conference

proceedings

21st USENIX Symposium on Networked Systems Design and Implementation (NSDI '24)

Santa Clara, CA, USA
April 16–18, 2024

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

ISBN 978-1-939133-39-7

Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI '24) Santa Clara, CA, USA April 16–18, 2024

NSDI '24 Sponsors

Gold Sponsors

Silver Sponsors

Bronze Sponsors

Open Access Sponsor

USENIX Supporters

USENIX Patrons

Futurewei • Google • Meta

USENIX Benefactor

Bloomberg

USENIX Partner

Thinkst Canary

Open Access Supporter

Google

Open Access Publishing Partner

PeerJ

USENIX Association

Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI '24)

April 16–18, 2024
Santa Clara, CA, USA

© 2024 by The USENIX Association

All Rights Reserved

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

ISBN 978-1-939133-39-7

Symposium Organizers

Program Co-Chairs

Laurent Vanbever, *ETH Zürich*

Irene Zhang, *Microsoft Research*

Program Committee

Sangeetha Abdu Jyothi, *University of California, Irvine, and VMware Research*

Mohammad Alizadeh, *Massachusetts Institute of Technology*

Peter Alvaro, *University of California, Santa Cruz*

Gianni Antichi, *Politecnico di Milano and Queen Mary University of London*

Maria Apostolaki, *Princeton University*

Vaibhav Bajpai, *CISPA Helmholtz Center for Information Security*

Tom Barbette, *Université catholique de Louvain*

Adam Belay, *MIT CSAIL*

Sonia Ben Mokhtar, *LIRIS CNRS*

Yérom-David Bromberg, *Inria de l'Université de Rennes*

Rodrigo Bruno, *INESC-ID and Instituto Superior Técnico, University of Lisbon*

Matthew Caesar, *University of Illinois at Urbana–Champaign*

Balakrishnan Chandrasekaran, *Vrije Universiteit Amsterdam*

Kang Chen, *Tsinghua University*

Rong Chen, *Shanghai Jiao Tong University*

Vijay Chidambaram, *The University of Texas at Austin and EPFL*

Marco Chiesa, *KTH Royal Institute of Technology*

Young-ri Choi, *UNIST (Ulsan National Institute of Science and Technology)*

Paolo Costa, *Microsoft Research*

Italo Cunha, *Universidade Federal de Minas Gerais*

Bruce Davie, *Systems Approach*

Shuwen Deng, *Yale University*

Aline Eid, *University of Michigan*

Eric Eide, *University of Utah*

Anja Feldmann, *Max Planck Institute for Informatics*

Ronaldo A. Ferreira, *Universidade Federal de Mato Grosso do Sul*

Fragkouli Georgia, *ETH Zurich*

Yasaman Ghasempour, *Princeton University*

Soudeh Ghorbani, *Johns Hopkins University*

Brighten Godfrey, *University of Illinois at Urbana–Champaign and VMware*

Sergey Gorinsky, *IMDEA Networks Institute*

Lisandro Zambenedetti Granville, *Universidade Federal do Rio Grande do Sul*

Arpit Gupta, *University of California, Santa Barbara*

Dongsu Han, *Korea Advanced Institute of Science and Technology (KAIST)*

Haitham Hassanieh, *EPFL*

Oliver Hohlfeld, *University of Kassel*

Yu Hua, Huazhong *University of Science and Technology*

Qun Huang, *Peking University*

Romain Jacob, *ETH Zurich*

Junchen Jiang, *University of Chicago*

Xin Jin, *Peking University*

Raj Joshi, *National University of Singapore*

Myoungsoo Jung, *Korea Advanced Institute of Science and Technology (KAIST)*

Kostis Kaffes, *Google*

Anuj Kalia, *Microsoft*

Zerina Kapetanovic, *Stanford University*

Sanidhya Kashyap, *EPFL*

Antoine Kaufmann, *Max Planck Institute for Software Systems (MPI-SWS)*

Daehyeok Kim, *The University of Texas at Austin*

Shir Landau Feibish, *The Open University of Israel*

Yanfang Le, *AMD*

Ben Leong, *National University of Singapore*

Philip Levis, *Stanford University and Google*

Amit Levy, *Princeton University*

Huaicheng Li, *Virginia Tech*

Jialin Li, *National University of Singapore*

Yuanjie Li, *Tsinghua University*

John Liagouris, *Boston University*

Kate Ching-Ju Lin, *National Yang-Ming Chiao Tung University*

Alan Zaoxing Liu, *Boston University*

Guyue Liu, *New York University Shanghai*

Ming Liu, *University of Wisconsin—Madison*

Mohammad Mazaheri, *University of Waterloo*

James Mickens, *Harvard University*

Akshay Narayan, *University of California, Berkeley*

Srinivas Narayana, *Rutgers University*

Deepak Narayanan, *NVIDIA*

Neha Narula, *Massachusetts Institute of Technology*

Ravi Netravali, *Princeton University*

Dave Oran, *Network Systems Research & Design*

Amy Ousterhout, *University of California, San Diego*

Kay Ousterhout, *Lightstep*

Jonggyu Park, *University of Washington*

Seo Jin Park, *University of Southern California and Google*

Luis Pedrosa, *INESC-ID and Instituto Superior Técnico, University of Lisbon*

Cristel Pelsser, *Université catholique de Louvain*

Amar Phanishayee, *Microsoft Research*

Dan Ports, *Microsoft Research*

Lili Qiu, *The University of Texas at Austin*

Rachee Singh

Barath Raghavan, *University of Southern California*

Costin Raiciu, *Politehnica University of Bucharest*

Gábor Rétvári, *Budapest University of Technology and Economics*

Michael Schapira, *The Hebrew University of Jerusalem*

Yizhou Shan, *Huawei Cloud*

Naveen Kr. Sharma, *Google*

Anirudh Sivaraman, *New York University*

Georgios Smaragdakis, *Delft University of Technology*

Elahe Soltanaghai, *University of Illinois Urbana–Champaign*

Ravi Soundararajan, *VMware*

Theano Stavrinos, *University of Washington*

Adriana Szekeres, *VMware Research*

Mina Tahmasbi Arashloo, *University of Waterloo*

Alain Tchana, *Grenoble INP*

Deepak Vasisht, *University of Illinois Urbana–Champaign*

Rashmi Vinayak, *Carnegie Mellon University*

Stefano Vissicchio, *University College London*
Zhaoguo Wang, *Shanghai Jiao Tong University*
Xingda Wei, *Shanghai Jiao Tong University*
Chenren Xu, *Peking University*
Hong Xu, *The Chinese University of Hong Kong*
Le Xu, *The University of Texas at Austin*
Neeraja J. Yadwadkar, *The University of Texas at Austin and VMware Research*
Francis Y. Yan, *Microsoft Research*
Suli Yang, *Google*
Nofel Yaseen, *Meta*
Yasir Zaki, *New York University*
Lixia Zhang, *University of California, Los Angeles*
Yongle Zhang, *Purdue University*
Wenting Zheng, *Carnegie Mellon University*
Danyang Zhuo, *Duke University*

Poster Session Co-Chairs
Kostis Kaffes, *Google*
Chenren Xu, *Peking University*

Test of Time Awards Committee
Mahesh Balakrishnan, *Yale University*
Manya Ghobadi, *MIT CSAIL*
Jon Howell, *VMware Research*
Jay Lorch, *Microsoft Research*
James Mickens, *Harvard University*
Amar Phanishayee, *Microsoft Research*
Minlan Yu, *Harvard University*

Steering Committee
Mahesh Balakrishnan, *Yale University*
Manya Ghobadi, *MIT CSAIL*
Casey Henderson-Ross, *USENIX Association*
Jon Howell, *VMware Research*
Jay Lorch, *Microsoft Research*
James Mickens, *Harvard University*
Amar Phanishayee, *Microsoft Research*
George Porter, *University of California, San Diego*
Renata Teixeira, *Netflix*
Minlan Yu, *Harvard University*

External Reviewers

Ryan Beckett
Chen Ding
Kyle Jamieson
Kyle Kingsbury
Lefteris Kokoris-Kogias
Jonathan Mace
Ilias Marinos
Ying Zhang

Message from the NSDI '24 Program Co-Chairs

Welcome to NSDI '24! This was a year of firsts for the conference. This is the first NSDI to have more than 100 PC members and accept more than 100 papers! It is also the first time that we have had an open call for PC members, leading to more PC members from different continents, countries, and institutions than ever before. Finally, for the first time ever, we had all PC members sign a code of conduct, which encouraged PC members to treat everyone involved in the review process (i.e., other PC members and authors) with respect and empathy. The reviewing process for a prestigious conference like NSDI is a stressful time for everyone, and we believe that the code of conduct led to a more productive outcome for everyone.

We are extremely pleased to have accepted a record 112 papers out of 601 submissions (a 7% increase with respect to NSDI '23, which was already a record year), yielding an 18.6% acceptance rate. 115 PC members were required over two deadlines to review this record number of submissions. For each of the two deadlines, the reviewing process consisted of two rounds of double-blind reviewing (which led to over 2500 written reviews) and an extensive online discussion. To accommodate the large number of PC members, both PC meetings were held over a three-day period. We are especially grateful to all of the PC members who called in (especially at odd hours) and made the PC meetings exceptionally smooth and pleasant. We'd also like to thank our Ph.D. students and post-doc scribes who managed PC members over 14 time zones (some of them changing time zones during the meeting!) for the online discussions.

Co-chairing NSDI is a hugely collaborative effort, and while it would be impossible for us to acknowledge every single person who helped us, we are incredibly indebted to the USENIX staff for all their hard work behind the scenes. We would also like to acknowledge the previous NSDI chairs and the NSDI steering committee members for sharing their experience and guiding us throughout the process. Last but not least, we would like to thank all the authors for submitting their best work to NSDI.

We look forward to seeing you all in Santa Clara for the NSDI '24!

Laurent Vanbever, *ETH Zürich*
Irene Zhang, *Microsoft Research*
NSDI '24 Program Co-Chairs

21st USENIX Symposium on Networked Systems Design and Implementation (NSDI '24)

April 16–18, 2024
Santa Clara, CA, USA

Tuesday, April 16

Clouds but Faster

Horus: Granular In-Network Task Scheduler for Cloud Datacenters 1
Parham Yassini, *Simon Fraser University;* Khaled Diab, *Hewlett Packard Labs;* Saeed Zangeneh and Mohamed Hefeeda, *Simon Fraser University*

Fast Vector Query Processing for Large Datasets Beyond GPU Memory with Reordered Pipelining 23
Zili Zhang, Fangyue Liu, Gang Huang, Xuanzhe Liu, and Xin Jin, *School of Computer Science, Peking University*

LoLKV: The Logless, Linearizable, RDMA-based Key-Value Storage System 41
Ahmed Alquraan and Sreeharsha Udayashankar, *University of Waterloo;* Virendra Marathe, *Oracle Labs;* Bernard Wong and Samer Al-Kiswany, *University of Waterloo*

Making Kernel Bypass Practical for the Cloud with Junction 55
Joshua Fried and Gohar Irfan Chaudhry, *MIT CSAIL;* Enrique Saurez, Esha Choukse, and Íñigo Goiri, *Azure Research – Systems;* Sameh Elnikety, *Microsoft Research;* Rodrigo Fonseca, *Azure Research – Systems;* Adam Belay, *MIT CSAIL*

Scheduling the Network

Sifter: An Inversion-Free and Large-Capacity Programmable Packet Scheduler 75
Peixuan Gao, Anthony Dalleggio, Jiajin Liu, and Chen Peng, *New York University;* Yang Xu, *Fudan University;* H. Jonathan Chao, *New York University*

Flow Scheduling with Imprecise Knowledge 95
Wenxin Li, Xin He, Yuan Liu, and Keqiu Li, *Tianjin University;* Kai Chen, *Hong Kong University of Science and Technology and University of Science and Technology of China;* Zhao Ge and Zewei Guan, *Tianjin University;* Heng Qi, *Dalian University of Technology;* Song Zhang, *Tianjin University;* Guyue Liu, *New York University Shanghai*

Pudica: Toward Near-Zero Queuing Delay in Congestion Control for Cloud Gaming. 113
Shibo Wang, *Xi'an Jiaotong University and Tencent Inc.;* Shusen Yang, *Xi'an Jiaotong University;* Xiao Kong, Chenglei Wu, and Longwei Jiang, *Tencent Inc.;* Chenren Xu, *Peking University;* Cong Zhao, *Xi'an Jiaotong University;* Xuesong Yang, *Bonree;* Jianjun Xiao and Xin Liu, *Tencent Inc.;* Changxi Zheng, *Pixel Lab, Tencent America, and Columbia University;* Jing Wang and Honghao Liu, *Tencent Inc.*

Revisiting Congestion Control for Lossless Ethernet 131
Yiran Zhang, *Tsinghua University and Beijing University of Posts and Telecommunications;* Qingkai Meng, *Tsinghua University and Beihang University;* Chaolei Hu and Fengyuan Ren, *Tsinghua University*

Serverless

Autothrottle: A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices 149
Zibo Wang, *University of Science and Technology of China and Microsoft Research;* Pinghe Li, *ETH Zurich;* Chieh-Jan Mike Liang, *Microsoft Research;* Feng Wu, *University of Science and Technology of China;* Francis Y. Yan, *Microsoft Research*

Jolteon: Unleashing the Promise of Serverless for Serverless Workflows 167
Zili Zhang, Chao Jin, and Xin Jin, *School of Computer Science, Peking University*

Can't Be Late: Optimizing Spot Instance Savings under Deadlines 185
Zhanghao Wu, Wei-Lin Chiang, Ziming Mao, and Zongheng Yang, *University of California, Berkeley;* Eric Friedman and Scott Shenker, *University of California, Berkeley, and ICSI;* Ion Stoica, *University of California, Berkeley*

Towards Intelligent Automobile Cockpit via A New Container Architecture 205
Lin Jiang and Feiyu Zhang, *Xi'an Yunzhiji Technology;* Jiang Ming, *Tulane University*

MuCache: a General Framework for Caching in Microservice Graphs **221**
Haoran Zhang, Konstantinos Kallas, Spyros Pavlatos, Rajeev Alur, Sebastian Angel, and Vincent Liu, *University of Pennsylvania*

Network Protocols

A large-scale deployment of DCTCP **239**
Abhishek Dhamija and Balasubramanian Madhavan, *Meta;* Hechao Li, *Netflix;* Jie Meng, Shrikrishna Khare, and Madhavi Rao, *Meta;* Lawrence Brakmo*;* Neil Spring, Prashanth Kannan, and Srikanth Sundaresan, *Meta;* Soudeh Ghorbani, *Meta and Johns Hopkins University*

TECC: Towards Efficient QUIC Tunneling via Collaborative Transmission Control **253**
Jiaxing Zhang, *Alibaba Group and University of Chinese Academy of Sciences;* Furong Yang, *Alibaba Group;* Ting Liu, *Alibaba Group and University of Chinese Academy of Sciences;* Qinghua Wu, *University of Chinese Academy of Sciences and Purple Mountain Laboratories, China;* Wu Zhao, Yuanbo Zhang, Wentao Chen, Yanmei Liu, Hongyu Guo, and Yunfei Ma, *Alibaba Group;* Zhenyu Li, *University of Chinese Academy of Sciences and Purple Mountain Laboratories, China*

iStack: A General and Stateful Name-based Protocol Stack for Named Data Networking **267**
Tianlong Li, Tian Song, and Yating Yang, *Beijing Institute of Technology*

Cloudcast: High-Throughput, Cost-Aware Overlay Multicast in the Cloud. **281**
Sarah Wooders and Shu Liu, *UC Berkeley;* Paras Jain, *Genmo AI;* Xiangxi Mo and Joseph Gonzalez, *UC Berkeley;* Vincent Liu, *University of Pennsylvania;* Ion Stoica, *UC Berkeley*

Understanding Routable PCIe Performance for Composable Infrastructures **297**
Wentao Hou, *University of Wisconsin-Madison;* Jie Zhang and Zeke Wang, *Zhejiang University;* Ming Liu, *University of Wisconsin-Madison*

Distributed Systems: Part 1

Alea-BFT: Practical Asynchronous Byzantine Fault Tolerance **313**
Diogo S. Antunes, Afonso N. Oliveira, André Breda, Matheus Guilherme Franco, Henrique Moniz, and Rodrigo Rodrigues, *Instituto Superior Técnico* (*ULisboa*) *and INESC-ID*

Harmony: A Congestion-free Datacenter Architecture **329**
Saksham Agarwal, Qizhe Cai, Rachit Agarwal, and David Shmoys, *Cornell University;* Amin Vahdat, *Google*

SwiftPaxos: Fast Geo-Replicated State Machines **345**
Fedor Ryabinin, *IMDEA Software Institute and Universidad Politécnica de Madrid;* Alexey Gotsman, *IMDEA Software Institute;* Pierre Sutra, *Télécom SudParis and INRIA*

The Bedrock of Byzantine Fault Tolerance: A Unified Platform for BFT Protocols Analysis, Implementation, and Experimentation **371**
Mohammad Javad Amiri, *Stony Brook University;* Chenyuan Wu, *University of Pennsylvania;* Divyakant Agrawal and Amr El Abbadi, *UC Santa Barbara;* Boon Thau Loo, *University of Pennsylvania;* Mohammad Sadoghi, *UC Davis*

DINT: Fast In-Kernel Distributed Transactions with eBPF **401**
Yang Zhou, *Harvard University;* Xingyu Xiang, *Peking University;* Matthew Kiley, *Harvard University;* Sowmya Dharanipragada, *Cornell University;* Minlan Yu, *Harvard University*

Programming the Network: Part 1

Brain-on-Switch: Towards Advanced Intelligent Network Data Plane via NN-Driven Traffic Analysis at Line-Speed. **419**
Jinzhu Yan and Haotian Xu, *Tsinghua University* Zhuotao Liu, Qi Li, Ke Xu, Mingwei Xu, and Jianping Wu, *Tsinghua University and Zhongguancun Laboratory*

The Eternal Tussle: Exploring the Role of Centralization in IPFS **441**
Yiluo Wei, *Hong Kong University of Science & Technology* (*GZ*)*;* Dennis Trautwein and Yiannis Psaras, *Protocol Labs;* Ignacio Castro, *Queen Mary University of London;* Will Scott, *Protocol Labs;* Aravindh Raman, *Brave Software;* Gareth Tyson, *Hong Kong University of Science & Technology* (*GZ*)

BBQ: A Fast and Scalable Integer Priority Queue for Hardware Packet Scheduling. **455**
Nirav Atre, Hugo Sadok, and Justine Sherry, *Carnegie Mellon University*

Sirius: Composing Network Function Chains into P4-Capable Edge Gateways **477**
Jiaqi Gao, Jiamin Cao, Yifan Li, Mengqi Liu, Ming Tang, Dennis Cai, and Ennan Zhai, *Alibaba Cloud*

Empower Programmable Pipeline for Advanced Stateful Packet Processing **491**
Yong Feng and Zhikang Chen, *Tsinghua University;* Haoyu Song, *Futurewei Technologies;* Yinchao Zhang, Hanyi Zhou, Ruoyu Sun, Wenkuo Dong, Peng Lu, Shuxin Liu, and Chuwen Zhang, *Tsinghua University;* Yang Xu, *Fudan University;* Bin Liu, *Tsinghua University*

Video

GRACE: Loss-Resilient Real-Time Video through Neural Codecs **509**
Yihua Cheng, Ziyi Zhang, Hanchen Li, Anton Arapin, and Yue Zhang, *The University of Chicago;* Qizheng Zhang, *Stanford University;* Yuhan Liu, Kuntai Du, and Xu Zhang, *The University of Chicago;* Francis Y. Yan, *Microsoft;* Amrita Mazumdar, *NVIDIA;* Nick Feamster and Junchen Jiang, *The University of Chicago*

LiFteR: Unleash Learned Codecs in Video Streaming with Loose Frame Referencing **533**
Bo Chen, *University of Illinois at Urbana-Champaign;* Zhisheng Yan, *George Mason University;* Yinjie Zhang, Zhe Yang, and Klara Nahrstedt, *University of Illinois at Urbana-Champaign*

MadEye: Boosting Live Video Analytics Accuracy with Adaptive Camera Configurations **549**
Mike Wong and Murali Ramanujam, *Princeton University;* Guha Balakrishnan, *Rice University;* Ravi Netravali, *Princeton University*

Gemino: Practical and Robust Neural Compression for Video Conferencing **569**
Vibhaalakshmi Sivaraman, Pantea Karimi, Vedantha Venkatapathy, and Mehrdad Khani, *Massachusetts Institute of Technology;* Sadjad Fouladi, *Microsoft Research;* Mohammad Alizadeh, Frédo Durand, and Vivienne Sze, *Massachusetts Institute of Technology*

ARTEMIS: Adaptive Bitrate Ladder Optimization for Live Video Streaming **591**
Farzad Tashtarian, *Christian Doppler Laboratory ATHENA, Alpen-Adria Universität Klagenfurt;* Abdelhak Bentaleb, *Concordia University;* Hadi Amirpour, *Christian Doppler Laboratory ATHENA, Alpen-Adria Universität Klagenfurt;* Sergey Gorinsky, *IMDEA Networks Institute;* Junchen Jiang, *University of Chicago;* Hermann Hellwagner and Christian Timmerer, *Christian Doppler Laboratory ATHENA, Alpen-Adria Universität Klagenfurt*

Sharing the Network

Credence: Augmenting Datacenter Switch Buffer Sharing with ML Predictions **613**
Vamsi Addanki, Maciej Pacut, and Stefan Schmid, *TU Berlin*

Seer: Enabling Future-Aware Online Caching in Networked Systems **635**
Jason Lei and Vishal Shrivastav, *Purdue University*

Reverie: Low Pass Filter-Based Switch Buffer Sharing for Datacenters with RDMA and TCP Traffic **651**
Vamsi Addanki, *TU Berlin;* Wei Bai, *Microsoft Research;* Stefan Schmid, *TU Berlin;* Maria Apostolaki, *Princeton University*

Precise Data Center Traffic Engineering with Constrained Hardware Resources **669**
Shawn Shuoshuo Chen, *Carnegie Mellon University;* Keqiang He, *Airbnb;* Rui Wang, *Google;* Srinivasan Seshan and Peter Steenkiste, *Carnegie Mellon University*

Multitenant In-Network Acceleration with SwitchVM **691**
Sajy Khashab, Alon Rashelbach, and Mark Silberstein, *Technion*

Wednesday, April 17

ML at Scale

Characterization of Large Language Model Development in the Datacenter **709**
Qinghao Hu, *Shanghai AI Laboratory and S-Lab, Nanyang Technological University;* Zhisheng Ye, *Shanghai AI Laboratory and Peking University;* Zerui Wang, *Shanghai AI Laboratory and Shanghai Jiao Tong University;* Guoteng Wang, *Shanghai AI Laboratory;* Meng Zhang and Qiaoling Chen, *Shanghai AI Laboratory and S-Lab, Nanyang Technological University;* Peng Sun, *Shanghai AI Laboratory and SenseTime Research;* Dahua Lin, *Shanghai AI Laboratory and CUHK;* Xiaolin Wang and Yingwei Luo, *Peking University;* Yonggang Wen and Tianwei Zhang, *Nanyang Technological University*

QuickUpdate: a Real-Time Personalization System for Large-Scale Recommendation Models 731
Kiran Kumar Matam, Hani Ramezani, Fan Wang, Zeliang Chen, Yue Dong, Maomao Ding, Zhiwei Zhao, Zhengyu Zhang, Ellie Wen, and Assaf Eisenman, *Meta, Inc*.

MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs 745
Ziheng Jiang and Haibin Lin, *ByteDance;* Yinmin Zhong, *Peking University;* Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, and Jianxi Ye, *ByteDance;* Xin Jin, *Peking University;* Xin Liu, *ByteDance*

Resiliency at Scale: Managing Google's TPUv4 Machine Learning Supercomputer 761
Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, Steve Lacy, Hang Wang, Aaron Wisner, Chris Lewis, and Henri Bahini, *Google*

Satellites and Things

NN-Defined Modulator: Reconfigurable and Portable Software Modulator on IoT Gateways 775
Jiazhao Wang and Wenchao Jiang, *Singapore University of Technology and Design;* Ruofeng Liu, *University of Minnesota;* Bin Hu, *University of Southern California;* Demin Gao, *Nanjing Forestry University;* Shuai Wang, *Southeast University*

Democratizing Direct-to-Cell Low Earth Orbit Satellite Networks 791
Lixin Liu, *Tsinghua University;* Yuanjie Li and Hewu Li, *Tsinghua University and Zhongguancun Laboratory;* Jiabo Yang, Wei Liu, Jingyi Lan, Yufeng Wang, and Jiarui Li, *Tsinghua University;* Jianping Wu, Qian Wu, Jun Liu, and Zeqi Lai, *Tsinghua University and Zhongguancun Laboratory*

Known Knowns and Unknowns: Near-realtime Earth Observation Via Query Bifurcation in Serval 809
Bill Tao, Om Chabra, Ishani Janveja, Indranil Gupta, and Deepak Vasisht, *University of Illinois Urbana-Champaign*

Spectrumize: Spectrum-efficient Satellite Networks for the Internet of Things 825
Vaibhav Singh, Tusher Chakraborty, and Suraj Jog, *Microsoft Research;* Om Chabra and Deepak Vasisht, *UIUC;* Ranveer Chandra, *Microsoft Research*

Wide-Area and Edge

Application-Level Service Assurance with 5G RAN Slicing 841
Arjun Balasingam, *MIT CSAIL;* Manikanta Kotaru and Paramvir Bahl, *Microsoft*

CHISEL: An optical slice of the wide-area network 859
Abhishek Vijaya Kumar, *Cornell University;* Bill Owens, *NYSERnet;* Nikolaj Bjørner, Binbin Guan, Yawei Yin, and Paramvir Bahl, *Microsoft;* Rachee Singh, *Cornell University*

LuoShen: A Hyper-Converged Programmable Gateway for Multi-Tenant Multi-Service Edge Clouds 877
Tian Pan, Kun Liu, Xionglie Wei, Yisong Qiao, Jun Hu, Zhiguo Li, Jun Liang, Tiesheng Cheng, Wenqiang Su, Jie Lu, Yuke Hong, Zhengzhong Wang, Zhi Xu, Chongjing Dai, Peiqiao Wang, Xuetao Jia, Jianyuan Lu, Enge Song, and Jun Zeng, *Alibaba Cloud;* Biao Lyu, *Zhejiang University and Alibaba Cloud;* Ennan Zhai, *Alibaba Cloud;* Jiao Zhang and Tao Huang, *Purple Mountain Laboratories;* Dennis Cai, *Alibaba Cloud;* Shunmin Zhu, *Tsinghua University and Alibaba Cloud*

Sprinter: Speeding Up High-Fidelity Crawling of the Modern Web 893
Ayush Goel and Jingyuan Zhu, *University of Michigan;* Ravi Netravali, *Princeton University;* Harsha V. Madhyastha, *University of Southern California*

Hairpin: Rethinking Packet Loss Recovery in Edge-based Interactive Video Streaming 907
Zili Meng, *Tsinghua University, Hong Kong University of Science and Technology, and Tencent;* Xiao Kong and Jing Chen, *Tsinghua University and Tencent;* Bo Wang and Mingwei Xu, *Tsinghua University;* Rui Han and Honghao Liu, *Tencent;* Venkat Arun, *UT Austin;* Hongxin Hu, *University at Buffalo, SUNY;* Xue Wei, *Tencent*

Verification

Finding Adversarial Inputs for Heuristics using Multi-level Optimization 927
Pooria Namyar, *Microsoft and University of Southern California;* Behnaz Arzani and Ryan Beckett, *Microsoft;* Santiago Segarra, *Microsoft and Rice University;* Himanshu Raj and Umesh Krishnaswamy, *Microsoft;* Ramesh Govindan, *University of Southern California;* Srikanth Kandula, *Microsoft*

Towards provably performant congestion control 951
Anup Agarwal, *Carnegie Mellon University;* Venkat Arun, *University of Texas at Austin;* Devdeep Ray, Ruben Martins, and Srinivasan Seshan, *Carnegie Mellon University*

EPVerifier: Accelerating Update Storms Verification with Edge-Predicate 979
Chenyang Zhao, Yuebin Guo, Jingyu Wang, Qi Qi, Zirui Zhuang, Haifeng Sun, and Lingqi Guo, *State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications;* Yuming Xie, *Huawei Technologies Co., Ltd;* Jianxin Liao, *State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications*

Netcastle: Network Infrastructure Testing At Scale 993
Rob Sherwood, *NetDebug.com;* Jinghao Shi, Ying Zhang, Neil Spring, Srikanth Sundaresan, Jasmeet Bagga, Prathyusha Peddi, Vineela Kukkadapu, Rashmi Shrivastava, Manikantan KR, Pavan Patil, Srikrishna Gopu, Varun Varadan, Ethan Shi, Hany Morsy, Yuting Bu, Renjie Yang, Rasmus Jönsson, Wei Zhang, Jesus Jussepen Arredondo, and Diana Saha, *Meta Platforms Inc.;* Sean Choi, *Santa Clara University*

MESSI: Behavioral Testing of BGP Implementations 1009
Rathin Singha and Rajdeep Mondal, *University of California Los Angeles;* Ryan Beckett, *Microsoft;* Siva Kesava Reddy Kakarla, *Microsoft Research;* Todd Millstein and George Varghese, *University of California Los Angeles*

Networking at Scale

A High-Performance Design, Implementation, Deployment, and Evaluation of The Slim Fly Network 1025
Nils Blach and Maciej Besta, *ETH Zürich;* Daniele De Sensi, *ETH Zürich and Sapienza University of Rome;* Jens Domke, *RIKEN Center for Computational Science* (*R-CCS*)*;* Hussein Harake, *Swiss National Supercomputing Centre* (*CSCS*)*;* Shigang Li, *ETH Zürich and BUPT, Beijing;* Patrick Iff, *ETH Zürich;* Marek Konieczny, *AGH-UST;* Kartik Lakhotia, *Intel Labs;* Ales Kubicek and Marcel Ferrari, *ETH Zürich;* Fabrizio Petrini, *Intel Labs;* Torsten Hoefler, *ETH Zürich*

Crescent: Emulating Heterogeneous Production Network at Scale. 1045
Zhaoyu Gao and Anubhavnidhi Abhashkumar, *ByteDance;* Zhen Sun, *Cornell University;* Weirong Jiang and Yi Wang, *ByteDance*

Reasoning about Network Traffic Load Property at Production Scale 1063
Ruihan Li, *Peking University and Alibaba Cloud;* Fangdan Ye, Yifei Yuan, Ruizhen Yang, Bingchuan Tian, Tianchen Guo, Hao Wu, Xiaobo Zhu, Zhongyu Guan, Qing Ma, and Xianlong Zeng, *Alibaba Cloud;* Chenren Xu, *Peking University;* Dennis Cai and Ennan Zhai, *Alibaba Cloud*

Poseidon: A Consolidated Virtual Network Controller that Manages Millions of Tenants via Config Tree 1083
Biao Lyu, *Zhejiang University and Alibaba Cloud;* Enge Song, Tian Pan, Jianyuan Lu, Shize Zhang, Xiaoqing Sun, Lei Gao, Chenxiao Wang, Han Xiao, Yong Pan, Xiuheng Chen, Yandong Duan, Weisheng Wang, Jinpeng Long, Yanfeng Wang, Kunpeng Zhou, and Zhigang Zong, *Alibaba Cloud;* Xing Li, *Zhejiang University and Alibaba Cloud;* Guangwang Li and Pengyu Zhang, *Alibaba Cloud;* Peng Cheng and Jiming Chen, *Zhejiang University;* Shunmin Zhu, *Tsinghua University and Alibaba Cloud*

OPPerTune: Post-Deployment Configuration Tuning of Services Made Easy 1101
Gagan Somashekar, *Stony Brook University;* Karan Tandon and Anush Kini, *Microsoft Research;* Chieh-Chun Chang and Petr Husak, *Microsoft;* Ranjita Bhagwan, *Google;* Mayukh Das, *Microsoft365 Research;* Anshul Gandhi, *Stony Brook University;* Nagarajan Natarajan, *Microsoft Research*

ML but Faster

Parcae: Proactive, Liveput-Optimized DNN Training on Preemptible Instances 1121
Jiangfei Duan, *The Chinese University of Hong Kong;* Ziang Song, *ByteDance;* Xupeng Miao and Xiaoli Xi, *Carnegie Mellon University;* Dahua Lin, *The Chinese University of Hong Kong;* Harry Xu, *University of California, Los Angeles;* Minjia Zhang, *Microsoft;* Zhihao Jia, *Carnegie Mellon University*

Accelerating Neural Recommendation Training with Embedding Scheduling 1141
Chaoliang Zeng, Xudong Liao, Xiaodian Cheng, Han Tian, Xinchen Wan, Hao Wang, and Kai Chen, *iSING Lab, Hong Kong University of Science and Technology*

DistMM: Accelerating Distributed Multimodal Model Training 1157
Jun Huang, *The Ohio State University;* Zhen Zhang, *Amazon Web Services;* Shuai Zheng, *Boson AI;* Feng Qin, *The Ohio State University;* Yida Wang, *Amazon Web Services*

Approximate Caching for Efficiently Serving Text-to-Image Diffusion Models 1173
Shubham Agarwal and Subrata Mitra, *Adobe Research;* Sarthak Chakraborty, *UIUC;* Srikrishna Karanam, Koyel Mukherjee, and Shiv Kumar Saini, *Adobe Research*

THC: Accelerating Distributed Deep Learning Using Tensor Homomorphic Compression 1191
Minghao Li, *Harvard University;* Ran Ben Basat, *University College London;* Shay Vargaftik, *VMware Research;* ChonLam Lao, Kevin Xu, Michael Mitzenmacher, and Minlan Yu, *Harvard University*

Distributed Systems: Part 2

Accelerating Skewed Workloads With Performance Multipliers in the TurboDB Distributed Database 1213
Jennifer Lam, Jeffrey Helt, and Wyatt Lloyd, *Princeton University;* Haonan Lu, *University at Buffalo*

SIEVE is Simpler than LRU: an Efficient Turn-Key Eviction Algorithm for Web Caches 1229
Yazhuo Zhang, *Emory University;* Juncheng Yang, *Carnegie Mellon University;* Yao Yue, *Pelikan Foundation;* Ymir Vigfusson, *Emory University and Keystrike;* K.V. Rashmi, *Carnegie Mellon University*

Harvesting Idle Memory for Application-Managed Soft State with Midas 1247
Yifan Qiao, *UCLA;* Zhenyuan Ruan, *MIT CSAIL;* Haoran Ma, *UCLA;* Adam Belay, *MIT CSAIL;* Miryung Kim and Harry Xu, *UCLA*

Efficient Exposure of Partial Failure Bugs in Distributed Systems with Inferred Abstract States 1267
Haoze Wu and Jia Pan, *Johns Hopkins University;* Peng Huang, *University of Michigan*

Load is not what you should balance: Introducing Prequal 1285
Bartek Wydrowski, *Google Research;* Robert Kleinberg, *Google Research and Cornell;* Stephen M. Rumble, *Google* (*YouTube*)*;* Aaron Archer, *Google Research*

Wireless Hardware

Orthcatter: High-throughput In-band OFDM Backscatter with Over-the-Air Code Division 1301
Caihui Du and Jihong Yu, *Beijing Institute of Technology;* Rongrong Zhang, *Capital Normal University;* Ju Ren, *Tsinghua University;* Jianping An, *Beijing Institute of Technology*

EdgeRIC: Empowering Realtime Intelligent Optimization and Control in NextG Cellular Networks 1315
Woo-Hyun Ko, *Texas A&M University;* Ushasi Ghosh, *University of California San Diego;* Ujwal Dinesha, *Texas A&M University;* Raini Wu, *University of California San Diego;* Srinivas Shakkottai, *Texas A&M University;* Dinesh Bharadia, *University of California San Diego*

ADR-X: ANN-Assisted Wireless Link Rate Adaptation for Compute-Constrained Embedded Gaming Devices . . 1331
Hao Yin, *University of Washington;* Murali Ramanujam, *Princeton University;* Joe Schaefer, Stan Adermann, Srihari Narlanka, and Perry Lea, *Microsoft;* Ravi Netravali, *Princeton University;* Krishna Chintalapudi, *Microsoft Research*

RFID+: Spatially Controllable Identification of UHF RFIDs via Controlled Magnetic Fields 1351
Donghui Dai, *The Hong Kong Polytechnic University;* Zhenlin An, *The Hong Kong Polytechnic University and Princeton University;* Zheng Gong, *The Hong Kong Polytechnic University;* Qingrui Pan, *The Hong Kong Polytechnic University and The University of Edinburgh;* Lei Yang, *Shenzhen Research Institute, The Hong Kong Polytechnic University*

SMUFF: Towards Line Rate Wi-Fi Direct Transport with Orchestrated On-device Buffer Management 1369
Chengke Wang, *Peking University;* Hao Wang, *Shenzhen Kaihong Digital Industry Development Co., Ltd.;* Yuhan Zhou and Yunzhe Ni, *Peking University;* Feng Qian, *University of Southern California;* Chenren Xu, *Peking University, Zhongguancun Laboratory, and Key Laboratory of High Confidence Software Technologies, Ministry of Education (PKU)*

Thursday, April 18

ML Scheduling

Vulcan: Automatic Query Planning for Live ML Analytics. 1385
Yiwen Zhang and Xumiao Zhang, *University of Michigan;* Ganesh Ananthanarayanan, *Microsoft;* Anand Iyer, *Georgia Institute of Technology;* Yuanchao Shu, *Zhejiang University;* Victor Bahl, *Microsoft Corporation;* Z. Morley Mao, *University of Michigan and Google;* Mosharaf Chowdhury, *University of Michigan*

CASSINI: Network-Aware Job Scheduling in Machine Learning Clusters 1403
Sudarsanan Rajasekaran and Manya Ghobadi, *Massachusetts Institute of Technology;* Aditya Akella, *UT Austin*

Towards Domain-Specific Network Transport for Distributed DNN Training 1421
Hao Wang and Han Tian, *iSING Lab, Hong Kong University of Science and Technology;* Jingrong Chen, *Duke University;* Xinchen Wan, Jiacheng Xia, and Gaoxiong Zeng, *iSING Lab, Hong Kong University of Science and Technology;* Wei Bai, *Microsoft;* Junchen Jiang, *University of Chicago;* Yong Wang and Kai Chen, *iSING Lab, Hong Kong University of Science and Technology*

Swing: Short-cutting Rings for Higher Bandwidth Allreduce 1445
Daniele De Sensi, *Sapienza University of Rome;* Tommaso Bonato, *ETH Zurich;* David Saam, *RWTH Aachen University;* Torsten Hoefler, *ETH Zurich*

Cloud Scheduling

LitePred: Transferable and Scalable Latency Prediction for Hardware-Aware Neural Architecture Search 1463
Chengquan Feng, *University of Science and Technology of China;* Li Lyna Zhang, *Microsoft Research;* Yuanchi Liu, *University of Science and Technology of China;* Jiahang Xu and Chengruidong Zhang, *Microsoft Research;* Zhiyuan Wang, *University of Science and Technology of China;* Ting Cao and Mao Yang, *Microsoft Research;* Haisheng Tan, *University of Science and Technology of China*

Harmonic: Hardware-assisted RDMA Performance Isolation for Public Clouds 1479
Jiaqi Lou, *University of Illinois Urbana-Champaign;* Xinhao Kong, *Duke University;* Jinghan Huang, *University of Illinois Urbana-Champaign;* Wei Bai, *Microsoft;* Nam Sung Kim, *University of Illinois Urbana-Champaign;* Danyang Zhuo, *Duke University*

LDB: An Efficient Latency Profiling Tool for Multithreaded Applications 1497
Inho Cho, *MIT CSAIL;* Seo Jin Park, *University of Southern California;* Ahmed Saeed, *Georgia Tech;* Mohammad Alizadeh and Adam Belay, *MIT CSAIL*

UFO: The Ultimate QoS-Aware Core Management for Virtualized and Oversubscribed Public Clouds 1511
Yajuan Peng, *Southern University of Science and Technology and Shenzhen Institutes of Advanced Technology, Chinese Academy of Science;* Shuang Chen and Yi Zhao, *Shuhai Lab, Huawei Cloud;* Zhibin Yu, *Shuhai Lab, Huawei Cloud, and Shenzhen Institutes of Advanced Technology, Chinese Academy of Science*

Programming the Network: Part 2

Automatic Parallelization of Software Network Functions 1531
Francisco Pereira, Fernando M.V. Ramos, and Luis Pedrosa, *INESC-ID, Instituto Superior Técnico, University of Lisbon*

AutoSketch: Automatic Sketch-Oriented Compiler for Query-driven Network Telemetry 1551
Haifeng Sun and Qun Huang, *National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University;* Jinbo Sun, *Institute of Computing Technology, Chinese Academy of Sciences;* Wei Wang, *Northeastern University, China;* Jiaheng Li, *National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University;* Fuliang Li, *Northeastern University, China;* Yungang Bao, *Institute of Computing Technology, Chinese Academy of Sciences;* Xin Yao and Gong Zhang, *Huawei Theory Department*

Leo: Online ML-based Traffic Classification at Multi-Terabit Line Rate 1573
Syed Usman Jafri, Sanjay Rao, Vishal Shrivastav, and Mohit Tawarmalani, *Purdue University*

Sequence Abstractions for Flexible, Line-Rate Network Monitoring 1593
Andrew Johnson, *Princeton University;* Ryan Beckett, *Microsoft Research;* Xiaoqi Chen, *Princeton University;* Ratul Mahajan, *University of Washington;* David Walker, *Princeton University*

OctoSketch: Enabling Real-Time, Continuous Network Monitoring over Multiple Cores 1621
Yinda Zhang, *University of Pennsylvania;* Peiqing Chen and Zaoxing Liu, *University of Maryland*

Wireless Sensing

NR-Surface: NextG-ready μW-reconfigurable mmWave Metasurface 1641
Minseok Kim, Namjo Ahn, and Song Min Kim, *KAIST*

Cyclops: A Nanomaterial-based, Battery-Free Intraocular Pressure (IOP) Monitoring System inside Contact Lens 1659
Liyao Li, *University at Buffalo SUNY and Northwest University;* Bozhao Shang and Yun Wu, *Northwest University and Shaanxi International Joint Research Centre for the Battery-Free Internet of Things;* Jie Xiong, *University of Massachusetts Amherst and Microsoft Research Asia;* Xiaojiang Chen, *Northwest University and Shaanxi International Joint Research Centre for the Battery-Free Internet of Things;* Yaxiong Xie, *University at Buffalo SUNY*

Habitus: Boosting Mobile Immersive Content Delivery through Full-body Pose Tracking and Multipath Networking **1677**
Anlan Zhang, *University of Southern California;* Chendong Wang, *University of Wisconsin — Madison;* Yuming Hu, *University of Minnesota — Twin Cities;* Ahmad Hassan and Zejun Zhang, *University of Southern California;* Bo Han, *George Mason University;* Feng Qian, *University of Southern California;* Shichang Xu, *Google*

BFMSense: WiFi Sensing Using Beamforming Feedback Matrix **1697**
Enze Yi and Dan Wu, *Peking University;* Jie Xiong, *University of Massachusetts Amherst;* Fusang Zhang, *Institute of Software, Chinese Academy of Sciences and University of Chinese Academy of Sciences;* Kai Niu, *Beijing Xiaomi Mobile Software Company Ltd.;* Wenwei Li, *Peking University;* Daqing Zhang, *Peking University and Institut Polytechnique de Paris*

mmComb: High-speed mmWave Commodity WiFi Backscatter **1713**
Yoon Chae and Zhenzhe Lin, *George Mason University;* Kang Min Bae and Song Min Kim, *Korea Advanced Institute of Science and Technology* (*KAIST*)*;* Parth Pathak, *George Mason University*

Security

Where The Wild Things Are: Brute-Force SSH Attacks In The Wild And How To Stop Them **1731**
Sachin Kumar Singh and Shreeman Gautam, *University of Utah;* Cameron Cartier, *University of Utah and Black Hills Information Security;* Sameer Patil and Robert Ricci, *University of Utah*

A System to Detect Forged-Origin BGP Hijacks **1751**
Thomas Holterbach and Thomas Alfroy, *University of Strasbourg;* Amreesh Phokeer, *Internet Society;* Alberto Dainotti, *Georgia Tech;* Cristel Pelsser, *UCLouvain*

NetVigil: Robust and Low-Cost Anomaly Detection for East-West Data Center Security **1771**
Kevin Hsieh, *Microsoft;* Mike Wong, *Princeton University and Microsoft;* Santiago Segarra, *Microsoft and Rice University;* Sathiya Kumaran Mani, Trevor Eberl, and Anatoliy Panasyuk, *Microsoft;* Ravi Netravali, *Princeton University;* Ranveer Chandra and Srikanth Kandula, *Microsoft*

Tango: Secure Collaborative Route Control across the Public Internet **1791**
Henry Birge-Lee, Sophia Yoo, Benjamin Herber, Jennifer Rexford, and Maria Apostolaki, *Princeton University*

Sidekick: In-Network Assistance for Secure End-to-End Transport Protocols **1813**
Gina Yuan, Matthew Sotoudeh, and David K. Zhang, *Stanford University;* Michael Welzl, *University of Oslo;* David Mazières and Keith Winstein, *Stanford University*

Mobile Things

VILAM: Infrastructure-assisted 3D Visual Localization and Mapping for Autonomous Driving **1831**
Jiahe Cui, *Beihang University, The Chinese University of Hong Kong, and Tianmushan Laboratory;* Shuyao Shi and Yuze He, *The Chinese University of Hong Kong;* Jianwei Niu, *Beihang University;* Guoliang Xing, *The Chinese University of Hong Kong;* Zhenchao Ouyang, *Tianmushan Laboratory and International Innovation Institute of Beihang University*

Catch Me If You Can: Laser Tethering with Highly Mobile Targets **1847**
Charles J. Carver, Hadleigh Schwartz, and Qijia Shao, *Columbia University;* Nicholas Shade, Joseph Lazzaro, Xiaoxin Wang, Jifeng Liu, and Eric Fossum, *Dartmouth College;* Xia Zhou, *Columbia University*

MobileConfig: Remote Configuration Management for Mobile Apps at Hyperscale **1867**
Matt Guo, *Meta Platforms;* Soteris Demetriou, *Imperial College London;* Joey Yang, Michael Leighton, Diedi Hu, Tong Bao, Amit Adhikari, Thawan Kooburat, Annie Kim, and Chunqiang Tang, *Meta Platforms*

Passengers' Safety Matters: Experiences of Deploying a Large-Scale Indoor Delivery Monitoring System **1883**
Xiubin Fan, *City University of Hong Kong;* Zhongming Lin, *The Hong Kong University of Science and Technology;* Yuming Hu, *University of Minnesota - Twin Cities;* Tianrui Jiang, *The Hong Kong University of Science and Technology;* Feng Qian, *University of Southern California;* Zhimeng Yin, *City University of Hong Kong;* S.-H. Gary Chan, *The Hong Kong University of Science and Technology;* Dapeng Wu, *City University of Hong Kong*

Augur: Practical Mobile Multipath Transport Service for Low Tail Latency in Real-Time Streaming **1901**
Yuhan Zhou, *School of Computer Science, Peking University and Tencent Inc.;* Tingfeng Wang, *Tencent Inc.;* Liying Wang, *School of Computer Science, Peking University;* Nian Wen, Rui Han, Jing Wang, Chenglei Wu, Jiafeng Chen, and Longwei Jiang, *Tencent Inc.;* Shibo Wang, *Xi'an Jiaotong University and Tencent Inc.;* Honghao Liu, *Tencent Inc.;* Chenren Xu, *School of Computer Science, Peking University and Zhongguancun Laboratory and Key Laboratory of High Confidence Software Technologies, Ministry of Education* (*PKU*)

Cloud Systems

Zombie: Middleboxes that Don't Snoop 1917
Collin Zhang, *Cornell;* Zachary DeStefano, Arasu Arun, and Joseph Bonneau, *NYU;* Paul Grubbs, *University of Michigan;* Michael Walfish, *NYU*

Solving Max-Min Fair Resource Allocations Quickly on Large Graphs 1937
Pooria Namyar, *Microsoft and University of Southern California;* Behnaz Arzani and Srikanth Kandula, *Microsoft;* Santiago Segarra, *Microsoft and Rice University;* Daniel Crankshaw and Umesh Krishnaswamy, *Microsoft;* Ramesh Govindan, *University of Southern California;* Himanshu Raj, *Microsoft*

Cloud-LoRa: Enabling Cloud Radio Access LoRa Networks Using Reinforcement Learning Based Bandwidth-Adaptive Compression 1959
Muhammad Osama Shahid, Daniel Koch, Jayaram Raghuram, and Bhuvana Krishnaswamy, *University of Wisconsin-Madison;* Krishna Chintalapudi, *Microsoft Research;* Suman Banerjee, *University of Wisconsin-Madison*

Cloudy with a Chance of Cyberattacks: Dangling Resources Abuse on Cloud Platforms 1977
Jens Frieß, *National Research Center for Applied Cybersecurity ATHENE and Technische Universität Darmstadt;* Tobias Gattermayer, *National Research Center for Applied Cybersecurity ATHENE and Fraunhofer Institute for Secure Information Technology SIT;* Nethanel Gelernter, *IONIX;* Haya Schulmann, *National Research Center for Applied Cybersecurity ATHENE and Goethe-Universität Frankfurt;* Michael Waidner, *National Research Center for Applied Cybersecurity ATHENE, Technische Universität Darmstadt, and Fraunhofer Institute for Secure Information Technology SIT*

Modeling Networks

CAPA: An Architecture For Operating Cluster Networks With High Availability. 1995
Bingzhe Liu, *UIUC;* Colin Scott, Mukarram Tariq, Andrew Ferguson, Phillipa Gill, Richard Alimi, Omid Alipourfard, Deepak Arulkannan, Virginia Jean Beauregard, and Patrick Conner, *Google;* P. Brighten Godfrey, *UIUC;* Xander Lin, Joon Ong, Mayur Patel, Amr Sabaa, Arjun Singh, Alex Smirnov, Manish Verma, Prerepa V Viswanadham, and Amin Vahdat, *Google*

NetAssistant: Dialogue Based Network Diagnosis in Data Center Networks. 2011
Haopei Wang, Anubhavnidhi Abhashkumar, Changyu Lin, Tianrong Zhang, Xiaoming Gu, Ning Ma, Chang Wu, Songlin Liu, Wei Zhou, Yongbin Dong, Weirong Jiang, and Yi Wang, *ByteDance Inc*

Klonet: an Easy-to-Use and Scalable Platform for Computer Networks Education 2025
Tie Ma, Long Luo, and Hongfang Yu, *University of Electronic Science and Technology of China;* Xi Chen, *Southwest Minzu University;* Jingzhao Xie, Chongxi Ma, Yunhan Xie, Gang Sun, and Tianxi Wei, *University of Electronic Science and Technology of China;* Li Chen, *Zhongguancun Laboratory;* Yanwei Xu and Nicholas Zhang, *Theory Lab, Central Research Institute, 2012 Labs, Huawei Technologies Co., Ltd.*

ExChain: Exception Dependency Analysis for Root Cause Diagnosis. 2047
Ao Li, *Carnegie Mellon University;* Shan Lu, *Microsoft Research and University of Chicago;* Suman Nath, *Microsoft Research;* Rohan Padhye and Vyas Sekar, *Carnegie Mellon University*

Horus: Granular In-Network Task Scheduler for Cloud Datacenters

Parham Yassini*[1], Khaled Diab*[2], Saeed Zangeneh[1] and Mohamed Hefeeda[1]

[1]*School of Computing Science, Simon Fraser University, Burnaby, BC, Canada*
[2]*Hewlett Packard Labs, United States*

Abstract

Short-lived tasks are prevalent in modern interactive datacenter applications. However, designing schedulers to assign these tasks to workers distributed across the whole datacenter is challenging, because such schedulers need to make decisions at a microsecond scale, achieve high throughput, and minimize the tail response time. Current task schedulers in the literature are limited to individual racks. We present Horus, a new in-network task scheduler for short tasks that operates at the datacenter scale. Horus efficiently tracks and distributes the worker state among switches, which enables it to schedule tasks in parallel at line rate while optimizing the scheduling quality. We propose a new distributed task scheduling policy that minimizes the state and communication overheads, handles dynamic loads, and does not buffer tasks in switches. We compare Horus against the state-of-the-art in-network scheduler in a testbed with programmable switches as well as using simulations of datacenters with more than 27K hosts and thousands of switches handling diverse and dynamic workloads. Our results show that Horus efficiently scales to large datacenters, and it substantially outperforms the state-of-the-art across all performance metrics, including tail response time and throughput.

1 Introduction

The slowdown of Moore's law and the end of Dennard scaling have changed how cloud datacenters deploy and manage their hardware resources and software services. Instead of continually increasing the frequency of CPU cores, microprocessor vendors have been shipping more cores per processor with only slight frequency increases. As a result, the available number of cores in datacenters has been steadily and substantially increasing over the last several years [30]. From the software perspective, numerous interactive and user-facing datacenter applications have been deployed. Examples of such latency-sensitive applications include key-value stores [11–13], multimedia applications [16, 34], distributed interactive analytics [49, 56], network function virtualization [56], and web search [9, 21]. To take advantage of the availability of many cores and reduce deployment costs, designers of these large-scale applications have recently started to embrace various practices such as micro-services and function-as-a-service.

The emerging hardware trends and the requirements of recent large-scale applications have increased the demands for fine-grain management of the datacenter computing resources. This is sometimes referred to as the *granular computing* paradigm [52]. In this paradigm, many applications are decomposed into large numbers of *short-lived* tasks, which are executed in parallel on 100's–1000's of cores that potentially *span multiple server racks*. Each task typically has a tight *response time*, in the order of 10's–100's of microseconds [20, 52]. And since the performance of applications is affected by their slowest tasks, granular computing platforms strive to minimize the task *tail response time* [20]. Granular computing platforms are also expected to support *high scheduling throughput* as the number of concurrent tenants and their application demands are rapidly growing [69].

Granular computing can be viewed as the generalization of recent initiatives from academia and industry for offering more flexible, finer-grain, cost-effective, and shorter latency computing infrastructures. For example, Amazon Lambda [2] handles task execution times in the order of 100's of milliseconds using the Firecracker microVMs [14]. Apache OpenWhisk [1] offers a serverless framework to seamlessly execute functions while handling the provisioning of the underlying computing resources. Efforts from Microsoft [68] and others [45, 70] have introduced mechanisms to reduce the cost of cold-starts in serverless frameworks to support low-latency applications. Furthermore, recent operating system schedulers, e.g., [43, 64], offer support for microsecond-scale tasks within individual servers. Granular computing aims at pushing the boundaries even further, by efficiently supporting microsecond tasks at a *datacenter scale*. This paper contributes to the realization of granular computing.

As illustrated in Figure 1, datacenter operators deploy mul-

*Both authors contributed equally to this work.

tiple software components to manage applications and computing resources [1, 71, 73], including a resource manager, worker pools, and task schedulers. The resource manager allocates a worker pool for each application according to its required level of fault tolerance and performance using mechanisms such as [43, 68, 70]. Then, a task scheduler assigns each submitted task of an application to a worker from its worker pool. Resource managers, such as YARN [72] and Mesos [39], decouple resource allocation from task scheduling. This enables deploying multiple task schedulers targeting different application needs. This paper presents a granular task scheduler designed for latency-sensitive applications.

Designing a granular task scheduler is, however, challenging because such a scheduler needs to make decisions at a microsecond scale, achieve high throughput, and minimize the task tail response time. Minimizing the tail response time requires balancing the load across workers, which is difficult to achieve because of the substantial diversity in the task execution times and the scale of modern datacenter applications that could have thousands of workers distributed across many racks. Further, since tasks are short-lived, the load on workers is highly dynamic. Thus, naively tracking the load on workers could result in substantial communication, processing, and memory overheads on the scheduler.

Current *software* schedulers, e.g., Borg [73], Twine [71], and Atoll [70], introduce significant network and processing delays. These delays are sometimes larger than the task execution time itself, which makes software schedulers unsuitable for short-lived tasks. In addition, scheduling granular workloads requires a substantial amount of computing resources [66, 73], which is difficult to realize using traditional application-layer schedulers. For example, consider a system with 20K tasks and a task mean execution time of 100 μs. The system would need to make 200M scheduling decisions per second and handle around the same number of packets for processing the state update messages. This scale of throughput is not possible to achieve using software schedulers [66].

In-network schedulers, on the other hand, schedule tasks in the data plane as packets carrying these tasks pass through switches. Thus, they significantly reduce the scheduling latency and, in turn, the response time of tasks. However, recent in-network schedulers, e.g., [48, 50, 78], can only schedule tasks within individual racks. Therefore, they cannot meet the growing demand of large-scale applications that require executing thousands of tasks across multiple racks.

To support current latency-sensitive applications and future granular computing platforms, we propose **Horus**, the *first* datacenter-wide in-network scheduling system in the literature, to the best of our knowledge. One of our key insights in designing Horus is that scheduling operations should run at different time scales for efficiency and scalability. For example, assigning a task to a worker should be done at a microsecond scale, whereas tracking the load on workers can occur at a millisecond scale. Leveraging this insight, we divide the operations of Horus into two components. The first assigns tasks to workers, and the second tracks, aggregates, and maintains the load on workers. Horus *offloads* both components to switches in the network and optimizes them independently, which enables efficient scheduling of granular tasks.

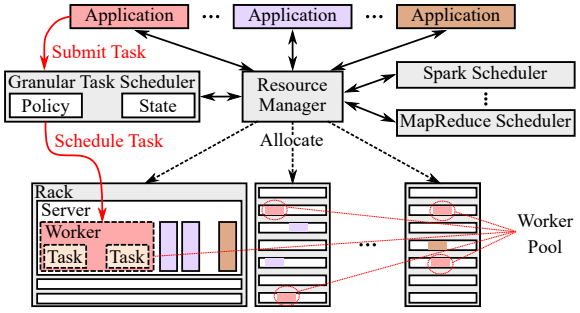


Fig. 1: Overview of resource management in datacenters.

The contributions of this paper are as follows.

- We propose an in-network task scheduling architecture for latency-sensitive datacenter applications, in §3.
- We present a new scheduling policy that does not queue tasks at switches and runs at line rate, in §3.3.
- We design multiple data structures to realize the proposed policy in modern programmable switches, which have a restricted programming model and limited memory resources.
- We propose efficient mechanisms to distribute the load information among schedulers, which maintain the freshness of load values and minimize overheads on switches, in §3.4.
- We implement Horus in a testbed using a Tofino switch and compare it against the state-of-art in-network scheduler, which is RackSched [78], in §4. In single-rack settings, our results show that Horus reduces the tail response time by up to 75% and increases the throughput by up to 1.9X compared to RackSched, for the considered realistic workloads. Since RackSched does not support multiple racks, we show that Horus substantially outperforms two natural extensions of RackSched in multi-rack settings.
- We also conduct large-scale simulations for datacenters with more than 27K hosts and thousands of switches handling diverse and dynamic workloads, in §5. Our results show that Horus outperforms RackSched and its extensions across all performance metrics. We also show the robustness of Horus against failures, packet losses, and link delays.

Due to space limitations, some details and evaluation results are presented in the **Appendix**.

2 Background and Related Work

2.1 Task Scheduling in Datacenters

Granular applications, e.g., key-value stores and microservices, create many short-lived tasks with *diverse* execution

times, ranging from tens of microseconds to hundreds of milliseconds or even longer. These tasks need to be assigned to workers for execution. Recent intra-server schedulers, e.g., Shinjuku [43] and ZygOS [64], support microsecond tasks. These task schedulers are, however, limited to single servers.

A natural question is then: can we use load balancers with server schedulers to scale beyond individual servers? Zhu et al. [78] showed that this policy is ineffective and may yield long tail response time. This is because most load balancers, e.g., [19, 33, 57], typically make their decisions based on hashing various fields in the packets. Zhu et al. [78] proposed RackSched to extend task scheduling to the rack level, where the top-of-rack (ToR) switch approximates the load on servers within its rack and assigns tasks to them accordingly.

Scaling task scheduling beyond single racks is an important and challenging research problem. There are many practical scenarios where applications require and/or benefit from execution on cores across racks. For example, running applications across racks in different fault domains improves their fault tolerance and availability [4,7,22]. This is especially critical for latency-sensitive applications since most of them are user-facing. A recent study from Facebook [6] indicates that the traffic of many latency-sensitive applications is mostly not rack-local. In addition, in public datacenters, it is not uncommon that tenants' VMs are placed on different racks due to unavailable resources at the time or for improved fault tolerance [40]. Therefore, there is a need to run tasks across racks in the datacenter. However, simple extensions of rack-level schedulers that use load balancers to distribute tasks to racks and then to servers may lead to long tail response time for the same reason mentioned above: load balancers are oblivious to the current queue lengths of workers, which could be impacted by the diversity in the task execution times.

To demonstrate the limitations of using the state-of-the-art approach, which is RackSched [78], for scheduling tasks across racks, we conduct simulations with representative workloads and datacenter configurations similar to prior works [43,54,67,78]; the details of our simulations are given in §5.1. Briefly, we simulate a tree-based datacenter with 27,648 servers, each having 32 cores, and we consider large workloads and diverse task distribution times and arrival patterns. We implemented the scheduling policy of RackSched, which we refer to as RS. We complemented RS with a datacenter load balancer that uniformly (at random) distributes tasks to racks. Tasks are then scheduled to servers within racks using RS. We refer to this scheduling system as RS-LB.

In addition, to show the potential performance gains, we simulate a *global* version of the Join Shortest Queue (JSQ) policy. JSQ tracks queue lengths at individual servers, and it schedules tasks to the server with the shortest queue. As analyzed in [79], JSQ produces optimal results across different performance metrics, e.g., waiting time and throughput, and for tasks with low- and high-dispersion execution times. We simulate an *ideal/theoretical* version of JSQ that immediately updates queue lengths to show the performance bounds. The results of our simulation are presented in Figure 2 for two representative task execution time distributions: Exponential and Bimodal. The figure plots the tail (99%) response time as the normalized workload increases for different task scheduling systems. The results reveal the substantial performance gap between JSQ and RS-LB. For example, in Figure 2a, the tail response time increases rapidly as the normalized load exceeds 50% when RS-LB is used, whereas it stays low for JSQ even for a load around 90%.

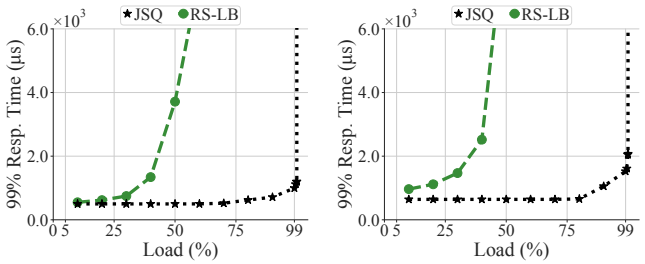


Fig. 2: Limitations of current rack-level task scheduling systems when scaled to the whole datacenter.

Although JSQ theoretically provides optimal results, it is not possible to implement in practice for large-scale datacenters, especially for microsecond tasks. This is because JSQ requires knowing the queue lengths at all servers, which takes time to either probe servers (one RTT) or wait for servers to send their updates (which may arrive asynchronously and/or delayed/aggregated). Thus, by the time the scheduler determines the server with the least load, the situation might have already changed. In addition, JSQ and similar global policies may cause task *herding*, which occurs because once a server is reported to have the least load, the scheduler may keep sending tasks to it until the server sends an update. Since updates may take a relatively long time compared to task execution times, a burst of too many tasks could have already been sent to the server, which leads to severe load imbalance and long response time. Finally, implementing JSQ would require extensive computation resources at the scale of a datacenter.

2.2 Related Work

Software Schedulers. Traditional schedulers are designed as software processes that run on one or multiple servers [37, 62, 73]. These schedulers focus on service times in the range of seconds to hours, and they can make complex decisions and support a wide range of resource allocation policies. To scale up and support higher throughput, distributed schedulers have been proposed [28, 29, 31, 46, 60]. The shortest task service time supported by distributed schedulers is still in the order of hundreds of milliseconds [31, 52, 60]. This is because of the relatively long time such schedulers take to

allocate tasks to servers. Sparrow [60] and Eagle [28], for example, maintain queues of submitted tasks at schedulers. Each scheduler then pushes reservation probes to randomly-selected workers. When a worker becomes idle, it pulls the next reserved task from that scheduler's queue, introducing a scheduling delay of at least two RTTs.

In contrast, Horus is designed for granular tasks with execution times in the order of tens of μs, which can be smaller than a single RTT [20]. Horus is also designed to support millions of scheduling decisions per second. In addition, multiple prior works, e.g., YARN [72] and Mesos [39], decouple resource allocation from task scheduling. Horus can complement such works by offering fine-grain task scheduling on the allocated resources for large-scale datacenter applications.

In-network Computing. Emerging programmable switches enable offloading of various operations and functions to the network to achieve high throughput and low latency for datacenter applications, such as caching [41], data aggregation [65], concurrency control [53], and lock management [77].

Similarly, in-network task scheduling has been considered before [48, 50, 78]. RackSched and prior in-network schedulers, however, support only single racks. In contrast, Horus scales to multiple racks across the whole datacenter network. RackSched [78] was shown to outperform prior works, and it is considered state-of-the-art. Thus, we compare a simple version of Horus against RackSched in single racks. We also extend RackSched to support scheduling across multiple racks and compare Horus against these extensions.

3 Proposed In-Network Scheduling

In this section, we first summarize the principles that guided the design of Horus. Then, we describe the proposed in-network task scheduling approach. This is followed by describing our efficient methods for distributing state among various components of Horus. Finally, we describe various deployment options for Horus. Due to space limitations, we present some details in the **Appendix**, including handling failures and packet losses (§A.3), supporting multi-packet tasks (§A.4), overhead analysis (§A.5), and pseudo code (§A.6).

3.1 Design Principles

The design of Horus is based on the following principles:

- ***P1: Load-aware Scheduling***. The load on workers in datacenters is subject to spatial and temporal variations due to resource allocation policies, application requirements, and the seasonality of workloads [71, 73]. And as shown in prior works, e.g., [78], and by our simulations in §2.1, not considering the actual worker load in the scheduling decisions may lead to long tail response times. We propose a *zero-queue* scheduling system that efficiently tracks the load of workers, minimizes the task tail response time, and avoids task herding. By not buffering tasks in switches, the memory requirements become *independent* of the task rate, which helps Horus to scale.
- ***P2: Lazy State Update***. Horus makes scheduling decisions based on the maintained state without queuing tasks. Thus, updating this state is important to reflect the latest changes. This, however, may increase the communication overhead and limit scalability. Our idea is that a switch may not need to immediately update its state *if* it can make accurate decisions using its current state. Our approach identifies when an update is needed by calculating a *drift* between actual load values and the load information available at schedulers, and it only updates the state of a scheduler when the drift may negatively impact the scheduling quality.
- ***P3: Localized State***. Horus avoids the complexity of replicating state across all switches in the datacenter by logically grouping the distributed schedulers and maintaining the state within each group. This allows each scheduler to update its view of a subset of workers without querying other schedulers. Localization of state enables Horus to further reduce the overheads on switches, achieve high throughput, and handle failures efficiently.

3.2 Overview and Workflow

Overview. Horus is a *distributed*, in-network, granular task scheduling architecture designed for datacenters. It can be viewed as one of the components in the software suite managing computing resources in datacenters, as illustrated in Figure 1. For example, Horus can be integrated with existing platforms such as OpenWhisk [1], and it can coexist with schedulers of long-lived tasks such as Borg [73].

As shown in Figure 3, Horus consists of a set of schedulers, a centralized controller, an agent per server, and APIs.

Schedulers in Horus are distributed to handle high task rates and tolerate failures. Horus decomposes task scheduling into two components. The first maintains and aggregates the load information of workers, whereas the second executes the scheduling policy to assign tasks to workers using this maintained information. Both components run in the data plane of the switches. Schedulers do not require specific network topology, and thus, Horus can easily be deployed on different datacenter networks. For clarity of the presentation, however, we focus on the widely-deployed leaf-spine topology [15, 42], which is shown in Figure 3. In this case, Horus schedulers run as data plane programs on leaf and spine switches; no schedulers run on core switches. Leaf schedulers track and use the load information about workers in their racks (**P1**), and they aggregate and efficiently distribute this information to spine schedulers (**P2**).

The ***centralized controller*** realizes various functions, such as addressing and handling failures. It assigns a fixed ID to each scheduler, and it interacts with the resource manager to

retrieve the placement information for the workers of each application. Using this information, the controller assigns a leaf scheduler to each rack that has workers. It then divides leaf schedulers into disjoint groups, where each group is assigned a spine scheduler and forms a logical tree. Horus uses a simple approach that aligns groups with datacenter pods, where a pod usually has 32-64 racks. In this case, the worker state is localized and maintained by schedulers within each pod (**P3**). This approach is efficient because packets exchanged between the spine and leaf schedulers within a pod traverse only one hop, which reduces packet latency and load on links, compared to the case where the spine is in a different pod.

Horus ***agents*** are lightweight processes that run on servers to track the load of workers. They also run a health check mechanism with the control plane of the leaf schedulers, which enables Horus to detect and react to worker failures. Agents are not involved in the scheduling decisions.

Horus offers ***APIs*** to datacenter applications to seamlessly submit tasks for execution and receive their results. To submit a task, Horus attaches a layer-4 header to packets, which makes Horus compatible with various routing protocols. The header includes a unique ID for each task, `taskID`. The uniqueness of task IDs is ensured by concatenating the application ID and a monotonically increasing sequence number. The application ID is computed by hashing the application pathname and adding a random number to ensure uniqueness. A sequence number of 32 bits is sufficient for all practical scenarios. Since tasks are expected to finish within micro or milliseconds, by the time the sequence number wraps around, if it ever does, earlier tasks would have been long completed.

Horus is designed for granular tasks, which are mostly contained within single packets as they typically carry parameters and paths to data. For example, in key-value stores [11–13], segments of the data are typically pre-distributed to workers, and tasks carry the queries to be executed on the data. For the common case of single-packet tasks, Horus does not maintain any per-task state at schedulers. Horus does support multi-packet tasks and maintains task-worker affinity using ideas similar to prior works, e.g., [57], as discussed in §A.4.

Horus Workflow. Workers of latency-sensitive applications are pre-deployed on CPU cores and initialized to be ready to execute tasks. A worker runs in a virtualized environment such as a container or microVM [14], and is allocated one or more cores. Horus assigns tasks to worker queues and does not dictate any intra-worker scheduling policy for distributing the tasks across the worker's cores. All workers of an application are assigned an anycast IP address by the controller. Tasks use the anycast address as the destination address in their packets. Tasks are scheduled in a recursive manner. When a task is submitted for execution, its packets are randomly forwarded to one of the spine schedulers assigned to this application (Step ❶ in Figure 3). Since the controller knows the distribution of workers across racks, the random selection of spine schedulers is weighted in proportion to the number of workers per pod.

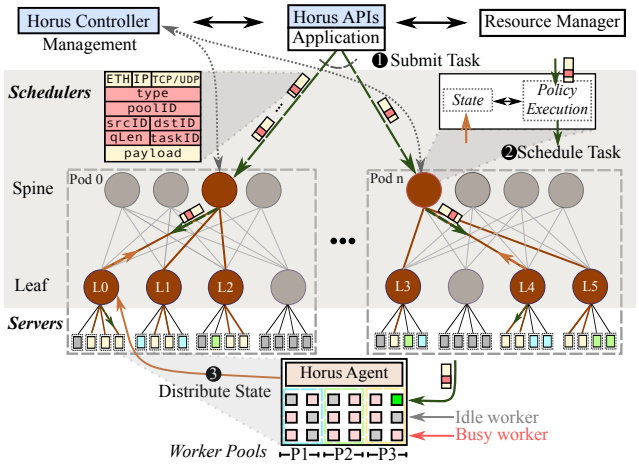


Fig. 3: The proposed Horus scheduling architecture.

This ensures load balancing across pods, which is important in case workers are non-uniformly distributed. The spine scheduler runs its policy to select a downstream leaf scheduler to handle the task, which in turn assigns the task to one of the workers in its rack (Step ❷).

After a worker completes a task, the agent running on that worker includes the latest load information into the header of the reply packet and sends it to the leaf scheduler in the rack (Step ❸). Upon receiving a reply packet, the leaf scheduler updates the load information in its memory and sends an update message to upstream spine schedulers, *if needed*, using our aggregated state update method in §3.4.

3.3 Scheduling Tasks in the Network

Switch Model. We consider the common switch model used by several switch vendors [10, 24, 61], in which a packet goes through a pipeline of multiple stages, where each stage has processing and memory components. This switch model supports line-rate packet processing by realizing two main design choices: (i) atomic memory updates and (ii) bounded packet latency. The first allows a packet to access *up to one* memory location at each stage, and the second limits the number of processing stages. As a result, the total available processing and memory resources for the stages are limited. To mitigate the impact of these design choices, a packet may have to be *recirculated* from the egress to the ingress for additional processing or to access the same memory block again at the cost of increased delay [74]. While this restrictive model enables line-rate packet processing, it makes it difficult to implement in-network task scheduling.

Proposed Scheduling Policy. In Horus, all spine and leaf schedulers employ the same scheduling policy and maintain the same data structures. Thus, we abstractly present the scheduling policy as follows. A scheduler assigns an arriving task to a lower-layer *node*. A node for a spine scheduler

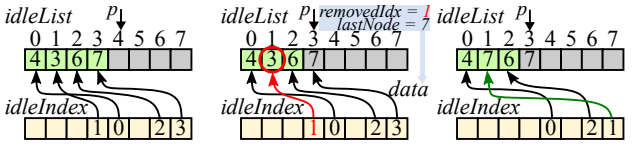
is a rack of servers, whereas it is a worker for a leaf scheduler. Optimally scheduling a task requires knowing the load of all downstream nodes and assigning the task to the least loaded one, i.e., implementing a JSQ-like policy. As we discussed in §2, JSQ is difficult to realize at the datacenter scale because it imposes high communication and processing overheads, and it may introduce task herding, where a burst of many tasks is sent to a node leading to periods of significant load imbalance.

Horus strives to *approximate* the JSQ scheduling policy at the datacenter scale while considering the dynamic nature of workloads and the restrictions of programmable switches. Specifically, Horus divides the scheduling decisions into two cases: (i) when some idle nodes are available and (ii) when all nodes are busy. In the first case, when a task arrives at a scheduler and the scheduler is aware of some idle nodes, it will send the task to one of them. For spine schedulers, an idle node is a rack that has at least one idle worker. In this case, the spine scheduler will send the task to the leaf scheduler of that rack, which in turn will select an idle worker within the rack, resulting in zero queuing time and minimizing the response time under light load. The challenge here is to track idle nodes at a large scale and in a way that can be implemented in programmable switches. We present the details of our solution later in this section.

In the second case, when a task arrives at a scheduler and all nodes are busy, the scheduler takes 2 *random* samples from the queue lengths of nodes and selects the least loaded node among the sampled values. This is referred to as the power-of-2 policy and is known to reduce the response time [58]. In addition, the randomization in taking samples prevents task herding, since it is unlikely that the same node will be repeatedly chosen for several consecutive tasks. Randomization is critically important for scheduling *granular* tasks as they are more susceptible to task herding. This is because granular tasks have short execution times, and load updates from workers may take relatively long times to reach various schedulers distributed across the datacenter. Implementing the power-of-2 policy at scale and in programmable switches with limited resources and strict constraints is challenging. We present new data structures to realize this policy later in this section and efficient methods to distribute load information among schedulers in §3.4.

Scheduling Tasks to Idle Nodes. We design a data structure, called *idleNodes*, for schedulers to track the IDs of idle nodes. Our data structure supports fast addition, removal, and retrieval of nodes with constant time in the switch data plane. It also requires a small number of memory accesses and minimal dependencies among memory blocks, which reduces the number of allocated processing stages in switches.

An insight that we used is that a scheduler needs *only* to know whether there exists an idle node in the list. It does not need to identify the temporal order of when nodes became idle. Building on this insight, we design a data structure that guarantees the following ***invariant***:



(a) Main structure (b) Remove: pass #1 (c) Remove: pass #2

Fig. 4: The proposed *idleNodes* data structure, which is designed to support fast operations in programmable switches.

If there are idle nodes, they will be contiguously stored at the top of the list.

Maintaining this invariant allows the scheduler to quickly find idle nodes. Notice that the data structure has to satisfy the invariant even when any arbitrary node is removed, i.e., becomes busy.

Figure 4a illustrates the main components of the proposed data structure. It has an *idleList* array of N items of node IDs, where N is the number of downstream nodes. It also has a pointer p that points to the first non-idle item in the list. p is initialized to zero. In addition, the data structure has an *idleIndex* array, which stores the indices of idle nodes. In Figure 4a, nodes 4, 3, 6, and 7 are idle, and *idleIndex* contains their locations in *idleList*.

Adding a node to *idleNodes* occurs when a scheduler receives an *idleAdd* message about a node n becoming idle. The scheduler writes the ID of n in *idleList[p]* and p in *idleIndex[n]*. It then increments p. Current programmable switches support an atomic *read-modify-write* operation in a single stage. This means that a scheduler can read the current p, increment it, and write it back in one stage. Thus, adding an idle node requires three processing stages only, one to update each of p, *idleList*, and *idleIndex*.

To schedule an incoming task to an idle node, a scheduler retrieves the node at *idleList[p-1]*. If this is a leaf scheduler, the retrieved node is removed from *idleList* by decrementing p and clearing the corresponding location of the retrieved node in *idleIndex*. This is because a node in this case represents a single worker, which will no longer be idle after sending the task to it. On the other hand, a node in a spine scheduler represents a whole rack, and sending a task to an idle leaf does not necessarily mean that the rack no longer has idle workers. Therefore, a spine scheduler does not remove the retrieved node *idleList[p-1]*. An idle node is removed from a spine scheduler only when it receives an explicit *idleRemove* message from the leaf scheduler represented by this node.

For both the spine and leaf schedulers, an incoming task is scheduled to an idle node *without* any delay or buffering and at line rate, since all operations on *idleNodes* are performed in a few consecutive processing stages in the switch. In addition, it is straightforward to show that the operations of adding a new idle node at *idleList[p]* and removing the idle node at *idleList[p-1]* maintain the invariant mentioned above.

Finally, the *idleNodes* data structure should support remov-

ing any arbitrary node n, while still maintaining the invariant. This is needed when a scheduler receives an *idleRemove* message for n. A scheduler removes node n by replacing it with the last idle node in the list. This ensures maintaining the invariant that all idle nodes are contiguously placed at the top of the list. It also allows the scheduler to conserve memory and use a constant number of processing stages. This operation needs to be performed in two passes because current programmable switches do not allow a packet to access the same memory location more than once in the same pass. In the first pass, the scheduler examines *idleIndex* to get the index of n in *idleList*, which is referred to as *removedIdx*. The scheduler also decrements p to retrieve the ID of the last idle node, which is called *lastNode*. In the second pass, the scheduler resubmits the packet with the additional data *lastNode* and *removedIdx*, where *lastNode* is then written into *idleIndex[removedIdx]*. An example is shown in Figures 4b and 4c.

We note that current programmable switches do not preserve the order of processing of resubmitted packets. For example, before processing the second pass of an *idleRemove* packet A, the first pass of another packet B could be processed by the switch, which may result in incorrect values of the indices. To prevent this potential race condition, we use a single memory location as a logical lock. The lock is acquired in the first pass and released in the second one. An *idleRemove* packet is dropped if it fails to acquire the lock. The sending node will resend a new *idleRemove* packet after receiving another task from the scheduler.

Scheduling Tasks to Busy Nodes. When there are no idle nodes ($p = 0$), a scheduler needs to realize the power-of-2 policy, which is more challenging than the case of idle nodes. This is because a scheduler needs to read two randomly selected indices from the *loadList*, while the switch model does not allow reading more than one item per packet from the same memory block. To address this restriction, Horus maintains two identical copies of the *loadList* in two different stages, where each array stores the load of all downstream nodes (one node per slot). Storing two copies allows the scheduler to read one random index from each copy and then compare them. When a new state update for a node arrives at a scheduler, it writes the updated load value to both copies.

After making a decision, a scheduler should update its view on the load information, which is stored in the *loadList*. This requires the scheduler to write back the updated load value to the corresponding *loadList* slot. As described earlier, the same memory block cannot be accessed twice per packet. A straightforward solution is to resubmit each packet to the pipeline and update the load state on the second pass. This, however, results in additional processing overhead and increases the scheduling latency. To address this issue, we propose a ***lazy state update*** algorithm, which resubmits a packet *only* if it will impact future scheduling decisions. That is, a scheduler keeps processing tasks using the potentially stale view of the *loadList* until it detects an update is needed.

Specifically, for a node m, we decompose its actual queue length q_m into a load value l_m and a drift value d_m, where $q_m = l_m + d_m$. We define the drift value as the number of tasks scheduled to a node that has not been reflected in its load value, and we store the drift values of all nodes in a data structure called *driftList*. Each scheduler maintains two copies of the *driftList* placed in two stages. Next, we find the *necessary condition* to resubmit a packet to update the load values. Upon receiving the first packet of a new task, a scheduler picks two random nodes m and n, and it reads their load values l_m and l_n from *loadList*. Without loss of generality, assume that $l_m < l_n$. Then, the scheduler should resubmit the packet *iff* $q_m > q_n$. That is, $l_m + d_m > l_n + d_n$. We rearrange the inequality to find the ***necessary condition*** to resubmit a packet as:

$$d_m > (l_n - l_m) + d_n. \quad (1)$$

The above condition means that as far as the drift in the load of node m is less than or equal to the difference between the load of n and the load of m plus the drift in the load of n, the scheduler will make the correct decision by choosing the node with the smaller load, which is m, by comparing their l_m and l_n stored in the *loadList*.

The proposed lazy state update algorithm works as follows. It first reads the load values of nodes m and n. It then identifies the node with the smaller load, say m, and it computes the difference $(l_n - l_m)$. Then, it reads d_m from the first copy of *driftList* to check how many more tasks are actually queued at m. If the drift value is lower than the difference between load values, i.e., $d_m < (l_n - l_m)$, the algorithm increments the corresponding drift value in each copy of the *driftList* for node m. Otherwise, when $d_m > (l_n - l_m)$, the algorithm resubmits the packet to update l_n and l_m. The algorithm does not include d_n in its calculations because it would violate the atomicity requirement in current programmable switches. The algorithm, however, does guarantee selecting the least loaded node among m and n. This is because $d_m > (l_n - l_m)$ still satisfies the necessary condition $d_m > (l_n - l_m) + d_n$. The less restrictive condition used by our algorithm allows implementing the power-of-2 policy in programmable switches at the cost of possibly resubmitting some extra packets than absolutely needed by the necessary condition.

An **example** illustrating this algorithm is given in §A.1.

3.4 Distributing State Among Schedulers

We design efficient mechanisms to distribute the necessary information among leaf and spine schedulers to update their states. This enables the execution of the proposed scheduling policy using fresh information with minimal overheads.

Distributing Worker State to the Leaf Layer. Since scheduling tasks to workers is only done by leaf schedulers, each leaf scheduler updates its state when selecting a worker for a task (§3.3). When a task is done executing on a worker, the agent modifies the `qLen` field in the header (Figure 3) and uses the

reply packet to report the updated load to the leaf scheduler. If the reply packet indicates the worker is idle, the leaf scheduler adds the `srcId` to the *idleList* and updates the *loadList* for the corresponding index.

Distributing Rack State to the Spine Layer. We consider two types of information to be distributed to the spine layer: (i) idleness of the rack and (ii) average load of the rack.

Idle State Update. When a leaf scheduler becomes aware of an idle worker, it sends an *idleAdd* packet to the spine scheduler it is linked with. This simple strategy balances and localizes the information about idle racks among the upper-layer spine schedulers. Once there are no more idle workers in a rack, the leaf scheduler in that rack sends an *idleRemove* packet to the linked spine to remove the leaf from its *idleList*.

Load State Update. Each spine scheduler tracks the load of a subset of racks that contain workers. Each leaf scheduler calculates the *average* load across workers in its racks and sends it to the linked spine. Directly calculating averages in programmable switches is, however, infeasible due to their restrictive programming model. We describe how we approximate averages in §A.2.

Since Horus is designed for short-lived tasks, continually sending every updated average load value to the spine would result in large communication and processing overheads on switches, without significantly modifying the average at the spine. Instead, we make each leaf scheduler locally compute the current average and maintain the previously sent average to the spine. Then, a leaf scheduler sends the update message only if the difference between the current and previous average load values is greater or equal to one, because this is the smallest integer value of load changes that could impact the scheduling decision. This ***aggregated update mechanism*** substantially reduces the number of update packets sent to spine schedulers without sacrificing the scheduling performance. In addition, we *piggyback* the state update information with the response packets sent by workers after completing tasks to minimize the communications overhead of Horus.

3.5 Horus Deployment Options

Horus does not rely on the structure of the datacenter network in its operation, and thus, it can be deployed in various networks. In addition, it does not dictate a specific routing protocol since it utilizes layer-4 headers. Furthermore, Horus can be incrementally deployed in datacenters. Suppose a fraction of the spine and leaf switches are upgraded to be programmable to support latency-sensitive applications. In this case, workers of these applications can be allocated in the racks with programmable switches, and the centralized controller in Horus can be configured to only use the programmable spine switches.

A more restrictive deployment scenario occurs when only a fraction of the spine switches are upgraded and all leaf switches are legacy. In this case, we can implement the leaf scheduler on a server in each rack using efficient techniques such as kernel bypassing, as in prior works, e.g., Maglev [33]. These techniques were shown to process packets in microseconds. Workers and Horus agents in each rack would be configured to first direct their messages through the software leaf scheduler in the rack. This adds an additional delay, but it is deterministic and limited by the small RTT within racks.

Finally, the ideas of Horus can be used without any programmable switches. For example, leaf schedulers can be implemented in software as above, and the functionality of the spine schedulers can be integrated with the datacenter load balancers handling the submission of tasks.

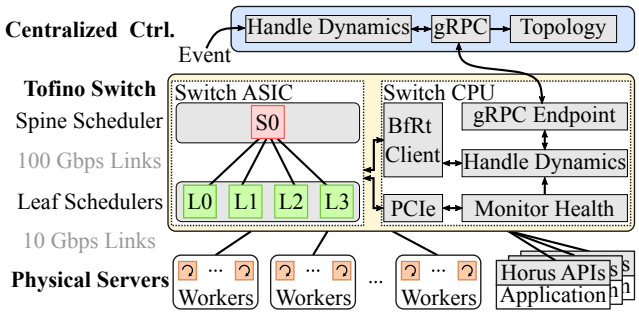


Fig. 5: Testbed setup.

4 Evaluation in a Testbed

4.1 Experimental Setup

Testbed. Our testbed, illustrated in Figure 5, has one 3.2 Tbps Intel Tofino switch, which has two hardware pipelines. We configure one of the hardware pipelines as a spine switch and run the spine scheduler of Horus on it. We emulate four leaf switches on the other hardware pipeline, where each switch represents a rack of servers. Leaf switches run the leaf schedulers of Horus. We connect the leaf switches to the spine switch using logical 100 Gbps links. In addition, the testbed has seven servers connected to the leaf switches through 10 Gbps links, where each server is equipped with an Intel 82599ES 10 GbE NIC. These servers are used as clients to generate tasks and as workers to execute tasks.

Horus Implementation. We have implemented a proof-of-concept of Horus consisting of leaf and spine schedulers, switch and centralized controllers, agents, and client APIs. All source code, testing scripts, and datasets are open source [3].

The leaf and spine schedulers are implemented in P4 [23] and deployed to the switch ASIC; a brief description of our P4 implementation can be found in §A.6. We implemented the switch and centralized controllers using Golang in about 6K lines of code. The controllers handle failures and dynamics, and they update the switch data structures accordingly. We implemented a set of APIs in C using DPDK to submit tasks and receive their results. We implemented the Horus agent in

about 100 lines of C code. The agent adds the worker load information to the reply packet after task execution is done.

For workers, we need to dispatch and run microsecond tasks. Recent intra-server OSes, e.g., ZygOs [64] and Shinjuku [43], support microsecond tasks and offer various scheduling policies. We modified the more recent Shinjuku [43] to dispatch tasks to worker queues based on Horus headers. In our experiments, we use one core per worker, bypassing Shinjuku's scheduling policy. We note that Horus does not dictate the use of any specific intra-server scheduling policy or OS.

Systems Compared Against. To the best of our knowledge, Horus is the first task scheduler that scales to the whole datacenter. RackSched [78] is the state-of-the-art in-network task scheduler, but it only scales to a single rack. We compare a simple version of Horus versus RackSched (referred to as RS) for single rack settings. We use the open-source P4 implementation of RackSched [8].

In addition, we consider two natural extensions of RackSched to support multiple racks. The first one integrates RackSched with a load balancer that uniformly distributes tasks to top-of-rack switches, which in turn run RackSched to assign tasks to workers within their racks. We refer to this system as RS-LB. In the second extension, we use RackSched *hierarchically*, meaning that we deploy it on both spine and leaf switches, where spine switches assign tasks to leaf switches using RackSched, which in turn assign tasks to workers also using RackSched. We refer to this system as RS-H. To ensure fair comparisons, we make each leaf scheduler send the load state update from workers to the spine scheduler immediately after each task is done.

Worker Placement. We consider two setups for worker distribution across racks: (1) **Uniform:** a total of 32 workers are uniformly distributed across all racks, where each rack has eight workers running on a physical server attached to the leaf switch (1 worker/core). The 32 workers run on four identical servers, each has Intel Xeon E-2186G CPU, 3.80 GHz, 12 cores, and 32 GB memory. (2) **Skewed:** a total of 48 workers are distributed as follows: two racks have four workers each, one rack has eight workers, and one rack has 32 workers (running on three physical servers, where one of the servers has Intel Xeon E5-2650 CPU, 2.3 GHz, 40 cores, 128 GB memory and runs 16 workers and the other two servers have the same specifications as the ones used in the Uniform setup and run 8 workers each).

Workloads. We evaluate Horus using two practical workload scenarios. The first scenario runs real tasks on the **RocksDB** engine [13], which is a high-performance key-value store developed by Facebook and is widely deployed in production [25]. The second scenario is synthetic and uses the **TPC-C** benchmark [5], which is an online transaction processing benchmark emulating e-commerce systems.

We create multiple RocksDB workloads with parameters similar to prior works [32, 43, 44, 78]. Specifically, our RocksDB workloads contain SCAN and GET tasks, where the first scans a range of objects (i.e., a relatively long task), and the second retrieves a specified number of objects (i.e., a short task). We construct the SCAN and GET tasks such that their dispersion is one order of magnitude: a SCAN request scans 5K objects with a median service time of 650 μs, whereas a GET request retrieves 60 objects where the median time of each request is 40 μs. We then employ two realistic distributions to generate concurrent long and short tasks. The first distribution is similar to workload A in the YCSB benchmark [26], and it consists of 50% GET and 50% SCAN tasks. The second one consists of 90% GET and 10% SCAN tasks, which is similar to Facebook's USR workload [17].

The TPC-C benchmark [5] consists of five tasks (or transactions) with different service times. We employ the profiled model in [32] of the benchmark to build a synthetic workload following the same task distribution and dispersion ratios. The five tasks have service times of 21.6, 22.68, 71.28, 332.64, and 378 μs, distributions of 44%, 4%, 44%, 4%, and 4%, and dispersion ratios of 1X, 1.05X, 3.3X, 15.4X, and 17.5X, respectively. We scaled the service times, compared to the profiled model in [32], so that clients in our testbed can smoothly generate tasks at high rates.

Task Arrival Model. We stress the system by generating tasks following a **Poisson** arrival process [78]. The Poisson process results in non-uniform inter-arrival delays and generates **bursts** that can cause temporary queue imbalance and impact the tail latency [32, 64].

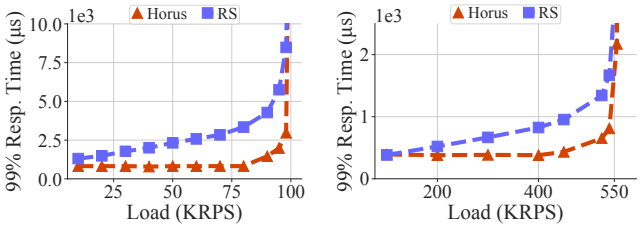
Horus schedulers have *no prior knowledge* about the service times, workloads, or arrival distributions.

Methodology and Performance Metrics. We vary the *system load* by incrementally increasing the number of tasks submitted for scheduling. The system load is measured in kilo requests per second (KRPS). We keep increasing the system load until we reach the capacity of the system, where the response time becomes unacceptably high for latency-sensitive applications (e.g., seconds or even minutes for tasks that should complete in micro or milliseconds). The *response time* is the period between submitting a task to a spine scheduler until it finishes execution on a worker.

An important metric for scheduling systems is the *achievable throughput*, which we define as the maximum system load that can be processed while meeting a given bound on a target performance metric, e.g., the tail (99th percentile) response time should not exceed 3ms.

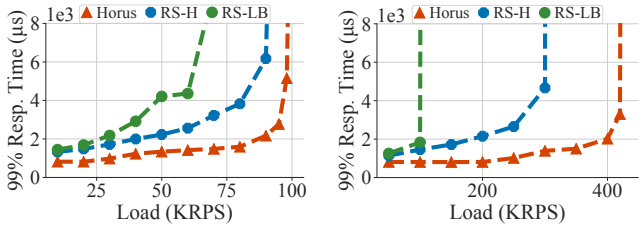
4.2 Comparison against State-of-the-Art

Horus vs. RackSched: Single Rack. We report the tail response time achieved by Horus and RackSched for the RocksDB and TPC-C workloads in Figure 6. There is no worker placement method used in this case, as all workers are located within the same rack. The results show that Horus consistently achieves much lower tail response times than RackSched, especially at high system loads. For example,



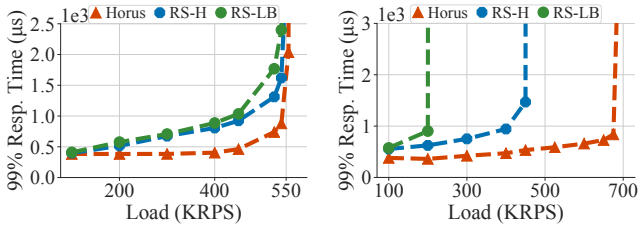
(a) RocksDB (b) TPC-C

Fig. 6: Comparing Horus vs. RackSched in single-rack setups.



(a) Uniform (b) Skewed

Fig. 7: Comparing Horus vs. RackSched extensions in multi-rack settings: Sample results from the RocksDB workloads.



(a) Uniform (b) Skewed

Fig. 8: Comparing Horus vs. RackSched extensions in multi-rack settings: Sample results from the TPC-C workloads.

for the RocksDB workload with 50% GET and 50% SCAN (Figure 6a) and at 80 KRPS system load, Horus reduces the tail response time by up to 75% compared to RackSched.

Horus achieves these significant gains because it tracks the loads on workers in a more accurate and efficient way than RackSched. Specifically, Horus identifies and uses idle workers within the rack, which results in low and constant response times at moderate loads. In contrast, RackSched does not explicitly track idle workers and relies only on the power of 2 policy of its scheduler, which may not always assign tasks to idle workers. In addition, RackSched relies on response messages from workers to update the load values. Since response messages are generated only after the completion of tasks, they may not capture the current state of workers by the time they arrive at the scheduler. In contrast, Horus updates the load values once it assigns a task to a worker, which enables it to have a real-time view of the workers' loads.

Horus vs. RackSched's Extensions: Datacenter. We compare Horus against RS-H and RS-LB for the different workloads and worker placements mentioned in §4.1. Representative samples of our results are shown in Figure 7 and Figure 8; the plots for all other scenarios are similar and given in §B. The results show that Horus consistently and substantially outperforms RS-H and RS-LB across all workloads and worker setups. For example, in the Uniform worker setup with 50% GET and 50% SCAN RocksDB requests (Figure 7a), Horus reduces the tail response time by up to 50% compared to RS-H when the system load is 80 KRPS; RS-LB could not support this load. This also means that Horus can achieve much higher throughputs than RS-H and RS-LB. For the same example in Figure 7a, if the target tail response time is 2 ms, Horus can achieve a throughput of up to 80 KRPS, whereas RS-LB and RS-H can only achieve up to 25 and 40 KRPS, respectively. That is, Horus can improve the throughput by up to 3.2X and 2X compared to RS-LB and RS-H, respectively, in this case. The results for the Skewed worker placement for the RocksDB workloads exhibit even higher gains, as shown in Figure 7b for the 90% GET and 10% SCAN workload. Similar gains are observed for TPC-C workloads with Unifrom and Skewed placements as shown in Figure 8.

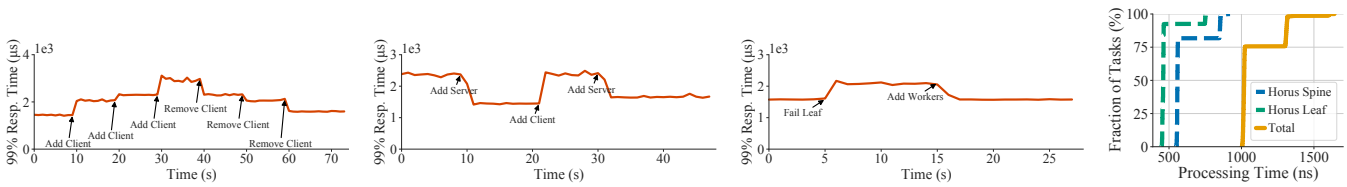
Horus achieves these gains across various workloads because its scheduling policy uses idle information to schedule tasks to any available idle workers, and it implements the efficient power-of-two method whenever there are no idle workers. In addition, Horus schedulers utilize the lazy update method to maintain up-to-date information about the worker loads, which significantly reduces the load imbalance across workers, as we show using large-scale simulations in §5.

4.3 Responsiveness and Overheads of Horus

We assess the performance of Horus in response to various dynamic events and analyze its overheads.

Dynamic Task Rate. We start with one client generating tasks at a rate of 30 KRPS. Every 10 seconds, a new client joins the system and sends additional tasks at a rate of 20 KRPS till the total task rate reaches 90 KRPS after 30s. Starting at 40s, we remove one client at a time at the same 10-second intervals. Figure 9a depicts the tail response time per second for the considered scenario. The results show that Horus can quickly react to the workload dynamics as the tail response time quickly drops after the task rate is decreased.

Dynamic Resource Scaling. In this scenario, one client starts sending tasks at a rate of 45 KRPS, where Horus schedules them to two racks of 16 workers. After 10s, we add a new server with 8 workers from another rack to the available worker pools. As Figure 9b shows, the tail response time drops to 1,416 μs after adding the server. We increase the task rate after 20s to 65 KRPS, which increases the response time. After 30s, we add another server with 8 workers, which reduces the response times to 1,644 μs. It takes between 1–2



(a) Dynamic task rate (b) Dynamic resource scaling (c) Handling failures (d) Latency

Fig. 9: Responsiveness and robustness of Horus to various dynamic events in (a)–(c), and its scheduling latency in (d).

seconds from when a resource allocation request is sent to the Horus controller till the response time is reduced.

Handling Failures. Initially, we use four leaf switches, and the client sends tasks at a rate of 65 KRPS. After 5s, we inject one leaf switch failure while the other three leaf schedulers remain active. We fail a switch by disabling its ports. Since the testbed has only one spine switch, we could not fail it. However, we analyze spine failures using simulations in §C.

As shown in Figure 9c, Horus can effectively use the remaining resources across other available racks to schedule tasks without interruption. While the spine scheduler has not been updated yet, a small fraction of the tasks sent to the failed leaf switch will be lost. At the rate of 65 KRPS, a leaf switch failure results in 1,444 failed tasks which are 2% of the total submitted tasks per second. At time 15s, we add another leaf scheduler with a rack of 8 workers. The response time is reduced within two seconds of adding the leaf scheduler.

Latency Overheads. We measure the total scheduling latency for a task at a switch by collecting the hardware timestamps at the beginning and end of the switch pipeline. Figure 9d shows the CDF of the measured scheduling latency at spine and leaf switches. As shown in the figure, the total latency is less than 1.6 μs for all tasks. We note that the small step increase observed in Figure 9d is due to the resubmitted tasks; recall that Horus resubmits a small fraction of tasks through the switch to update its state.

Other Results. We present more results in §B, including analyzing the fraction of resubmitted tasks.

5 Evaluation using Simulation

5.1 Simulation Setup

Datacenter Topology. We simulate a large datacenter with characteristics similar to the ones used in prior works, e.g., [54, 67]. Specifically, we simulate a network with a multi-rooted Clos topology composed of common 48-port switches with fully connected pods. The network has 1,152 leaf switches interconnected with the same number of spine switches. Each leaf switch manages a rack of 24 servers, leading to a network with a total of 27,648 servers. Each server has 32 cores and accommodates a maximum of 32 workers. The average per-hop delay between switches is set to 5 μs [35, 55], and the average packet loss rate is set to 1e$-$3% [38, 80].

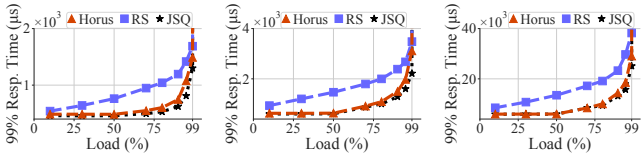
Worker Placement. We simulate 1K concurrent worker pools, where each worker pool processes tasks of a large-scale datacenter application. Similar to [54, 67], we allocate workers to pools following an exponential distribution with min=50, max=20K, and mean=685. The total number of workers is 685K. Each worker has a private task queue and runs an FCFS policy to process tasks.

Workloads. To analyze the performance in realistic settings, we generate three workloads with different distributions for the task processing time: (1) **Exp (100)** is an exponential distribution with mean=100 μs, which represents the processing time of a single type of tasks with its variability, such as tasks that occur in in-memory key-value stores and caching servers [43, 78], (2) **Bimodal** (50%–50 μs, 50%–500 μs), and (3) **Trimodal** (33.$\overline{3}$%–50 μs, 33.$\overline{3}$%–500 μs, 33.$\overline{3}$%–5000 μs), which together simulate patterns observed in a mix of simple and complex tasks such as *get/put* and *scan* operations [63]. These workloads are similar to the ones used to evaluate RackSched, which ensures fair comparisons.

Task Arrival Model. We generate tasks following a **Poisson** process to stress the system under **bursty** arrival and non-uniform patterns. We keep increasing the system load until we reach the maximum for each worker pool, which is given by $\lambda = n/\bar{s}$, where n is the number of workers in the pool and $\bar{s}$ is the mean task execution time. In the figures, we report the system load as a percentage of the maximum load.

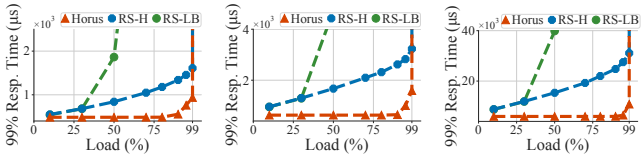
5.2 Comparison against the State-of-Art

Horus vs. RackSched: Single Rack. We compare the performance of Horus versus RackSched in single rack settings. Additionally, we compare against JSQ to show how far Horus is from the theoretical performance bounds; as we discussed in §2, JSQ is not implementable in real environments. We simulate a hypothetical switch that executes JSQ with zero-delay state updates. To be able to compute the optimal results by JSQ, we consider only 10 concurrent applications with workers deployed within the same rack. Figure 10 shows the tail response times for different workloads. Horus substantially outperforms RackSched, and its performance is close to JSQ. This is because Horus tracks the loads on workers more accurately than RackSched, as discussed before in §4.2.



(a) Exponential (b) Bimodal (c) Trimodal

Fig. 10: Comparing Horus against RackSched and JSQ in single-rack settings using simulation.



(a) Exponential (b) Bimodal (c) Trimodal

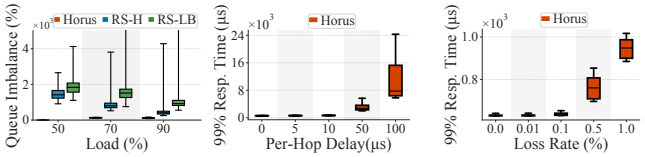
Fig. 11: Comparing Horus against RackSched's extensions in multi-rack settings using simulation.

Horus vs. RackSched's Extension: Datacenter. We compare Horus versus the extensions of RackSched described in 4. To ensure fair comparisons with RS-H, we make each leaf scheduler send the load state of the rack to every available spine scheduler. Figure 11 depicts the tail response time observed by the worker pool with the median size for Horus, RS-LB, and RS-H. The figure shows that Horus reduces the tail response time by up to 3X at moderate loads, and it achieves higher throughput for any target tail response time. The gains are more significant when the dispersion in the task execution times is high, as shown for the Bimodal (Figure 11b) and Trimodal distributions (Figure 11c) compared to the exponential distribution (Figure 11a). This is because the high diversity in the task execution time, coupled with the variability introduced by the Poisson inter-arrivals of tasks, may introduce imbalance in the queues at workers, which are better addressed by Horus.

To analyze the reasons behind the achieved gains, we measure the imbalance in the queues at workers. We define the *imbalance* as the ratio between the maximum queue length and the average queue length within each worker pool. We measure the imbalance every 50 μs and plot the results in Figure 12a, as an interquartile range (1st and 99th percentiles). The results show that Horus spreads the load more uniformly across workers because it tracks their loads more accurately.

5.3 Analysis of Horus

Impact of Network Delay. We analyze the impact of network delays on the scheduling performance of Horus. We increase the average per-hop delay between switches from 0 to 100 μs and measure the tail response time. Figure 12b shows the interquartile range of the tail response time of different applications. As the figure shows, even when the per-hop delay is 10 μs, which is twice the average in real environments [35,55], Horus maintains a low tail response time with small variation. As the per-hop delay increases further to unrealistically high values (50 and 100 μs), the delayed updates from the leaf to spine layer start to impact the scheduling quality of Horus, because such updates may deliver stale information about the worker status in different racks, as shown in Figure 12b.



(a) Queue imbalance (b) Impact of delay (c) Impact of losses

Fig. 12: Analysis of Horus.

Impact of Packet Losses. Datacenter networks have very low loss rates, up to 0.01% [38, 80]. We vary the average packet loss rate from 0 to 1%, which is 100X the loss rate in real datacenters. Figure 12c shows the tail response time of different applications versus the packet loss rate. Even when the average loss rate is 10X (i.e., 0.1%) the normal rate, Horus is not significantly impacted by packet losses. This is because Horus re-transmits the important information such as *idleAdd* and *idleRemove* messages. However, with extreme loss rates, states maintained at switches can be stale for short periods, which causes an increase in response times.

Other Results. We present more results in §C, including analyzing the impact of worker placement, scheduler failures, and the contributions of Horus components to its performance.

6 Conclusions

We presented the design, implementation, and evaluation of Horus, a granular task scheduler for multi-tenant datacenters. In contrast to traditional schedulers, Horus offloads the scheduling of latency-sensitive tasks to network switches, which enables scheduling them at high rates in real time. Horus distributes the load information of workers among network switches, and it introduces a new scheduling policy that minimizes the task response time and does not buffer tasks in switches. We presented multiple ideas and data structures to efficiently realize the scheduling policy in programmable switches. We also designed methods to propagate updated load values among schedulers. We implemented Horus in a testbed with a modern programmable switch and compared its performance against RackSched [78], the state-of-art in-network task scheduler. We also evaluated the performance of Horus in large-scale simulations. Our experimental and simulation results showed that Horus is scalable and robust, and it substantially outperforms RackSched across all performance metrics in both single- and multi-rack settings.

References

[1] Apache openwhisk. https://openwhisk.apache.org/. [Online; accessed September 2022].

[2] Aws lambda. https://aws.amazon.com/lambda/. [Online; accessed September 2022].

[3] Network and Multimedia Systems Lab (NMSL) at SFU. https://nmsl.cs.sfu.ca/.

[4] Placement groups - amazon elastic compute cloud. https://bit.ly/3zMuugf. [Online; accessed April 2023].

[5] Tpc-c. https://www.tpc.org/tpcc/. [Online; accessed September 2022].

[6] Data sharing on traffic pattern inside facebook's data center network. https://bit.ly/3tLgIqz, Jan 2017. [Online; accessed April 2023].

[7] Fault tolerance through optimal workload placement. https://bit.ly/2VMR6wQ, Sep 2020. [Online; accessed April 2023].

[8] Racksched Git Repository. https://github.com/netx-repo/RackSched, 2020. [Online; accessed September 2022].

[9] Apache Lucene search engine. https://lucene.apache.org/, 2021. [Online; accessed September 2022].

[10] Intel Tofino ASIC. https://intel.ly/3fmIP8Q, 2021. [Online; accessed September 2022].

[11] Memcached key-value store. http://memcached.org/, 2021. [Online; accessed September 2022].

[12] Redis in-memory data structure store. https://redis.io/, 2021. [Online; accessed September 2022].

[13] RocksDB. https://rocksdb.org/, 2021. [Online; accessed September 2022].

[14] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proc. of USENIX NSDI'20*, pages 419–434, Santa Clara, CA, February 2020.

[15] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. In *Proc. of ACM SIGCOMM'14*, page 503–514, Chicago, IL, August 2014.

[16] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A Serverless Video Processing Framework. In *Proc. of ACM SoCC'18*, pages 263–274, Carlsbad, CA, October 2018.

[17] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proc. of ACM SIGMETRICS'12*, page 53–64, London, United Kingdom, June 2012.

[18] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. Chatty tenants and the cloud network sharing problem. In *Proc. of USENIX NSDI'13*, pages 171–184, Lombard, IL, April 2013.

[19] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. A High-Speed Load-Balancer design with guaranteed Per-Connection-Consistency. In *In Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 667–683, Santa Clara, CA, February 2020. USENIX Association.

[20] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, March 2017.

[21] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *IEEE micro*, 23(2):22–28, 2003.

[22] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A Maltz, and Ion Stoica. Surviving failures in bandwidth-constrained datacenters. In *Proc. of ACM SIGCOMM'12*, pages 431–442, Helsinki, Finland, August 2012.

[23] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[24] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *Proc. of ACM SIGCOMM'13*, pages 99–110, Hong Kong, China, August 2013.

[25] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In

Proc. of USENIX FAST'20, pages 209–223, Santa Clara, CA, February 2020.

[26] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proc. of ACM SoCC'10*, page 143–154, Indianapolis, IN, June 2010.

[27] Penglai Cui, Heng Pan, Zhenyu Li, Jiaoren Wu, Shengzhuo Zhang, Xingwu Yang, Hongtao Guan, and Gaogang Xie. Netfc: Enabling accurate floating-point arithmetic on programmable switches. In *Proc. of IEEE ICNP'21*, pages 1–11, Virtual Event, November 2021.

[28] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *Proc. of ACM SOCC'16*, pages 497–509, Santa Clara, CA, October 2016.

[29] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proc. of USENIX ATC'15*, pages 499–510, Santa Clara, CA, July 2015.

[30] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proc. of ACM ASPLOS'14*, page 127–144, Salt Lake City, UT, February 2014.

[31] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *Proc. of ACM SoCC'15*, pages 97–110, Kohala Coast, HI, August 2015.

[32] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proc. of ACM SOSP'21*, pages 621–637, Virtual Event, October 2021.

[33] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *In Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, pages 523–535, Santa Clara, CA, March 2016. USENIX Association.

[34] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads. In *Proc. of USENIX NSDI'17*, pages 363–376, Boston, MA, March 2017.

[35] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Proc. of OSDI'16*, volume 16, pages 249–264, Savannah, GA, 2016.

[36] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proc. of ACM SIGCOMM'11*, pages 350–361, Toronto, Canada, August 2011.

[37] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proc. of USENIX OSDI'16*, pages 99–115, Savannah, GA, November 2016.

[38] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proc. of ACM SIGCOMM'15*, pages 139–152, London, United Kingdom, August 2015.

[39] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. of USENIX NSDI'11*, pages 22–22, Boston, MA, March 2011.

[40] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable message latency in the cloud. In *Proc. of ACM SIGCOMM'15*, pages 435–448, London, United Kingdom, August 2015.

[41] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proc. of ACM SOSP'17*, pages 121–136, Shanghai, China, October 2017.

[42] Sangeetha Abdu Jyothi, Mo Dong, and P. Brighten Godfrey. Towards a flexible data center fabric with source routing. In *Proc. of ACM SOSR'15*, pages 1–8, Santa Clara, CA, June 2015.

[43] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *Proc. of USENIX NSDI'19*, pages 345–360, Boston, MA, February 2019.

[44] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proc. of ACM SOSP'21*, pages 605–620, Virtual Event, October 2021.

[45] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Practical Scheduling for Real-World Serverless Computing. November 2021. arXiv: 2111.07226.

[46] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proc. of USENIX ATC'15*, pages 485–497, Santa Clara, CA, July 2015.

[47] D. Katz and D. Ward. Bidirectional forwarding detection (bfd). RFC 5880, RFC Editor, June 2010.

[48] Ibrahim Kettaneh, Sreeharsha Udayashankar, Ashraf Abdel-hadi, Robin Grosman, and Samer Al-Kiswany. Falcon: Low latency, network-accelerated scheduling. In *Proc. of EuroP4'20*, pages 7–12, Barcelona, Spain, December 2020.

[49] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *Proc. of USENIX OSDI'18*, pages 427–444, Carlsbad, CA, October 2018.

[50] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *Proc. of USENIX ATC'19*, pages 863–880, Renton, WA, July 2019.

[51] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[52] Collin Lee and John Ousterhout. Granular Computing. In *Proc. of ACM HotOS'19*, pages 149–154, Bertinoro, Italy, May 2019.

[53] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proc. of ACM SOSP'17*, pages 104–120, Shanghai, China, October 2017.

[54] Xiaozhou Li and Michael J Freedman. Scaling ip multicast on datacenter topologies. In *Proc. of ACM CoNEXT'13*, pages 61–72, Santa Barbara, CA, December 2013.

[55] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. Sundial: Fault-tolerant clock synchronization for datacenters. In *Proc. of OSDI'20*, pages 1171–1186, 2020.

[56] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient microservices on SmartNIC-Accelerated servers. In *Proc. of USENIX ATC'19*, pages 363–378, Renton, WA, July 2019.

[57] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proc. of ACM SIGCOMM '17*, pages 15–28, Los Angeles, CA, August 2017.

[58] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[59] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: A scalable fault-tolerant layer 2 data center network fabric. In *Proc. of ACM SIGCOMM'09*, pages 39–50, Barcelona, Spain, August 2009.

[60] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proc. of ACM SOSP'13*, pages 69–84, Farminton, Pennsylvania, November 2013.

[61] Recep Ozdag. Intel® Ethernet Switch FM6000 Series - Software Defined Networking. *Intel*, page 8, 2012.

[62] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proc. of ACM EuroSys'18*, pages 1–14, Porto, Portugal, April 2018.

[63] Markus Pilman, Kevin Bocksrocker, Lucas Braun, Renato Marroquin, and Donald Kossmann. Fast scans on key-value stores. *Proc. of the VLDB Endowment*, 10(11):1526–1537, 2017.

[64] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proc. of ACM SOSP'17*, pages 325–341, Shanghai, China, October 2017.

[65] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with in-network aggregation. In *Proc. of USENIX NSDI'21*, pages 785–808, Virtual Event, April 2021.

[66] Malte Schwarzkopf. Cluster scheduling for data centers: Expert-curated guides to the best of cs research: Distributed cluster scheduling. *ACM Queue*, 15(5):78–89, October 2017.

[67] Muhammad Shahbaz, Lalith Suresh, Jen Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. Elmo:

Source-routed multicast for public clouds. In *Proc. of ACM SIGCOMM'19*, pages 458–471. Beijing, China, August 2019.

[68] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proc. of USENIX ATC'20*, pages 205–218, Virtual Event, July 2020.

[69] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proc. of ACM SIGCOMM'15*, page 183–197, London, United Kingdom, August 2015.

[70] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A Scalable Low-Latency Serverless Platform. In *Proc. of ACM SoCC'21*, pages 138–152, Seattle, WA, November 2021.

[71] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A unified cluster management system for shared infrastructure. In *Proc. of USENIX OSDI'20*, pages 787–803, Virtual Event, November 2020.

[72] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proc. of SOCC'13*, pages 1–16, Santa Clara, California, October 2013.

[73] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proc. ACM EuroSys'15*, pages 1–17, Bordeaux, France, April 2015.

[74] Dingming Wu, Ang Chen, T. S. Eugene Ng, Guohui Wang, and Haiyong Wang. Accelerated Service Chaining on a Single Switch ASIC. In *Proc. of ACM HotNets'19*, pages 141–149, Princeton, NJ, November 2019.

[75] Dingming Wu, Yiting Xia, Xiaoye Steven Sun, Xin Sunny Huang, Simbarashe Dzinamarira, and TS Eugene Ng. Masking failures from application performance in data center networks with shareable backup. In *Proc. of ACM SIGCOMM'18*, pages 176–190, Budapest, Hungary, August 2018.

[76] Yiting Xia, Xiaoye Steven Sun, Simbarashe Dzinamarira, Dingming Wu, Xin Sunny Huang, and TS Eugene Ng. A tale of two topologies: Exploring convertible data center network architectures with flat-tree. In *Proc. of ACM SIGCOMM'17*, pages 295–308, Los Angeles, CA, August 2017.

[77] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using programmable switches. In *Proc. of ACM SIGCOMM'20*, pages 126–138, Virtual Event, August 2020.

[78] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *Proc. of USENIX OSDI'20*, pages 1225–1240, November 2020.

[79] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers(technical report). 10 2020.

[80] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proc. of ACM SIGCOMM'17*, pages 362–375, Los Angeles, CA, 2017.

Appendix A More Details of Horus

This appendix describes additional details that we could not fit into the main paper.

A.1 Example: Lazy State Update

We illustrate the implementation of the lazy state update algorithm in programmable switches in Figure 13. The figure shows the processing of a resubmitted packet. The two copies of *loadList* are maintained in stages 1 and 2, and the two copies of *driftList* are in stages 4 and 5. Each of stages 1 and 2 randomly selects a node and passes the ID of this node and its load to the subsequent stages as metadata (shown at the bottom of the figure). The selected nodes in this example are $m = 0$ from stage 1 and $n = 2$ from stage 2. Stage 3 chooses the node with the smaller load, m, and computes the difference in the load between m and n, $l_n - l_m = 1$.

In stage 4, the drift, $d_m = 3$, is found to be greater than the load difference between m and n, i.e., $d_m > (l_n - l_m)$. This means that the initial scheduling decision (assuming $q_m < q_n$) may not be accurate, since it could be underestimating the actual load of m. Thus, a flag is set to resubmit this packet to update all lists and select the correct node. The drift for the second node, d_n, is read in stage 5 from the second copy of *driftList* and passed to stage 6, which calculates the actual load of both nodes q_m and q_n as the sum of drift and load values and resubmits the packet with this data.

In the resubmission pass, the scheduler becomes aware of the actual load values of the two nodes, i.e., $q_m = 3+3 = 6$ and $q_n = 4+1 = 5$, and it selects the least loaded node, which is now $n = 2$. The scheduler then increments the *loadList* of the least loaded node. The *loadList* is updated with the new values, and the corresponding entries in the *driftList* are reset to 0. Notice that in the resubmission pass, we do not read items from the *loadList* or *driftList* as these values are injected in the resubmitted packet. This allows us to increment the load list values while respecting the strict memory access requirement of programmable switches.

Finally, we note that memory updates in programmable switches are done atomically: a packet may update memory locations at different stages of the pipeline, but the following packet will not observe such updates until they re fully completed. *loadList* and *driftList* values are only updated upon receiving Load Update packets from downstream nodes. *driftList* is only incremented* in the first pass for task packets, and *loadList* is only incremented in the resubmission pass. Therefore, even with re-ordering of resubmitted packets and other state updates, Horus does not introduce any race conditions.

*An atomic read-modify-write operation that prevents race conditions in the pipeline.

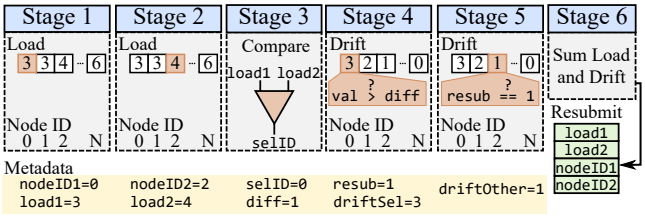


Fig. 13: Example of resubmitting a packet while scheduling tasks to busy workers.

A.2 Realizing Average Queue Length

Calculating an average value in the switch data plane is infeasible due to the lack of support for floating-point arithmetic in programmable switches. To mitigate this issue, we approximate the calculation of the average by using a *fixed-point* representation. We use a 32-bit number to represent an average load value, which is communicated among switches using the `qLen` field in Figure 3. The 32 bits are divided equally among the integer and fraction parts, which can support an accuracy of 2^{-16} that is sufficient for most practical cases. We note that Horus can support existing approximations of floating-point operations, e.g., [27], at the cost of additional switch resources.

When the resource manager allocates w workers in a rack for a worker pool, the Horus controller calculates an additive factor with a value of $1/w$ using the fixed-point representation, and updates the corresponding leaf switch with this value, which is maintained in a table. For example, when a worker pool has 8 workers in a rack, then the additive factor is 0x00002000, which is interpreted as $1/8$. Notice that this value can be maintained in switches as a 32-bit number.

The calculation of an average value is a sequence of increments/decrements of the corresponding additive factor. The leaf scheduler uses the `poolID` as a key to access the table, and uses add/subtract of the additive factor to increase/decrease the average value for every started/finished task. As an example, when the 8 workers have 11 tasks in their queues, then the average queue length is $11/8 = 1+3/8$, which can be represented as 0x00016000.

A.3 Handling Failures and Packet Losses

Horus employs simple mechanisms to notify the impacted schedulers about various failures to reduce disrupting ongoing tasks. These mechanisms are deployed at the centralized controller and switches' controllers. We rely on existing protocols [47, 59, 75] to detect failures and notify the fabric manager. We assume that the centralized controller is replicated on multiple servers using algorithms such as Paxos [51], and it receives failure events from the fabric manager.

Switch Failures. Upon receiving a leaf or spine switch failure event, the centralized controller instructs all impacted

switches' controllers to remove the failed switch from their memories and reset any state linkage information. Thus, the failed switch will not be used for scheduling incoming tasks. We note that leaf switch failures usually result in network partitioning. Therefore, no new tasks could be assigned to workers inside the rack during leaf switch failures, and Horus does not attempt to retrieve the workers' soft state stored at the failed leaf switch.

Server Failures. The leaf switch controller locally detects the failures of servers in the rack based on heartbeat packets and a fixed timeout value. This localizes the state maintenance at the rack level. After detecting a failure, the leaf controller decreases the number of available workers and instructs its data plane to remove the failed worker from its data structures.

Packet Losses. Packets carrying the submitted tasks as well as packets sent to update the state at different schedulers can be lost. Horus delegates handling the first case to the submitting applications. Since Horus is designed for short-lived (μ or millisecond sec) tasks, applications typically submit multiple tasks in parallel and ignore the failed ones, as the latency of resubmitting a failed task can be larger than the execution time of the task itself.

Horus only retransmits the lost Idle Add/Remove messages that are sent from a leaf to a spine. To detect a lost update packet, the leaf and spine schedulers use a simple protocol: Spine schedulers set the *qLen* field of the scheduled task to *idleSelected* when it selects a rack from the *idleList*. When the task arrives at the leaf, it checks the header field to determine the state of the rack in the spine's memory. If it mismatches the current state, it will resend an Idle Add/Remove packet to the spine based on the correct state. Horus frequently sends Load Update packets to maintain the state at different schedulers. Horus does not retransmit lost Load update packets, since this would be costly and ineffective for the target application environment. Rather, it relies on subsequent update packets carrying fresh information to bring the state up to date. Recall that Horus strives to *approximate* the current load on workers; it is nearly impossible to make schedulers track the exact load on every single worker, given the very short execution time of tasks and the high dynamics nature of the workload.

A.4 Handling Multi-packet Tasks

When a task is composed of multiple packets, Horus schedulers need to send these packets to the same worker. This is known as task affinity. Applications expecting to submit multi-packet tasks, set the *isLastPacket* flag to 0 for all packets of the task except the last one. Also, for the non-first packets of the task they should set the *type* in the Horus header to *taskContinuation* (Figure 3).

Similar to prior approaches, e.g., [57, 78], Horus maintains a connection table at switches. When the first packet of a task arrives at a spine or leaf scheduler, the scheduler assigns it to a node using its normal operation. Let us denote the ID of this node by *nodeID*, which can be an ID of a rack (in the case of a spine scheduler) or an ID of a worker (in the case of a leaf scheduler). If the *isLastPacket* field is not set for the first packet of task, the scheduler adds the entry $\langle hash(poolID, taskID),\ nodeID\rangle$ to its connection table. For subsequent packets with the same *taskID*, the scheduler forwards them to the same node using the connection table. We note that the leaf scheduler adds another field to each entry of the connection table: *spineID*, which specifies the ID of the spine switch from which the packet came.

Entries in the connection table are removed in one of two ways. First, when a task submission by client completes, the final packet with *type* of *taskContinuation* comes and the *isLastPacket* flag is set to 1, then the scheduler removes the corresponding connection table entry. The second way to remove an entry from the connection table is through timeout. Each entry automatically disappears after a pre-specified period of time (in the order of 10s of milliseconds). This takes care of failed tasks and lost final packets.

We note that the connection table is only maintained for multi-packet tasks, not for short-lived granular tasks that are composed of single packets.

A.5 Horus Overheads

Horus imposes multiple types of overheads. First, it attaches a small header of size 11 bytes to include information such as task ID and queue length as shown in Figure 3. Second, it maintains state at switches to enable realizing a load-aware scheduling policy. This consumes part of the memory of the programmable switches. For granular tasks, which is the main target of Horus, the maintained state at switches is *independent* of task rates, which is an important property that makes Horus scalable.*

The state, however, grows with the number of applications submitting tasks and the number of workers assigned to each of them. Let us consider one datacenter application with a total of W workers allocated across R racks, where typically $R \ll W$. Spine schedulers do not maintain state about individual workers. Rather, they maintain the average worker load in different racks. Thus, the memory requirements on spine schedulers are in the order of $O(R)$. Specifically, each spine scheduler maintains one copy of the *idleList* and two copies of each of the *loadList* and *driftList* data structures. It also maintains a table mapping each rack to the fractional fixed-point additive factor for updating the load values. The number of entries in each list is R, each is 16 bits.

Leaf schedulers maintain a state about workers in their racks, which is on average $O(W/R)$ when workers are uniformly allocated across racks, and $O(W)$ in the worst case

*For multi-packet tasks, Horus maintains a connection table to ensure task affinity. As discussed in §A.4, applications expected to submit multi-packet tasks set the `isLastPacket` field to 0 in the Horus header (Figure 3), and only state about such tasks are maintained in the connection table.

Algorithm 1 Scheduling and state updates for idle nodes

```
    - p: reg. pointing to the next available slot in idleList
    - idleList: reg. array that holds the IDs of idle nodes
    - idleIndex: reg. array holding indices of the nodes in
    idleList.
1:  // On an idleAdd pkt received
2:  function ADD(pkt)
3:      readInc(p)
4:      idleList[p] ← pkt.srcID // pkt.srcID is idle node ID
5:      idleIndex[pkt.srcID] ← p
6:  function SCHEDULETASKIDLE(pkt)
7:      readDec(p) // Only a leaf scheduler decrements p
8:      selectedNode ← idleList[p − 1]
9:      Update pkt with IP of selectedNode and Forward
10: (A) On an idleRemove pkt received: First Path
11: function REMOVE(pkt)
12:     readDec(p)
13:     lastNodeID ← idleList[p]
14:     removedNodeIdx ← idleIndex[pkt.srcID]
15:     resubmit(lastNodeID, removedNodeIdx)
16: (B) On an idleRemove pkt received: Resubmit Path
17: function REMOVE(lastNodeID, removedNodeIdx)
18:     idleList[removedNodeIdx] ← lastNodeID
19:     idleIndex[lastNodeID] ← removedNodeIdx
```

when all workers are in the same rack. Leaf schedulers maintain the same *idleList*, *loadList*, and *driftList* data structures as spine schedulers.

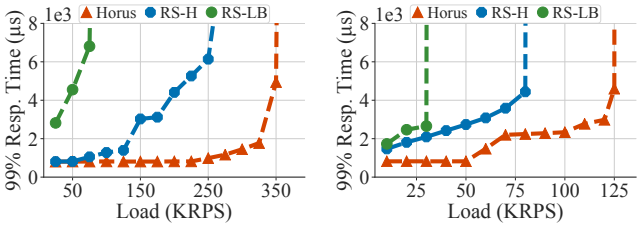
For illustration, consider an application with $W = 10,000$ workers uniformly distributed across $R = 10$ racks. A spine scheduler would need up to 120 bytes of memory, whereas a leaf scheduler would need approximately 1000 bytes. The recent Tofino switch has a few hundred Megabytes of memory.

In addition, Horus exchanges messages to update the state at schedulers. However, following our design principles, we keep the worker state localized within individual racks, and we only send aggregated updates to spine schedulers. In addition, Horus piggybacks the update messages with response packets of tasks.

A.6 Pseudo Code and P4 Implementation

Algorithm 1 and Algorithm 2 show the high-level pseudo code of Horus for handling idle nodes and busy nodes. Our implementation only uses the stages in the ingress pipeline of the switch. Upon receiving a task, Horus schedulers will access and read the number of available idle noads and only use the procedure in Algorithm 2 (line 15) if there are no available idle nodes.

Recall that removing an idle node after being selected by SCHEDULETASKIDLE depends on the scheduler type. A leaf scheduler removes the selected idle worker immediately after sending a task to it because the worker is no longer idle. A spine scheduler, however, only reads p and does not decrement it, since removing an idle node in a spine scheduler is triggered by an *idleRemove* packet sent by a leaf scheduler.



(a) Uniform (b) Skewed

Fig. 14: Testbed results for the RocksDB workload. Comparing Horus vs RackSched extensions in multi-rack settings: (a) 90%-GET, 10%-SCAN, (b) 50%-GET, 50%-SCAN distributions.

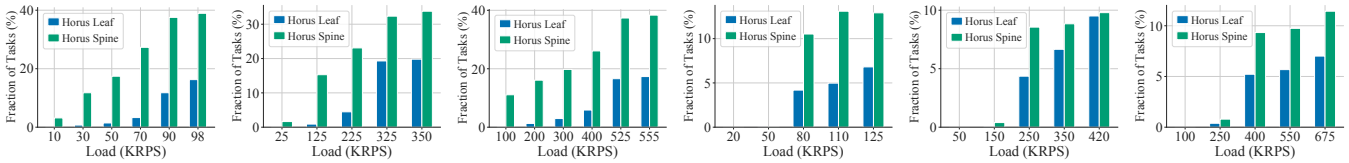
The UPDATE procedure (Algorithm 2, line 2) is triggered by receiving a reply packet from the worker at leaf schedulers. At the spine layer, updates are explicitly triggered by *loadUpdate* packets sent by leaf schedulers. Note that leaf schedulers increment the load or drift values after selecting the target worker by one, which reflects the selected worker queue length after the assignment. At the spine layer, the values are incremented by the additive factor for the selected rack ($1/\#workers$), based on the value stored in the tables as described in §A.2, which is not shown in the pseudo codes for simplicity.

Generating state update messages in the data plane can be challenging since packet generation is triggered based on the real-time state of the workers. Limiting the processing to the ingress pipeline of the switch enables us to realize this efficiently, without recirculating the packets. The leaf switch checks the trigger conditions when processing each arriving packet. If the condition to send an update is met, it duplicates the original packet via the traffic manager and sends the original copy to its destination. The switch then modifies the header fields of the other copy and sends it as an update message to the upper layer. As an example, *idleRemove* packet is generated when the switch receives a task and becomes aware that no more idle workers are available. The switch duplicates the packet, forwards the original packet to the worker, changes the *type* field to *idleRemove*, sets the *srcId* to the ID of the leaf switch, and sends the copied packet header to the spine.

Appendix B More Results from Testbed

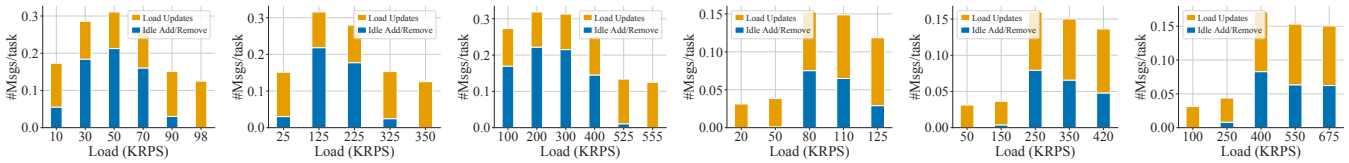
This appendix provides more evaluation results obtained from the testbed.

Response Time. We present additional results for comparing Horus against RackSched's extensions in the multi-rack set-



(a) RocksDB (50-50) (b) RocksDB (90-10) (c) TPC-C (d) RocksDB (50-50) (e) RocksDB (90-10) (f) TPC-C

Fig. 15: Fraction of resubmitted tasks for the Uniform worker placement in (a)-(c) and for the Skewed worker placement (d)-(f).



(a) RocksDB (50-50) (b) RocksDB (90-10) (c) TPC-C (d) RocksDB (50-50) (e) RocksDB (90-10) (f) TPC-C

Fig. 16: Rate of state update messages for the Uniform work placement in (a)-(c) and for the Skewed worker placement in (d)-(f).

tings in Figure 14. As shown in Figure 14a, for the workload with 90%-GET and 10%-SCAN requests, even using uniform worker placement across the racks, Horus achieves significantly lower tail latency at moderate and high loads. Similarly, Figure 14b shows that for the workload with 50%-GET and 50%-SCAN requests, under a skewed worker placement, the system maintains a lower latency and sustains up to 125 KRPS throughput, which is 60% higher than RS-H and 400% higher than RS-LB.

More Overhead Results. A Horus scheduler may selectively resubmit a fraction of task packets to the switch pipeline to update the scheduler view after making a scheduling decision. We measure the fraction of task packets that are processed twice by a spine or leaf switch in our experiments. Figure 15 shows that the maximum fractions of resubmitted tasks are 38% and 13% for the Uniform and Skewed placement setups, respectively. When the load is low, the scheduler does not need to resubmit tasks as most of the tasks are scheduled based on idle nodes, and the rate of reply packets is high enough to automatically update the scheduler state. We note that simple solutions to update the state would result in a 100% resubmission rate because they resubmit every packet after scheduling a task.

Further, we note that the rate of resubmitted packets is correlated with the placement of workers and the size of worker pool. Recall that resubmissions are triggered when the drift value is greater than the difference between the two sampled load values while scheduling a task. Therefore, the rate of resubmissions at the spine is impacted by the closeness of the average load values of the racks in the spine memory. In addition, the number of available racks impacts the rate of resubmissions at the spine layer. That is, having a small number of racks increases the probability of triggered resubmissions as this increases the likelihood of drawing the same two random samples for multiple arriving tasks before the state update arrives, which results in a resubmission.

Next, we analyze the different types of state update messages processed by the spine scheduler in our testbed. Since leaf schedulers passively track the load by processing tasks and reply packets passing through them, there are no extra overheads for state updates inside the racks. We measure and plot the rate of messages normalized by the task rate across different workloads and worker settings in Figure 16. There are two types of messages sent to the spine layer. Selective Load Updates are sent to update the average load of the rack, and Idle Add/Remove messages are sent when the state of the rack changes from busy to idle and vice versa.

As shown in Figure 16, the rate of the Load Update messages at the highest workload is less than 0.15. Horus significantly reduces the overheads compared to previous works (e.g., [48, 60, 78]), which require at least processing one message per task. The figure also shows a small rate for the Idle Add/Remove messages. Similar to the fraction of resubmitted packets, the rate of messages is impacted by the placement of workers and the size of the worker pool. The number of workers in each rack impacts the rate of oscillation between the idle states as well as the rate of required average load updates.

Appendix C More Results from Simulation

This appendix includes additional results from our simulation.

Analysis of Horus Components. We analyze the contributions of the two components of the proposed scheduling policy: (i) scheduling tasks to idle nodes using idleness information and (ii) scheduling tasks to busy nodes using the power-of-two policy. In this experiment, we focus on five sample worker pools with sizes ranging from 50 to 20,000 workers each, and we use the Bimodal task distribution.

Algorithm 2 Scheduling and state updates for busy nodes

- loadList: reg. array that holds the queue length of nodes (two identical copies maintained).
- driftList: reg. array holding the difference between values in loadList and actual load (two identical copies maintained).

```
1:  // On a taskReply or loadUpdate pkt received
2:  function UPDATE(pkt)
3:      loadList1[pkt.srcID] ← pkt.qlen
4:      loadList2[pkt.srcID] ← pkt.qlen
5:      driftList1[pkt.srcID] ← 0
6:      driftList2[pkt.srcID] ← 0
7:  // Atomic, read-modify-write operation on the reg.
8:  function CHECKDRIFT(selectedIdx, diffSamples)
9:      if driftList1[selectedIdx] ≤ diffSamples then
10:         Increment driftList1[selectedIdx]
11:         return NORESUB
12:     else
13:         return driftList1[selectedIdx]
14: (A) On an task pkt received: First Path
15: function SCHEDULETASKBUSY(pkt)
16:     randIdx1, randIdx2 ← genRandomSamples()
17:     sample1 ← loadList1[randIdx1]
18:     sample2 ← loadList2[randIdx2]
19:     selectedIdx, diffSamples ← CompareSamples()
20:     driftSelected ← CheckDrift()
21:     if driftSelected == NORESUB then
22:         Increment driftList2[selectedIdx]
23:         Update pkt headers and Forward
24:     else
25:         load1 ← driftSelected + loadSelected
26:         load2 ← driftList2[otherIdx] + loadOther
27:         resubmit(selectedIdx, load1, otherIdx, load2)
28: (B) On an task pkt received: Resubmit Path
29: function SCHEDULETASKBUSY(selectedIdx, load1,
    otherIdx, load2)
30:     selectedIdx ← CompareSamples()
31:     Increment loadList1[selectedIdx]
32:     Increment loadList2[selectedIdx]
33:     Update pkt headers and Forward
```

We simulate two variants of the power-of-two policy. The first relies only on reply packets from workers to update the state, as done in RackSched [78]. This is referred to as Pow-of-2 Delayed Updated (DU). The second variant, which is used in Horus, updates the state while scheduling tasks, and it may require resubmitting packets through the switch. We also simulate a scheduler that uses the idle node selection only: it schedules tasks to idle nodes, and if there are no idle nodes, it will assign tasks to nodes randomly.

The results are presented in §C, where we present the average response time versus the system load. The results show

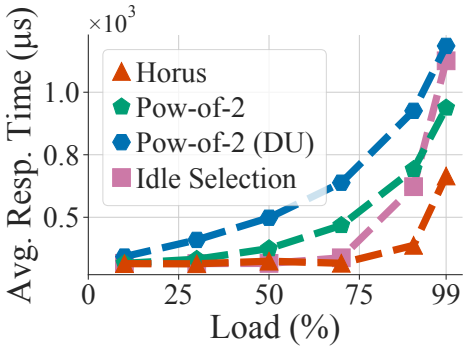


Fig. 17: Analysis of Horus components.

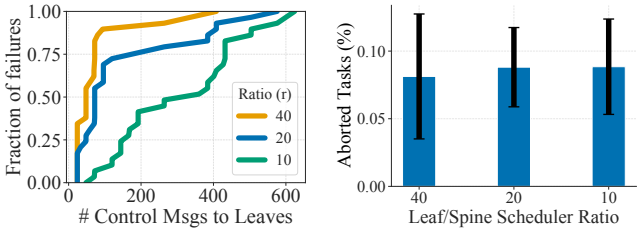
the impact and importance of the two components of Horus. That is, Horus achieves its performance by effectively tracking the load on nodes and assigning tasks to idle nodes whenever they are available. And when there are no idle nodes, Horus uses the power-of-2 policy with accurate load information to assign tasks to nodes with lower loads. Further, the randomness in the power-of-2 policy makes Horus robust against the task herding problem.

Impact of Scheduler Failures. We analyze the impact of spine scheduler failures while all of the 1K worker pools are running. Upon detecting a failure, the centralized controller notifies the leaf schedulers impacted by the failed spine scheduler. Similar to [38], we add latency to the control messages carrying the failure notices based on the number of hops between the failed spine and each impacted leaf. The experiment is repeated 30 times; each time, we fail a random spine scheduler. Notice that burst failures of spine switches are rare; the median time between failures of such switches is multiple hours [36]. Therefore, we only consider single spine failures.

We define r as the ratio of leaf to spine schedulers to control the number of spine schedulers per worker pool. For example, $r = 40$ indicates using four times fewer spine schedulers compared to $r = 10$, for the same number of leaf schedulers.

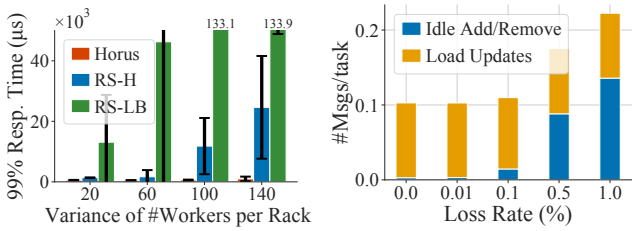
Figures 18a and 18b show the impact of spine scheduler failures and the trade-off for using different numbers of spine schedulers for worker pools. Figure 18a shows the number of messages sent from the centralized controller to the leaf switches as a result of the failure. When the state is distributed among more spine schedulers (i.e., small r), a failure results in more control messages sent. This is because the Horus controller sends a message to each leaf switch that needs to update its state. In the worst case ($r = 10$), an average of 306 (maximum 624) messages need to be sent for each spine failure event, which is a small message rate; centralized controllers in today's datacenters can send thousands of updates per second [59, 76].

Figure 18b shows the fraction of aborted scheduling tasks during spine switch failures. For all r values, less than 0.1% of the total submitted tasks are aborted on average during a



(a) Overhead during failure (b) Aborted tasks during failure

Fig. 18: Impact of spine scheduler failures.



(a) Worker placement (b) Packet losses

Fig. 19: Impact of worker placement and packet losses.

spine failure. This is because of the distributed scheduling in Horus, which enables multiple spine schedulers to handle the tasks submitted to a worker pool. Horus equally distributes the tasks belonging to a worker pool among spine schedulers. Therefore, before the failed switch is removed from the list of schedulers, only a small fraction of tasks that were sent to that switch may be affected. The aborted tasks can be re-launched by the application, which will be routed to the other active spine schedulers. Using more spine schedulers per worker pool provides better availability for the worker pools that were using the failed spine scheduler, because the scheduling requests of each worker pool are distributed uniformly among a larger number of spine schedulers.

Impact of Worker Placement. Datacenter operators may use different policies, e.g., [18], to allocate workers to worker pools, which can result in various worker distributions across racks. We analyze the impact of worker distribution on the performance of the task scheduler. Recall that we simulate 1K worker pools that have different numbers of workers (according to exponential distribution) randomly distributed across racks. We quantify the diversity in worker distribution by computing the variance in the number of workers per worker pool. A high variance indicates more scattered workers. We group worker pools that observed similar variance in the worker distributions together.

We plot, in Figure 19a, the average of the tail response time (at 90% load) observed by various worker pools. We also plot in the same figure, as error bars, the average plus/minus one standard deviation. The figure shows that as the variance in the worker distribution increases, RS-H and RS-R result in higher and more variable tail response times, whereas the response time of Horus remains stable across all worker distributions. For example, when the workers are highly scattered across racks (i.e., variance is 140), Horus reduces the tail response times by up to 94% and 99% compared to RS-R and RS-H, respectively.

The results in Figure 19a imply that the performance of Horus is robust against different worker distributions, which offers flexibility to datacenter operators to employ various worker allocation policies. The robustness of Horus is achieved by more accurately tracking the load on workers compared to RS-R and RS-H.

Impact of Packet Losses on Update Messages. Figure 19b shows the impact of packet loss on the rate of different types of update messages. Since Horus sends average load updates periodically based on the number of tasks that enter and exit the rack, the rate of load updates is not sensitive to packet losses. The *idleAdd* and *idleRemove* messages are, however, re-transmitted in case of loss. For example, when an *idleRemove* packet from a leaf to a spine is lost, it re-transmits the message until the leaf is removed from the idle list. This adds extra overhead on switches. As the figure shows, the rate of *idleAdd* and *idleRemove* messages increased by only 15% when the loss rate is 0.1%. Even with the much higher loss rates of 0.5 and 1%, Horus still functions properly, albeit at increased rates of *idleAdd* and *idleRemove* messages.

Fast Vector Query Processing for Large Datasets Beyond GPU Memory with Reordered Pipelining

Zili Zhang Fangyue Liu Gang Huang Xuanzhe Liu Xin Jin

School of Computer Science, Peking University

Abstract

Vector query processing powers a wide range of AI applications. While GPUs are optimized for massive vector operations, today's practice relies on CPUs to process queries for large vector datasets, due to limited GPU memory.

We present RUMMY, the first GPU-accelerated vector query processing system that achieves high performance and supports large vector datasets beyond GPU memory. The core of RUMMY is a novel *reordered pipelining* technique that exploits the characteristics of vector query processing to efficiently pipeline data transmission from host memory to GPU memory and query processing in GPU. Specifically, it leverages three ideas: (i) *cluster-based retrofitting* to eliminate redundant data transmission across queries in a batch, (ii) *dynamic kernel padding* with *cluster balancing* to maximize spatial and temporal GPU utilization for GPU computation, and (iii) *query-aware reordering and grouping* to optimally overlap transmission and computation. We also tailor GPU memory management for vector queries to reduce GPU memory fragmentation and cache misses. We evaluate RUMMY with a variety of billion-scale benchmarking datasets. The experimental results show that RUMMY outperforms IVF-GPU with CUDA unified memory by up to 135×. Compared to the CPU-based solution (with 64 vCPUs), RUMMY (with one NVIDIA A100 GPU) achieves up to 23.1× better performance and is up to 37.7× more cost-effective.

1 Introduction

The breakthroughs in Deep Learning (DL) [1] enable unstructured data (e.g., images, videos, and audios) [2, 3] to be represented as high-dimensional feature vectors for serving a wide range of AI applications [4–12]. In particular, recent advancements in Large Language Models (LLMs) [13, 14] have catalyzed the emergence of a new generation of AI applications. However, LLMs only support a short-term memory (32k tokens limit for GPT-4 [15]). Vector databases [16–20] can provide persistence and long-term memory for LLMs. Consequently, Retrieval Augmented Generation (RAG) is applied to augment LLMs by dynamically retrieving relevant documents from a database during the generation process, thereby expanding LLMs' knowledge base and contextual understanding. Specifically, RAG first converts personal or organizational documents into vectors to build a vector database. When users post a question, it first issues a *vector query* to identify a set of documents that may contain the answer (i.e., find the most similar vectors from a vector database for a given vector). These documents, along with the original question, are then fed into an LLM, which analyzes the text information and returns the final answer. Additionally, vector query processing is also adopted in a wide range of applications, such as recommendation systems [4–6, 21], recognition [7–9, 22], and information retrieval [10–12].

With the explosive growth of the dataset scale, vector query becomes a performance bottleneck for AI applications. To reduce query overhead, GPUs, optimized for massive vector operations, are a natural choice to process vector queries. This is exemplified by early GPU-based vector query processing systems that load vector datasets into GPU memory [23]. A key problem is that they cannot support large datasets due to limited GPU memory. The explosion of unstructured data [3] and the increasing adoption of DL in production [4–9, 21, 22] make this problem particularly acute. Even high-end GPUs like NVIDIA H100 and A100, with tens of GB of memory, fall short of today's large vector datasets [24, 25] containing billions of items with a memory footprint of hundreds of GB. Thus, CPU-based solutions [16, 19, 23] are still the de facto choice for billion-scale datasets in production.

Conceivably, GPU memory can be augmented with host memory to support large datasets. A straightforward solution is to divide the dataset into multiple parts in host memory. Each part can be transmitted into GPU memory in rotation to process vector queries. This solution performs transmission and computation sequentially and cannot well utilize GPU resources. Another possible solution is to integrate a GPU-based solution with CUDA unified memory [26], which automatically handles GPU memory swapping and supports parallel transmission and computation. As CUDA unified memory is unaware of vector queries, this solution would incur massive GPU memory page faults, leading to low performance.

We present RUMMY, the first GPU-accelerated vector query processing system that (i) supports large vector datasets beyond GPU memory, and (ii) achieves higher performance and is more cost-effective than CPU-based solutions. The core of RUMMY is a novel *reordered pipelining* technique that

exploits the characteristics of vector query processing to efficiently pipeline data transmission from host memory to GPU memory and query processing in GPU.

Three primary technical challenges must be overcome to realize RUMMY, each pertaining to a different aspect: transmission, computation, and pipelining. First, redundant transmission occurs when processing vector queries in a batch. Due to limited GPU memory, the same data subset is transmitted and evicted repeatedly for different queries in the same batch. Leveraging the fact that each subset's processing is independent, we design *cluster-based retrofitting* to restructure the query plan for each vector query and *completely* eliminate redundant transmission.

Second, the Streaming Multiprocessors (SMs) in the GPU suffer from low utilization due to pipelining-caused load imbalance. Specifically, pipelining segregates the query processing into groups, each corresponding to a GPU kernel. A kernel comprises a grid of thread blocks, and each block is executed within an SM. A kernel for a small group is launched with a small grid of thread blocks, resulting in idle SMs (i.e., spatial underutilization). Besides, the discrepancy between the size of different data parts causes stragglers—some thread blocks execute longer than others (i.e., temporal underutilization). To maximize utilization, we introduce *dynamic kernel padding* with *cluster balancing*. Cluster balancing equalizes data part sizes offline. Dynamic kernel padding fills idle SMs with padding thread blocks online to enhance GPU occupancy.

Third and most importantly, directly applying pipelining suffers from limited overlapping between transmission and computation, and has high pipelining overhead due to frequent API invocations and synchronizations. The problem is exacerbated by that the pipelining plan can only be decided at runtime when queries arrive, as the quality of a particular plan is query-dependent. We design *query-aware reordering and grouping*, which dynamically reorders the query plan and divides the plan into groups with two algorithms. The reordering algorithm hides small transmission operations with large computation operations as many as possible to maximize the overlapping. The grouping algorithm finds the best tradeoff between pipelining efficiency and pipelining overhead. The two algorithms are lightweight with negligible runtime overhead. We prove that the two algorithms achieve optimal pipelining performance, respectively.

In addition, we also tailor the GPU memory management for vector query processing. RUMMY pre-allocates GPU memory and re-allocates the memory to each transmission task, without involving the general-purpose but costly GPU memory manager. We also exploit the vector query processing to reduce GPU memory fragmentation and cache misses.

We remark that the runtime *reordered pipelining* technique in RUMMY is distinct from canonical pipelining techniques adopted in microprocessor architectures [27, 28], DL systems [29, 30], and task schedulers [31, 32]. Those techniques rely on dependent and deterministic execution flows. For instance, DL models have a pre-defined layer-by-layer computation graph, which enables solutions like PipeDream [30] and PipeSwitch [29] to compute a pipelining plan *offline*. Besides, to ensure correctness, the deterministic computation graph cannot be *reordered*. In contrast, a vector query plan is *non-deterministic* and can be executed in any order since the processing of each subset is *independent*. The key novelty of RUMMY is that it leverages the independent and non-deterministic nature of vector query processing to dynamically decide the pipelining plan with *reordering* at *runtime*, and achieves optimal parallelism between transmission and computation.

In summary, we make the following contributions.

- We present RUMMY, to the best of our knowledge, the first GPU-accelerated vector query processing system that achieves high performance and supports large vector datasets beyond GPU memory.
- We introduce a new runtime *reordered pipelining* technique, which leverages *cluster-based retrofitting*, *dynamic kernel padding* with *cluster balancing*, and *query-aware reordering and grouping* to efficiently pipeline data transmission and query processing.
- We implement a RUMMY prototype. The evaluation shows that RUMMY outperforms IVF-GPU with CUDA unified memory by up to 135×. Compared to the state-of-the-art CPU solution (with 64 vCPUs), RUMMY (with one NVIDIA A100 GPU) achieves up to 23.1× better performance and is up to 37.7× more cost-effective.

2 Background and Motivation

In this section, we first introduce the background of vector query processing. We then describe GPU-based solutions to accelerate vector query processing. Finally, we summarize the challenges to support billion-scale datasets for GPU-based vector queries, which motivate the design of RUMMY.

2.1 Vector Query Processing

DL models convert unstructured data into high-dimensional feature vectors to serve applications [4–12, 21, 22]. These vector datasets [24, 25] are utilized to construct a vector database [16–20] as the persistence and long-term memory. Specifically, RAG leverages these datasets to enhance the quality of LLM-based AI applications. For instance, TEXT1B [24] contains one billion data entries from both textual and visual modalities, which can be used to enhance multimodal LLMs. Vector queries are referred to as the memory retrieval part in RAG and are widely used in emerging LLM-based AI applications [33–35].

Specifically, a vector query is to find the top-k nearest neighbors (KNN) in a vector dataset that are most similar to the given vector. KNN returns the exact top-k results and requires searching on the entire dataset. KNN becomes impractical for large datasets due to high query latency. Approximate top-k nearest neighbor (ANN) search trades query accuracy

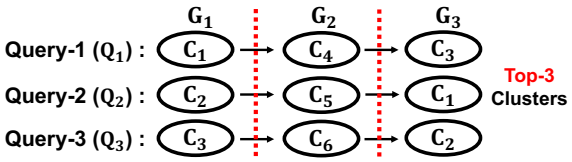


Figure 1: Query plan for a batch of three queries.

for query latency. Previous works [36] show that ANN can achieve 99% accuracy while only searching on 1% of the dataset. Even for accuracy-sensitive scenarios, ANN can meet the requirements of applications and outperform KNN by hundreds of times. Since DL models inherently introduce errors when generating vectors, ANN is widely used in existing vector databases for large datasets [16, 18, 19, 37–40].

The basic idea of ANN search is to build an index structure [41–44] to sample a subset of the dataset for answering a vector query. There are two representative ANN indexes, inverted file index (IVF) [41,45,46] and graph index [42,47,48]. While graph index is more accurate, it is more expensive to build and maintain due to its large memory footprint and building time. When confronting billion-scale datasets, IVF is proved to be more efficient and accurate than the state-of-the-art graph index [48] with the same memory consumption according to recent work [46]. Thus, IVF is more suitable for very large datasets. While IVF has many variants, their general workflow is similar. At offline, IVF trains a list of clusters ($\{C_1, C_2, ...C_l\}$) by k-means clustering [49], and each vector is assigned to the closest cluster. Online, it performs pairwise comparison between the query vector and each vector in the top-n closest clusters to produce the approximate top-k nearest neighbor vectors. The number of sampled clusters n is a configurable parameter to make a tradeoff between accuracy and latency.

For example, Figure 1 shows a query plan of the batch of three queries ($\{Q_1, Q_2, Q_3\}$). IVF selects top-3 nearest clusters (i.e., $n = 3$) for each query, e.g., $\{C_1, C_4, C_3\}$ for Q_1. Q_1 is compared with each vector in the three clusters, in which the top-k nearest vectors are returned as the result.

2.2 GPU Acceleration

Vector query processing is compute-intensive and involves massive vector operations. GPUs are a natural choice to accelerate vector processing [23]. The pairwise distance computation between high-dimensional vectors is a good fit for the GPU architecture. A detailed comparison between CPUs and GPUs on IVF and graph index in terms of performance and cost is listed in Appendix A.2, which demonstrates the benefits of GPU-based vector query processing systems.

Vector query processing in GPUs. A GPU consists of many SMs, each including numerous CUDA cores for extensive vector operations. Such hardware architecture involves a hierarchical parallel computation model [50]: each computation consists of multiple CUDA kernels; each CUDA kernel includes a grid of thread blocks; and each block comprises a number of threads. A block of threads can be executed simultaneously within one SM, where each thread is executed on a CUDA core. To process a vector query in a GPU, the computation of the query on one cluster (e.g., $Q_1 \rightarrow C_1$ in Figure 1) is a thread block. The computations of different queries on their corresponding clusters (e.g., $\{Q_1 \rightarrow C_1, Q_2 \rightarrow C_2, Q_3 \rightarrow C_3\}$) form a grid of thread blocks, i.e., a CUDA kernel.

Limitation: GPU memory capacity. Existing GPU-based vector query systems [23] load the entire dataset in the GPU. Today's large vector datasets [24, 25] contain billions of vectors, which have a memory footprint of hundreds of GB. GPU memory is too limited compared to large datasets. For example, NVIDIA A100 GPUs are equipped with up to 80GB of memory, while the size of the TEXT1B dataset [24] is 750GB. It requires at least ten NVIDIA A100 GPUs to store the TEXT1B dataset. Using multiple GPUs causes a mismatch between GPU compute resources and GPU memory resources. In particular, as we have explained in §2.1 and Figure 1, ANN search only performs vector operations on a small subset of the dataset, which exacerbates this mismatch and causes low utilization of expensive GPU resources.

The natural idea is to expand GPU memory with host memory. For example, the dataset is stored in host memory and is divided into multiple parts, each of which can fit GPU memory. To process a query, each part is transmitted from host memory to GPU memory *in rotation*. After iterating over all parts, the intermediate results of each part are aggregated to produce the final result. This solution has a long transmission time, and because it performs transmission and computation sequentially, it has low GPU utilization.

Another strawman solution is to integrate existing GPU-based query systems with CUDA unified memory [26]. CUDA unified memory automatically handles data transmission at runtime, and performs parallel transmission and computation to fully utilize the GPU copy engine and kernel engine. However, it is a general memory swapping technique that is unaware of vector queries. It incurs massive GPU memory page faults for data transmission. Besides, the overlapping between transmission and computation is also limited as it is agnostic of vector queries. Therefore, applying CUDA unified memory introduces a large gap from the ideal performance.

2.3 Challenges

Opportunity: pipelining transmission and computation. We can pipeline transmission and computation to increase GPU utilization and improve performance. Specifically, we divide the query processing into groups, e.g., $\{G_1, G_2, G_3\}$ in Figure 1. The transmission of one group can be pipelined with the computation of the preceding group. Figure 2(a) shows how to pipeline the three groups in Figure 1, where PCIe represents data transmission from host to GPU memory over PCIe, and GPU represents computation in GPU. The transmission of G_2 is pipelined with the computation of the

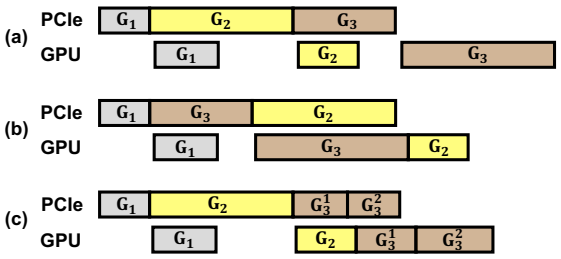


Figure 2: Examples for the pipelining plans (ignore data reuse).

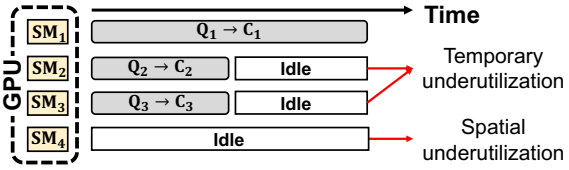


Figure 3: Spatial and temporal GPU underutilization.

preceding group G_1, partly hiding the transmission overhead. We summarize three challenges to exploit this opportunity and achieve optimal pipelining efficiency, which focus on three system aspects: transmission, computation, and pipelining.

Challenge 1: cross-query redundant transmission. The first challenge is that processing a batch of vector queries introduces redundant transmission. As Figure 1 shows, if the GPU memory capacity equals three clusters and the execution order is $[G_1, G_2, G_3]$. Initially, the GPU is empty. The system transmits $G_1 = \{C_1, C_2, C_3\}$ to the GPU, which fills up the GPU memory. Next, to transmit $\{C_4, C_5, C_6\}$ for G_2, it first evicts $\{C_1, C_2, C_3\}$ and then starts the transmission. As for G_3, it clears the GPU memory again and then transmits $\{C_1, C_2, C_3\}$ back to GPU memory. In total, the order $[G_1, G_2, G_3]$ transmits nine clusters, among which C_1, C_2 and C_3 are transmitted twice each. Alternatively, swapping the order between G_2 and G_3 only needs to transmit six clusters. In this case, the system reuses the data $\{C_1, C_2, C_3\}$ for G_3 after processing G_1. Reordering G_2 and G_3 eliminates the redundant transmission.

Challenge 2: spatial and temporal GPU underutilization. The second challenge is the SMs in the GPU have low utilization due to the load imbalance of thread blocks. The computation of G_1 in Figure 1 corresponds to a kernel that contains three thread blocks (i.e., $\{Q_1 \to C_1, Q_2 \to C_2, Q_3 \to C_3\}$). $Q_i \to C_j$ represents the thread block of the computation of query Q_i on cluster C_j. Each thread block is executed within one SM and cannot be migrated to other SMs [50]. As shown in Figure 3, when processing G_1, there are only three thread blocks to run, if the GPU has four SMs, then one SM (i.e., SM_4) is idle, causing *spatial underutilization*. Besides, because the clusters have different sizes, the thread blocks for larger clusters take a longer time to run, i.e., stragglers. As shown in Figure 3, SM_2 and SM_3 are idle after completing their thread blocks, leading to *temporal underutilization* as they wait for SM_1 to finish. In summary, spatial underutilization stems from a mismatch between the number of SMs and thread blocks, while temporal underutilization is caused by cluster heterogeneity.

Challenge 3: transmission and computation overlapping. The third challenge is to maximize the overlapping between transmission and computation in the pipeline. The overlapping is affected by both the ordering and granularity of the groups in the pipeline. We illustrate each of them in Figure 2, and we do not consider data reuse in this example. Figure 2(a) shows a primitive pipelining plan based on the execution order $[G_1, G_2, G_3]$. First, as shown in Figure 2(b), if we reorder G_2 and G_3, then the transmission of G_2 is completely hidden by the computation of G_3, which increases the overlapping. Second, as shown in Figure 2(c), dividing G_3 into two smaller groups, G_3^1 and G_3^2, also improves the overlapping, because finer-grained grouping can hide more transmission operations with computation operations. However, finer-grained grouping also introduces more system overhead due to frequent API invocations and synchronizations.

The challenge is exacerbated by the runtime nature of the problem. The pipelining plan depends on the input vector queries, as different queries process different clusters, which affects the ordering and grouping decisions of the pipeline. Let N be the number of thread blocks. The search space of exhaustive search contains $O(N! \times 2^N)$ choices, as there are $O(N!)$ ordering cases and $O(2^N)$ grouping cases. It is challenging to find the best plan among these choices. In addition, the system also needs an accurate *profiler* to precisely estimate the transmission and computation time for a given group at runtime.

3 RUMMY Overview

We present RUMMY, a GPU-accelerated vector query processing system to support large vector datasets beyond GPU memory. RUMMY exploits the characteristics of vector queries to achieve fast and cost-effective query processing. RUMMY achieves so by pipelining transmission and computation through a novel *reordered pipelining* technique. This technique eliminates redundant data transmission (§4.1), maximizes GPU utilization (§4.2), and finds out optimal pipelining plans with negligible runtime overhead (§4.3). Here we provide a brief overview of RUMMY as Figure 4 shows. RUMMY consists of an offline part and an online part.

Offline. RUMMY first builds a primitive IVF index for the dataset, which can be extended into many variants of IVF. IVF divides the vectors into a few clusters. RUMMY extends the index building with *cluster balancing* to alleviate the straggler problem (§4.2). The dataset is stored in host memory and is transmitted to the GPU memory over PCIe online under the control of the GPU *memory management system* (§4.4). RUMMY also builds a *profiler* offline to measure the computation and transmission time (§4.3.1) for online decision making.

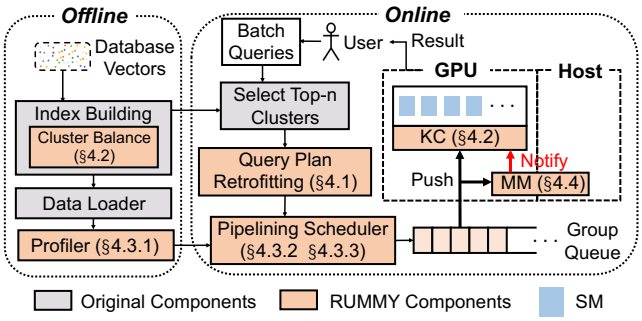


Figure 4: RUMMY overview.

Symbol	Description
C_i	The i_{th} cluster
Q_i	The i_{th} query vector
G_i	The i_{th} group
$\{C_1, C_2 ...\}$	A set of original clusters
$[C_1, C_2 ...]$	The execution order of original clusters
$\{B_1, B_2 ...\}$	A set of balanced clusters
$[B_1, B_2 ...]$	The execution order of balanced clusters
ρ	The fixed size of balanced clusters (the number of vectors)
$T(G)$	The transmission (time) of group G
$E(G)$	The computation (time) of group G
$Q_i \rightarrow B_j$	The computation (thread block) of Q_i on B_j

Table 1: Key notations in the design.

Online. At runtime, RUMMY processes vector queries that arrive at the system. RUMMY has the following three major components in the query runtime.

Query plan. After the top-n closest clusters of each query are determined through IVF, RUMMY retrofits the query plan (i.e., queries to clusters as shown in Figure 1) of each query, which eliminates the redundant data transmission (§4.1).

Pipelining scheduler. The runtime *pipelining scheduler* receives the retrofitted query plan. Based on the prediction time provided by the *profiler*, the scheduler reorders the plan with a greedy algorithm, which finds the optimal order that minimizes the total time on per-cluster granularity (§4.3.2). After the order is determined, the scheduler groups the query plan with a dynamic programming algorithm, which finds the best tradeoff between pipelining efficiency and pipelining overhead (§4.3.3). Each group is pushed into a global *group queue* for transmission and computation.

GPU Runtime. The GPU runtime of RUMMY consists of two components: *kernel controller (KC)* and GPU *memory management system (MM)*. The two components maintain two local group queues respectively. They pull tasks from the global group queue. MM starts transmission immediately as long as its local group queue is not empty (§4.4). When finishing the transmission of a group G_i (i.e., T(G_i)), it notifies KC to set the computation of G_i (i.e., E(G_i)) executable and pops the next task for transmission. KC pops the task $E(G_i)$ and executes it if the kernel engine is idle and $E(G_i)$ is executable. Besides, when launching $E(G_i)$, KC will estimate whether $E(G_i)$ is able to saturate all SMs in the GPU. If not, KC dynamically pads the kernel with more thread blocks to utilize idle SMs (§4.2). After finishing the computation of all groups in a batch, KC returns the final result to the user.

4 RUMMY Design

In this section, we present the design of RUMMY, i.e., *reordered pipelining* that includes three main techniques. The first is *cluster-based retrofitting* to eliminate redundant transmission (§4.1). The second is *dynamic kernel padding* with *cluster balancing* to maximize GPU utilization (§4.2). The third is *query-aware reordering and grouping* to overlap transmission and computation (§4.3). Besides, we describe how to tailor GPU memory management for vector query processing (§4.4) based on IVF. The key notations are listed in Table 1.

4.1 Cluster-based Retrofitting

The key observation behind our idea, namely cluster-based retrofitting, is that the processing of each subset is independent of vector query processing. This observation allows us to retrofit the query plan of each query without changing its correctness. We retrofit the query plan from two aspects, *intra-batch* and *inter-batch*, to completely eliminate the cross-query redundant transmission.

Intra-batch. Figure 1 shows an original vector query plan. The order, $[G_2, G_1, G_3]$, eliminates the three redundant transmission (§2.3). A strawman approach might attempt to find the optimal order by enumerating all possibilities. It is impractical at runtime due to the problem's combinatorial nature.

RUMMY's solution, namely cluster-based retrofitting, is based on the key insight: the optimal number of cluster transmissions is no smaller than the number of involved clusters. In Figure 1, six clusters $\{C_1, C_2 ... C_6\}$ necessitate a minimum of six transmissions, where each cluster is transmitted at least once. As such, our goal is to transmit each involved cluster only once. Figure 5(a) depicts a retrofitted query plan based on Figure 1. We represent this process by a matrix M. $M[i, j] = 1$ means that C_j is involved in the i-th query Q_i, otherwise $M[i, j] = 0$. When transmitting a cluster C_j, the retrofitted plan instantly processes the corresponding queries $\{Q_i ...\}$ for $i \in \{1, 2 ... BS\}$ and $M[i, j] = 1$, with BS as the batch size. For instance, it processes Q_1 and Q_2 immediately after transmitting C_1. In this way, cluster-based retrofitting completely eliminates the redundant transmission. Besides, the retrofitting only traverses the original query plan *once*, resulting in a negligible linear time complexity.

Inter-batch. In intra-batch retrofitting, the cluster order is irrelevant, and every permutation of $[C_1, C_2 ... C_6]$ transmits six clusters. Conversely, inter-batch considers order. Let GPU memory accommodate two clusters C_5 and C_6 initially. Then it additionally eliminates two redundant transmissions

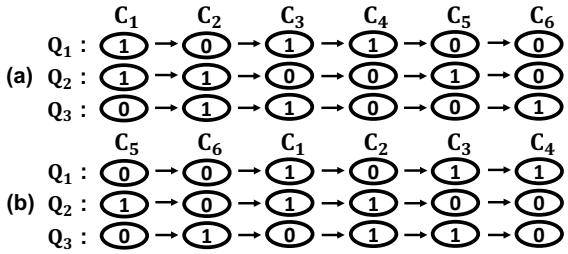


Figure 5: Retrofit the vector query plan.

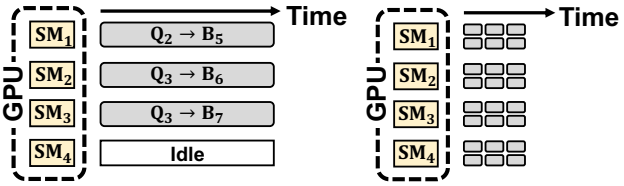
by moving C_5 and C_6 to the front, i.e., $[C_5, C_6, C_1 ..., C_4]$ as shown in Figure 5(b). To maximize data reuse from preceding batches, RUMMY prioritizes moving the clusters already present in the GPU to the forefront of the execution order. Moreover, this mechanism works together with the *reordering* algorithm, which will be described in §4.3.2.

4.2 Dynamic Kernel Padding with Cluster Balancing

Cluster balancing. The straggler causes temporal GPU underutilization by delaying the entire kernel. To address this issue, we design a cluster balancing technique that equilibrates the sizes of various clusters *offline*. Specifically, a cluster C_i is divided into a set of balanced clusters $\{B_{i_1}, B_{i_2} ...\}$. Each balanced cluster B_j is in equal size, denoted by ρ. The detailed balancing algorithm is summarized in Appendix A.3. We emphasize that this technique balances the clusters in *host memory* offline, and the balanced cluster is transmitted into the GPU at runtime to answer the queries. After eliminating the discrepancy between different clusters, the straggler thread block is split into multiple balanced blocks, which maximizes temporal GPU utilization. For instance, suppose that the group G consists of two clusters, C_5 and C_6, in Figure 5(b). C_5 is split into $\{B_5\}$, and C_6 is split into $\{B_6, B_7\}$. The computation of G comprises three balanced thread blocks: $\{Q_2 \to B_5, Q_3 \to B_6, Q_3 \to B_7\}$. Each thread block completes in the same time as illustrated by Figure 6(a). However, the spatial GPU underutilization still exists (SM_4 is always idle).

Dynamic kernel padding. Runtime kernel padding is a technique to address spatial GPU underutilization. An intuitive solution is to process different groups simultaneously through GPU space sharing and provision idle SMs to the subsequent group of computation. This principle resembles the kernel padding technique [51] in DNN schedulers, which shares the GPU with various jobs. However, it is infeasible in the context of pipelining, as the subsequent group remains unprepared until its transmission is finalized. Two groups cannot be executed simultaneously. We design a dynamic kernel padding technique to pad one individual group of kernel execution.

The thread block, $Q_i \to B_j$, receives Q_i and a pointer P_j to B_j as inputs and returns the top-k nearest vectors as the intermediate result. The inputs and outputs are both resident in GPU memory. Suppose the size of each balanced cluster is ρ and the dimension of each vector is d. The balanced cluster is divided into two parts with two pointers: P_j and $P_j + \rho \times d/2$. Thus, $Q_i \to B_j$ is divided into two smaller thread blocks, namely kernel padding. The split number is decided by *Kernel Controller* according to the number of SMs and the query plan to achieve 100% SM utilization and high GPU occupancy [52]. As shown in Figure 6(b), the split number is 8 and the number of padded blocks is 24. Every SM is utilized to achieve 100% spatial utilization. As for GPU occupancy [52], two blocks run on each SM concurrently to further improve the computation efficiency. Since dynamic kernel padding only operates the data pointers, its runtime overhead is negligible.



(a) Cluster balancing. (b) Dynamic kernel padding.

Figure 6: Dynamic kernel padding with cluster balancing.

A natural question is why not solve the straggler problem through kernel padding online only, given that straggler blocks can easily be split at runtime. The reason is that computing the split strategy incurs extra runtime overhead, as different clusters need different split numbers. Thus, our design shifts some work offline to simplify the online decision.

4.3 Pipelining Scheduler

As we describe in §2.3, it is impossible to enumerate every pipelining plan to find out the optimal solution at runtime. We break the entire problem into two individual tractable subproblems and design *query-aware reordering* and *grouping* algorithms that reorder the retrofitted query plan and divide the plan into groups. The algorithms compute optimal solutions for each sub-problem, and the combination of them achieves near-optimal results empirically (§6).

4.3.1 Profiler

Transmission profiler. The goal of the transmission profiler is to measure $T(G_i)$ for a given group G_i. The transmission time from host memory to GPU memory can be divided into two components: the propagation time and the overhead of API invocation. Let G_i consist of m balanced clusters. The propagation time is directly proportional to m. The overhead of API invocation can be estimated as a fixed value. Therefore, $T(G_i)$ is estimated by the following formula: $T(G_i) = a \times \rho \times m + b$, where a and b are parameters fitted offline, and ρ is the size of balanced clusters (§4.2). Querying the transmission profiler for $T(G_i)$ only costs constant time.

Computation profiler. The goal of the computation profiler is to measure $E(G_i)$. We define the computation quantity as $Com = \rho \times SIZE(G_i)$, where ρ is the size of balanced clusters and $SIZE(G_i)$ is the number of elements, whose value is 1, in the retrofitted and balanced query plan of G_i (i.e., the matrix

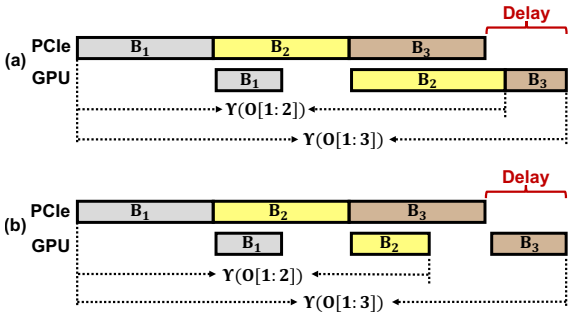


Figure 7: Two cases of the recurrence formula in formulation.

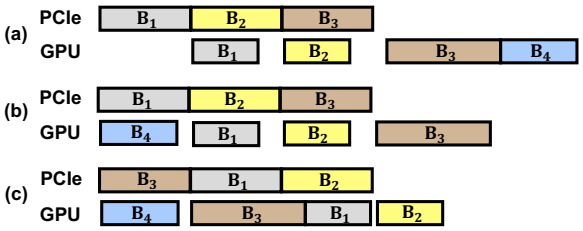


Figure 8: Examples for the reordering algorithm.

in Figure 5 with balanced clusters). Apparently, the degree of parallelism (i.e., the number of thread blocks) and *Com* dominate the computation time. Therefore, we map *Com* with different block numbers to computation time offline. For the intermediate point between the map keys, we use percentile-wise linear interpolation to estimate its value. Similarly to the transmission profiler, querying the unordered map of the computation profiler online also only costs constant time.

4.3.2 Reordering

Problem formulation. The first sub-problem is to reorder the clusters to maximize the overlapping between transmission and computation (i.e., minimize the total time). Let the set of the involved balanced clusters be $\{B_1, B_2 ... B_m\}$, and $[B_1, B_2 ... B_m]$ represents an execution order. $[o_1, o_2 ... o_m]$ is a permutation of $[1, 2 ... m]$. Due to cluster balancing, each B_i is in equal size and introduces the same $T(B_i) = \eta$, unless B_i is already in the GPU from the preceding batch (i.e., $T(B_i) = 0$). The order $[B_{o_1}, B_{o_2} ... B_{o_m}]$ is represented as O, and $O[i:j]$ is a slice of O and is also a new order. $\Upsilon(O)$ is the total time based on the order O, where $O = [B_{o_1}, B_{o_2} ... B_{o_m}]$.

Based on these definitions, we have the following recurrence formula to calculate $\Upsilon(O)$:

$$\Upsilon(O[1:i]) = max(\ \Upsilon(O[1:i-1]) + E(B_{o_i}),\ \sum_{k=1}^{i} T(B_{o_k}) + E(B_{o_i})\ ). \quad (1)$$

The entire computation of an arbitrary order is always later than its transmission, which introduces a *delay*. Figure 7 shows the delay of three clusters (i.e., $i = 3$). The first formula in the *max* function introduces that $T(B_{o_i})$ (i.e., the transmission of the last cluster in $O[1:i]$) is not able to compensate for the delay of $O[1:i-1]$ in Figure 7(a). Figure 7(b) demonstrates $T(B_{o_i})$ is able to compensate for the delay, causing the second formula in the *max* function. Consequently, the maximal case is the final result, i.e., $\Upsilon(O[1:i])$.

In summary, the reordering problem is formulated as the following optimization problem, where the objective is to minimize the total time $\Upsilon(O)$ based on Formula 1.

$$\begin{aligned} &Min. \quad \Upsilon(O). \\ &s.t. \quad O = [B_{o_1}, B_{o_2} ... B_{o_m}], \\ &\qquad [o_1, o_2 ... o_m] = permutation([1, 2 ... m]). \end{aligned} \quad (2)$$

To solve this optimization problem, one can use a search algorithm with pruning or leverage existing solvers for optimization problems. However, such solutions are complex and infeasible due to the vast search space ($m!$ possibilities), the recursive nature of the objective function, and the discrete constraint (a permutation array). Designing an algorithm capable of rapidly finding the optimal order during runtime poses a substantial challenge.

Insights. We leverage two insights to address the challenge. As described in the problem formulation, the transmission time of each balanced cluster is either a fixed value, η, or zero. However, the computation time of each balanced cluster $E(B_i)$ varies since each cluster relates to a different number of queries. Such characteristics provide an opportunity to simplify the problem with two *insights*. To demonstrate the insights, we give an example of four balanced clusters in Figure 8(a), and the execution order is $[B_1, B_2 ... B_4]$ where B_4 is already in the GPU (i.e., $T(B_4) = 0$).

The first insight is that moving a cluster with a transmission time of zero to the front increases the overlapping. In the example of Figure 8, $T(B_4) = 0$. When moving B_4 to the front, i.e., $[B_4, B_1, B_2, B_3]$, the overlapping increases as $T(B_1)$ is hidden by $E(B_4)$ as shown in Figure 8(b).

The second insight is that moving a cluster with a large computation time forward increases the overlapping. In the example of Figure 8, B_3 has the largest computation time. Moving B_3 to the second, i.e., $[B_4, B_3, B_1, B_2]$, further increases the overlapping as shown in Figure 8(c). The intuition is that $E(B_3)$ compensates the non-overlapping area between $E(B_i)$ and $T(B_i)$ for $i \in \{1, 2\}$.

Algorithm. Based on the two insights, we design a greedy algorithm that reorders the clusters to maximize the overlapping and minimize the total time, and we prove that the greedy algorithm is optimal. Algorithm 1 shows the pseudo-code. The function *FindOptOrder* finds the optimal execution order based on the greedy policy: moving the balanced clusters with zero transmission time and large computation time

Algorithm 1 Optimal greedy reordering algorithm

```
1:  function FINDOPTORDER({B_1, ..., B_m})
2:      opt_order ← ∅
3:      for i = 1 → m do
4:          if T[B_i] == 0 then
5:              B_i.priority ← +∞
6:          else
7:              B_i.priority ← E[B_i]
8:          opt_order.append(B_i)
9:      Sort opt_order in descending order based on B_i.priority
10:     return opt_order
```

to the front. It takes the m balanced clusters. It uses *opt_order* to store the execution order and initializes this variable with none (line 2). It iterates over the balanced clusters, and assigns a priority to each cluster based on the greedy policy (line 3-8). It then sorts *opt_order* in descending order based on the priority of each cluster (line 9). Given m balanced clusters, the critical path is the sort operation. Therefore, the time complexity of Algorithm 1 is $O(m \log m)$, which enables RUMMY to quickly find the optimal order at runtime. We have the following theorem for the reordering algorithm.

Theorem 4.1 *Algorithm 1 finds the optimal execution order of balanced clusters that minimizes the total time for the pipelining on per-cluster granularity.*

The main idea of the proof is to show that any transformation of the optimal execution order does not decrease the total time. The proof is in Appendix A.1. Since the reordering algorithm moves the clusters with zero transmission time to the front, it solves the inter-batch retrofitting problem described in §4.1.

4.3.3 Grouping

The second sub-problem is to group the balanced clusters after the order is determined. The basic way for pipelining is to pipeline on per-cluster granularity, i.e., transmitting and computes cluster by cluster. However, such fine-grain pipelining introduces two sources of pipelining overheads: the overhead of the frequent invocations and the synchronization overhead between each group of transmission and computation. Another way for pipelining is to group all clusters together. It eliminates the pipelining overhead, but there would be no overlapping between transmission and computation at all.

We employ a dynamic programming algorithm, similar to previous works [29, 30], to find the optimal grouping plan in the context of pipelining. To further reduce the search space, we use a heuristic pruning method, maintaining a global variable to record the best time in the current search space. During the search tree traversal, the lower bound of a node's subtree is calculated by completely overlapping leftover transmission and computation while ignoring the pipelining overhead. If such lower bound is greater than the best time, the subtree is pruned. The time complexity of the dynamic programming algorithm is polynomial, and the algorithm with the pruning heuristic enables RUMMY to quickly find the optimal grouping plan at runtime.

4.4 GPU Memory Management

Existing GPU-based vector query processing systems lack support for runtime GPU memory management and load the entire dataset into the GPU offline. RUMMY provides runtime GPU memory management tailored for vector query processing. It leverages GPU's native interfaces, like `cudaMalloc` and `cudaFree` in NVIDIA GPUs, to pre-allocate the entire GPU memory to RUMMY at system startup and manage the memory internally. This avoids frequent invocations to GPU's native interfaces. RUMMY handles data transmission tasks from its local task queue (§3), and notifies the kernel controller upon completion. Below, we describe the GPU memory layout and page replacement policy to reconcile vector query processing with the limited GPU memory capacity. RUMMY pins host memory to further reduce the transmission time.

GPU memory layout. The entire GPU memory is treated as a consecutive memory space in RUMMY. RUMMY's memory layout is able to eradicate both internal memory fragmentation and external memory fragmentation. RUMMY allocates GPU memory on page granularity for transmission tasks, enabling clusters to be stored in discontinuous space. RUMMY returns several discontinuous pages for the allocation of a transmission task. Thus, each free memory fragment (i.e., a free page) can be allocated for any task, which minimizes the external fragmentation issue. Besides, a large page size introduces internal memory fragmentation while a small page size introduces extra overhead of massive paging operations. Due to cluster balancing (§4.2), the cluster size is fixed. Setting the page size to the cluster size ensures that each page is fully utilized and the overhead of paging is minimized, thereby easily solving internal fragmentation issues.

GPU page replacement policy. When the GPU memory is full, RUMMY has to evict a page (balanced cluster) for the subsequent allocation. RUMMY's GPU page replacement policy accounts for both intra-batch and inter-batch to minimize the miss rate. For intra-batch, RUMMY traverses the local task queue of the current batch, and pins the intersection clusters between the future involvement in the current batch and those already in the GPU, without evicting pinned clusters. As for inter-batch, we observe that certain clusters (hot) are referred to frequently, while others (cold) are occasionally referred to. RUMMY records the referred count of each cluster. At runtime, RUMMY evicts the cluster with the smallest counter if it is not pinned, which is akin to LFU [53]. There are some canonical page replacement policies, e.g., LRU. However, they are not well-suited to vector query processing. Specifically, LRU evicts the cluster that is least recently used. A vector query with a large batch size uses every cluster. LRU may replace the hot clusters, those accessed early in the batch, with cold clusters, those accessed late in the batch. This leads to a high

miss rate in the subsequent batch that predominantly relies on hot clusters (i.e., small batch size).

Host memory pinning. The operating system swaps host memory pages to the disk if this page is inactive for a certain time. This mechanism causes severe page faults during GPU runtime data transmission, as GPU can't directly transfer pages from the disk. To mitigate the problem, RUMMY uses page-locked host memory (i.e., pin memory) to store the clusters through `cudaMallocHost` in NVIDIA GPUs. The pinned memory, accessible directly by GPU, allows read/write operations with higher bandwidth compared to pageable memory allocated by standard `malloc`.

5 Implementation

We implement a system prototype of RUMMY with 12K lines of codes in CUDA and C++, and integrate it with Faiss [23], a state-of-the-art vector query processing system adopted by many vector databases, like Milvus [16], Zilliz [17] and AnalyticDB-V [19]. RUMMY can be integrated with any vector query system. We choose Faiss as it is the most widely used GPU-based vector query processing system, and is adopted in production like Meta. RUMMY is built on top of IVF index rather than graph index, because the former is proved to be more efficient on billion-scale datasets [46] with limited memory capacity. Moreover, graph index (e.g., HNSW) cannot be easily integrated into GPU due to its random access pattern. The code of RUMMY is open-source and is publicly available at https://github.com/pkusys/Rummy.

Kernel controller. The kernel controller executes GPU kernels. It learns the GPU hardware information (e.g., the number of SMs) by reading CUDA macros. We extend CUDA kernels in Faiss to dynamically pad the thread blocks according to the split number. A larger split number means a larger block number to saturate the GPU (SMs).

Memory management. RUMMY divides GPU memory into two areas: auxiliary and primary memory. The auxiliary memory stores the query vectors, the query plan, and the intermediate results. The primary memory holds the clusters of vector datasets. The auxiliary memory is managed as a stack, which orchestrates memory for temporary data created and deleted in order during query processing. The primary memory is organized as a heap to cache the clusters. Each page in the heap is either free or allocated to a cluster. RUMMY uses an AVL tree to manage these pages to quickly retrieve an allocated page or allocate a free page.

System optimizations. Faiss spawns one CUDA stream to process a vector query and synchronizes all operations by default. In contrast, RUMMY uses three CUDA streams: one for launching CUDA kernel, another for GPU-to-host memory transmission, and the third for host-to-GPU memory transmission. This trio of streams allows RUMMY to parallelize computation and two types of transmission in the pipeline.

Dataset	Dimensions	Database Vectors	Query Vectors	Distance	Memory Footprint
SIFT1B [25]	128	1B	10K	Euclidean	480 GB
DEEP1B [24]	96	1B	10K	Euclidean	361 GB
TEXT1B [24]	200	1B	100K	Angular	748 GB
SIFT40M [25]	128	40M	10K	Euclidean	31 GB
DEEP50M [24]	96	50M	10K	Euclidean	29 GB
TEXT30M [24]	200	30M	100K	Angular	32 GB
SIFT10M [25]	128	10M	10K	Euclidean	9 GB

Table 2: Datasets used in the evaluation.

RUMMY wraps up each stream with a host thread to simplify function invocations and metadata (e.g., AVL tree) changes. RUMMY groups concurrent requests into one batch. The concurrent requests (i.e., one batch) are primarily processed on the GPU. The host thread is responsible for generating the query plan and directing the GPU to process the batch.

6 Evaluation

In this section, we first use end-to-end experiments to demonstrate the overall performance improvements of RUMMY over existing GPU-based and CPU-based solutions on billion-scale datasets. Next, we use microbenchmarks to deep dive into RUMMY and show the effectiveness of each component in RUMMY under a variety of settings. As we discuss in §2.2, IVF is more suitable for billion-scale datasets than graph index schemes (e.g., HNSW). Thus, we focus on the IVF index in the evaluation. The baselines are also implemented based on Faiss.

6.1 End-to-End Experiments

Setup. All experiments are conducted on AWS. The end-to-end experiments use two types of AWS EC2 instances. One is p4d.24xlarge configured with eight NVIDIA A100 GPUs with 40 GB GPU memory each, 1152 GB host memory and PCIe 4.0×16. The GPU instance is used to compare RUMMY with existing GPU-based solutions (§2.2). While we only use *one* GPU, we use p4d.24xlarge because it is the only type of instance on AWS that has high-end GPU, A100. The other is x1.16xlarge configured with 64 vCPUs (Intel Xeon E7-8880) and 976 GB of host memory. The CPU instance is used to compare RUMMY with the existing CPU-based solution.

Workloads. Table 2 summarizes the benchmarking datasets, and the top three are used in end-to-end experiments. The datasets include SIFT1B [25], DEEP1B [24], and TEXT1B [24]. They consist of one billion database vectors and some query vectors. These datasets are standard benchmarks for vector database retrieval used by both academic and industry. They can also be integrated with LLMs to support retrieval augmented generation, e.g., TEXT1B (a cross-model dataset) can be used to enhance multimodal LLMs. The memory footprint of the 1B datasets is in the hundreds of GB range, which *oversubscribes* the 40 GB GPU memory, but these datasets can be held in the host memory. Query vectors are divided into two batch sizes, 2048 (large) and 8 (small), representing offline and online workloads, respectively.

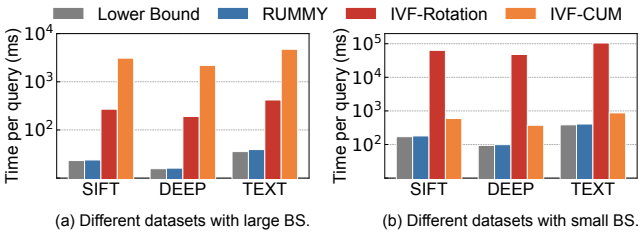


Figure 9: Overall performance on GPUs.

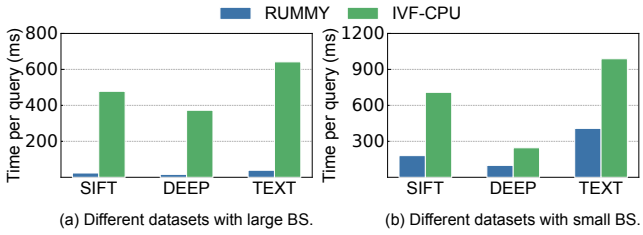


Figure 10: Comparison between RUMMY and IVF-CPU.

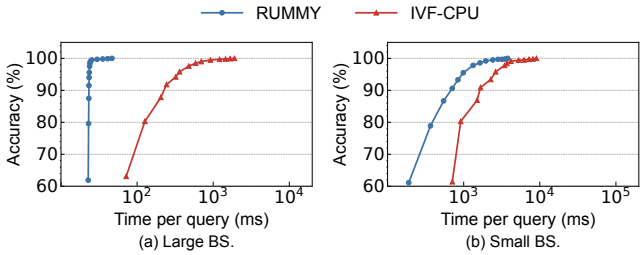


Figure 11: Comparison under different accuracy.

Metrics. We use the average end-to-end time per query as the main metric. In addition, when comparing GPU-based solutions to CPU-based solutions, we also use per-dollar performance, to evaluate the cost-effectiveness. Specifically, per-dollar performance is calculated as $\frac{Price}{Time\ per\ Query}$, where *Price* is the AWS EC2 On-Demand hourly rate [54].

Baselines. IVF is a state-of-the-art ANN index and has been widely deployed in production for billion-scale datasets. It is supported on both GPUs and CPUs in Faiss [23]. Its GPU version requires the dataset to be fully loaded into GPU memory. There are *no* existing GPU-based systems that support large datasets beyond GPU memory. We extend the GPU version of IVF to implement two baseline systems to compare with RUMMY. Specifically, we compare RUMMY to the following three baselines while keeping the same searching parameters under two different batch sizes.

- **IVF-Rotation.** It extends the GPU version of IVF with the strawman rotation method described in §2.2 to support large datasets beyond GPU memory.
- **IVF-CUM.** It extends the GPU version of IVF with CUDA unified memory [26] to expand GPU with host memory.
- **IVF-CPU.** It is the CPU version of IVF, which exploits Intel AVX [55] to speed up vector operations.

Overall performance. We first compare the time per query of RUMMY to the GPU-based baselines, IVF-Rotation and IVF-CUM, and keep the same searching parameters. The experiments are conducted on p4d.24xlarge with three billion-scale datasets. We only use one GPU in p4d.24xlarge since one GPU is enough for RUMMY and p4d.24xlarge is the only instance configured with A100 GPUs on AWS. RUMMY outperforms the baselines by hundreds of times and achieves near-optimal performance with *reordered pipelining*. The results are shown in Figure 9. We summarize as follows.

- The lower bound is calculated by the larger value of transmission and computation time per query. It represents an ideal case that has the maximum overlapping between transmission and computation without any pipelining overhead. Figure 9 shows that RUMMY processes a query within *a few milliseconds* and is close to the lower bound, i.e., it achieves near-optimal performance.
- RUMMY outperforms IVF-Rotation by 10.7-11.7× under large BS and hundreds of times under small BS. The performance gap is larger on small BS, as IVF-Rotation iterates over all data parts and introduces a fixed transmission time under any batch sizes. When confronting small BS, the query time is dominated by the fixed transmission time, which introduces a large performance gap between RUMMY and IVF-Rotation.
- RUMMY outperforms IVF-CUM by 121-135× under large BS and by 2.1-3.7× under small BS. RUMMY outperforms IVF-CUM by a large margin under large BS, since large BS introduces heavier computation and more interference between GPU SMs. This interference causes severe GPU memory page faults.

Comparison to CPU. We then compare RUMMY and IVF-CPU. The results are shown in Figure 10. RUMMY (with one A100 GPU) achieves 2.4-23.1× higher performance than IVF-CPU (with 64 vCPUs). Besides time per query, RUMMY also achieves better per-dollar performance (26.7-37.7× under large BS), i.e., RUMMY is more cost-effective. As the batch size grows, the performance gap becomes larger. This is because large BS introduces high computation parallelism for RUMMY with GPUs but high computation interference for CPU solutions. It is notable that the performance of two GPU baselines (IVF-CUM and IVF-Rotation) falls short of IVF-CPU. This is because the datasets exceed the GPU memory and existing GPU solutions are not optimized for such scenario (§2.2). This emphasizes the necessity for RUMMY on large datasets.

Moreover, Figure 11 demonstrates the performance improvement of RUMMY over IVF-CPU under different accuracy on SIFT1B. RUMMY constantly outperforms IVF-CPU under any configuration.

6.2 Deep Dive of RUMMY

Setup & workloads. We use two AWS EC2 GPU instances to deep dive into RUMMY and evaluate the effectiveness of each

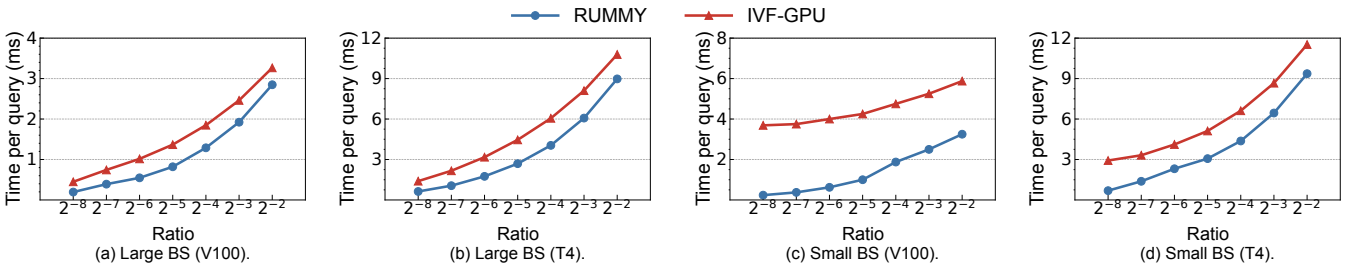


Figure 12: Effectiveness of RUMMY's dynamic kernel padding with cluster balancing.

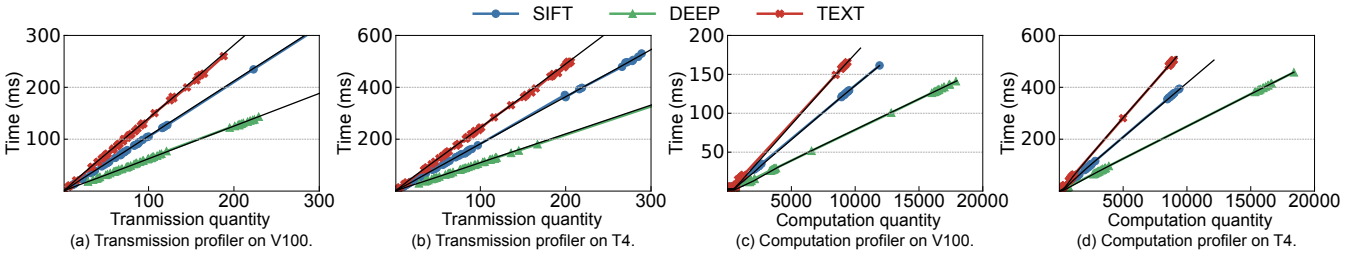


Figure 13: Effectiveness of RUMMY's profiler.

component in RUMMY. One is p3.2xlarge configured with one NVIDIA V100 GPU with 16 GB GPU memory. The other is g4dn.2xlarge configured with one NVIDIA T4 GPU with 16 GB GPU memory. In addition, we reduce large BS to 256 and the scale of the datasets since the computing power and memory of T4 and V100 is not a patch on that of A100. The four micro-datasets are listed at the bottom of Table 2, and the top three all oversubscribe the 16 GB GPU memory. The reason to use small GPUs and datasets is to reduce evaluation costs due to our *limited budget*. Besides, the experiments on million-scale datasets show that RUMMY maintains its effectiveness even with moderately sized datasets.

6.2.1 Dynamic Kernel Padding with Cluster Balancing

We measure the computation performance with SIFT10M, which can be held in GPU memory, and do not consider transmission in this experiment. We run the vector queries under various top-n. The baseline is IVF-GPU without the two techniques. The two both operate the memory pointers. Thus, the negligible overhead of index building and query is not included in the evaluation.

As shown in Figure 12, the x-axis represents the ratio of top-n to the number of all clusters. As the ratio grows, the performance decreases. Meanwhile, large BS introduces high computation density causing high time per query. As Figure 12(c) and Figure 12(d) show, RUMMY outperforms IVF-GPU by up to 15.5× under small BS. The performance improvement is mainly from the dynamic kernel padding since IVF-GPU cannot utilize the SMs under small BS. The performance gap becomes smaller under small BS as illustrated by Figure 12(a) and Figure 12(b). Both RUMMY and IVF-GPU fully utilize the SMs, but RUMMY still has a better performance (1.1-2.3×) due to the cluster balancing that mitigates the straggler block in the entire kernel.

6.2.2 Profiler

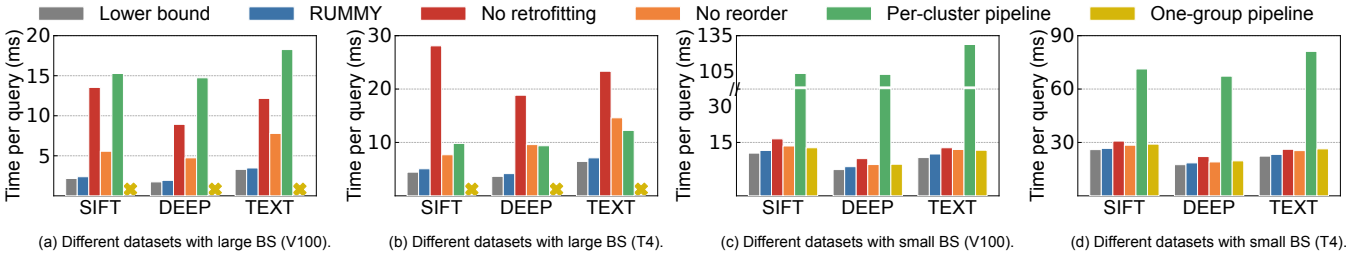
We use SIFT40M, DEEP50M, and TEXT30M to evaluate the effectiveness of RUMMY's profiler, and run the vector queries on V100 and T4 GPUs with two batch sizes. We record the profiler's estimated time of each group and measure the group's actual transmission or computation time. Figure 13 shows the results. The colored lines represent the actual time, and the black lines represent the estimated time. We plot the points of two batch sizes in one line. The x-axis represents the transmission quantity or the computation quantity (defined in §4.3.1). The results confirm that the profiler closely tracks the actual time, thereby demonstrating the effectiveness of the simple yet practical formulas in §4.3.1.

6.2.3 Pipelining Scheduler & Cluster-based Retrofitting

We use SIFT40M, DEEP50M, and TEXT30M in this experiment. We compare the following techniques discussed in §4.1 and §4.3, while keeping other components the same.

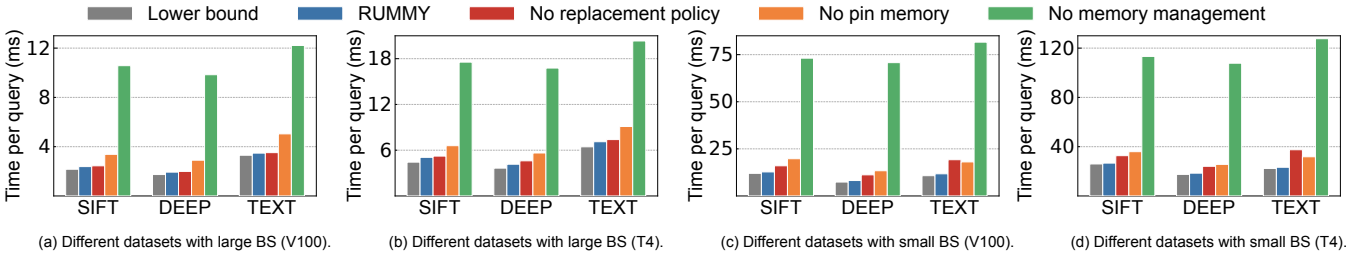
- **Lower bound.** It is the theoretical lower bound of time.
- **RUMMY.** It is RUMMY with all the techniques.
- **No retrofitting.** It does not use retrofitting.
- **No reordering.** It does not use Algorithm 1 for reordering.
- **Per-cluster pipeline.** It groups the clusters on per-cluster granularity and processes the query cluster by cluster.
- **One-group pipeline.** It groups all clusters into one group.

Figure 14 shows the performance of the above techniques under two batch sizes and two GPU instances. RUMMY's cluster-based retrofitting improves the performance by 1.1-5.7× through eliminating the redundant transmission. The reordering algorithm improves the performance by 1.5-2.4× under large BS while only having a slight improvement under small BS. This is because a larger batch size leads to a complex query plan (allowing the reordering algorithm to identify



(a) Different datasets with large BS (V100). (b) Different datasets with large BS (T4). (c) Different datasets with small BS (V100). (d) Different datasets with small BS (T4).

Figure 14: Effectiveness of RUMMY's reordering and grouping with cluster-based retrofitting.



(a) Different datasets with large BS (V100). (b) Different datasets with large BS (T4). (c) Different datasets with small BS (V100). (d) Different datasets with small BS (T4).

Figure 15: Effectiveness of RUMMY's GPU memory management.

Workload	Reorder Overhead	Group Overhead	Total Time
SIFT40M (BS=256, V100)	7.4 μs (0.31%)	10 μs (0.44%)	2.4 ms
SIFT40M (BS=256, T4)	6.3 μs (0.12%)	7.4 μs (0.15%)	5.1 ms
SIFT40M (BS=8, V100)	57 μs (0.45%)	47 μs (0.37%)	12 ms
SIFT40M (BS=8, T4)	55 μs (0.2%)	42 μs (0.16%)	27.3 ms

Table 3: Runtime overhead of reordering and grouping.

a greater scope for optimization), while a smaller batch size only has a few clusters for computation and the reordering algorithm has little room for optimization. The per-cluster and one-group pipelining are two extreme grouping plans, which lead to the maximum pipelining overhead and the minimum overlapping, respectively. RUMMY's grouping plan achieves the optimal tradeoff between them (1.1-13.8× performance improvement). The one-group pipelining fails to process vector queries under large BS since it requires a large amount of data transmission for one group, and thereby oversubscribing the GPU memory. The lower bound is the theoretical lowest time per query. RUMMY achieves the closest performance to the lower bound with the aforementioned techniques.

The runtime scheduling overheads of the reordering and grouping are listed in Table 3 on SIFT40M. The overhead per query is within tens of microseconds, which is relatively small compared to the total time per query ($\leq 0.5\%$).

6.2.4 Memory Management

The setup is the same as §6.2.3. We do not compare RUMMY's memory management to CUDA unified memory, since the comparison is already included in §6.1. We compare the following memory management mechanisms discussed in §4.3, while keeping other components the same.

- **Lower bound.** The same with the lower bound in §6.2.3.
- **RUMMY.** It is RUMMY with all the techniques.
- **No memory management.** It allocates and frees GPU memory at runtime through `cudaMalloc` and `cudaFree`.
- **No replacement policy.** It evicts a random page (cluster).
- **No pin memory.** It uses pageable host memory.

As shown in Figure 15, the replacement policy improves the performance by 1.2-1.6× under small BS while having little effect under large BS. This is because large BS transmits most of the clusters, including cold clusters. The differentiation between cold and hot clusters becomes meaningless. As for the pin memory and the memory layout with pre-allocated memory techniques, they improve the performance by 1.3-1.6× and 2.8-8.6×, respectively. This is because the pin memory improves the overall memory transmission bandwidth, and the memory layout reduces memory fragmentation and frequent invocations to CUDA native interfaces. This experiment demonstrates that all memory management techniques of RUMMY are effective and the combination of them achieves the closest performance to the ideal case.

7 Discussion

System, not index. We emphasize that RUMMY is not a new ANN index, but a new vector query processing system (with new techniques on query pipelining) to support billion-scale datasets beyond GPU memory. RUMMY is built on the vector query processing system, Faiss and ANN index, primitive IVF. Many works (e.g., SPANN [46] and Auncel [56]) propose variations of IVF to improve query efficiency. We do not compare these works to RUMMY, as the underlying hardware-backends and index algorithms differ significantly. However, RUMMY is orthogonal to these algorithmic works and can be integrated into these variations of IVF to support GPU acceleration. We do not directly compare RUMMY to vector

databases, since RUMMY focuses on the query processing part.

Vector quantization. Vector quantization [57] is proposed to reduce the memory footprint of large datasets. It compresses high-dimensional vectors into low-dimensional space, thereby reducing the memory usage. RUMMY supports vector queries beyond GPU memory through system techniques and is orthogonal to vector quantization. Besides, vector quantization can be integrated into RUMMY's primitive IVF index to further reduce memory usage and improve query efficiency.

Reordered pipelining. Pipelining, a common technique to enhance computer system performance, is uniquely implemented in RUMMY, which differs from traditional methods. RUMMY leverages the *independent* and *non-deterministic* nature of computational units in vector query processing on IVF. RUMMY's reordered pipelining is also applicable to other domains, such as batch request processing in LLM inference. Specifically, when using host memory to store the key-value tensors, it offers a strategy to decide the group (i.e., batch) and order of requests, which can efficiently parallelize the computation and key-value tensor transmission between host memory and GPU memory.

8 Related Work

A variety of ANN algorithms (e.g., inverted file [41, 45, 46], locality-sensitive hashing [44, 58–60], and graph algorithms [42, 47, 48]) are proposed for vector query processing. IVF and graph index are among the most popular ANN algorithms. They have different system characteristics. There are several reasons why we choose IVF to build RUMMY to support billion-scale datasets beyond GPU memory. First, IVF is proved to be more efficient than graph index on billion-scale datasets with the same memory footprint according to the recent work [46]. This work demonstrates the benefits of IVF and the limitations of graph index, which applies IVF and significantly outperforms the state-of-the-art graph index, DiskANN [48]. Second, graph index maintains a huge graph which introduces 4× memory usage [61] than IVF. As a result, graph index is not suitable for billion-scale datasets with limited GPU memory. Last, graph index (e.g., HNSW) cannot be easily integrated into GPU due to its random access pattern. This paper focuses on IVF-GPU, and we leave the support of different ANN algorithms in RUMMY as future work.

A set of works [56, 62, 63] focuses on parameter tuning to improve the accuracy and query latency. For example, LAET [63] leverages a decision tree model to early terminate a query when it is hard to improve query accuracy. These optimizations can be integrated into RUMMY since RUMMY focuses on system techniques and does not change any index characteristics. ANN algorithms can be accelerated by hardware accelerators, such as GPUs [23] and FPGAs [6]. However, prior solutions are not suitable for large vector datasets and require the datasets to be preloaded into the global memory of the accelerators. We can exploit RUMMY's core idea (i.e., reordered pipelining) to expand the accelerator memory with host memory as well.

With the proliferation of unstructured data and deep learning, ANN algorithms on embedding vectors become a key component (long-term memory retrieval) in many AI applications, such as recommendation systems [4, 5, 21], recognition [7–9, 22], information retrieval [10–12] and LLM-based AI applications [33–35]. RUMMY benefits these applications by accelerating vector query processing and reducing cost. Recent industrial vector databases [16, 17, 19, 64] adopt Faiss [23] as their query engine. RUMMY, based on Faiss, can be integrated into these systems. Big data processing [65–67] is prevalent in cloud services. RUMMY can be integrated into these services with GPUs to manage large unstructured data and accelerate vector query processing.

9 Conclusion

We present RUMMY, the first GPU-based system for billion-scale vector query processing beyond GPU memory. RUMMY expands GPU memory with host memory to achieve high performance and low cost by leveraging a novel *reordered pipelining* technique. We evaluate RUMMY on various benchmarking datasets with billions of items and show that it outperforms GPU-based baselines by up to 135× with near-optimal performance. Our experiments also show that RUMMY is capable of achieving better performance and cost than the state-of-the-art CPU-based solution.

Acknowledgments. We sincerely thank our shepherd Yu Hua and the anonymous reviewers for their valuable feedback on this paper. This work was supported by the National Natural Science Foundation of China under the grant numbers 62325201, 62172008, and the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas). Xin Jin and Xuanzhe Liu are the corresponding authors. Zili Zhang, Fangyue Liu, Gang Huang, Xuanzhe Liu, and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, 2015.

[2] R. Blumberg and S. Atre, "The problem with unstructured data," *Dm Review*, 2003.

[3] "Eighty Percent of Your Data Will Be Unstructured in Five Years," 2023. https://solutionsreview.com/data-management/.

[4] A. S. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: scalable online collaborative filtering," in *WWW*, 2007.

[5] M. Grbovic and H. Cheng, "Real-time personalization using embeddings for search ranking at airbnb," in *ACM SIGKDD*, 2018.

[6] C. Zeng, L. Luo, Q. Ning, Y. Han, Y. Jiang, D. Tang, Z. Wang, K. Chen, and C. Guo, "*FAERY*: An fpga-accelerated embedding-based retrieval system," in *USENIX OSDI*, 2022.

[7] R. He, Y. Cai, T. Tan, and L. Davis, "Learning predictable binary codes for face indexing," *Pattern recognition*, 2015.

[8] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[9] L. Zheng, L. Shen, L. Tian, S. Wang, J. Wang, and Q. Tian, "Scalable person re-identification: A benchmark," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[10] H. Chen, O. Engkvist, Y. Wang, M. Olivecrona, and T. Blaschke, "The rise of deep learning in drug discovery," *Drug discovery today*, 2018.

[11] A. C. Mater and M. L. Coote, "Deep learning in chemistry," *Journal of chemical information and modeling*, 2019.

[12] K. Berlin, S. Koren, C.-S. Chin, J. P. Drake, J. M. Landolin, and A. M. Phillippy, "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing," *Nature biotechnology*, 2015.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017.

[14] "Introducing chatgpt," 2023. https://openai.com/blog/chatgpt.

[15] OpenAI, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[16] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, *et al.*, "Milvus: A purpose-built vector data management system," in *ACM SIGMOD*, 2021.

[17] "Introducing zilliz," 2023. https://zilliz.com/.

[18] "Introducing pinecone," 2023. https://www.pinecone.io/.

[19] C. Wei, B. Wu, S. Wang, R. Lou, C. Zhan, F. Li, and Y. Cai, "Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data," in *Proceedings of the VLDB Endowment*, 2020.

[20] "Introducing qdrant," 2023. https://qdrant.tech/.

[21] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee, "Billion-scale commodity embedding for e-commerce recommendation in alibaba," in *ACM SIGKDD*, 2018.

[22] X. Liu, W. Liu, H. Ma, and H. Fu, "Large-scale vehicle re-identification in urban surveillance videos," in *ICME*, 2016.

[23] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, 2019.

[24] "yandex billion-scale datasets," 2023. https://research.yandex.com/datasets/biganns.

[25] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, "Searching in one billion vectors: re-rank with source coding," in *ICASSP*, 2011.

[26] "CUDA Unified Memory," 2023. https://devblogs.nvidia.com/unified-memory-cuda-beginners/.

[27] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill, "Mips: A microprocessor architecture," *ACM SIGMICRO Newsletter*, 1982.

[28] A. Hartstein and T. R. Puzak, "The optimum pipeline depth for a microprocessor," *ACM Sigarch Computer Architecture News*, 2002.

[29] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin, "Pipeswitch: Fast pipelined context switching for deep learning applications," in *USENIX OSDI*, 2020.

[30] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: generalized pipeline parallelism for DNN training," in *ACM SOSP*, 2019.

[31] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: Nimble task scheduling for serverless analytics," in *USENIX NSDI*, 2021.

[32] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters," in *USENIX OSDI*, 2020.

[33] "Chatgpt plugin: Retrieval," 2023. https://openai.com/blog/chatgpt-plugins#retrieval.

[34] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, and H. Wang, "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, 2023.

[35] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Advances in Neural Information Processing Systems*, 2020.

[36] "Benchmarking nearest neighbors," 2023. https://github.com/erikbern/ann-benchmarks/.

[37] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, "Practical and optimal lsh for angular distance," in *Advances in Neural Information Processing Systems*, 2015.

[38] M. Muja and D. Lowe, "Flann-fast library for approximate nearest neighbors user manual," *VISAPP*, 2009.

[39] "Annoy," 2023. https://github.com/spotify/annoy.

[40] L. Boytsov and B. Naidan, "Engineering efficient and effective non-metric space library," in *SISAP*, 2013.

[41] A. Babenko and V. Lempitsky, "The inverted multi-index," *TPAMI*, 2014.

[42] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *TPAMI*, 2018.

[43] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," *ACM TOG*, 2008.

[44] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*, 2004.

[45] D. Baranchuk, A. Babenko, and Y. Malkov, "Revisiting the inverted indices for billion-scale approximate nearest neighbors," in *ECCV*, 2018.

[46] Q. Chen, B. Zhao, H. Wang, M. Li, C. Liu, Z. Li, M. Yang, and J. Wang, "SPANN: Highly-efficient billion-scale approximate nearest neighbor search," in *Advances in Neural Information Processing Systems*, 2021.

[47] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," in *Proceedings of the VLDB Endowment*, 2019.

[48] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "Diskann: Fast accurate billion-point nearest neighbor search on a single node," in *Advances in Neural Information Processing Systems*, 2019.

[49] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the royal statistical society. series c (applied statistics)*, 1979.

[50] "CUDA Refresher: The CUDA Programming Model," 2023. https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/.

[51] M. Han, H. Zhang, R. Chen, and H. Chen, "Microsecond-scale preemption for concurrent gpu-accelerated dnn inferences," in *USENIX OSDI*, 2022.

[52] "Nvidia Achieved Occupancy," 2023. https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm.

[53] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Transactions on Computers*, 2001.

[54] "Amazon EC2 On-Demand Pricing," 2023. https://aws.amazon.com/ec2/pricing/on-demand/.

[55] "Intel Advanced Vector Extensions," 2023. https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html.

[56] Z. Zhang, C. jin, L. Tang, X. Liu, and X. Jin, "Fast, approximate vector queries on very large unstructured datasets," in *USENIX NSDI*, 2023.

[57] T. Ge, K. He, Q. Ke, and J. Sun, "Optimized product quantization for approximate nearest neighbor search," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2013.

[58] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: efficient indexing for high-dimensional similarity search," in *Proceedings of the VLDB Endowment*, 2007.

[59] Y. Zheng, Q. Guo, A. K. Tung, and S. Wu, "Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single index," in *ACM SIGMOD*, 2016.

[60] L. Gong, H. Wang, M. Ogihara, and J. Xu, "idec: indexable distance estimating codes for approximate nearest

neighbor search," in *Proceedings of the VLDB Endowment*, 2020.

[61] W. Zhao, S. Tan, and P. Li, "Song: Approximate nearest neighbor search on gpu," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020.

[62] A. Gogolou, T. Tsandilas, T. Palpanas, and A. Bezerianos, "Progressive similarity search on time series data," in *BigVis*, 2019.

[63] C. Li, M. Zhang, D. G. Andersen, and Y. He, "Improving approximate nearest neighbor search through learned adaptive early termination," in *ACM SIGMOD*, 2020.

[64] "Pinecone: Introduction to Facebook AI Similarity Search (Faiss)," 2023. https://www.pinecone.io/learn/series/faiss/faiss-tutorial/.

[65] "Google BigQuery," 2023. https://cloud.google.com/bigquery/.

[66] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, *et al.*, "The snowflake elastic data warehouse," in *Proceedings of the 2016 International Conference on Management of Data*, 2016.

[67] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan, "Amazon redshift and the case for simpler data warehouses," in *ACM SIGMOD*, 2015.

A Appendix

A.1 Proof of Theorem§ 4.1

Proof. To further elaborate the problem formulation in §4.3.2, let the final result of Algorithm 1 be $O = [B_{o_1}, B_{o_2} ... B_{o_m}]$, and there are m^* balanced clusters that are already in the GPU (i.e., $T(B_{o_i}) = 0,\ 0 < i \leq m^*$). η is the constant transmission time of the leftover balanced clusters since these clusters have the same size (i.e., $T(B_{o_i}) = \eta,\ m^* < i \leq m$).

Based on Formula 1 and Formula 2, we employ greedy exchange to show that *FindOptOrder* outputs the optimal order. The original order is denoted by O. The exchanged order is denoted by O^*. The general exchange can be transferred into the adjacent exchange. Our target is to prove that $\Upsilon(O) \leq \Upsilon(O^*)$ is always true if we exchange B_{o_i} and $B_{o_{i+1}}$ in O, $1 \leq i \leq m-1$. B_{o_i} has a higher priority than $B_{o_{i+1}}$, and the priority is shown in Algorithm 1. We classify the exchange paradigm into the following three cases.

Case 1: $T(B_{o_i}) = T(B_{o_{i+1}}) = 0$**.** The balanced clusters, B_{o_i} and $B_{o_{i+1}}$, are already in the GPU and their transmission time is both zero. Because of the delay, we have $\Upsilon(O[1:j]) \geq \sum_{k=1}^{j} T(B_{o_k}) \geq \sum_{k=1}^{j-1} T(B_{o_k})$ where $1 \leq j \leq m$. As such,

$$\begin{aligned}
&\Upsilon(O[1:i+1]) = max(\Upsilon(O[1:i]), \sum_{k=1}^{i-1} T(B_{o_k})) + E(B_{o_{i+1}}) \\
&= \Upsilon(O[1:i]) + E(B_{o_{i+1}}) \\
&= max(\Upsilon(O[1:i-1]), \sum_{k=1}^{i-1} T(B_{o_k})) + E(B_{o_i}) + E(B_{o_{i+1}}) \\
&= \Upsilon(O[1:i-1]) + E(B_{o_i}) + E(B_{o_{i+1}}) \qquad (3)
\end{aligned}$$

Similarly, we have:

$$\Upsilon(O^*[1:i+1]) = \Upsilon(O^*[1:i-1]) + E(B_{o_i}) + E(B_{o_{i+1}}) \quad (4)$$

Since O^* is the exchanged order of O which only swaps B_{o_i} and $B_{o_{i+1}}$, we get $O^*[1:i-1] = O[1:i-1]$ and $O^*[i+1:m] = O[i+1:m]$ (m is the number of input balance clusters). Based on Formula 3 and Formula 4, we conclude that $\Upsilon(O[1:i+1]) = \Upsilon(O^*[1:i+1])$. Besides, $O^*[i+1:m] = O[i+1:m]$, and $\sum_{k=1}^{j} T(B_{o_k})$ is the same in the recurrence formula of O and O^* when $j \geq i+1$. Therefore, $\Upsilon(O) = \Upsilon(O^*)$.

Case 2: $T(B_{o_i}) = 0$ **and** $T(B_{o_{i+1}}) = \eta$**.** First, B_{o_i} is already in the GPU and its transmission time is zero. η represents the constant time to transmit a balanced cluster. According to Formula 1,

$$\begin{aligned}
\Upsilon(O[1:i+1]) &= max(\Upsilon(O[1:i]), n \times \eta) + E(B_{o_{i+1}}) \\
\Upsilon(O^*[1:i+1]) &= max(\Upsilon(O^*[1:i]), n \times \eta) + E(B_{o_i}) \qquad (5)
\end{aligned}$$

where $n \times \eta = \sum_{k=1}^{i+1} T(B_{o_k})$. Also, we have $\Upsilon(O^*[1:i]) = max(\Upsilon(O^*[1:i-1]), n \times \eta) + E(B_{o_{i+1}})$ and $\Upsilon(O[1:i]) = max(\Upsilon(O[1:i-1]), (n-1) \times \eta) + E(B_{o_i}) = \Upsilon(O[1:i-1]) + E(B_{o_i})$ because $\Upsilon(O[1:i-1]) \geq (n-1) \times \eta$. As $\Upsilon(O^*[1:i]) = max(\Upsilon(O^*[1:i-1]), n \times \eta) + E(B_{o_{i+1}})$, we derive that

$$\Upsilon(O^*[1:i]) \geq n \times \eta + E(B_{o_{i+1}}) \qquad (6)$$

As $O[1:i-1] = O^*[1:i-1]$, we get $\Upsilon(O[1:i-1]) = \Upsilon(O^*[1:i-1]) \leq max(\Upsilon(O^*[1:i-1]), n \times \eta)$. Therefore, we conclude that

$$\begin{aligned}
&\Upsilon(O[1:i-1]) \leq max(\Upsilon(O^*[1:i-1]), n \times \eta) \\
\Longrightarrow\ &\Upsilon(O[1:i]) - E(B_{o_i}) \leq \Upsilon(O^*[1:i]) - E(B_{o_{i+1}}) \\
\Longrightarrow\ &\Upsilon(O[1:i]) + E(B_{o_{i+1}}) \leq \Upsilon(O^*[1:i]) + E(B_{o_i}) \qquad (7)
\end{aligned}$$

Besides, we have the following formula based on Formula 5.

$$\Upsilon(O[1:i+1]) = \begin{cases} n \times \eta + E(B_{o_{i+1}}), & n \times \eta \geq \Upsilon(O[1:i]) \\ \Upsilon(O[1:i]) + E(B_{o_{i+1}}), & n \times \eta < \Upsilon(O[1:i]) \end{cases}$$

No matter which case is true, we have $\Upsilon(O[1:i+1]) \leq \Upsilon(O^*[1:i]) + E(B_{o_i}) \leq max(\Upsilon(O^*[1:i]), n \times \eta) + E(B_{o_i}) = \Upsilon(O^*[1:i+1])$ according to Formula 6 and Formula 7. $\sum_{k=1}^{j} T(B_{o_k})$ is the same in the recurrence formula of O and O^* when $j \geq i+1$. Therefore, $\Upsilon(O) \leq \Upsilon(O^*)$.

Case 3: $T(B_{o_i}) = T(B_{o_{i+1}}) = \eta$ **and** $E(B_{o_i}) \geq E(B_{o_{i+1}})$**.** In this case, we have a similar formula to Formula 5:

$$\begin{aligned}
\Upsilon(O[1:i+1]) &= max(\Upsilon(O[1:i]) + E(B_{o_{i+1}}), \\
&\qquad \sum_{k=1}^{i+1} T(B_{o_k}) + E(B_{o_{i+1}})) \\
\Upsilon(O^*[1:i+1]) &= max(\Upsilon(O^*[1:i]) + E(B_{o_i}), \\
&\qquad \sum_{k=1}^{i+1} T(B_{o_k}) + E(B_{o_i})) \qquad (8)
\end{aligned}$$

Also, we have $\Upsilon(O^*[1:i]) = max(\Upsilon(O^*[1:i-1]), \sum_{k=1}^{i-1} T(B_{o_k}) + \eta) + E(B_{o_{i+1}})$ and $\Upsilon(O[1:i]) = max(\Upsilon(O[1:i-1]), \sum_{k=1}^{i-1} T(B_{o_k}) + \eta) + E(B_{o_i})$. As $O[1:i-1] = O^*[1:i-1]$, we have

$$\begin{aligned}
&\Upsilon(O[1:i]) - E(B_{o_i}) = \Upsilon(O^*[1:i]) - E(B_{o_{i+1}}) \\
\Longrightarrow\ &\Upsilon(O[1:i]) + E(B_{o_{i+1}}) = \Upsilon(O^*[1:i]) + E(B_{o_i}) \qquad (9)
\end{aligned}$$

Since B_{o_i} has a higher priority than $B_{o_{i+1}}$, we get

$$\begin{aligned}
&E(B_{o_i}) \geq E(B_{o_{i+1}}) \\
\Longrightarrow\ &\sum_{k=1}^{i+1} T(B_{o_k}) + E(B_{o_{i+1}}) \leq \sum_{k=1}^{i+1} T(B_{o_k}) + E(B_{o_i}) \qquad (10)
\end{aligned}$$

Combining Formula 8, Formula 9 and Formula 10, we conclude

$$\Upsilon(O[1:i+1]) \leq \Upsilon(O^*[1:i+1])$$

Because $\sum_{k=1}^{j} T(B_{o_k})$ is the same in the recurrence formula of O and O^* when $j \geq i+1$, we derive $\Upsilon(O) \leq \Upsilon(O^*)$.

Batch Size	IVF-GPU (T4) Time per Query (ms)	IVF-GPU (T4) Per-Dollar Performance	IVF-CPU (r5.2xlarge) Time per Query (ms)	IVF-CPU (r5.2xlarge) Per-Dollar Performance	HNSW-CPU (r5.2xlarge) Time per Query (ms)	HNSW-CPU (r5.2xlarge) Per-Dollar Performance
8	0.394	3370.81	20.62	96.2	11.37	174.42
32	0.366	3631.12	18.34	108.163	10.5	188.96
128	0.356	3729.54	17.07	116.23	9.79	202.52
512	0.354	3755.30	16.29	121.77	9.47	209.37
2048	0.355	3740.74	16.25	122.05	9.51	208.55

Table 4: Comparison between GPU and CPU on different ANN indexes.

It's easy to extend the three adjacent exchange cases to a general exchange paradigm, i.e., exchanging two arbitrary balanced clusters rather than an adjacent pair. We prove that exchanging will reduce the overlapping and Algorithm 1 is able to find the optimal order on per-cluster granularity.

A.2 Comparison between GPU and CPU

We perform a measurement to compare the performance and the price of GPU-based and CPU-based (IVF and the state-of-the-art graph index, HNSW) vector query processing systems. The current GPU vector query processing systems are based on IVF since the graph index requires random vector access and is not suitable for the continuous memory access pattern on GPUs. The setup is described as follows.

Setup. This measurement is conducted on AWS. We use three EC2 instance types, including two GPU instances (p3.2xlarge and g4dn.2xlarge) and one CPU instance (r5.2xlarge). p3.2xlarge is configured with one NVIDIA V100 GPU while g4dn.2xlarge is configured with one NVIDIA T4 GPU. Both of the GPUs are equipped with 16 GB GPU memory. r5.2xlarge is configured with 8 vCPUs (Intel Platinum 8259CL) with 64 GB host memory. We use SIFT10M [25] (with around 9GB memory footprint on GPU) in this measurement and fix the parameters in index building and query processing (e.g., similar memory footprint and accuracy, recall@10) for fair comparison. The per-dollar performance is calculated by $Price/Time\ per\ Query$. *Price* is AWS EC2 On-Demand hourly rate [54].

Table 4 shows the time per query and per-dollar performance of different devices under different batch sizes. The GPU-based system [23] has lower cost per query than the CPU-based system (IVF-CPU and HNSW-CPU). The larger batch size introduces higher computation parallelism, which fully utilizes the GPU resources. In summary, leveraging GPUs to accelerate vector query processing provides higher performance and is more cost-effective. It is notable that the dataset can be held in the GPU memory, but when the scale of datasets grows, existing GPU-based vector query processing systems cannot support large datasets beyond GPU memory.

A.3 Cluster Balancing Algorithm

The pseudo-code is in Algorithm 2. The function *BALANCE* balances the original clusters in *host memory* and returns the balanced clusters with equal size. It takes two inputs: the l original clusters and a configurable standard deviation. It uses *cluster_sizes* to store the original clusters' sizes, and uses *map_sizes* to store the balanced clusters' sizes and their related original clusters. ρ represents the fixed size of balanced clusters while χ is the minimal threshold specified by the user. The function first initializes array *cluster_sizes* through a for loop and initializes ρ as the minimum value in *cluster_sizes* (line 3-7). It then starts a while loop that splits the original clusters until the variance of the generated clusters is less than the given standard deviation, *dev* (line 9-25). Finally, it generates the balanced clusters with the appropriate sizes and returns them (line 26-32). The algorithm runs offline, and the overhead is negligible compared to the index building time.

Algorithm 2 Cluster balancing algorithm

```
1:  function BALANCE({C_1,...,C_l}, dev)
2:      cluster_sizes ← ∅, map_sizes ← ∅
3:      for i = 1 → l do
4:          // SIZE function returns the size of the cluster
5:          cluster_sizes.append(SIZE(C_i))
6:          map_sizes.append(∅)
7:      ρ ← Min(cluster_sizes)
8:      // χ is a configured value
9:      while ρ ≥ χ do
10:         for i = 1 → l do
11:             size ← cluster_sizes[i]
12:             while size ≥ 0 do
13:                 if size ≥ ρ then
14:                     map_sizes[i].append(ρ)
15:                 else
16:                     map_sizes[i].append(size)
17:                 size ← size − ρ
18:         // DEV function returns the standard deviation
19:         cur_dev ← DEV(map_sizes)
20:         if cur_dev ≤ dev then
21:             break
22:         else
23:             ρ ← ρ/2
24:             for i = 1 → l do
25:                 map_sizes[i] ← ∅
26:     BC ← ∅
27:     for i = 1 → l do
28:         start ← 0
29:         for j = 0 → SIZE(map_sizes[i]) do
30:             BC.append(C_i[start : start + map_sizes[i][j]])
31:             start ← start + map_sizes[i][j]
32:     return BC
```

LoLKV: The Logless, Linearizable, RDMA-based Key-Value Storage System

Ahmed Alquraan[*], *Sreeharsha Udayashankar*[*], *Virendra Marathe*[#], *Bernard Wong*[*], *Samer Al-Kiswany*[*]
[*]*University of Waterloo, Canada*
[#]*Oracle Labs, USA*

Abstract

We present LoLKV, a novel logless replicated key-value storage system. LoLKV follows a fundamentally different approach for designing a linearizable key-value storage system compared to state-of-the-art systems. LoLKV forgoes the classical log-based design and uses lock-free approach to allow multiple threads to concurrently update objects. It presents a novel leader election and consolidation approach to handle complex failure scenarios. LoLKV's followers are passive, reducing their overall CPU usage. Our evaluation shows that LoLKV achieves 1.7–10× higher throughput and 20–92% lower tail latency than other state-of-the-art low-latency key-value stores.

1 Introduction

Online services, such as financial services [1, 2] and interactive applications [3], have strict tail latency requirements in the microsecond ranges [4]. Systems supporting these services must achieve high throughput at a low tail latency. A central category of systems supporting this class of services is replicated and strongly consistent key-value stores [5, 6, 7].

State-of-the-art low-latency key-value stores such as APUS [5], DARE [6], Mu [7], and uKharon-KV[8] use RDMA-based consensus protocols, which replicate data across multiple nodes in a few microseconds. While these systems achieve acceptable tail latencies under low load, our evaluation (Section 6) shows that these systems are unable to maintain these low tail latencies under high load. The main reason for this inefficiency is that these systems follow the classical log-based design [9, 10] for building a linearizable storage.

The classical log-based design has two steps for executing operations that update objects: *replication* and *application*. During the replication step, a leader stores and *replicates* a new operation to a replicated log. Once the operation is replicated on a majority of followers, the application step *applies* the operation to the key-value store on all followers.

The log-based design [9, 10] has three fundamental inefficiencies when considering low tail latency workloads. First, the log represents a single point of serialization, limiting concurrency and prohibiting current systems from effectively leveraging multi-core machines. To address this inefficiency, modern systems resort to *sharding* [5, 6, 7, 11, 12, 13, 14], such that each shard serves a subset of the key space. Each shard has a separate process with its own memory, threads, and followers on other nodes. Current systems use sharding for two purposes. First, to distribute shards among nodes to scale to large clusters. Second, to have multiple *active* shards per node to leverage multiple CPU cores. Unfortunately, deploying multiple active shards per node leads to inefficient memory use, especially when paired with RDMA, as each shard process has its own pinned memory region causing memory fragmentation. Furthermore, having a large number of shards complicates supporting multi-key operations and leads to lower performance for skewed workloads, in which a few shards hold popular keys.

The second fundamental inefficiency is that the classical log-based design separates data replication from application. In key-value stores this leads to an extra memory copy as new objects need to be first stored in the log [5, 6, 7] and later, when committed, copied to the key-value store.

Finally, the classical design requires all followers to re-execute every operation. This significantly increases system overhead. For instance, with a replication level of three, every operation is executed three times in the cluster.

We present the Logless Linearizable Key-Value (LoLKV) storage system. LoLKV follows a fundamentally different design approach compared to the state-of-the-art systems. LoLKV forgoes the replicated log design and avoids placing multiple active shards on a node. In LoLKV, a node may have a single active leader shard and multiple passive follower shards. The leader shard is multi-threaded and utilizes all of the node's CPU cores. The passive follower shards are not involved in processing `put` and `get` operations. The leader shard uses one-sided RDMA to replicate new objects on followers. Once the new object is replicated on a majority of nodes, the hash table is updated with a pointer to the new object.

This approach overcomes the aforementioned shortcomings. First, it uses multiple threads to utilize all CPU cores and to concurrently replicate objects. It avoids extra memory copies by only having pointers in the hash table. Finally, the leader replicates the updates to the memory and hash table of followers without requiring the follower to re-execute operations. This enables LoLKV to use all available resources on all nodes to serve client requests, leading to higher performance.

While the proposed design increases system concurrency, it introduces new challenges for leader election and data consolidation. The proposed design complicates fault tolerance because LoLKV uses multiple concurrent threads that can leave the system in an inconsistent state under leader failure scenarios. Applying insert and delete operations directly in the key-value store complicates garbage collection as well. To realize LoLKV, we design novel leader election and data consolidation protocols that identify the latest updates for each memory segment and sync a new leader with the latest

system state. In addition, LoLKV uses a replicated garbage collection approach to maintain the storage system memory in sync during garbage collection.

We implemented LoLKV and compared it to the state-of-the-art systems. Our evaluation shows that for uniform workloads, LoLKV achieves *1.69–2.9*× higher throughput and *20–55%* lower tail latency over DARE [6], and *4–10*× higher throughput and *56–92%* lower tail latency over APUS [5], Mu [7] and uKharon [8]. In addition, our evaluation shows that LoLKV achieves similar results under skewed workloads and under different read and write ratios. Our evaluation of system scalability shows that LoLKV can scale to efficiently use all system resources and achieve up to *18 million op/s* which is 4×, 7×, 10×, and 36× higher than DARE, uKharon, Mu, and APUS, respectively.

The rest of the paper is organized as follows. Section 2 discusses RDMA and RDMA-based consensus protocols. Section 3 and 4 discuss the design and implementation of LoLKV. Section 5 discusses the correctness of LoLKV. Section 6 presents the results of our evaluation. Section 7 discusses additional related work before the paper concludes in Section 8.

2 Background and Related Work

2.1 Remote Direct Memory Access

Remote direct memory access (RDMA) [15] allows a machine to directly access the memory of a remote machine without involving the remote CPU. RDMA offers a low-latency and high throughput communication mechanism as it bypasses the kernel network stack.

Applications communicate over RDMA by establishing queue pairs. Each queue pair consists of a send and a receive queue. RDMA supports two main types of operations: two-sided and one-sided operations. Two-sided operations include `Send` and `Receive` operations and involve the CPUs of both the sender and receiver. One-sided operations do not involve the CPU on the receiving side. The sender specifies the data's remote address. One-sided operations have lower latency and higher throughput compared to two-sided operations [16].

Current implementations of RDMA support three transport protocols: Reliable Connected (RC), Unreliable Connected (UC), and Unreliable Datagram (UD). In RC and UC, a connection is established between two queue pairs, one on the sender and one on the receiver, which communicate exclusively with each other. RC guarantees that messages are delivered at most once, in order, and without corruption. In a UD protocol, one queue pair can communicate with one or more queue pairs. UD supports only two-sided operations.

In LoLKV, we use one-sided `Writes` over RC for data replication and two-sided operations over UD for client communication.

2.2 RDMA-based Consensus

Several RDMA-based leader-based consensus protocols have been recently proposed. In leader-based systems, one replica acts as a leader while others are followers. The leader is responsible for processing client requests. The leader appends new operations to the log and then replicates the log entry on followers. The operation is considered committed only if it is replicated on a majority of replicas. The leader then applies the operation to the state machine and replies to the client.

DARE [6] is an in-memory RDMA-based consensus protocol that adopts the Raft protocol [9]. Replication in DARE requires 2 RTT of one-sided `Write` operations. The first set of `Writes` appends the log entry to the follower logs while the second updates their tail indices. When both `Writes` succeed on a majority of replicas, the leader updates its commit index and posts another `Write` to update the followers' commit indices. Followers check their commit index periodically to apply newly committed entries to their state machines.

APUS [5] is a Paxos-based [10] consensus protocol which uses RDMA. The leader stores new operations to its local log and then replicates log entries to follower logs using one-sided `Writes` and waits for acknowledgments. In contrast to DARE, APUS followers actively participate in replication. Each follower notifies the leader when it accepts a log entry by sending an RDMA `Write` to update the entry in the leader's log. The leader commits an entry if it is accepted by a majority of replicas. Committing an entry in APUS requires 2 RTTs.

Mu [7] is an RDMA-based consensus protocol that targets microsecond-scale applications. The leader uses an RDMA `Write` to append operations to follower logs. The operation is considered committed if it is replicated on a majority. Followers poll committed requests from the log and pass them to the application. The leader does not start replicating a log entry until the previous one is committed. Mu requires a single RTT to replicate an operation in the common case.

uKharon [8] is an RDMA-based consensus protocol optimized for microsecond-scale failure recovery. uKharon uses one-sided RDMA-based Paxos to achieve fast leader failover times. To enable one sided Paxos, uKharon's consensus engine uses an RDMA Write and a Compare-and-Swap (CAS) operation in the ACCEPT phase while Mu [7] only uses an RDMA Write. While this additional CAS improves failure recovery times, the authors mention that it causes uKharon to perform worse than Mu in failure-free scenarios. In addition, uKharon does not offer ways for failed or network-partitioned nodes to rejoin the system.

These systems have three fundamental shortcomings. Firstly, at the core of each of these systems is a replicated log. The log limits concurrency because it introduces a serialization point. New operations are inserted serially in the log. Furthermore, in Mu, a new object is replicated only when the previous object is committed.

We evaluate the overhead of having multiple threads appending to a log. Figure 1 shows the time a thread waits until it acquires the lock protecting the log in APUS and Mu. Results show that the average wait time for 8 threads is 14.9× and 38.5× higher than that with one thread for APUS and

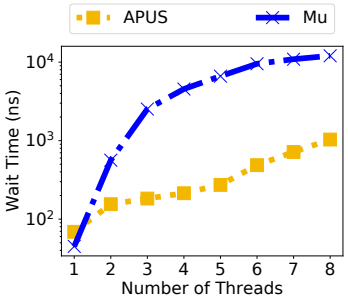


Figure 1: The average time needed to acquire the lock protecting the operation log in APUS and Mu.

Mu, respectively. Mu has a higher overhead as Mu holds onto the lock until the operation is replicated and committed, while APUS releases the lock after appending the request to the log. Given the low latency that RDMA provides, the delay imposed by the log serialization mechanism introduces a bottleneck that limits the performance of the system.

Second, the log-based design imposes an additional data copy. New objects are first inserted into the log and then copied to the key-value store. Finally, to efficiently leverage all CPU cores, multiple leader shards are deployed per node. Deploying multiple leader shards per node requires physical partitioning of memory between co-located shards, leading to inefficient memory usage. Also, sharding leads to lower performance under skewed workloads.

LoLKV adopts a fundamentally different design approach to overcome these shortcomings. LoLKV adopts a logless design that co-designs the storage and index data structures to avoid unnecessary memory copies. LoLKV deploys a single multi-threaded leader shard per node, which is able to efficiently utilize all CPU cores. It also uses novel techniques for leader election, data consolidation, and garbage collection. These design decisions lead to higher concurrency and lower system overhead.

3 LoLKV Design

LoLKV supports `get`, `put`, and `delete` requests, which read, write, and delete entire objects respectively. Figure 2 shows the architecture of LoLKV. The system consists of a set of nodes (i.e., replicas or replica set) connected using an RDMA-capable network, such as InfiniBand [15].

LoLKV adopts a leader-based replication protocol, in which one replica acts as a leader while others are followers. All client requests are handled by the leader, which runs multiple worker threads to process them. LoLKV does not partition the key space among the threads, i.e., any thread can process `put` or `get` requests for any object.

Similar to Raft [9] and Multi-paxos [17], LoLKV divides time into terms with a single leader per term. Each term has a unique `term_id` that is assigned in a strictly increasing order. When the leader fails, a new leader is elected using a leader election protocol (Section 3.4).

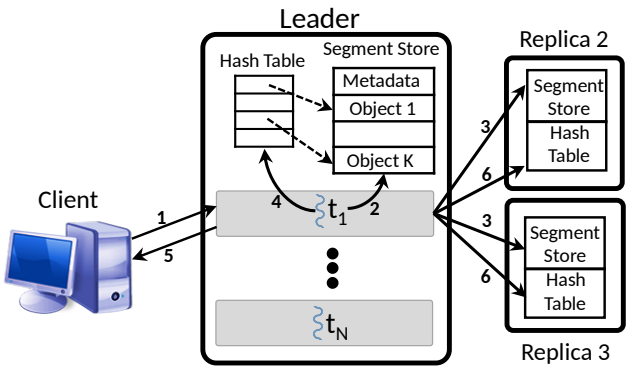


Figure 2: LoLKV Architecture.

LoLKV has two components: the *segment store*, which holds objects within memory segments; and the *hash table*, which contains pointers to these objects. To store a new value, the leader first stores the value in its local segment store (Step 2 in Figure 2) before replicating it on follower nodes (Step 3). A new value is considered committed only if it is stored in the segment store of a majority of followers. After an entry is committed, the leader updates its local hash table (Step 4) and replies to the client (Step 5). Updates to the hash table are replicated lazily to the followers (Step 6) since the object metadata stored in the segment store has enough information to recover the hash table in case of leader failure.

LoLKV guarantees linearizability at the object level; updates to the same object appear to be executed in a single global order. While updates to different objects can proceed concurrently. To manage concurrency within the segment store, each worker thread has exclusive access to a subset of the memory segments. The hash table is shared among all threads and uses CAS operations to serialize concurrent updates by multiple threads.

3.1 Segment Store

LoLKV divides allocated memory into equal sized segments which hold objects. Each segment contains a metadata section and an array of `ObjectEntries`. All objects within a single segment have the same size but objects across different segments may have different sizes. LoLKV's segment store design is inspired by the Hoard memory allocator [18].

A segment can be owned by a single worker thread at a time. The thread which owns a segment has exclusive write access to its objects. A segment can be *free* meaning it is unowned, *active* meaning it has space for new objects, or *sealed* meaning that it is full and its objects cannot be modified. As stored objects are immutable, any thread can read any object from any segment. Each thread typically owns multiple segments with different object sizes.

3.1.1 Segment Metadata

Figure 3a shows the fields of a segment's metadata:

(a) Segment Metadata (b) ObjectEntry

Figure 3: LoLKV data structures.

- `owner_id` : ID of the thread that owns the segment.
- `term_id` : Term during which the latest change to the segment metadata occurred.
- `status` : Indicates whether a segment is active or sealed.
- `seg_ver` : Updated when a thread owns a segment.
- `object_size` : Size of objects within the segment. Object size may differ between segments.
- `tail_idx` : Index of the latest inserted entry in the segment.

3.1.2 Owning a Segment

To claim ownership of a new segment, a thread traverses the segments array in the segment store to find a segment that is "free". The thread then uses a CAS operation to change the `owner_id` field of the metadata of the free segment. Following this, the thread sets the `seq_num` for all `ObjectEntries` to -1 to indicate that they are free. Then, it sets the fields of the segment metadata: `term_id` to the current term number, `status` to "active", and `tail_idx` to 0.

LoLKV commits segment ownerships one at a time on a majority of replicas. The thread owning a new segment acquires a lock, increments a counter shared across threads, copies the counter's value to the `seg_ver` field in the segment metadata, and then replicates this metadata. Once the operation is committed via replication on a majority of followers, the thread releases the lock and uses the segment to store new objects. This process guarantees that a segment's ownership is committed before it is used.

3.1.3 Sealing and Releasing a Segment

A thread seals a segment when it is full and all its objects have been applied to the hash table. A segment remains sealed until all its entries are garbage collected. After that, the thread can release the segment. To release a segment, a thread will change the `owner_id` to -1 and set the status to "free". When a segment is released, it becomes available for other threads to own. Sealing and releasing a segment are performed by changing the `status` field to "sealed" or "free", respectively.

Segment sealing and releasing operations are not replicated. If a leader fails after sealing or releasing a segment, a future leader will perform the same operations as the old leader. If it finds a full segment, it will seal the segment, and if it finds an empty segment, it will release the segment.

3.2 Hash Table

The hash table stores pointers to `ObjectEntries` stored in segments; The goal of the hash table is to implement efficient `get` operations. The size of each hash table entry is 64 bits.

A hash table entry stores the offset of an `ObjectEntry` within the segment store, to ensure portability across replicas. While segment stores may have different base addresses across replicas, an `ObjectEntry` in a segment will have the same offset on all replicas. The hash table is shared among all worker threads.

The main fields of an `ObjectEntry` are shown in Figure 3b:

- `<key,value>`: The object's key and value.
- `term_id` : Term during which this entry was written.
- `seq_num` : Operation sequence number assigned by the worker thread. Used to order all operations of a thread within a term.

3.3 Replication Protocol

The leader runs multiple worker threads to process client requests. Each request is processed by one worker thread. Worker threads are independent; each thread has its own RDMA resources (e.g., queue pairs and work completion queues).

3.3.1 Worker Thread Metadata

Each worker thread maintains the following metadata fields: `sequence_number` and `latest_operation`. The `sequence_number` is a strictly increasing counter that is used to assign a sequence number for `put` operations processed by the thread. The `latest_operation` field stores the address of the last `ObjectEntry` inserted by the thread.

The thread also maintains two metadata fields for each follower: `latest_replicated_operation` and `latest_applied_operation`. The thread uses these to identify operations not yet replicated or applied at the follower.

3.3.2 Put Requests

The processing of a `put` request has three main phases: creation, replication, and application phases.

Creation Phase. When a thread receives a `put` request, the thread creates an `ObjectEntry` and populates its fields. The thread increments its `sequence_number` and assigns it to the entry. The thread writes the object to a segment. The thread updates its `latest_operation` field and `tail_idx` field of the segment metadata to point to the newly inserted entry.

Replication Phase. The thread commits the `ObjectEntry` by replicating it on a majority of replicas. The thread replicates entries in order of `seq_num`. Threads use RC transport to replicate objects, which guarantees in-order message delivery.

Once an ObjectEntry is replicated on a majority of replicas, the request is considered committed. The thread updates its latest_replicated_operation field whenever it replicates an entry on a follower. The <term_id, seq_num> tuple specifies the order of all operations processed by a thread.

Application Phase. After an ObjectEntry is committed, the thread updates the hash table entry to point to the newly inserted object. To guarantee consistency, entries must be applied in the order specified by <term_id, seq_num> tuple.

LoLKV uses a linear probing hash table. The thread first checks if the key already exists in the hash table. The thread hashes the key *K* to determine the entry to begin probing from. The thread checks that hash table entry and all successive entries one by one. Probing terminates when the thread finds either a hash table entry pointing to an ObjectEntry of the key *K* or an empty entry. If the thread finds an entry pointing to the key *K*, the thread only updates the entry if the insert is a newer operation than the existing version (Section 3.3.3). If probing ends at an empty entry, the key is inserted into the empty entry.

As the hash table is shared among all worker threads, multiple threads may concurrently try to update the same hash table entry. To ensure correctness, hash table entries are updated atomically using CAS operations. If a CAS operation fails, the thread repeats the probing process.

After the thread applies the entry locally, it returns an acknowledgement to the client. In the background, the thread replicates the hash table updates to the followers. Periodically, the thread updates segments' tail_idx on followers.

3.3.3 Concurrent Put Requests to the Same Key

In LoLKV, multiple put requests for the same key might be replicated concurrently by different threads. As a result, an ordering mechanism is needed to ensure the correctness of the system. LoLKV uses the *incarnation array* to ensure operations are serialized. The incarnation array is an array of *K* atomic counters. LoLKV divides the key-space into groups and each group is mapped to a counter in the incarnation array. The counter is atomically incremented whenever a put operation for a key within the group is processed. This value is then stored in the incarnation field of the ObjectEntry.

Before applying an ObjectEntry to the hash table, its <term_id, incarnation> tuple is verified to ensure that it is larger than the <term_id, incarnation> of the ObjectEntry currently pointed to by the hash table. If it is not larger, the hash table update is discarded since the ObjectEntry currently pointed to by the hash table is newer.

The incarnation array is not replicated as it is only used to order writes by leader threads. When a new term starts, the new leader resets its incarnation array. The size of the incarnation array is far larger than the number of worker threads, which reduces the chance of false sharing between concurrent put operations.

3.3.4 Get Requests

In LoLKV, get requests are served locally by the leader. Any thread can process any get request for any key as a thread can read objects from all segments, even a segment it does not own. The worker thread probes the hash table in the same manner used for put requests. The thread will return a value to the client only if the hash table entry points to a valid ObjectEntry. It is safe to serve get requests using only the leader state because the leader obtains a lease [19] from a majority of replicas. During the lease, other replicas will not try to become a leader.

3.3.5 Delete Requests

Delete requests follow the same steps of the put requests with a small difference: a tombstone object is replicated in the segment store and the hash table is updated with to point to the tombstone object.

3.4 Leader Election

LoLKV divides time into terms, and each term has at most one leader. Before a replica becomes a leader, it must get the votes of a majority of replicas. Each replica can vote only once in a term. LoLKV relies on dynamically managing the state of the RC QPs to ensure that only the current leader has remote access to the system's data structures. Before accepting a vote request from a candidate, replicas revoke the old leader's access by transitioning the associated QPs into a non-operational state.

Failure detection. LoLKV employs a heartbeating mechanism to detect failures. Each replica maintains a heartbeat data structure which consists of three fields: term_id, leader_id, and counter. The counter field is incremented periodically. The leader election protocol starts when a follower suspects that the leader has failed (i.e., misses *N* heartbeats from the leader). To detect leader failure, followers read the leader's heartbeat data structure periodically via RDMA. Followers verify that the leader's counter field has changed and that the leader's ID and term_id match their data structure. If a follower cannot access the leader's heartbeat data structure or the leader's counter does not change for a specific duration, the follower assumes that the leader has failed. Similarly, the leader checks the followers heartbeat data structures periodically and steps down if it no longer has a majority of followers.

Leader election. Leader election protocols of some systems (e.g., Raft [20]) elect the most up-to-date replica as a leader. However, using this approach in LoLKV is not feasible. In LoLKV, different worker threads are independent. Different threads may replicate different requests on different replicas. That is, two committed operations could be replicated on two different majorities. As a result, after a leader failure, there could be no replica that is up-to-date for all threads.

In LoLKV, each replica scans its segments to find the the segment with the largest <term_id, seg_ver>. Replicas exchange their largest <term_id, seg_ver>. The replica with the largest <term_id, seg_ver> becomes a leader. If multiple nodes have the same <term_id, seg_ver>, the node with the smallest IP address among them becomes the leader.

Data consolidation. The elected leader may not have all the segments up-to-date. Updating segments requires identifying the most up-to-date replica for each thread. To do this, the leader sends a request to each follower asking for the highest <term_id, seq_num> of each thread. Using this information, the leader identifies its missing operations for each thread and the replicas which have these operations. The leader contacts the most up-to-date follower to get the missing operations. The leader commits any uncommited operations and applies them to the hash table.

Update Followers. During the leader election and data consolidation steps, the leader finds the highest <term_id, seg_ver> for each follower and the highest <term_id, seq_num> for all threads of the follower. The leader uses this information to complete the work of the previous leader by committing all uncommitted memory ownership operations and put operations.

Once the leader completes this process, the leader has all committed values and its segment store is up-to-date. After that, the leader starts serving new client requests.

3.5 Fault Tolerance

Follower Failure. LoLKV is designed to tolerate N replica failures given $2N + 1$ replicas. When a follower fails, the leader removes it from the active set and stops replicating operations on that replica. Hence, follower failures do not affect the safety of the system. When a follower rejoins the system, the leader first finds the highest <term_id, seg_ver> of the follower, and then it replicates all memory ownership operations that the follower misses. Then, the leader recovers the worker threads metadata for that follower (i.e., latest_replicated_operation and latest_applied_operation). After that, the leader starts replicating ObjectEntries on the follower.

Leader Failure. When the leader fails, the system will not be available until a new leader is elected and the data consolidation process completes. For any thread, it is guaranteed that at least one replica in any majority has all committed entries that are committed by the failed leader. The new leader will use the data consolidation mechanism (Section 3.4) to bring itself up-to-date. Hence, although a leader failure might affect the availability of the system, it does not affect its correctness.

Message Loss. Client requests and responses are sent over UD QPs, which do not guarantee delivery of messages. If a request or a response is lost, the client times out and resubmits the request again. Note that it is safe to process requests again since they are idempotent. On the other hand, replication is implemented on top of RDMA RC transport which offers reliable and in-order message delivery.

Data Corruption. The data structures described in Section 3 can be accessed simultaneously by both local and remote threads. As a result, the correctness of any data read must be verified. LoLKV facilitates this by augmenting each structure with verification information. ObjectEntries are augmented with a byte appended to the end of each entry. For segment metadata entries and heartbeat data structures, a counter field is stored at the beginning of the data structure and again at the end of the data structure. If a failure occurs while updating this data structure, the counter field at the beginning will not match the one at the end.

Self-verifying data structures [13] are able to detect corruptions at the remote side as well (e.g., incomplete RDMA Write) because RDMA NICs guarantee that writes are performed in an increasing address order, i.e., the verification bytes at the end are not written before other bytes in the data structure.

4 Implementation Details

4.1 Garbage Collection

The system will eventually run out of space if segments are not released by owner threads. The garbage collection process is responsible for releasing segments. A segment is not released until all ObjectEntries in the segment become invalid, i.e., they are inaccessible through the hash table.

Each thread periodically checks its sealed segments to identify any segments that should be freed. In LoLKV, a segment is freed if the ratio of valid objects in a segment drops below a configurable threshold. Before a segment is released, all its valid entries must be moved to another active segment. The owner thread moves these entries to a new segment while preserving their incarnation and term_id values in order to avoid overwriting a newer entry of the same key. The thread then inserts the objects in the hash table using the same probing mechanism. An entry in the hash table is updated with a new address only if its incarnation equals that of the moved object. Once the segment does not have any valid entries, the thread releases the segment by updating its metadata.

4.2 Reusing Tombstones

While processing a put, if during the probing step the thread reaches an empty hash table entry, this indicates that the key does not exist. During probing, the thread records the first hash table entry that points to a tombstone with a term number smaller than the current term number. If probing terminates at an empty hash table entry (i.e., the key is not found), the thread updates the hash table tombstone to point to the new ObjectEntry. If no tombstone is found, the empty hash table entry is updated to point to the new ObjectEntry. We only reuse tombstones from previous terms to avoid conflicts with delete requests in the current term.

5 Correctness

LoLKV design is based on the following assumptions; the network is unreliable and asynchronous, as packets could be dropped or take arbitrary time to arrive at the destination. There is no limit on the time a node takes to process a request. We assume a non-byzantine failure model in which nodes may stop working but will never send erroneous messages.

LoLKV guarantees linearizability at the object level. All operations for an object appear to be executed in a single global order. We used the TLA+ model checking tool [21] to verify LoLKV's correctness. We use TLA+ model checking to verify the correctness of the segment store, the hash table, the replication protocol, and the leader election. In this section we sketch out the proof of LoLKV's correctness and discuss the correctness of the segment store and the hash table. To simplify the discussion, we sketch the proof without garbage collection.

Property I. *Memory ownership is safe*, meaning if a segment is owned by a thread, the segment ownership operation will be present in all future leaders. To prove this, we use three invariants:

1. *A leader in term i commits segment ownership operations in order of the seg_ver.* When a thread owns a new segment, it acquires a lock, increments *seg_ver* and assigns it to the new segment, commits the ownership operation on a majority of followers, then releases the lock. Using a lock guarantees that segments are assigned an increasing *seg_ver* and are committed in order of *seg_ver*.
2. *For a given term i, if a follower has segment with term_id = i and seg_ver = n, it is guaranteed that it also has all segments with term_id = i and seg_ver < n.* Following the argument in the previous invariant, segment ownership operations are committed in order of *seg_ver*. Furthermore, LoLKV uses RC transport to replicate memory ownership operation. RC guarantees ordered delivery of messages. This means if a node received *seg_ver*, then it must have received all previous messages sent on that channel.
3. *The last segment ownership operation committed in term i is present in the leader of term $i+1$.* The leader election protocol elects a leader with the highest <*term_id, seg_ver*> among a majority of replicas. If the last committed ownership operation on a majority in term i has $seg_ver = n$, then the leader of term $i+1$ will have an ownership operation with term id = i and $seg_ver >= n$. Following invariant 1 above, the leader of term $i+1$ will have the last committed entry in term i.

Invariant 3 indicates that a leader of term i has the latest memory ownership committed in term $i-1$. Hence, invariant 2 implies that the leader of term i also has all segment ownership committed in term $i-1$. *By induction, the leader of term i has all the segment ownership committed in all previous terms.*

Property II. *The data consolidation process guarantees that a leader has all committed entries from the previous term.* Without loss of generality, we sketch the proof for a single thread here using the following invariants:

4. *The leader thread in term i replicates put operations in order of the seq_num*. To commit an operation, the leader thread increments the *seq_num*, stores the operation in the local segment store, then replicates the operations to followers.
5. *If a follower in term i receives an operation with $seq_num = n$, then this follower has received all operations from this thread in term i with $seq_num < n$*. This follows from invariant 4, and that we use RDMA RC transport for replication. The RC transport guarantees in-order delivery of messages.
6. *For a given thread in term i, if an operation with $seq_num = n$ is committed, then all operations for that thread in term i with $seq_num < n$ are committed*. If operation with $seq_num = n$ in term i is committed, then this operation has been received by a majority of nodes. From invariant 5 above, those nodes that received $seq_num = n$ must have received all operations in term i with $seq_num < n$. Given that those are a majority of nodes, then all operations in term i with $seq_num <= n$ are committed.
7. *A node that has the highest seq_num in term i among a majority of nodes has all of the committed entries in term i.* Assume that in term i, the largest *seq_num* on a majority of nodes is n, and that the last committed entry in term i is m. Given that $seq_num = m$ is replicated on a majority, then $m <= n$. Following invariant 5 above, the node that has n also has all operations with $seq_num < n$, including m. Following invariant 6, it has all committed entries in term i.

The newly elected leader has the highest term number on a majority of nodes, but may not have all committed entries from the previous term. Before processing new requests, the leader runs the data consolidation process. For each thread, data consolidation looks for the highest <*term_id, seq_num*> on a majority of replicas. If a follower has a *seq_num* higher than the *seq_num* of the leader thread, then the consolidation process updates the leader with all missing operations. At the end of this process, for a given thread, the leader has all operations from the previous term with the highest *seq_num* among a majority of nodes. Following invariant 7 above, the leader has all committed entries in that term for that thread, verifying property II.

Property III. *LoLKV guarantees if an object is committed by a leader it will be present in the segment store of all future leaders.* From property II above, if a leader commits an operation in term i, the data consolidation process guarantees that this operation will exist in the segment store in the leader of term $i+1$. By induction, a committed entry will exist in the

segment store of all future leaders.

5.1 Correctness of the Hash Table

We do not need to replicate updates to the hash table entries for correctness. The object store has all the information needed to recover the hash table on a new leader. LoLKV threads lazily replicate updates to the hash table in the order of `seq_num` only to optimize leader recovery.

We discuss safety when multiple threads try to update the same slot in the hash table under two scenarios: concurrency between threads updating the same slot with different keys and concurrency between threads updating the same slot with the same key.

The hash table uses CAS to update its slots. If a thread finds an empty slot in the table and its CAS operation fails to update that slot, the thread repeats the probing operation to find the target slot for the new object.

Property IV. *LoLKV applies updates to the hash table following the order specified by the* `incarnation`. If two threads are processing a put request for the same key, one of them will insert the update first in the table using CAS. While probing, the second thread will find that the key already exists, but since the object was inserted by a different thread, we cannot use the `seq_num` to order these updates. For this scenario, we rely on the `incarnation` field. When processing a `put` request, a thread atomically increments the incarnation array to get a unique increasing `incarnation` for its `put` operation. We use this number to order operations on the same key. If a thread finds the object in the table with `incarnation` higher than the `incarnation` of its `put` operation, it discards its `put` operation, otherwise it updates the hash table.

Property V. *In LoLKV, A* `get` *request returns the latest committed object for the target key.* In LoLKV, `get` requests are served locally by the leader without contacting the followers. To serve a `get` request, a worker thread probes the hash table in the same manner used for a `put` request. If the thread finds the target key in the table, the thread reads the object from the object store. The thread verifies that the hash table slot was not modified while the thread is reading from the object store by verifying that the slot still contains the same object store address. If the slot has a different address, the thread repeats the process and reads the newer object of the key. Since a key can have at most one slot in the hash table, and since hash table slots point only to committed objects, the thread is guaranteed to return the latest committed object for the key.

Property VI. *LoLKV guarantees linearizability at the object level.* Properties I, II, and III guarantee that if a leader commits an entry in the segment store, the entry will be present in the segment store of all future leaders. Multiple threads may concurrently commit entries for the same key. Property IV shows how LoLKV uses the `incarnation` to create a global order of updates to the same key. Finally, Property V guarantees that `get` operations return the latest committed entry. These properties guarantee that LoLKV is linearizable for `put` and `get` operations per key.

6 Evaluation

We evaluate the performance of LoLKV and compare its latency and throughput against the state-of-the-art alternatives using uniform and skewed workload distributions.

Testbed. We conducted the experiments using a 12-node cluster on Cloudlab Utah [22]. Each machine has a Xeon E5-2450 8-core CPU with hyperthreading, 16GB of RAM, and a Mellanox Dual port FDR CX3 adapter. Machines are connected using a 56 Gbps Infiniband [15] network.

Alternatives. We compare LoLKV against DARE [6], APUS [5], Mu [7], and uKharon [8]. DARE and uKharon implementations come bundled with an in-memory key-value store. For APUS and Mu, we implemented an in-memory key-value store following the same design as LoLKV's in-memory key-value store. Newly inserted keys are stored in the APUS or Mu logs and then copied to the segment store during the apply phase. We also evaluated LogCabin [20], a strongly-consistent key-value store based on Raft [9]. LogCabin uses TCP/IP stack for communication. Our results with write-only workload show that LogCabin has a throughput of $12,500\ ops$, which is at least two orders of magnitude lower than other RDMA-based systems, and its latency is around $1.5ms$, which is three orders of magnitude higher latency than other RDMA-based systems. We also evaluated HERD [12], an unreplicated key-value store that uses a mixture of RDMA Writes and RDMA Send and Receive for communication with clients. Results show that HERD achieves $4\times$, $8\times$, $8\times$, and $50\times$ higher throughput compared to LoLKV, DARE, Mu, and APUS, respectively. This performance difference is mainly because HERD is an unreplicated key-value store, while other systems implement a replicated and linearizable key-value store. We omit adding the results of LogCabin and HERD to the figures for clarity.

Experiment Configuration. We run the systems in a multi-sharded setting, in which multiple processes of a system are deployed. Each process represents a replica of a single shard. We benchmark each system and select the number of shards and configurations that maximize its performance. We used a replication factor of three in all our experiments. All replicas of all shards were distributed among the same three nodes, while other nodes are used to run clients. Unless otherwise specified, we deploy the leader replicas of all shards on one node. However, we evaluate the scenario when the leaders are equally distributed among nodes in Section 6.6.

Shard Configuration. We run LoLKV with 8 threads, DARE with 8 shards, APUS with 7 shards, and Mu and uKharon with 4 shards. APUS does not scale to 8 shards as it runs a background process to establish RDMA multicast communication paths. Mu does not scale to 8 shards as each shard uses 4 threads; one thread performs log replication while 3 threads perform heart beating, monitoring, and permissions management. uKharon also does not scale to 8 shards as each

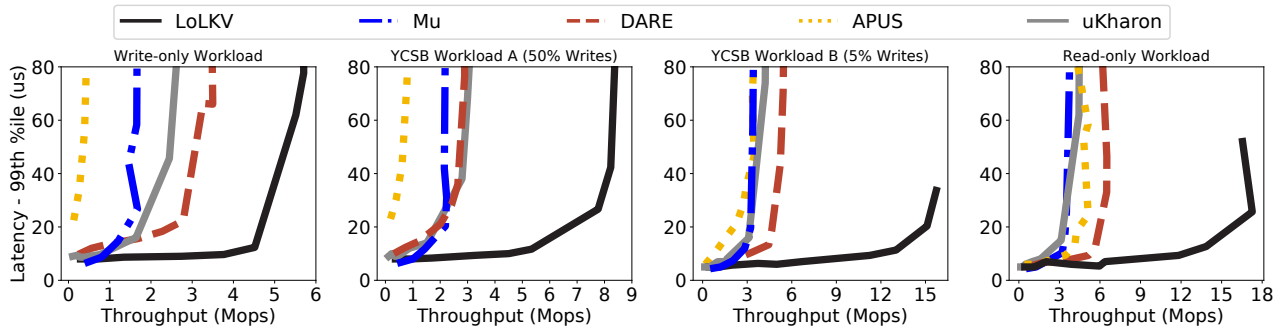


Figure 4: Throughput and tail latency using uniform YCSB workloads.

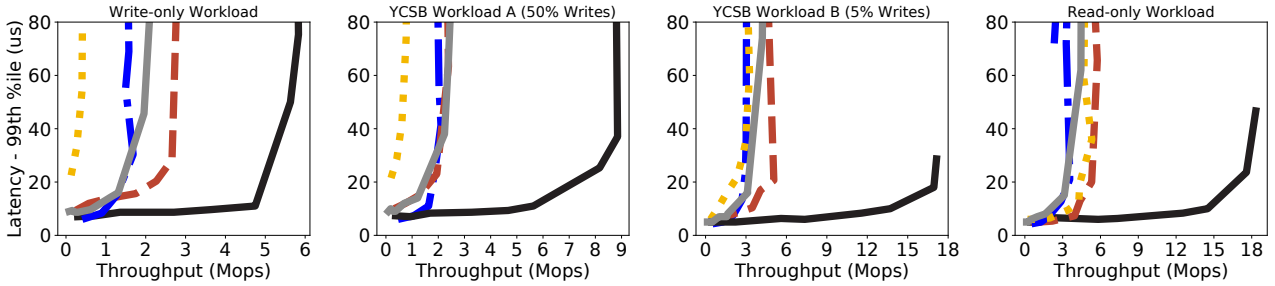


Figure 5: Throughput and tail latency using skewed YCSB workloads.

shard uses two threads, one for processing client requests and replication and one for monitoring and detecting failures.

Multi-shard deployment of DARE, APUS, Mu, and uKharon requires sharding the key space into partitions, with all requests to keys within a partition being served by the same replica set. In our experiments with these systems, a client picks the right shard for a request based on the hash of the key. On the other hand, as LoLKV does not have a fixed assignment of keys to threads, its clients send requests to threads in a round-robin fashion.

Workload. We used the YCSB benchmark [23] in our experiments with both uniform and skewed workloads. The skewed workload follows the Zipf distribution with a skewness parameter of 0.99. We experimented with 100,000, 1 million, and 10 million keys. As the results obtained in all these cases are similar, we present the results with 1 million keys. The key and value sizes are 24 and 64 bytes respectively. We report the averages from running 10 trials for each experiment. The standard deviation of all our experiments is below 5%.

6.1 Performance Evaluation

We compare the throughput and latency of all systems using the YCSB workloads with uniform and skewed key popularity distributions (Figure 4 and Figure 5).

For uniform workloads (Figure 4), LoLKV achieves *1.7–2.9*× higher throughput and *20–55%* lower latency compared to DARE. LoLKV also achieves *4–10*× higher throughput and *56–92%* lower latency compared to uKharon, Mu, and APUS. Figure 5 shows that LoLKV's performance is not affected by workload skewness. On the other hand, DARE, uKharon, Mu, and APUS achieve 14%, 16%, 6%, and 18% lower throughput with the skewed write-only workload compared to their throughput with uniform workload.

LoLKV outperforms other alternatives for several reasons. First, LoLKV requires only one RDMA Write operation to replicate an object. Second, LoLKV combines the replication phase and the apply phase by replicating operations directly to the object store, avoiding an extra memory operation to copy the data from the operation log to the object store.

APUS has the lowest performance because replication in APUS requires two RDMA Write operations: one from the leader to followers to replicate log entries and one from followers to the leader to indicate the acceptance of a log entry. Hence, in APUS, the CPUs of the followers are involved in the replication process, which reduces throughput and increases latency. Replication in Mu and uKharon requires only one RDMA write operation from the leader to the followers, which results in higher throughput and lower latency compared to APUS. However, Mu and uKharon do not employ batching in the replication process, which imposes long queuing delays under heavy workloads. DARE achieves better performance than APUS and Mu as it employs batching. However, DARE requires two RDMA Write operations to replicate a batch of operations: the first operation replicates log entries on followers and the second one updates the tail index of the followers.

LoLKV outperforms other systems even under read-heavy workloads due to two reasons. First, even with a small percentage of `put` requests, the overhead of replicating and applying operations limits the performance of `get` requests. Second, APUS, Mu, and uKharon are multi-threaded systems but not all threads are utilized to serve client requests. As a result, given the same amount of resources, LoLKV and DARE can

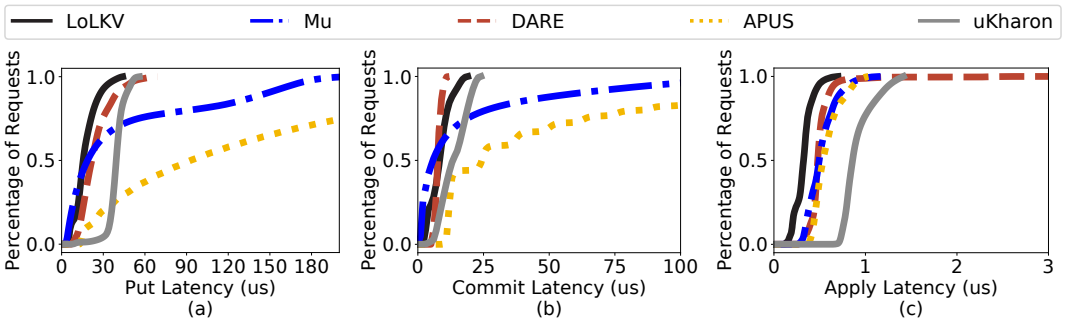


Figure 6: Latency CDF with a uniform write-only workload (a) `put` latency. (b) commit latency. (c) apply latency.

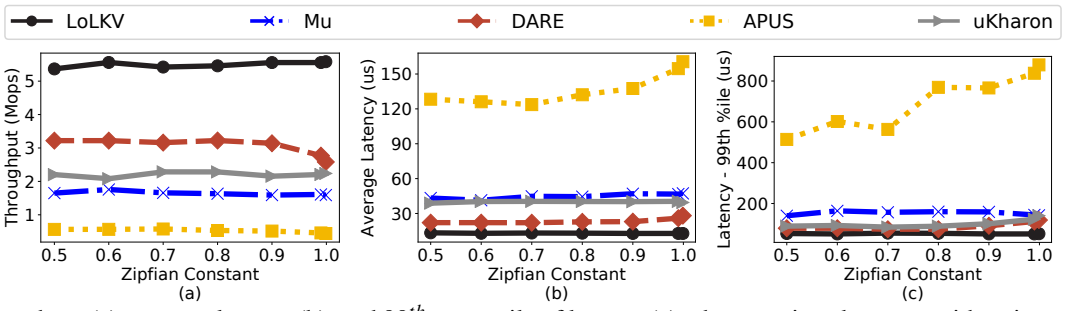


Figure 7: Throughput (a), average latency (b), and 99th percentile of latency (c) when varying skewness with write-only workload.

be deployed with larger number of threads/shards than APUS, Mu, and uKharon. Finally, DARE's batching mechanism batches similar operations only (i.e., a batch can have only `get` or `put` requests). This approach leads to small batches and degrades DARE's performance significantly when the workload has both `get` or `put` requests.

6.2 Latency Evaluation

Figure 6.a shows the CDF of `put` latency for a uniform write-only workload. Other workloads in the YCSB benchmark have a similar pattern. LoLKV lowers the 90th percentile of `put` latency by *30%*, *42%*, *80%* and *91%* compared to DARE, uKharon, Mu, and APUS, respectively.

Figure 6.b and Figure 6.c show the breakdown of the `put` latency into commit time and apply time under the uniform write-only workload. We measure the commit time as the time from when the leader polls a client request until the leader commits the request. LoLKV lowers the 90th percentile of commit latency by *33%*, *73%* and *89%* compared to uKharon, Mu, and APUS, respectively. DARE's commit latency is comparable to LoLKV. However, DARE has a large queuing delay as it polls requests one by one. The impact of this polling mechanism appears in Figure 6.a. LoLKV outperforms other systems as requests are replicated concurrently and committing an operation requires only one RDMA Write operation.

Figure 6.c shows the CDF of the apply time of different systems under the uniform write-only workload. The apply time is the time needed by the system to update the key-value store. LoLKV lowers the 90th percentile of the apply latency by *26%*, *63%*, *39%*, and *45%*, compared to DARE, uKharon, Mu and APUS respectively. In LoLKV, applying an operation requires only updating the hash table to point to the new object. In all other systems, in addition to updating the hash table, the object is copied from the log to the segment store.

6.3 Workload Skewness

We evaluate the impact of workload skewness on the throughput (Figure 7.a), the average latency (Figure 7.b), and the 99th percentile latency (Figure 7.c) when varying the skewness factor between 0.5 and 0.999 for the write-only workload. Results show that LoLKV handles extremely skewed workload efficiently; its throughput and latency are not affected by the workload skewness. With a skewness factor of 0.999, the throughput of DARE, uKharon, Mu, and APUS is reduced by 14%,16%, 6%, and 18%, and their average latencies is increased by 27%, 13%, 4%, and 27% compared to their throughput and latencies with a skewness factor of 0.5.

Increasing the skewness factor does not result in a significant performance degradation for all systems. The reason for this is that with a skewness factor of 0.999, 50% of the requests target only 209 keys. However, we found that these keys are distributed among different shards. Hence, the load generated by the clients is still distributed among all shards. As a result, we do not see a noticeable difference in the performance when comparing uniform and skewed workloads.

Modern key-value stores allow clients to select a partition function that maps keys to shards. These functions often group related keys in a single shards. To better understand the impact of workload skewness on the performance, we conducted an experiment in which we control the percentage of requests targeting a shard. Figure 8.a shows the systems throughput, Figure 8.b shows the average latency, and Figure 8.c shows the 99th percentile latency when we vary the percentage of requests that are handled by a specific shard. For ease of com-

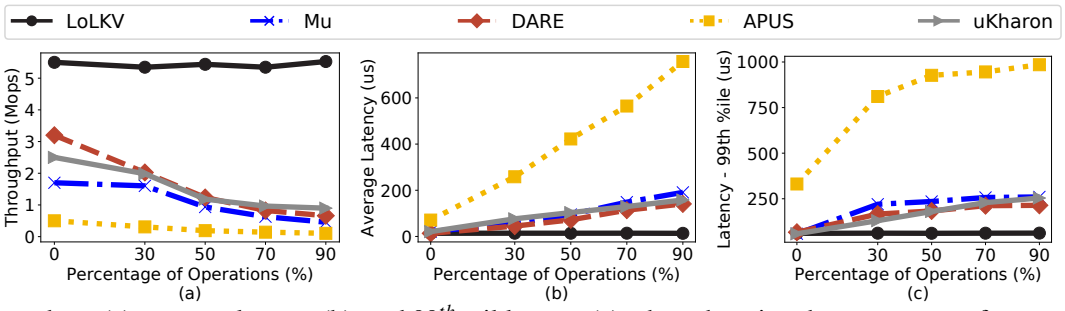


Figure 8: Throughput (a), average latency (b), and 99th tail latency (c) when changing the percentage of requests targeting a shard under write-only workload.

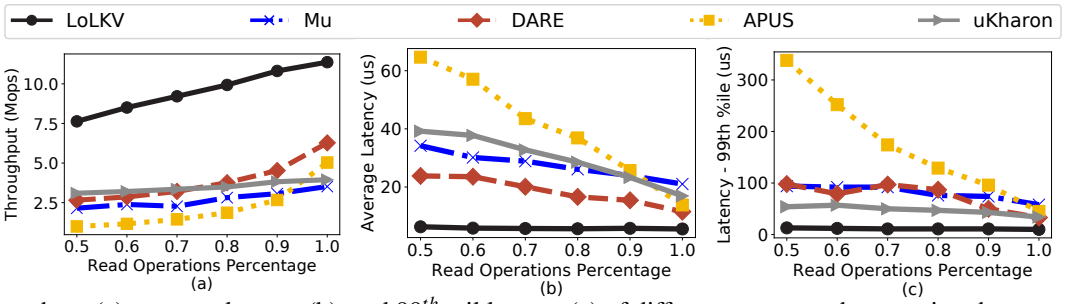


Figure 9: Throughput (a), average latency (b), and 99th tail latency (c) of different systems when varying the percentage of read requests.

parison, Figure 8 also shows the performance when requests are uniformly distributed among shards ("U" in Figure 8). Results show a significant degradation in the performance of DARE, Mu, uKharon, and APUS. For instance, comparing the uniform distribution to when 50% of requests are served by one shard, the throughput of DARE, uKharon, Mu, and APUS is reduced by 61%, 52%, 45%, and 62%, respectively. Similarly, the average latencies of DARE, uKharon, Mu, and APUS are 5.3× , 3.4×, 4.5×, and 5.9× their average latencies with uniform distribution, and their 99th percentile latencies are 2.7×, 3.2×, 4×, and 2.8× of their 99th percentile latencies with uniform request distribution.

LoLKV's performance is not affected by the workload skewness because LoLKV does not shard the key space among worker threads and all threads can participate in serving requests for popular objects. Concurrent `put` operations on the same key can be committed in parallel by different threads. Concurrent updates are ordered when updating the hash table. For all other systems, the commit and apply operations are serialized when there are concurrent updates to the same key and all operations for a popular key are handled by a single shard while other shards could be idle.

6.4 Read-to-Write Ratio

Figure 9 shows the throughput, the average latency, and the 99th percentile latency when varying the `get` requests percentage. Increasing the percentage of `get` requests increases the throughput and lowers the latency of all systems as `get` requests are served by the leader in all systems. Nonetheless, LoLKV outperforms other systems for all read-to-write ratios. For instance, when 70% of requests are `get` requests, LoLKV achieves *2.8*×, *2.75*×, *4*× and *6.3*× higher throughput compared to DARE, uKharon, Mu, and APUS, respectively. LoLKV outperforms Mu and uKharon for read-only workloads because LoLKV uses higher number of threads to serve requests. DARE polls one request at a time which limits its throughput even under read-only workloads.

6.5 LoLKV Failover

In Figure 10, we evaluate the recovery time of LoLKV when the leader fails, including the time it takes to elect a new leader and perform data consolidation. In this experiment, we use the same uniform write-only workload we use in Figure 4.a with the configuration that achieves the maximum throughput of 5.7 Mops. We kill the leader process at the 50 ms mark. The throughput drops to zero when the leader fails and the system remains unavailable until a new leader is elected.

The total recovery time is around *4.5ms*. Detecting the leader failure and electing a new leader takes around *1.4ms*. This is dominated by waiting for 3 heartbeat periods (300μs each) to start leader election. Around *3ms* are spent bringing the new leader up-to-date. Once the new leader is active, the throughput of the system returns to 5.7 Mops. We note that the current implementation of LoLKV is not optimized for fast failure recovery. For instance, in the current implementation one thread sequentially performs data consolidation for all thread segments before the leader can begin serving client requests. This can be parallelized by making each thread update the segments it owns. We will explore optimizing failure recovery in future work.

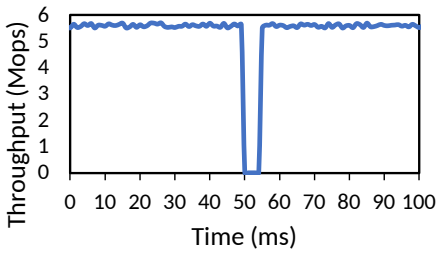


Figure 10: Recovery time of LoLKV.

6.6 Scalability

In this section, we evaluate the scalability as we increase the number of shards or threads. We deploy the systems on three nodes and uniformly distribute shard leaders across nodes, i.e., each node is a leader for one third of the shards and a follower for others. Figure 11 shows the throughput of different systems with increasing number of shards/threads. The results show that LoLKV can scale to efficiently use all hardware threads (i.e., 24 threads) on the three machines and achieve up to 18 Mops which is 4× higher than DARE, 7× higher than uKharon, 10× higher than Mu, and over 36× higher than APUS. LoLKV's performance drops when having more than 24 threads due to fully utilizing all CPU cores.

LoLKV efficiently use all CPU cores because followers in LoLKV are not active during normal operations. Consequently, all available CPU cores on each node are used to host leader shards serving client requests. The other systems do not scale as well because each node is a leader for n shards and a follower for $2\times n$ shards. Thus, two thirds of a node's resources are used by follower processes. Furthermore, in Mu, uKharon and APUS not all threads are used for processing client requests. For each shard process, Mu uses 4 threads on the leader and 4 on each follower, out of which only one of the leader's threads processes client requests. APUS and uKharon use 2 threads on the leader and every follower per shard. DARE is a single-threaded system and thus it scales to a larger number of shards. Each DARE shard uses one thread on the leader and each follower.

7 Additional Related Work

Consensus optimizations. Numerous software-based approaches towards optimizing consensus protocols exist in current literature. EPaxos [24] is a leaderless replication protocol designed to optimize commit latencies. CURP [25] utilizes commutativity to improve replication speeds. While these optimizations improve performance, they still result in latencies in the hundreds of microseconds or millisecond ranges.

Network-accelerated consensus. FLAIR [26], NetChain [27] and NetPaxos [28] utilize programmable switches to accelerate consensus. However, FLAIR only optimizes read operations while NetChain and NetPaxos are only suitable for systems with data sizes of a few megabytes or less.

Micro-scale optimizations. Several software optimizations have been developed to support microsecond range workloads, such as low-latency network stacks [29, 30] and intra-node schedulers [31, 32]. These optimizations are orthogonal to LoLKV, as we explore a novel design for micro-scale linearizable key-value storage.

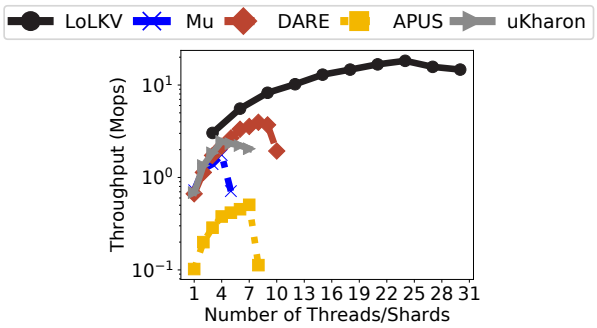


Figure 11: The throughput of different systems when varying the number of shards/threads with leaders distributed on three nodes.

Unreplicated RDMA KV Stores. The clients in Pilaf [13] and XStore [14] use RDMA to access server-side data structures to reduce latency. HERD [12] uses RDMA writes when clients submit requests to the server. The server polls these requests for processing. To leverage multi-core machines, scaling these systems requires key-space sharding and processing all requests for a shard by a thread/process.

Distributed Transactional Systems. DrTM [33] and DrTM+R [34] use RDMA and hardware transactional memory to support distributed transactions. RDMA-based CAS operations are used to replicate local transactions supported by HTM. These works are orthogonal to ours as we focus on using RDMA to build a logless replicated key-value store.

8 Conclusion

We present LoLKV, a novel logless key-value storage system which provides data replication and linearizability guarantees while avoiding the shortcomings of modern log-based key-value stores. LoLKV uses multi-threading to scale and handle client requests, applies operations directly to the underlying data structures of the object store, and uses RDMA to accelerate its operations. LoLKV followers are passive, enabling the efficient use of their resources. Our evaluation shows that LoLKV achieves significantly lower tail latency and higher throughput compared to the state-of-the-art systems. LoLKV shows that strong consistency guarantees can be achieved without being restricted to classical log-based consensus designs.

Acknowledgments

We thank Alex Kogan, Mina Tahmasbi Arashloo, and the anonymous reviewers for their insightful feedback. This research was supported by grants from Oracle and NSERC. Ahmed Alquraan is supported by an IBM PhD fellowship.

References

[1] Xinhui Tian, Rui Han, Lei Wang, Gang Lu, and Jianfeng Zhan. Latency critical big data computing in finance. *The Journal of Finance and Data Science*, 1(1):33–41, 2015.

[2] Stephen F. Elston and Melinda J. Wilson. Big data and smart trading. https://www.risktech-forum.com/media/download/61681/download.

[3] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 645–650, Boston, MA, July 2018. USENIX Association.

[4] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[5] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: Fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 94–107, New York, NY, USA, 2017. Association for Computing Machinery.

[6] Marius Poke and Torsten Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, page 107–118, New York, NY, USA, 2015. Association for Computing Machinery.

[7] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 599–616. USENIX Association, November 2020.

[8] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. uKharon: A membership service for microsecond applications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 101–120, Carlsbad, CA, July 2022. USENIX Association.

[9] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 305–319, 2014.

[10] Leslie Lamport. Paxos made simple. 2001.

[11] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *NSDI*, pages 79–94, 2019.

[12] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. *SIGCOMM Comput. Commun. Rev.*, 44(4):295–306, aug 2014.

[13] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA reads to build a fast, CPU-Efficient Key-Value store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, San Jose, CA, June 2013. USENIX Association.

[14] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based ordered Key-Value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 117–135. USENIX Association, November 2020.

[15] Rajkumar Buyya, Toni Cortes, and Hai Jin. An introduction to the infiniband architecture. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 616–632, 2002.

[16] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association.

[17] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):1–36, 2015.

[18] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, nov 2000.

[19] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, page 202–210, New York, NY, USA, 1989. Association for Computing Machinery.

[20] Diego Ongaro. LogCabin. *https://github.com/logcabin/logcabin*.

[21] Leslie Lamport. The TLA+ Home Page. *https://lamport.azurewebsites.net/tla/tla.html*.

[22] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of Cloudlab. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, pages 1–14, 2019.

[23] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154, 2010.

[24] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372, 2013.

[25] Seo Jin Park and John K Ousterhout. Exploiting commutativity for practical fast replication. In *NSDI'19*, pages 47–64, 2019.

[26] Hatem Takruri, Ibrahim Kettaneh, Ahmed Alquraan, and Samer Al-Kiswany. Flair: Accelerating reads with consistency-aware network routing. In *NSDI'20*, pages 723–737, 2020.

[27] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 35–49, 2018.

[28] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, pages 1–7, 2015.

[29] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[30] Dominik Scholz. A look at intel's dataplane development kit. https://api.semanticscholar.org/CorpusID:11483651, 2014.

[31] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. Efficient scheduling policies for Microsecond-Scale tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1–18, Renton, WA, April 2022. USENIX Association.

[32] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for usecond-scale tail latency. pages 345–360, 2019.

[33] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 87–104, New York, NY, USA, 2015. Association for Computing Machinery.

[34] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

Making Kernel Bypass Practical for the Cloud with Junction

Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez[†], Esha Choukse[†], Íñigo Goiri[†],
Sameh Elnikety[‡], Rodrigo Fonseca[†], Adam Belay
MIT CSAIL *[†]Azure Research – Systems* *[‡]Microsoft Research*

Abstract. Kernel bypass systems have demonstrated order of magnitude improvements in throughput and tail latency for network-intensive applications relative to traditional operating systems (OSes). To achieve such excellent performance, however, they rely on dedicated resources (*e.g.,* spinning cores, pinned memory) and require application rewriting. This is unattractive to cloud operators because they aim to densely pack applications, and rewriting cloud software requires a massive investment of valuable developer time. For both reasons, kernel bypass, as it exists, is impractical for the cloud.

In this paper, we show these compromises are not necessary to unlock the full benefits of kernel bypass. We present Junction, the first kernel bypass system that can pack thousands of instances on a machine while providing compatibility with unmodified Linux applications. Junction achieves high density through several advanced NIC features that reduce pinned memory and the overhead of monitoring large numbers of queues. It maintains compatibility with minimal overhead through optimizations that exploit a shared address space with the application. Junction scales to 19–62× more instances than existing kernel bypass systems and can achieve similar or better performance without code changes. Furthermore, Junction delivers significant performance benefits to applications previously unsupported by kernel bypass, including those that depend on runtime systems like Go, Java, Node, and Python. In a comparison to native Linux, Junction increases throughput by 1.6–7.0× while using 1.2–3.8× less cores across seven applications.

1 Introduction

Network-intensive applications have experienced remarkable performance improvements (*i.e.,* order of magnitude better tail latency and throughput) from kernel bypass systems [6, 15, 28, 29, 37, 43, 44, 67]. Their key idea is to map network queues into userspace, so applications can communicate directly with the NIC and avoid kernel overheads.

Junction is a new kernel bypass system that targets cloud applications (*e.g.,* microservices, serverless, etc.). Like previous kernel bypass systems, Junction delivers significant performance improvements, including higher throughput and greater CPU efficiency, as well as order of magnitude reductions in tail latency relative to traditional OSes. At the same time, Junction is the first kernel bypass system that retains compatibility with unmodified Linux binaries and is capable of achieving high density (*i.e.,* the ability to scale to thousands of instances on a machine). Junction delivers these benefits while maintaining strict isolation between applications with a narrower attack surface than existing cloud isolation schemes.

Prior kernel bypass systems make compromises that render them impractical for use in the cloud. For example, they require dedicated, busy-spinning cores and pinned memory, so very few instances can be packed on a machine. Moreover, they make significant changes to the programming model [6, 14, 21, 33, 52] that break compatibility and sacrifice the enormous investment made in existing software. Finally, most kernel bypass systems provide no isolation of their own, so they must be combined with virtualization—and its associated overheads (*e.g.,* VM exit costs, nested page tables, etc.)—to be deployed safely in a cloud setting.

Junction is able to retain the full performance benefits of kernel bypass without these compromises through a set of design contributions that target isolation, density, and compatibility. To achieve strong isolation while avoiding virtualization overheads, Junction runs each instance inside a normal Linux process and installs a filter that limits access to system calls. Within an instance, Junction runs as a library that shares an address space with the application. Because Junction is able to build all of its OS abstractions on top of kernel bypass hardware (*e.g.,* NIC queues, CPU features, etc.) it requires only minimal interactions with the kernel, just enough to enable resource multiplexing (≈ a dozen system calls).

To achieve high density, Junction efficiently multiplexes both cores and memory. For cores, Junction builds upon prior work on a dedicated scheduler core, but overcomes the previously unaddressed challenge of scaling to a large number of instances. To do so, Junction makes novel use of a NIC hardware feature that delivers packet arrival notifications on a dedicated queue instead of requiring each receive queue to be polled individually. For memory, Junction employs a variety of new techniques to reduce the footprint of packet buffers, including configuring NIC hardware to share a queue of receive buffers across multiple cores, as well as allowing multiple packets to be posted in each receive buffer. Junction also securely exposes the Linux page cache to share read-only memory across instances.

To achieve Linux compatibility, Junction provides its own implementation of the Linux Kernel system call interface. This was challenging to do in a way that maintains the performance benefits of kernel bypass. Junction exploits the fact that it runs in the same address space as the application to unlock optimizations that minimize the cost of compatibility. For example, Junction safely converts system calls into function calls, avoids transient execution mitigations,

System	Density			Compatibility				
	Mem. overhead (single core)	Mem. overhead per instance per core	Max instances (128GB RAM)	Ported Application	Lines of code			Compatible w/ existing clients
					Added	Removed	Modified	
eRPC [30]	24 MB	24 MB	# of cores	Masstree	551	0	0	✗
Demikernel [67]	167 MB	–	# of cores	Redis	926	819	213	✓
Caladan [15]	648 MB	9 MB	200	Memcached	393	539	637	✓
Junction	29 MB	0.47 MB	3,500	*No porting*	–	–	–	✓

Table 1: Existing kernel bypass systems require large amounts of dedicated host resources and invasive changes to applications.

accesses arguments directly without copying them, exploits undefined behavior to eliminate locking, and uses vector instructions without the need to save and restore register state. Junction also provides many OS features that are missing in existing kernel bypass systems (*e.g.,* signals, thread-local storage, randomness, file systems, timers, etc.) but are crucial to supporting cloud applications. It maintains a kernel bypass approach to delivering these features by exploiting modern CPU extensions to avoid traps into the kernel.

Results. Junction helps to pave a path toward practical deployment of kernel bypass in the cloud by showing it is possible to deliver high performance without sacrificing security, density, or compatibility. For example, Junction can run unmodified Linux binaries while matching or exceeding the performance of three state-of-the-art kernel bypass systems that require significant code changes. Moreover, Junction's reliance on kernel bypass hardware allows it to reduce the number of system calls in its attack surface by 69%–87% relative to two security-focused library OSes, and Junction's buffer management and queue polling optimizations allow it to pack thousands of instances on a machine. Finally, Junction is the first system to bring the performance benefits of kernel bypass to unmodified applications with complex language runtimes (*e.g.,* Python, Node, Go, Java, etc.). It improves throughput by 1.6–7.0× and reduces CPU use by 1.2–3.8× for seven applications relative to Linux. Junction is available as open source software: https://github.com/JunctionOS/junction.

2 Background & Motivation

Kernel bypass systems eliminate the kernel from the network datapath, and replace it with an optimized user-level networking stack that communicates directly with the NIC. In this section, we first discuss why the existing approach to kernel bypass has shortcomings that hinder adoption, especially in a cloud setting where density and compatibility are crucial issues. Next, we discuss current progress in making kernel bypass more general purpose. Finally, we discuss how a lack of security further compounds these issues.

Density challenges. Cloud providers commonly pack many instances on a machine to improve density [16, 55, 62, 68]. This is necessary because latency-sensitive applications have variability in their demand for resources, and utilization can only be kept high by filling in idle resources with best-effort applications. Furthermore, in serverless environments, it is typical for thousands of instances to remain active on a single machine to prevent cold start delays [1].

Unfortunately, kernel bypass systems today can only support a limited number of instances on a machine (left side of Table 1). One major problem is the widespread use of busy spinning and dedicated cores [30, 67]. Because this approach requires a minimum of one core per instance (and often many more), the maximum number of instances is limited by the number of cores.

New approaches to CPU scheduling that speed up core allocation can overcome this limitation and eliminate waste from busy spinning without sacrificing tail latency [15, 42, 50]. However, these systems rely on a dedicated core to make scheduling decisions, so their scalability is still limited. For example, Caladan is unable to scale beyond a few hundred instances because of bottlenecks in its scheduler core.

Finally, the memory footprint of kernel bypass systems is also a significant barrier to achieving high density. This is especially true for kernel bypass systems that handle TCP connections, like the Demikernel and Caladan, because they cannot guarantee the application will consume packet buffers the moment they arrive (*e.g.,* what if they arrive out of order?), so they must reserve a significant number of buffers to avoid packet drops. There is also an additional per-core cost for packet buffers in multicore kernel bypass systems because each core must post enough receive buffers to handle a worst case burst in traffic, which could be uneven across cores.

Compatibility challenges. Ideally, a kernel bypass system should support unmodified binaries. Existing kernel bypass systems instead require applications to be ported by developers (right side of Table 1). This is a significant barrier to adoption. For example, at the time of writing, three recent, state-of-the-art kernel bypass systems (eRPC, Demikernel, and Caladan) support just a handful of applications. Moreover, all of these applications are key-value stores written in C or C++, which are less challenging to port than typical cloud applications. A recent survey of serverless functions found that Node and Python were among the most popular languages [57], and both require a full language runtime with a complex set of OS dependencies that cannot be met by existing kernel bypass systems.

Breaking compatibility is not easy in the cloud because of

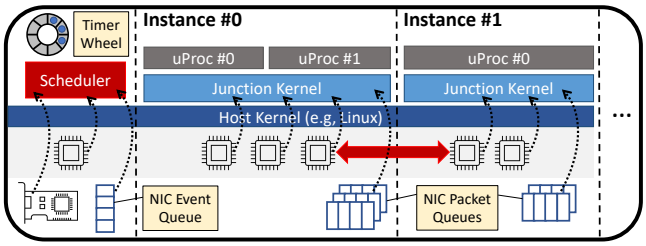


Figure 1: Junction's system architecture.

the significant investment developers have made in existing software. One obvious solution would be to focus on rewriting just the most popular applications. Unfortunately, this would have little impact on efficiency overall because, even together, these applications do not account for a large enough fraction of overall resource usage [31].

Making kernel bypass general purpose. There have been several recent efforts to make kernel bypass more general purpose. For example, normally kernel bypass systems use run-to-completion to optimize for short requests [6], but this leaves them vulnerable to high tail latency when there is dispersion in request service times. Shinjuku [29] and Persephone [9] solve this through efficient, fine-grained preemption and steering of short requests to separate cores respectively. Many kernel bypass systems also use "share nothing" designs that harm tail latency under load imbalance, an issue that ZygOS addresses through work stealing [47]. Demikernel provides unifying abstractions on top of different hardware backends (*e.g.,* RDMA vs. Ethernet) to reduce developer effort [67]. Finally, Arachne [50] and Shenango [42] show that threading can be made fast enough to be used with kernel bypass networking. Junction adopts several ideas from these systems while solving the previously unaddressed challenges of density and compatibility.

Security challenges. Most kernel bypass systems eschew isolation and must be run as root. As a result, they depend on other isolation mechanisms to be deployed safely in the cloud. The most viable option is to run each instance in a separate VM, but this increases overheads including extra TLB misses, VM exit costs, and a larger memory footprint caused by the guest kernel. In most cases, VMs also cannot take advantage of the page cache, which further limits their density.

3 Junction Overview

Figure 1 provides an overview of Junction and highlights its main components. Junction is designed to handle thousands of instances on a machine. An *instance* is an isolated container that runs one or more application binaries. From the host kernel's perspective, this container consists of a single process (called a kProc) with a fixed set of threads (called kThreads) that are statically initialized at startup time. The kThreads are scheduled on cores by a centralized scheduler (left side of Figure 1). An instance can load and run multiple binaries in its shared address space (placing each at different offsets in virtual memory). Each binary within an instance runs in a userspace process abstraction called a uProc.

A copy of the *Junction kernel* runs inside each instance and shares an address space with its uProcs. It directly handles system calls from uProcs and provides OS abstractions (*e.g.,* threading, networking, filesystems, signals, etc.) in userspace, similar to a library OS [10, 46]. The Junction kernel supports the Linux system call interface so that it can run existing software without modifications.

The Junction kernel uses kernel bypass hardware (both networking queues and CPU features) to provides its OS abstractions. As a result, most system calls can be handled entirely in userspace and it is only necessary to make system calls into the host kernel to multiplex resources (*i.e.,* cores and memory)—all other host kernel system calls can be blocked. Shifting OS functionality into userspace improves performance by reducing the frequency of boundary crossings, and limits the attack surface by allowing untrusted programs to exercise only very small parts of trusted host kernel code.

Throughout this paper, we assume Linux is used as the host kernel, but any standard OS could serve this purpose. Junction can coexist with normal Linux processes that do not use Junction. As a result, Junction can take full advantage of existing debugging and profiling tools; and the control plane and management functions of a standard Linux environment.

Networking and communication. Like other kernel bypass systems, Junction instances are provisioned with one NIC send and receive queue pair per kThread. This improves performance by allowing concurrent access to the NIC without synchronization. The Junction kernel provides a high-performance TCP/IP and UDP networking stack, which enable uProcs to communicate with the outside world. uProcs within the same instance can communicate with each other using standard inter-process communication (IPC) primitives (*e.g.,* pipes), but different instances on the same host may only communicate via loopback networking through the NIC.

Threading. The Junction kernel includes a high-performance, user-level threading library that uses work stealing to balance light weight user-level threads (uThreads) across kThreads. uProc threads (*i.e.,* those created when starting or via `clone3()`) are mapped to uThreads. uThreads are also used for various internal tasks like network protocol processing. Each kThread runs a scheduling loop that polls local queues for packets and timeouts, and runs pending uThreads.

Core scheduling. Junction relies on a microkernel-style scheduler to make core allocation decisions [15, 23, 42, 48]. It runs on a dedicated core and busy polls control signals to decide how and when cores should be allocated to each instance (indicated by the red double arrow in Figure 1). Instances can use as few as zero cores when idle, or more than one if demand justifies it (up to a per-instance limit). For each instance, the scheduler monitors timer expirations and queueing delays in both thread and network queues, which are made visible

to the scheduler through shared memory. When the scheduler grants a core to an instance, it selects one of its idle kThreads and pins it to the core for the duration of the grant.

In addition to performing core allocation, the scheduler assists the threading library in implementing fine-grained timeslicing by sending user IPIs (UIPIs) [25] to cores where a running uThread has exceeded its timeslice quantum. This ensures that all uThreads can make progress, and that packet queues are drained in a timely manner. It is also beneficial for reducing tail latency when service time dispersion is high [29]. As an optimization, these interrupts are only sent when queued packets or runnable threads are waiting to be processed.

Because Junction aims to support thousands of instances, it employs novel techniques to ensure that control signals can be monitored in a scalable way. A *timer wheel* keeps track of the next timeout for each instance and a *NIC event queue* provides notifications of packets arrivals. These reduce how often the scheduler has to poll shared state inside each instance to determine if it could benefit from additional cores. We discuss these optimizations in more detail in §5.

4 Security

The conventional wisdom is to use virtual machines as the isolation boundary for the cloud to reduce interactions between untrusted code and the host kernel. However, virtualization still exercises a large amount of trusted code in the host kernel, resulting in a significant attack surface [3, 65]. Junction, by contrast, delivers OS abstractions directly on top of kernel bypass hardware, significantly reducing its reliance on the host kernel. In this section, we discuss Junction's security design in more detail.

4.1 Threat Model

We assume each instance can run arbitrary and potentially malicious code, so strong memory isolation is needed between instances and the host kernel. We are particularly concerned about a malicious user exploiting a bug in the host kernel through a system call or VM exit. Today's host kernels have wide attacks surfaces that can exercise millions of lines of code, and transient execution attacks potentially expose additional vulnerabilities in this code [4]. This poses a large security risk in cloud deployments [1, 17].

The Junction kernel, by contrast, runs as a separate copy in each instance. Each copy shares an address space and has fate sharing with the uProcs it handles, but is strictly isolated from other instances. Therefore, if a uProc corrupts its memory or exploits a bug in a system call implementation, it can only harm its own availability. Through the same reasoning, validating system call arguments or preventing transient executions attacks is not required for isolation with the Junction kernel. This unlocks many opportunities for optimization that are not available to the host kernel (§6.2). uProcs that are mutually trusting can run in the same instance at lower overhead, or in separate instances if isolation is needed.

Category	Syscalls
CPU scheduling	yield_core()
Memory management	mmap(), munmap(), madvise(), mprotect(), mremap()
Loading binaries	open(), close(), pread64()
Logging (optional)	write()
Process management	exit_group()

Table 2: System calls that Junction requires from the host kernel.

The NIC is trusted and we assume it can provide network virtualization and packet scheduling across multiple instances. NICs with these capabilities are commonly deployed in public clouds today (*e.g.,* Microsoft Catapult [12] and Amazon Nitro [2]).

4.2 Host Kernel Isolation

Isolation mechanisms. Junction relies on kProcs, which are normal Linux processes, for isolation of its instances. Each kProc installs a strict seccomp filter that limits access to a restrictive set of system calls. Because Junction uses the page cache to improve density, an instance is given read-only access to a `chroot` jail directory with needed binaries and shared libraries. Junction instances also have direct access to the NIC through a set of queues and a dedicated page for MMIO doorbell writes to PCIe. The NIC is aware of distinct Junction instances, and provides each instance with its own queues, pinned memory, and doorbell page.

Allowed system calls. Junction requires only 11 system calls, as shown in Table 2. Two of these are used only rarely: `write()` is used for debug logging to `stdout`, and `exit_group()` is used only once when the instance exits. Similar to Caladan [15], Junction uses a single blocking system call, `yield_core()`, to yield and allocate cores. This call is provided by a custom Linux Kernel module, and coordinates with the scheduler—informing it when a kThread yields voluntarily. The module allows the scheduler to wake a kThread on a specific core and unblock it from its `yield_core()` call.

Junction requires five system calls to manage memory. This is important for density because instances rarely need all of the memory they reserve [1]. First, Junction relies on `mmap()` to allocate anonymous memory and to map files. Second, it relies on `mprotect()` and `munmap()` to change mapping permissions and remove mappings respectively. Third, `mremap()` is needed to adjust existing mappings. Finally, `madvise()` is exposed to provide hints to the Linux Kernel, such as releasing unused memory without modifying the VMA, which has lower overhead than `munmap()`.

Page cache. To minimize the attack surface, we initially considered a design where Junction did not access the Linux Kernel file system at all, instead delivering files over the net-

work (*e.g.,* 9P) or in memory. However, we realized that to achieve density, limited access to the Linux file system is necessary. This is because the host kernel is the only layer that can mediate access to the page cache, allowing different processes to share read-only mappings to the same disk blocks. Reducing the memory footprint in this way allows Junction to achieve higher density (§8.6).

Our goal, therefore, is to expose just enough of the file system to make it possible to leverage the page cache. Using the `chroot` jail, an instance can use `open()` to access allowed files in read-only mode and `close()` when done. `mmap()` is used to map the files into memory (*i.e.,* ELF segments), enabling use of the page cache. `pread64()`, which takes an offset and is designed to scale well across cores, is also made available to allow the contents of files to be read before they are mapped into memory. Before adopting its syscall filter, an instance scans the files and folders in its jail and caches the metadata (*e.g.,* file size), so it does not need access to any further file-system calls when running.

5 Optimizing for Density

Junction's goal is to deliver high networking throughput and low latency, like existing kernel bypass systems, while also packing significantly more instances on a machine. This required us to resolve several problems related to the use of a large number of NIC receive queues.

Normally, kernel bypass systems assign a separate receive queue to each core in order to avoid synchronization overheads so they can scale linearly with the number of cores. Although Junction differs in that it adjusts cores dynamically based on load, it still must assign enough receive queues to each instance so that one will be available for each kThread that might be running. As a result, the number of queues needed is the maximum number of cores per instance times the number of instances.

Modern NICs can easily scale to thousands of queues, but using them to pack many instances on a machine still poses significant challenges. First, buffer memory consumption is a key limit for density because each receive queue must post enough buffers to accommodate a worst-case burst in arriving packets, which is exacerbated by the unevenness in traffic across cores. Second, the cost of polling every queue in the core scheduler becomes prohibitive, with cache pollution causing a performance collapse. We now discuss our solution to each of these problems in more detail.

5.1 Minimizing Buffer Memory Consumption

Existing kernel bypass systems maintain large pools of memory for buffers that are used to send and receive packets through the NIC. These buffers must be available to the NIC for direct memory access, so the backing memory must be *pinned* to prevent the host kernel from swapping it out. Pinning buffers at the time of sending/receive packets is prohibitively expensive, so it is instead better to maintain a pool of pinned buffers. Sizing this pool requires consideration of several factors, including the number of open connections (*i.e.,* the total outstanding send and receive windows for a protocol like TCP), the worst-case delay in packet processing time, and the network round trip time. To handle variability in processing time and to absorb bursts, it is common to keep receive rings full with buffers so that packets are not dropped.

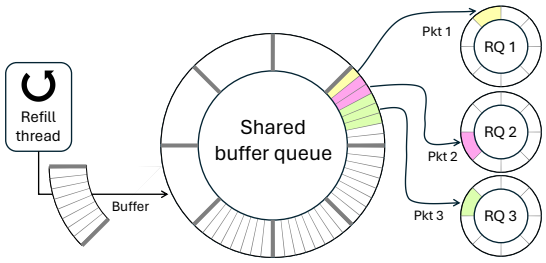


Figure 2: Junction reduces memory use through (A) a shared buffer queue that supplies per-core receive queues (RQs), and (B) packing many packets in each buffer (even when delivered to different RQs). A refill thread replenishes buffers.

Kernel bypass systems often provision pinned memory well in excess of the minimum amount needed. One major factor that contributes to this is the use of per-core receive queues. Traffic is distributed among receive queues using RSS, which is susceptible to skew and can lead to bursts on some queues. Each queue must have enough posted buffers to ensure that it can handle any traffic distributed to it. For example, kernel bypass network stacks can drive a single TCP flow at speeds greater than 100Gb/s. Because such a flow would be hashed to one receive queue at random, each queue must have enough buffers available to handle this high rate of traffic, despite the fact that only one queue's buffers will be consumed. Thus the amount of buffer memory needs to scale linearly with the number of cores an instance may be using.

A second factor that contributes to wasted memory is the need to support variably-sized packets. Normally, each packet that arrives consumes a single posted buffer, which must be at least MTU-sized. Small packets consume only a fraction of these buffers, so to ensure that enough bytes are available to buffer small packets that arrive at high rates, the buffer pool must be made proportionally larger.

As shown in Figure 2, Junction takes advantage of recent NIC hardware to overcome both of these problems. First, it uses a per-instance shared buffer queue to post buffers instead of posting them separately in each receive queue. This allows posted buffers to be shared amongst the receive queues, eliminating the need to scale the buffer pool with the number of cores. Second, it avoids fragmentation in large packet buffers by allowing many packets to be placed consecutively in each buffer, reducing the minimum memory consumed per-packet from MTU size (between 1500B to 9000B) to 256B.

Using a shared buffer queue requires coordination across cores, which can add overhead and limit scalability. The host cannot refill a slot in the buffer queue until the NIC has fin-

ished writing to it, which is detected by tracking arrivals at each receive queue. Additionally, because many packets can be stored in a single buffer, a buffer cannot be reused until all of its packets are freed. This is especially challenging because packets from a single buffer can be spread across many receive queues, requiring coordination across cores when freeing packets and refilling the shared buffer queue.

Prior work on ShRing [45] proposed hardware modifications to the NIC to enable efficient coordination. However, Junction reduces synchronization overhead with a software-only approach by using per-core reference counters for each slot in the shared buffer queue and each buffer in the buffer pool (*i.e.,* no cachelines are shared). Because buffer sizes can be large (we picked 16KB), the shared buffer queue can be kept small, resulting in low per-core memory overhead (about 512B). A high-priority *refill thread* is responsible for managing the shared buffer queue: it scans the counters and replenishes queue slots with free buffers. We provide more details on our approach and how it compares to ShRing in Appendix B.

5.2 Scalable Queue Polling

Using a spinning core for rapid core scheduling eliminates the need for each instance to spin-poll its own receive queues, allowing idle cores to be reallocated to applications that need them (§2). The scheduler polls shared memory locations in a loop to monitor queueing delays across each instance's network receive queues, thread runqueues, and timers. With large numbers of instances, the polling loop can take a long time to complete, leading to delays in wakeups and increases in latency. Furthermore, polling many locations pollutes the cache for the scheduler, leading to additional slowdowns. Junction avoids polling thread runqueues for inactive applications, but must still track pending timers and arriving packets as both may warrant an immediate core allocation for an idle instance.

Junction makes two main modifications to the scheduler core model to dramatically improve its scalability by reducing the amount of memory that the scheduler must inspect in each scheduling pass. First, it uses a set of NIC features in a novel way to avoid continuously polling idle network queues. Junction allocates a single *event queue* and a dedicated doorbell page for the scheduler core. Each time the scheduler observes an empty receive queue, it *arms* the queue by marking the index of the current head pointer and writing to the doorbell. When a packet arrives on an armed queue, the NIC writes an event into the event queue and disarms the queue. The scheduler continually polls the event queue and can react immediately when a packet arrives at an idle queue. This feature is available on modern Mellanox NICs.

Expiring timers may also require cores to be allocated to idle instances. High resolution timers are important for datacenter workloads; for example, they may used to detect when TCP segments need to be retransmitted, or an RPC has failed. In order to ensure that instances with timers are woken with minimal delay, Junction's scheduler employs a second optimization: a high resolution (16 μs) hierarchical timer wheel [61]. The timer wheel allows the scheduler to ignore instances with timers that expire far out in the future and monitor only instances that are either active or have immediately pending timeouts.

Subsystem	Hardware Feature	Syscall Alternative
Networking	NIC Queues	socket(),recv(),send()
Randomness	`RDRAND`, `RDSEED`	getrandom()
Threading (TLS)	`WRFSBASE`	arch_prctl()
Signals	`SENDUIPI`, `XSAVEC`	tgkill()
	`XRSTOR`, `UIRET`	rt_sigreturn()

Table 3: Kernel bypass hardware features used by Junction.

We also optimize the memory footprint of the data structures used by Junction's scheduler to ensure that the state from each instance occupies as few cache lines as possible. Additionally, state is arranged to avoid false sharing, *i.e.,* state needed only when an application is active is kept separate from state needed when an instance is idle. Together, these optimizations overcome the scalability bottlenecks of a centralized core scheduler and enable Junction's scheduler to manage thousands of active instances without latency issues.

6 Linux Compatibility

In line with experiences reported by other researchers, we found that it was tractable for a small team to implement enough of the Linux interface to run a wide range of applications [20, 22, 46]. Constraining our goal to only supporting cloud applications made this easier. For example, we could ignore desktop features like graphics, input devices, and sound, which would have required significantly more developer effort. In addition, most cloud applications are built for specific runtime systems and do not perform system calls directly. Therefore, targeting all the system calls needed by a particular runtime can enable a broad swath of compatibility. For example, we found Junction could run any Go program after implementing the set of system calls needed by its runtime.

We had to overcome two challenges to achieve Linux compatibility. First, we had to provide OS features that are not available in prior kernel bypass systems, but are necessary for cloud applications. Second, we had to minimize the overhead of compatibility in order to not squander the performance benefits of kernel bypass. We address the first challenge by shifting OS abstractions into userspace and by building them on top of kernel bypass hardware (Table 3). We address the second challenge by exploiting fate sharing and a shared address space to unlock performance optimizations. We discuss each solution in more detail next.

6.1 Adapting OS Features to Kernel Bypass

Loader and multiprocess support. The Junction kernel includes its own ELF loader to load uProcs inside an instance. The ELF loader is invoked automatically at startup to load the first uProc—a path to the program image is provided as a configuration parameter. The ELF loader can also be invoked later by the `execve()` system call, which enables loading of additional uProcs. During ELF loading, Junction creates a new uThread and populates its stack with environment variables, arguments, and an auxiliary vector. The auxiliary vector contains several important parameters that are needed to emulate a Linux process environment [35].

In an earlier prototype of Junction, the Junction kernel was instead deployed as a library that was linked with the application, similar to a Unikernel [24, 34]. We decided against this approach, however, because it required compiling a new binary for each program. This breaks compatibility with existing Linux binaries and also makes it more difficult to support multiple processes within an instance.

Multiprocess support is an important feature for some cloud applications (*e.g.,* microservices) because they rely on sidecars for RPC handling, logging, and other services. Sidecars are normally trusted by the application, so it is acceptable to run them together, potentially without memory isolation [56]. In Linux, a new process is created via `fork()`, which spawns a separate address space. However, Junction is a single address space OS, so it cannot fork to create new uProcs.

To launch multiple uProcs, Junction instead relies on `vfork()`, an optimized version of `fork()` that delays the creation of a new address space until `execve()` is called. Linux uses this to fuse the fork operation with the loading of the program, eliminating the need to clone the page table. However, Junction transparently co-opts the `vfork()` + `execve()` sequence to provide a different behavior. Instead of creating a new address space, it finds an empty location in the existing address space and loads the program there, allowing it to support multiple uProcs in one instance. To avoid collisions, a program must work correctly in any location in the address space, so at most one uProc can be compiled without position-independent code (PIC) enabled. Fortunately, for security reasons, PIC is enabled by default in most datacenters.

Threading. User-level threading packages are gaining traction because of their increased performance (*e.g.,* Java's Project Loom [49] and the Go Runtime [39]). Junction brings the same benefits to unmodified binaries by shifting all threading operations (*e.g.,* creating threads, acquiring mutexes, context switching, etc.) into userspace instead of going through the host kernel. Junction maintains a separate uThread runqueue in each kThread and uses packet arrivals, timeouts, signals, and other events to wake uThreads.

Junction provides threading support at two layers of abstraction. First, it supports the low-level Linux system calls that are required for threading (*e.g.,* `futex()`, `clone()`, etc.). This is needed to achieve compatibility with programs that use nonstandard threading libraries. Second, for greater efficiency, it overrides glibc's pthread library, and provides a custom implementation that is integrated directly with Junction and does not make use of these system calls. For example, `futex()` must normally do a hash table lookup to find if a thread is blocking on an address, but Junction can instead reference the mutex object directly to find the blocking uThread. Another challenge not addressed in prior kernel bypass systems is support for thread-local storage (TLS). Junction's solution is to rely on the `WRFSBASE` instruction to switch between thread-local regions during each uThread context switch. Without this instruction, performance would be significantly worse as the host kernel's `arch_prctl()` system call would have to be invoked at each context switch (§8.6).

Signals. Surprisingly, we found that many cloud applications depend on signals for normal operation. For example, Go uses signals to preempt and reschedule Go Routines, and the Hotspot JVM uses signals as an optimization to avoid explicit NULL pointer checks. This reflects two separate forms of signals supported by Linux: 1) those that are sent internally by an application (*e.g.,* via `tgkill()`) and 2) those that are generated by CPU exceptions (*e.g.,* page faults).

The difficulty in supporting signals is that their behavior is highly customizable (*e.g.,* setting handlers, masking signals, using alternate stacks, etc.). However, exposing `sigaction()` and `sigaltstack()` through the host kernel would significantly widen the attack surface. Signals also compose poorly with Junction's threading layer, because signals must normally wake or preempt a specific uThread, but the host kernel is only aware of kThreads.

Instead, Junction uses UIPIs [25] to reduce the involvement of the host kernel. When a uThread invokes `tgkill()` to send an internal signal, the Junction kernel uses the `SENDUIPI` instruction to send a UIPI (if preemption is necessary). We discuss our implementation with UIPIs further in Appendix A.

CPU exceptions, however, still requires involvement from the host kernel because the CPU does not support user-level handling of these faults. Junction statically configures the host kernel with handlers for every possible CPU exception signal (using an alternate signal stack) before dropping privileges. When a uThread triggers a CPU exception, the host kernel sets up a trapframe on an alternate stack and invokes Junction's signal handler. Thankfully, UIPI's `UIRET` instruction obviates the need for the host kernel's `rt_sigreturn()` system call since it can atomically restore stack and instruction pointers. Though a system call is typically needed to unblock signals that are masked during delivery, Junction configures its Linux signal handlers to never alter the signal mask. This is appropriate because CPU exceptions cannot be masked.

When an interrupt or Linux signal is delivered, Junction uses its internal knowledge of uProcs and their signal configuration to route the signal to the right handler, which pushes the signal's trap frame onto a uThread's stack. This potentially

includes waking a uThread or preempting a running uThread.

Other OS features. Junction provides support for networking sockets, IPC (*e.g.,* `pipe()` and loopback networking), waiting for events (*e.g.,* `select()`, `poll()`, `epoll()`, and `eventfd()`), a virtual filesystem, memory management (*e.g.,* `mmap()`), time keeping, and many more OS features not found in prior kernel bypass systems. We adopted several novel strategies to reduce overheads, discussed next in more detail.

6.2 Performance Optimizations

System call handling. The main mechanism for intercepting system calls in Linux is `seccomp`, which can generate a signal for each system call it intercepts. Junction uses this mechanism as an occasional fallback, but we found that it had a prohibitive level of overhead for intercepting all system calls. A more efficient alternative would be to patch occurrences of the `SYSCALL` instruction so they jump directly into the Junction kernel. Researchers recently discovered a clever trick to support this without exceeding the instruction length of `SYSCALL`, eliminating the need for binary rewriting [66].

However, Junction uses a different strategy to squeeze out even more performance. When a program is loaded, Junction's ELF loader transparently replaces glibc with a modified version. We found that nearly all system calls are performed through glibc so this was a good place to intercept them. The modified library is designed to call into Junction during each system call. This includes the use of a trampoline page that finds the location of Junction, which is randomized for ASLR.

One key benefit to this approach is that it allows each system call to be invoked like a normal function call with standard calling conventions. This is favorable for performance because some general purpose registers and all floating point and vector registers can be safely clobbered. As a result, unlike the Linux Kernel, the Junction kernel can be compiled with all optimizations enabled, including those that use vector instructions. This also means that significantly less register state has to be saved and restored when context switching between threads that are blocked. However, preempted threads still require all state to be restored. Because of fate sharing, there is also no need to apply transient execution attack mitigations during system calls [4]. Instead, these are only necessary when entering the host kernel.

Reducing compatibility overheads. Junction further reduces overheads by changing the way system calls are implemented. First, Junction does not need to mitigate time-of-check to time-of-use (TOCTOU) attacks, again because of fate sharing. As a result, Junction does not copy system call arguments. Second, the UNIX standard has many examples of undefined behavior. In the Linux Kernel these have to be implemented carefully to avoid compromising security, but in Junction they can be implemented in whatever way achieves the best performance.

A good example of this opportunity is that if a file descriptor is closed while `select()` is monitoring it, the behavior is undefined. The Linux Kernel still requires extra locking to prevent race conditions. On the other hand, Junction allows these race conditions to happen and avoids the cost of locking because it assumes a correctly written program will never trigger undefined behavior. However, Junction must still perform all standard argument checking (*e.g.,* is a file descriptor valid) because Linux programs depend on these behaviors.

7 Implementation

Junction is implemented in about 12,000 lines of C++23 code and runs on modern x86 CPUs. It is linked against Caladan's runtime (14,000 LOC), which it uses as a low-level library for networking and threading routines. Caladan provided useful support for centralized core scheduling. However, we had to heavily modify it to scale to more instances and to run under our restricted host kernel interface. Moreover, we completely replaced its `mlx5` driver (for modern Mellanox NICs) to enable shared buffer queues and multi-packet receive buffers (5,000 LOC); and we modified its kernel module to support UIPIs (500 LOC). The modified NIC driver replaces Linux's ibverbs and tightly integrates with the scheduler, allowing it to expose the NIC event queue.

Linux compatibility. Junction's current implementation supports 126 Linux system calls. We found this subset sufficient to run language frameworks including Python, Go, Node.js, and Java, as well as a variety of applications written in C, C++, and Rust. With the exception of Go, Junction runs all of these as unmodified Linux binaries. Go programs belong to a rare class of applications that make most of their system calls outside of libc (the only one we encountered). To deliver the best performance, we added a new OS target to the Go compiler that uses function calls and our trampoline instead of system calls. However, `seccomp` can still handle system calls even for unmodified Go binaries.

8 Evaluation

Our evaluation aims to answer the following questions:

1. How does Junction's performance compare to state-of-the-art kernel bypass systems (§8.2)?
2. How many active instances can Junction pack on a machine (§8.3)?
3. Can Junction achieve compatibility with unmodified cloud applications (§8.4)?
4. Can Junction be securely deployed in a multi-tenant cloud (§8.5)?
5. What factors contribute to Junction's better performance and higher density (§8.6)?

8.1 Methodology

Experimental setup. Most experiments are run on a server with an Intel Xeon 6354 3.6 GHz 18-core CPU, 64GB of RAM, and a 200GbE Mellanox ConnectX-6 NIC. Scaling

	Files	Net.	Mem.	Process	Threads/ Sync.	Signals	Rand.	Timers	Multi-threaded	Garbage Collected	Kernel Time (%)
Memcached (C)	6	20	4	12	4	2	1	1	✓	✗	86.4
Redis (C)	11	14	5	9	5	3	1	1	✗	✗	84.2
Masstree (C++)	4	14	5	6	4	4	0	1	✓	✗	83.2
nginx (C)	5	9	4	10	2	0	1	1	✗	✗	76.1
Node.js HTTP	7	12	6	14	4	2	1	0	✗	✓	44.7
Python HTTP	11	10	4	11	3	1	1	0	✗	✗	63.5
Go HTTP	4	9	4	6	4	3	1	1	✓	✓	55.3
Rocket (Rust)	7	13	5	7	4	0	1	0	✓	✗	44.7
Tomcat (Java)	15	18	5	15	5	4	1	2	✓	✓	51.6

Table 4: Characteristics of evaluated applications, including unique system calls used grouped by kernel subsystem, whether an application is multithreaded or uses garbage collection, and ratio of time spent in the kernel (when running in Linux).

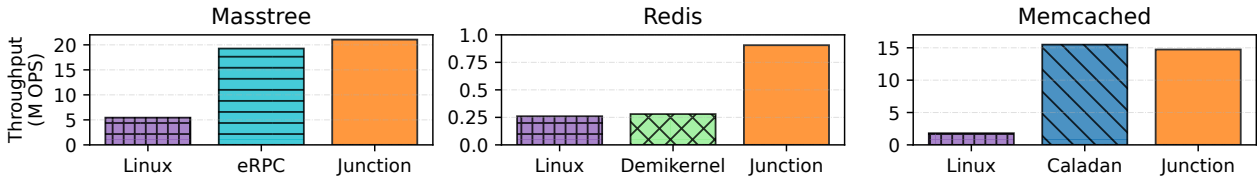


Figure 3: Performance comparison of Junction with state-of-the-art kernel bypass systems. Junction offers higher throughput than eRPC and Demikernel and is competitive with Caladan despite requiring no code modifications.

experiments run on a server with an Intel Xeon 5420+ 2.7GHz 28-core CPU and 128GB of RAM. Unless otherwise noted, we use a set of client machines connected to a 100GbE switch to generate load. The server runs Linux Kernel version 6.2, with the default mitigations enabled for CPU vulnerabilities. We use an open-loop kernel bypass load generator with Poisson arrivals [42] for latency measurements.

Systems. We compare Junction to three state-of-the-art kernel bypass systems: eRPC [30], Demikernel [67], and Caladan [15]. We also compare Junction to two state-of-the-art cloud isolation systems: Firecracker [1], a micro-VM isolation system, and gVisor [17], a secure container isolation system that also implements the Linux system call interface in userspace. Finally, we show performance relative to native Linux. We made every effort to tune each system for its maximum performance, and ensured that performance matched what is reported in other studies.

Applications. Table 4 characterizes the applications that we use throughout the evaluation, which include a range of web servers and various in-memory database and key-value stores.

8.2 Comparison to other kernel bypass systems

Figure 3 shows that Junction delivers performance to unmodified binaries that is on-par with or better than existing kernel bypass systems that require modifications to applications. The set of applications that have been ported to existing kernel bypass systems is limited and disjoint, making it difficult to use the same application to evaluate each system. Instead, we select one of the ported applications that was used originally to evaluate each kernel bypass system and compare to an unmodified Linux binary running in Junction and Linux. Table 1 shows the porting effort required for each of them. Except for Demikernel, which doesn't support multiple cores, we configure each system to use up to 16 cores.

eRPC. We compare against eRPC using the in-memory database called Masstree [36]. eRPC's port of Masstree replaces its included TCP/UDP server with its own wrapper around the database and a custom wire format. We compare the performance of eRPC running with RoCE versus Junction and Linux using standard TCP. We provision 128 client threads across 8 machines to issue a mixture of 50% GET and 50% PUT requests to the server (with four outstanding requests per thread). Because of eRPC's custom wire protocol, we use eRPC's client to measure eRPC's performance, and the Linux binary running in Junction to measure Linux and Junction. Junction achieves higher throughput (over 21 MOp/s) than eRPC (19.3 MOp/s), both significant improvements over the Linux baseline. We observe that eRPC spends up to 10% of its CPU time handling futexes while allocating memory, an overhead that Junction largely eliminates.

Demikernel. We benchmark Demikernel using Redis and its included redis-benchmark utility. We run the redis-benchmark client inside Junction to measure throughput of GET and SET requests over TCP connections. The Linux baseline achieves 260,000 RPS while consuming 1.5 CPU cores (this includes softirq time). Demikernel improves throughput by about 7% while using only 1 CPU core. Junction improves throughput by 248% over Linux, while also using only 1 CPU core. Junction's TCP stack is more efficient (*e.g.,* it uses a fast-path [32]), and we suspect this is the main contributor to its performance advantage relative to Demikernel in this setting.

Caladan. We use memcached [13] to compare performance to Caladan [15], with additional detail on tail latency in Fig-

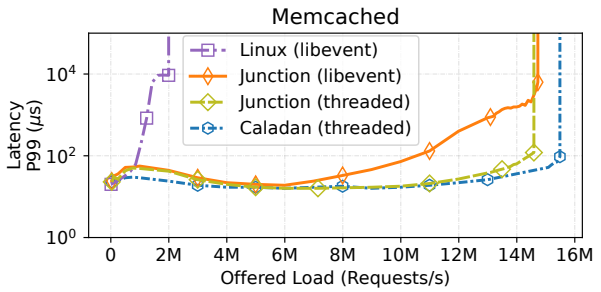


Figure 4: Latency for native memcached binaries (libevent) running in Linux and Junction, as well as two ported versions that replace libevent with per-connection threads (threaded) written against POSIX and Caladan interfaces. Junction can nearly match the throughput of Caladan in both cases, but the added overheads of libevent harm latency at higher throughputs.

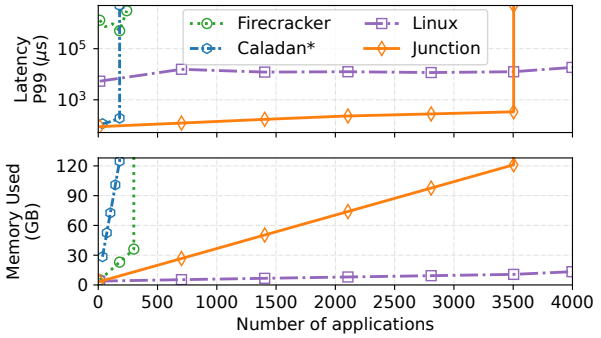


Figure 5: Performance and resource consumption while varying the number of single-threaded Rocket web servers packed into one machine. Load is fixed at 150K RPS and spread evenly across all instances. Junction supports up to 3,500 active instances while using 3.33× less memory per instance than Firecracker and keeping tail latency 35× lower than Linux.

ure 4. This comparison highlights the performance costs of using unmodified binaries because the two systems share a user threading library and TCP stack. We found that the while the native Linux binary in Junction could nearly match the throughput of Caladan, its tail latency degraded more quickly. This is because the Caladan version of memcached entirely removes `libevent` and `epoll()`, which were used to multiplex large numbers of connections across a smaller set of threads. We performed the same set of modifications to the Linux binary and evaluated its performance on Junction. The modified version (threaded) shows that Junction can match the latency profile of Caladan, suggesting that limiting the number of threads is important for OSes with high kernel crossing overheads, but can be detrimental otherwise due to head-of-line blocking. Thus, an optimal version of memcached for Junction would merely spawn a thread for each connection. Nevertheless, this result shows that Junction can offer dramatic improvement for low-latency applications even without modifications.

8.3 Density

We now demonstrate that Junction can densely pack many active instances on a machine and deliver low tail latency. Figure 5 shows an experiment where we provision increasing numbers of instances of a multi-threaded HTTP server written in Rust called Rocket. We use a host with 128GB of RAM. Each instance is provisioned with 8 threads. We offer a total load of 150,000 RPS to the machine, evenly divided across all instances. At each instance count, we show the aggregate tail latency experienced across all instances and the total memory consumption. We compare Junction to a Linux baseline and to Firecracker (designed to have a low per-instance memory footprint while providing an isolated VM environment). We also show *Caladan**, which mixes Junction with Caladan's network stack that is not optimized for density; this scales to only 180 instances. Linux consumes hardly any memory per instance (<1MB) and can scale far beyond Junction, but has poor tail latency. Firecracker's memory overhead is 8.6× lower than Caladan's but suffers from even poorer tail latency. Junction scales to 3,500 instances before running out of memory, with p99 latency below 350 μs, a 35× improvement over Linux.

8.4 Compatibility

We demonstrate that Junction achieves broad compatibility for cloud workloads by benchmarking a suite of HTTP servers written in several popular languages/frameworks. This collection includes (1) nginx [51], a popular load-balancer and proxy written in C, (2) Node.js's built-in web server [40], (3) a simple web server written in Python [60], (4) Go's built-in HTTP package [39], (5) the Rocket framework in Rust [53], and (6) the Apache Tomcat framework in Java [58]. These applications cannot be supported by any existing kernel bypass system without a large porting effort. We compare them to native Linux, and to both gVisor and Firecracker. We configure each application to use up to 8 cores, though several are single-threaded. All are configured to deliver small static responses (14–600 bytes); nginx is the only one configured to read its response from a file (stored in a RAM-backed file system). We use 200 concurrent connections with connection keep-alives to avoid the cost of additional TCP handshakes.

Figure 6 shows the p99 latency and total CPU utilization across varying load for each application. For all applications, Junction provides superior throughput, latency, and CPU efficiency. Relative to Linux, Junction improves throughput by 1.62-3.69× and uses 19-65% less CPU when handling Linux's peak loads. Junction shows even larger gains against Firecracker and gVisor.

8.5 Attack Surface

Junction's use of kernel bypass reduces the host kernel attack surface relative to existing security-focused library

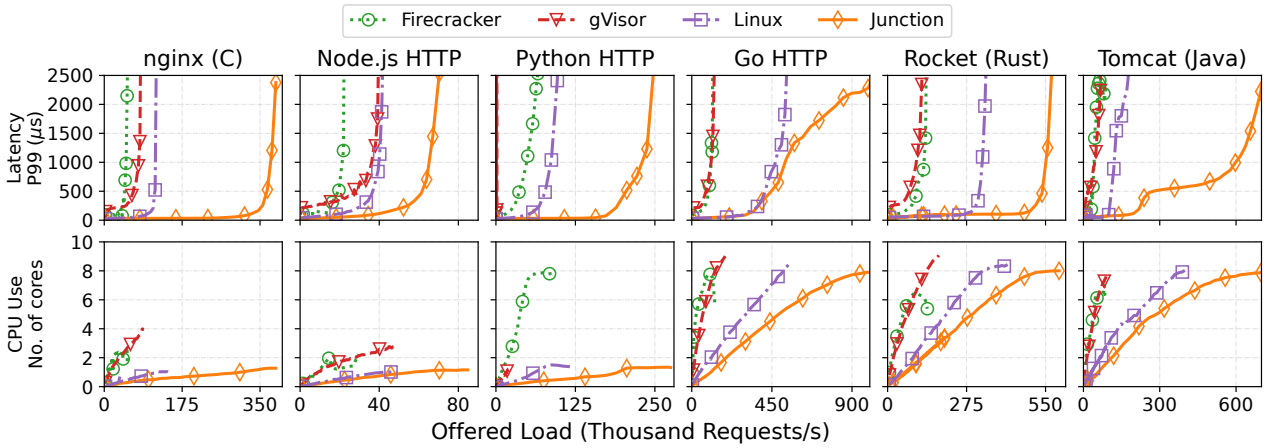


Figure 6: Service response-time and corresponding CPU usage at varying offered loads. Junction offers higher throughput and lower tail latency while also using fewer cores compared to other systems.

	Total	Threads/ Mem.	Procs	Sync	I/O	Misc.
Junction	11	5	2	0	4	0
gVisor (Sentry) [19]	64	7	11	2	34	10
gVisor (Gofer) [18]	57	4	11	1	36	5
Drawbridge	36	3	7	7	12	7

Table 5: Comparison of allowed syscalls to the host kernel for Junction and two other library operating systems.

	Unique syscalls	Total syscalls (/s)	`VMEXIT`s (/s)
Junction	4	9,603	n/a
Firecracker [1]	5	68,162	102,027
gVisor [17]	21	99,879	13,084
Linux	14	34,087	n/a

Table 6: Kernel crossings (per second) when running Rocket for 10s with an offered load of 10,000 RPS.

OSes. To demonstrate this, Table 5 shows the total number of syscalls required to run Junction, compared to gVisor and Drawbridge [46]. Drawbridge is a library OS that provides applications with a Windows 7 interface (in total over 100,000 API calls) using 36 system calls. gVisor is broken down into its two components, the Sentry and Gofer; the Sentry implements much of the system call interface but proxies access to files through the Gofer to add defense in depth. Because Junction uses kernel bypass NIC and CPU features to implement OS functionality, it requires 3.2–7.6× fewer syscalls to run.

Table 6 further demonstrates Junction's lack of reliance on the host kernel by showing profiling output from `strace` while running an 8-threaded Rocket instance at 10K RPS (after initialization). We report host kernel interactions through both system calls and VMExits. 99% of Junction's syscalls are to `yield_core()`, which is called when the application idles between requests, enabling other applications to run. The remaining 1% of calls are exclusively allocating and releasing memory. Linux, gVisor, and Firecracker all rely heavily on system calls that read from, send to, and block on file descriptors. Firecracker and gVisor interact with a Linux TAP device and virtio queues, while the Linux instance interacts with sockets. Both gVisor and Firecracker use `ioctl()` to interact with KVM. gVisor also heavily uses futexes and timers.

8.6 Performance Analysis

To better understand Junction's performance, we evaluated the impact of several of our design choices and mechanisms.

Performance optimizations. Figure 7 demonstrates how various aspects of Junction's design contribute to its performance. Both (1) using the Linux TAP driver to send and receive packets (as gVisor and Firecracker do) and (2) using seccomp filters to trap and intercept syscalls severely limit throughput for Junction instances. Per-core kernel bypass queues allow it to scale to significantly higher rates. Additional optimizations to enable compatibility with unmodified binaries further improve performance: both the use of hardware instructions to support TLS (`WRFSBASE`) and the optimization to allow it to clobber floating point state during system calls. These techniques allow Junction to nearly match the performance of Caladan (*i.e.,* the same TCP stack without compatibility).

Pinned memory. We quantify the improvements from adopting shared buffer queues and multi-packet buffers on the memory footprint of a Junction instance in Figure 8a, which shows the amount of memory consumed by 8-threaded instances with buffer pools sized to accommodate peak throughput. Enabling shared buffer queues shrinks the footprint by 35%, and enabling multi-packet buffers shrinks the footprint by an additional 48%, yielding a 33MB footprint per instance.

Scaling core allocation. We evaluate the two techniques discussed in §5.2 to scale the core allocator in Figure 8b. This experiment uses the same scenario as Figure 5. Enabling notifications from the NIC to a centralized queue allows the scheduler to scale to handle an additional 4× the number of

Page Cache Sharing	Total CPU Utilization
None	65%
+ Sharing Junction Kernel	60%
+ Sharing common libraries	59%
+ Sharing Rocket binary	36%

Table 7: Impact on CPU consumption of varying page cache sharing strategies for Rocket in Junction with 1500 instances.

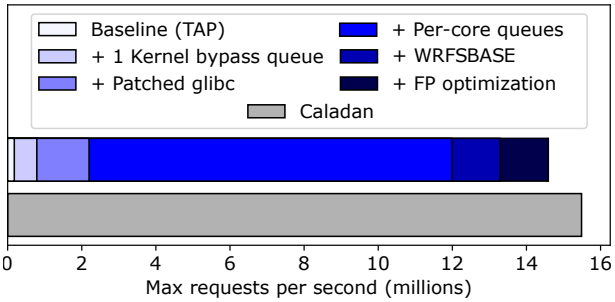
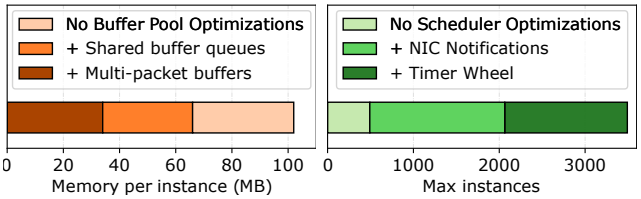


Figure 7: Contributions of several elements of Junction's design to its performance (using memcached).



(a) Memory optimizations. **(b)** Scheduler scalability.

Figure 8: Contributions of techniques to improve density (allowing more instances to be packed on a machine).

instances; the timer wheel scales it by an additional 1.69×.

Impact of page cache sharing We investigate the impact of Junction's page cache sharing (unsupported by VMs) in Table 7 by varying which binaries and dynamically linked libraries can be deduplicated. This experiment shows that in extreme case where a large number of identical binaries are operating at once, allowing page cache sharing results in a 44% decrease in CPU consumption. This is because data in instruction caches can be shared across instances, which suggests that operators that favor statically-linked binaries may be squandering performance under dense deployments.

9 Discussion

Hardware features. The CPU features Junction relies on are available on all modern Intel and AMD x86 CPUs. The only exception is UIPIs, which were recently introduced in Intel's Sapphire Rapid CPUs, so their availability is more limited. On older hardware, Junction falls back to the host kernel (`tgkill()` and `rt_sigreturn()`) for signals. Junction also relies on features in recent Mellanox NICs (ConnectX-5 and later) to reduce buffer memory consumption and make polling scalable, but can still support other NICs with potentially higher memory use and less scalability.

In the future, Junction could benefit from additional hardware support. For example, Junction could take advantage of the ability to handle CPU exceptions in userspace without involving the host kernel, as discussed in §6.1 and below.

Host kernel platform. Building Junction on Linux has many advantages. However, better security might be achievable by adopting a clean-slate approach, creating a purpose-built kernel for Junction's restricted host kernel interface. Such a kernel could be extremely minimal and even formally verified, reducing the chances of an exploit. uKVM provides a *monitor* similar to this idea for the Solo5 library OS [64], and we plan to investigate this for Junction in future work.

Another interesting tradeoff that such a platform could allow us to investigate is whether Junction should use Intel's VT-x and EPT extensions. An advantage could be that it would enable Junction to manipulate its page table and handle CPU exceptions directly, further reducing its reliance on the host kernel. Indeed, Dune demonstrates that these extensions can be configured in a way that maintains a process-like environment (*i.e.,* not a full VM) [5]. However, this would incur extra costs when entering and exiting the host kernel.

Performance isolation. Providing performance isolation across tenants is a challenging and important goal for cloud providers. Tenants can contend for CPU time, memory, network bandwidth, and microarchitectural CPU resources (caches, memory bandwidth, etc.). Junction is in a good position to handle this because it builds on Caladan, which already controls multiple forms of microarchitectural CPU interference [15]. While not in our current prototype, Junction's centralized core scheduler can easily implement priority scheduling and enforce CPU shares, and Linux cgroups can be used to enforce memory allocation limits. Finally, network bandwidth allocation can be offloaded to the NIC (§4).

10 Conclusion

This paper presented Junction, a system that retains the performance benefits of kernel bypass while scaling to thousands of instances and maintaining compatibility with existing Linux applications. Junction uses kernel bypass hardware (NIC queues and CPU features) to reduce its reliance on the host kernel, and it restricts the host kernel interface to enable efficient resource multiplexing with a minimal attack surface. It exploits a variety of advanced NIC features and scheduler optimizations to enable higher density. It also adapts OS subsystems to a kernel bypass setting and reduces system call overheads to maintain compatibility without sacrificing performance. Our evaluation shows that Junction can bring large benefits to existing applications without modifications, delivering superior tail latency, throughput, CPU efficiency, and density relative to state-of-the-art kernel bypass and cloud isolation systems.

Acknowledgements

We thank our shepherd Kostis Kaffes, the anonymous reviewers, Frans Kaashoek, Robert Morris, and other members of the MIT PDOS group for their helpful feedback. This work was funded in part by a Facebook Research Award; a Google Faculty Award; the DARPA FastNICs program under contract #HR0011-20-C-0089; the NSF under award CNS-2104398 and CNS-2212099; and VMware.

A UIPI Support

Junction uses UIPIs to support preemptive core allocation, timeslicing, and realtime POSIX signals between uThreads in the same instance. UIPIs are a recent Intel CPU feature that allow IPIs to be directly sent and received in userspace. At the time of writing, UIPI support is not yet a part of the mainline Linux Kernel, but Intel has proposed a patchset to add support for it [38]. Junction's design (*e.g.,* the assumption that one kThread runs on a core at a time) unlocks simplifications to host kernel support for UIPI that diverge from this patchset. Therefore, we implemented UIPI support from scratch as a small Linux Kernel module (§7) that is integrated with the central core scheduler.

Hardware interface. UIPIs build upon Intel's existing posted interrupt hardware, which was used previously to deliver interrupts directly into VMs without involving the host kernel. UIPIs instead allow IPIs to be sent between *normal OS threads* (kThreads in Junction) without going through the host kernel. Each interrupt receiver (*i.e.,* a thread or vCPU) is associated with a posted interrupt descriptor in memory, which serves as a link between the interrupt sender and receiver. When the receiver is scheduled on a core, the host kernel configures the core with several pieces of information: (1) the address of the receiver's descriptor, (2) a physical interrupt vector to use for posted interrupts, and (3) the handler (or guest interrupt vector) that should be invoked when a posted interrupt is received. At the same time, the host kernel writes information about the core into the descriptor (*i.e.,* its APIC ID and the physical interrupt vector). This allows a sender to send interrupts by only referencing a receiver's descriptor, which can be inspected by the CPU at send time to find up-to-date information about which core is running the receiver.

The ability to send UIPIs is restricted by target tables managed by the host kernel, which enumerate the addresses of descriptors that a thread is allowed to send to. To send a UIPI, a program invokes the `SENDUIPI` instruction with an operand that indexes this table. When this instruction is invoked, hardware writes a pending interrupt flag to the target descriptor. It then inspects the descriptor to determine whether or not to send an IPI—if another interrupt is pending or the receiver is not running, it skips sending an IPI. Otherwise, it uses the information in the descriptor to deliver an IPI to the receiver's core. When a core receives an interrupt on its posted interrupt vector, it inspects its current descriptor for pending interrupts and delivers them to the receiver.

UIPI Driver. UIPI support for Junction is managed by its UIPI kernel module. The kernel module receives notifications from the core scheduling kernel module whenever a core is reallocated, so it can appropriately program the core for the next running kThread. Junction does not need to send IPIs to inactive Junction kthreads, so we opt to use a single interrupt descriptor for each core, instead of using one for each kThread. Each Junction instance is provisioned with its own target table for `SENDUIPI` with one entry for each core's descriptor. Rows that correspond to cores actively allocated to this Junction instance are marked as valid, and all other rows are invalidated. The core scheduler's target table gives it permission to interrupt any core on the machine so it can use `SENDUIPI` when it needs to send an IPI. To correctly implement support for UIPIs, the driver must interpose on all context switches and system call entry and exit points for Junction kthreads. It does so by registering itself for callbacks through the Linux Kernel's lightweight tracepoint system.

How Junction uses UIPIs. Both the scheduler and the Junction kernel can send UIPIs to *kick* cores that are running kThreads and force them into the Junction kernel's handler. The handler determines the reason for the IPI and performs the appropriate action (*e.g.,* rescheduling uThreads, scanning network queues, or yielding the core).

Junction's core scheduler uses IPIs to notify kThreads when their core is being revoked or when a uThread running on the core has exceeded its timeslice. This works well when the scheduler needs to send just a single IPI at a time. However, because the scheduler implements microsecond-scale scheduling, it often needs to send multiple IPIs. While the interrupt controller supports IPI multicasting, a feature that reduces the cost of sending multiple IPIs, there is no instruction that exposes this functionality to userspace.

Our workaround is to take advantage of the fact that any interrupt sent on a core's posted interrupt vector can trigger user interrupt handlers on the receiving side. We mapped the interrupt descriptors for each core into the scheduler's memory so it can directly post the interrupt information, and exposed a custom system call through the UIPI kernel module that sends a multicast interrupt on the posted interrupt vector (we use the same vector on all cores). Upon receipt of the interrupt, hardware observes the interrupt in the descriptor and transfers control to the userspace interrupt handler. This approach still requires the scheduler to perform a system call, but this cost is amortized across multiple interrupts. If necessary, the APIC could be exposed directly to the scheduler in the future since it is a trusted component. The receiver side still benefits because it avoids interacting with the kernel.

Register saving. uThreads that are interrupted by UIPIs must save their registers as well as *extended* CPU state, which includes all vector registers and can total many kilobytes. x86

provides an instruction pair, `XSAVE` and `XRSTOR`, that can be used in usermode to save and restore extended CPU states. It also provides two variants of `XSAVE` that aim to reduce overhead. `XSAVEOPT` avoids saving states that were not modified since the previous `XRSTOR`, while `XSAVEC` reduces memory fragmentation by saving to a compacted memory layout. Both instructions save only *active* states. Intel warns that `XSAVEOPT` is not recommended for user applications, because it is possible for the processor to mistakenly correlate save/restore pairs between different applications [26]. We invested some effort to ensure that `XSAVEOPT` was only used when it was paired with a correct `XRSTOR`, however, we found that `XSAVEC` was just as fast as `XSAVEOPT`, so Junction uses this instruction when saving extended states.

Interactions with system calls. POSIX signals delivered during a blocking interruptible system call should cause the system call to return immediately with an error, so that signal handlers can run without delay. Junction must support this at two levels. First, system calls that block inside the Junction kernel (*e.g.,* blocking network socket reads) must be unblocked by signals. Each uThread in Junction has an atomic flag that is used to coordinate blocking with wakeups from signal senders. Second, a uProc system call that results in a host kernel system call (*e.g.,* `mmap()`) must also be unblocked when a signal is sent. Junction's UIPI kernel module handles this by redirecting the local core's posted interrupt vector whenever a Junction kThread enters a Linux system call. This causes future interrupts on this vector (*i.e.,* those sent by `SENDUIPI` or multicast) to be delivered to the kernel module. Because the centralized core scheduler assigns kThreads to cores with exclusive grants, the kernel module is able to easily locate and wake up the blocked kThread.

Evaluation. We examined the impact of UIPIs on core allocation, timeslicing, and inter-thread signalling. We found that it only marginally improved latency and efficiency when replacing Linux signals for preempting kThreads to reallocate cores. We also have yet to find any real-world applications that use signals frequently enough to benefit from the speedups of UIPIs. We did observe, however, a significant reduction in overhead for Junction's uThread timeslicing. In Figure 9, we measure this overhead for a range of scheduler quantums with UIPIs and Linux signals. We observe that UIPIs reduce timeslicing overhead by an average of 2.35×. In practice, this unlocks the ability to timeslice uThreads at much finer granularity, which is beneficial for reining in the tail latency of microsecond-scale workloads with high service time dispersion [9, 27, 29].

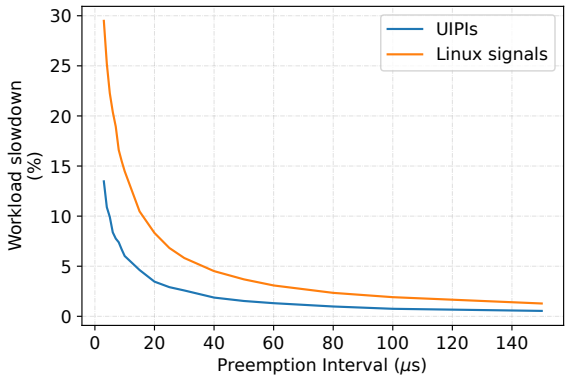


Figure 9: Comparison of preemption overheads for UIPIs versus Linux signals for a synthetic workload performing square root computations. UIPIs on average reduce slowdowns by 2.35×.

B Buffer Management

Keeping the buffer memory footprint of a kernel bypass application low is critical for density. To address this, Junction leverages two NIC hardware features simultaneously—shared buffer queues and multi-packet receive buffers.

Using a shared buffer queue with per-core receive queues introduces the need to coordinate across cores. Posting buffers to the shared buffer queue involves two steps: writing the buffer address into an available descriptor in the queue, and advancing the index pointer forward to notify the NIC that the descriptor is ready. When a core receives a packet on its receive queue, it can immediately refill the corresponding descriptor in the shared buffer queue with a new buffer. However, it must wait to update the index pointer because the preceding buffers may have not yet been received by other cores. Therefore, if a core is slow to poll its receive queue, it could stall the entire buffer posting process.

Using multi-packet buffers adds additional complexity, since posted contiguous buffers are split into many smaller packets and distributed to multiple cores. In addition to coordinating updates to the index pointer, cores must coordinate (A) when to refill an individual descriptor, which cannot happen until all of its individual packets are delivered; and (B) when to mark a buffer as eligible to be reposted, which cannot happen until packet processing has freed each individual packet in that buffer.

Prior work on ShRing also explored using these hardware features, but for a different goal of optimizing DDIO cache usage [45]. The authors argued for hardware changes to enable efficient coordination. They proposed out-of-order posting to shared buffer queues, to prevent a busy core from delaying freeing slots. Instead, other cores can continue to supply buffers to the NIC without waiting. This change is coupled with a change to reduce the high cost of reference counting buffers. Their solution is for the hardware to batch multiple packets into a single buffer, but not deliver multiple packets within one buffer to multiple cores. Instead, a buffer becomes associated with a core when the first packet is delivered, and only future deliveries to that core can use the buffer.

To the contrary, Junction shows that low coordination overhead can be achieved with existing hardware (*i.e.,* without these changes). First, Junction relies on both UIPIs and work

stealing across kThread receive queues to ensure that processing delays cannot starve the shared buffer queue. Second, Junction's per-core reference counting reduces synchronization overhead by allowing kThreads to update reference counters without frequently modifying shared cache lines.

When a kThread receives a packet, it increments a core-local counter corresponding to the packet's descriptor by the number of bytes consumed. Once the sum of the per-core counters is equal to the full size of the buffer, no further data can be written, so it is safe to replenish the slot with a new buffer. Similarly, after a packet is dequeued from the NIC, it undergoes protocol processing and is delivered to the application. Eventually, a core frees a packet by incrementing its local counter corresponding to the buffer backing the packet. Once all the packets in a buffer are free, the buffer can be reused. The scheduler core monitors the number of posted buffers and sends a notification over shared memory when the queue needs to be refilled. This notification triggers a high-priority refill thread that is responsible for managing the shared buffer queue, which collects the reference counter sums and refills the queue. This thread's logic is shown in algorithm 1.

Algorithm 1: Refill Pool Algorithm

```
1  Function RefillPool():
2      // Refill free slots in the shared queue
3      while true do
4          index ← CompletedBufs mod |RQ|
5          cnt ← 0
6          foreach cpu do
7              cnt += SlotCompletions[cpu][index]
8          end
9          if cnt = PKTS_PER_BUF then
10             ResetSlotRefs(index)
11             BusyBufs.append(SharedQ[index])
12             SharedQ[index] ← FreeBufs.pop()
13             CompletedBufs += 1
14         else
15             break
16         end
17     end
18     // Find free buffers
19     foreach buf in BusyBufs do
20         index ← index_of(buf)
21         cnt ← 0
22         foreach cpu do
23             cnt += BufCompletions[cpu][index]
24         end
25         if cnt = PKTS_PER_BUF then
26             ResetBufRef(index)
27             FreeBufs.push(buf)
28             BusyBufs.erase(buf)
29         end
30     end
```

C Additional Benchmarks

C.1 Threading & Networking

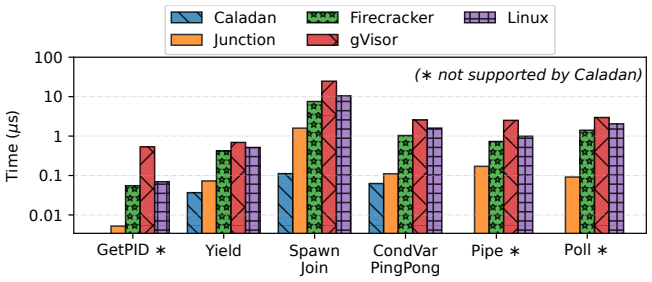
In this section, we evaluate the performance of Junction's threading and networking primitives.

Threading microbenchmarks. Figure 10a shows a set of microbenchmarks that measure the performance of several common threading operations. We use the same binary for each system except Caladan, which required a custom implementation due to its lack of compatibility. Each operation is performed in a loop for 1,000,000 iterations to determine the time per operation. The GetPID benchmark measures the baseline cost of performing a syscall (`getpid()`). In Junction, syscalls are replaced with function calls, resulting in low overheads (≈ 10 ns). Linux, Firecracker, and gVisor pay a penalty for switching between user and kernel mode, while gVisor pays additional penalties for its system call interception techniques.

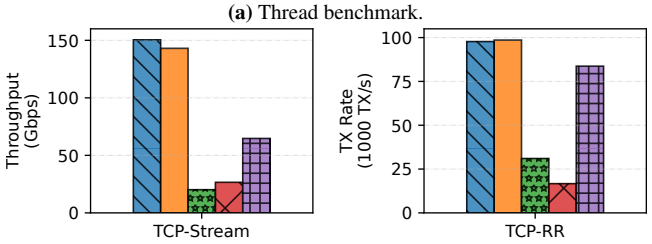
Yield and SpawnJoin measure overheads of the threading subsystems. Relative to Caladan, Junction can context switch between threads nearly as fast, but pays additional costs for thread creation and teardown because of POSIX compatibility and support for TLS—each thread must allocate and initialize thread local variables when created. CondVar and Pipe measure the costs of synchronizing two threads using condition variables and pipes; the Poll benchmark does the same but uses `poll()` on non-blocking pipes. In most cases, Junction fares at least an order of magnitude better than the systems that rely on kernel crossings. Caladan's performance on the CondVar benchmark is slightly better than Junction's in part due to its ability to inline synchronization functions (while in Junction these calls must traverse the trampoline page).

PARSEC. We now show how these threading primitives translate to Junction's end-to-end performance in the PARSEC benchmark suite [7], which consists of a set of compute-bound applications that exhibit high degrees of parallelism with varying synchronization strategies. Figure 11 shows the time taken to execute each PARSEC benchmark relative to Linux. As expected, due to the low overall frequency of system calls, Junction, Firecracker, and gVisor achieve mostly comparable performance to Linux. However, in benchmarks where there is heavier thread synchronization, Junction outperforms Firecracker and gVisor because of its better load balancing and more efficient system calls.

TCP Microbenchmarks. We evaluate the performance of Junction's kernel bypass networking and TCP/IP stack using the standard netperf benchmark (TCP-Stream and TCP-RR). For each experiment, we use *a single flow* and configure each system with a single thread (but do not restrict SoftIRQ processing on other cores). We use the same binary for every system but Caladan, for which we wrote a compatible imple-



(a) Thread benchmark.



(b) Networking benchmark.

Figure 10: Performance comparison with threading and networking microbenchmarks. On most threading benchmarks, Junction improves performance nearly 10× relative to non-kernel bypass systems. Junction is also able to to sustain TCP throughputs of up to 146Gbps with a single flow on a single core, a 2.2× improvement over Linux. Junction's low overhead network stack enables it to achieve high transaction rates (100,000 messages per second), a 3.2–6.0× speedup over Firecracker and gVisor.

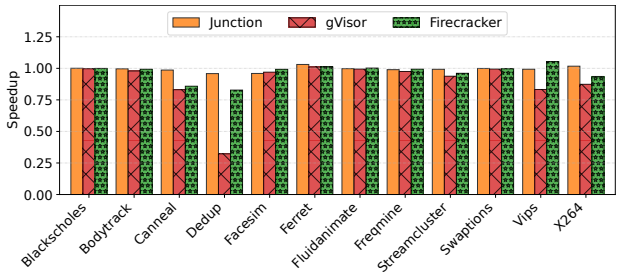


Figure 11: PARSEC benchmark speedup, normalized to Linux. Junction is able to offer comparable performance to Linux, while other systems perform worse in situations with heavier synchronization.

mentation. This benchmark runs between two machines that are connected back-to-back with a 200GbE link.

The TCP-Stream benchmark measures data transfer rates. Figure 10b shows that Junction and Caladan are able to achieve 146 and 154 Gbps respectively, greater than 2.2× the performance of the next best system (Linux). Junction's improvements are even more pronounced relative to Firecracker and gVisor, improving throughput 5.4–7.1×.

TCP-RR benchmarks the request/response rate which is determined by the round trip latency between two TCP stacks. Both Junction and Caladan achieve close to 100,000 transactions per second, indicating a round trip time of 10 μs. Linux adds marginal overhead with round trip times of 11 μs. Firecracker and gVisor, however, have 32 μs and 60 μs latencies respectively. They both use TAP devices for networking, resulting in additional scheduling and processing hops on top of Linux.

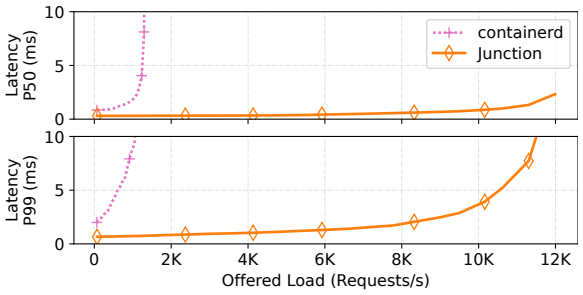


Figure 12: Benefits of using Junction in an end-to-end experiment with the serverless framework faasd. Accelerating each component in the system using Junction leads to compounding benefits for tail latency and throughput, with 3.5× lower tail latency and 10× higher total system throughput.

C.2 FaaS Benchmark

We also evaluate the end-to-end performance of Junction with a cloud framework to study the impact of adopting it across a distributed system instead of just one application. We use faasd [11], an open source serverless orchestration framework based on OpenFaaS [41]. faasd uses Linux containers to sandbox untrusted user applications, deployed by containerd [8]. It includes two orchestration services written in Go: a front-end load balancer and a per-host provider that communicates with containerd. To demonstrate the performance benefits of Junction, we replace containerd with *junctiond*, a C++ component that manages local instances of Junction. We also run the two orchestration services inside Junction. We evaluate the setup using invocations of a serverless function from vSwarm [59,63] that encrypt a 600 byte input with AES. We do not evaluate cold-starts here, but we separately profiled the startup costs for a single-threaded Junction instance and found that Junction takes 3.4ms to initialize. These experiments used two machines with 10 core Xeon 4114 CPUs running at 2.2GHz, 48GB of RAM, and 100GbE NICs.

Figure 12 shows the median and tail latency across varying request rates offered via the front-end load balancer. Junction can sustain up to 10× more throughput while lowering the latency by ∼ 2× at the median and ∼ 3.5× at the tail. This reflects the compounding end-to-end benefit of using Junction across multiple components running in separate instances. More discussion of the faasd architecture and this experiment are available in [54].

References

[1] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *NSDI*, 2020.

[2] Amazon Web Services. The Security Design of the AWS Nitro System: AWS Whitepaper. Technical report, November 2022.

[3] Anjali, Tyler Caraza-Harter, and Michael M. Swift. Blending Containers and Virtual Machines: A Study of Firecracker and GVisor. In *VEE*, 2020.

[4] Jonathan Behrens, Adam Belay, and M. Frans Kaashoek. Performance evolution of mitigating transient execution attacks. In *EuroSys*, 2022.

[5] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *OSDI*, 2012.

[6] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *OSDI*, 2014.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[8] containerd. containerd overview, 2023.

[9] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with PerséPhone. In *SOSP*, 2021.

[10] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *SOSP*, 1995.

[11] faasd. A lightweight & portable FaaS engine, 2023.

[12] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*, 2018.

[13] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004.

[14] Linux Foundation. Data Plane Development Kit (DPDK), 2015.

[15] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *OSDI*, 2020.

[16] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. Memory-Harvesting VMs in Cloud Platforms. In *ASPLOS*, 2022.

[17] Google. gVisor Documentation, 2019.

[18] Google. gVisor Gofer Syscalls, 2022.

[19] Google. gVisor Sentry Syscalls, 2023.

[20] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al. Biscuit: A framework for near-data processing of big data workloads. In *ISCA*, 2016.

[21] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *OSDI*, 2012.

[22] Steven Hand, Andrew Warfield, Keir Fraser, Evangelos Kotsovinos, and Daniel J Magenheimer. Are Virtual Machine Monitors Microkernels Done Right? In *HotOS*, 2005.

[23] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. GhOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *SOSP*, 2021.

[24] Takayuki Imada. Mirageos unikernel with network acceleration for iot cloud environments. In *ICCBDC*, 2018.

[25] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2B*, December 2022.

[26] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 1*, June 2023.

[27] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling. In *SOSP*, SOSP '23, New York, NY, USA, 2023.

[28] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems. In *NSDI*, 2014.

[29] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for μSecond-Scale Tail Latency. In *NSDI*, 2019.

[30] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter RPCs Can Be General and Fast. In *NSDI*, 2019.

[31] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. Profiling a warehouse-scale computer. In *ISCA*, 2015.

[32] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *EuroSys*, 2019.

[33] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv: Optimizing the Operating System for Virtual Machines. In *USENIX ATC*, 2014.

[34] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, et al. Unikraft: fast, specialized unikernels the easy way. In *EuroSys*, 2021.

[35] H.J. Lu, Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface. *AMD64 Architecture Processor Supplement*, 2018.

[36] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, 2012.

[37] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network Stack Specialization for Performance. In *SIGCOMM*, 2014.

[38] Sohil Mehta. x86 User Interrupts support, 2021. Available at https://lore.kernel.org/lkml/20210913200132.3396598-1-sohil.mehta@intel.com/.

[39] Jeff Meyerson. The Go programming language. *IEEE software*, 31(5):104–104, 2014.

[40] Node.js. Node.js, 2023.

[41] OpenFaaS. Serverless functions made simple, 2023.

[42] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. In *NSDI*, 2019.

[43] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3), aug 2015.

[44] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System Is the Control Plane. In *OSDI*, 2015.

[45] Boris Pismenny, Adam Morrison, and Dan Tsafrir. ShRing: Networking with Shared Receive Rings. In *OSDI*, 2023.

[46] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. Rethinking the library OS from the top down. In *ASPLOS*, 2011.

[47] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *SOSP*, 2017.

[48] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 342–355, New York, NY, USA, 2015. Association for Computing Machinery.

[49] Ron Pressler. On the performance of user-mode threads and coroutines, 8 2020. Accessed: September 2023.

[50] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-Aware Thread Management. In *OSDI*, 2018.

[51] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008.

[52] Luigi Rizzo. netmap: a novel framework for fast packet I/O. In *USENIX Security*, 2012.

[53] Rocket. Rocket, 2016.

[54] Enrique Saurez, Joshua Fried, Gohar Irfan Chaudhry, Esha Choukse, Íñigo Goiri, Sameh Elnikety, Adam Belay, and Rodrigo Fonseca. Junctiond: Extending FaaS Runtimes with Kernel-Bypass, March 2024.

[55] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *EuroSys*, 2013.

[56] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *ASPLOS*, 2019.

[57] Mike Stemle. The State of Serverless, 2023.

[58] Tomcat. Tomcat, 1999.

[59] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, and Edouard Bugnionand Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS*, 2021.

[60] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[61] George Varghese and Anthony Lauck. Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility. *IEEE/ACM transactions on networking*, 5(6):824–834, 1997.

[62] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-Scale Cluster Management at Google with Borg. In *EuroSys*, 2015.

[63] vSwarm. Serverless benchmarking suite, 2023.

[64] Dan Williams and Ricardo Koller. Unikernel Monitors: Extending Minimalism Outside of the Box. In *HotCloud*, 2016.

[65] Dan Williams, Ricardo Koller, and Brandon Lum. Say goodbye to virtualization for a safer cloud. In *HotCloud*, 2018.

[66] Kenichi Yasukata, Hajime Tazaki, Pierre-Louis Aublin, and Kenta Ishiguro. zpoline: a system call hook mechanism based on binary rewriting. In *USENIX ATC*, 2023.

[67] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *SOSP*, 2021.

[68] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *EuroSys*, 2013.

Sifter: An Inversion-Free and Large-Capacity Programmable Packet Scheduler

Peixuan Gao
New York University

Anthony Dalleggio
New York University

Jiajin Liu
New York University

Chen Peng
New York University

Yang Xu *
Fudan University

H. Jonathan Chao
New York University

Abstract

Packet schedulers play a crucial role in determining the order in which packets are served. They achieve this by assigning a rank to each packet and sorting them based on these ranks. However, when dealing with a large number of flows at high packet rates, sorting functions can become extremely complex and time-consuming. To address this issue, fast-approximating packet schedulers have been proposed, but they come with the risk of producing scheduling errors, or packet inversions, which can lead to undesirable consequences. We present Sifter, a programmable packet scheduler that offers high accuracy and large capacity while ensuring inversion-free operation. Sifter employs a unique sorting technique called "Sift Sorting" to coarsely sort packets with larger ranks into buckets, while accurately and finely sorting those with smaller ranks using a small Push-In-First-Out (PIFO) queue in parallel. The sorting process takes advantage of the "Speed-up Factor", which is a function of the memory bandwidth to output link bandwidth ratio, to achieve Sift Sorting and ensure accurate scheduling with low resource consumption. Sifter combines the benefits of PIFO's accuracy and FIFO-based schedulers' large capacity, resulting in guaranteed delivery of packets in an accurate scheduling order. Our simulation results demonstrate Sifter's efficiency in achieving inversion-free scheduling, while the FPGA-based hardware prototype validates that Sifter supports a throughput of 100Gbps without packet inversion errors.

1 Introduction

The Programmable Data Plane has seen increased interest in both academia and industry to enhance the flexibility and the programmability of ultra-high-speed networks without compromising the throughput and latency performance [9] [27] [56] [53] [55] [23]. The academic and industry communities have built a sophisticated ecosystem for the programmable data plane including the P4 programming language (P4) [8], cross-platform high-level languages and their compilers [42] [38], programmable switch devices [11], [17], and a variety of re-configurable network chip architectures [18] [10].

Recent research has focused on enhancing the programmability of packet scheduling in the data plane. Various packet scheduling disciplines sort packets based on a "rank" value assigned by the scheduler to represent their transmission order. [48] [49] [35].

However, implementing a priority queue that sorts packets in an ultra-high-speed data plane is challenging. The scheduler needs to serve each packet within a very limited time budget. For a 64-byte packet on a 100 Gbps Ethernet link, the packet processing time is less than 7 ns. Packet schedulers such as Sequencer [13] [14] [15] and PIFO[48] [49] manage to sort packets with ultra-low time complexity by sacrificing buffer capacity. To accommodate larger buffer sizes and reduce the implementation complexity, the community proposed a number of approximating packet schedulers such as PCQ [45], SP-PIFO [2], Gearbox [24] and AIFO [54] that trade-off scheduling accuracy for simplicity and scalability.

Scheduling errors, also known as packet inversions [2], are introduced by approximating packet schedulers and may impact network performance. These packet inversions cause throughput fluctuation, affect fairness, introduce delay, and slow down the flow completion time (FCT). Failure to rigorously adhere to the scheduling order of bandwidth allocation algorithms can negatively affect performance and fairness, which are essential for network isolation [31] [6] [46] [30] [41] [50]. Moreover, emerging applications, such as self-driving vehicles and remote surgery, exhibit a high sensitivity to packet inversions. As a result, there is a pressing requirement for precise packet scheduling to maintains strict packet order, not only for delay-sensitive applications but also for algorithms that react adversely to packet inversions.

Accuracy and scalability of the scheduler are often conflicting requirements. We want a scheduler that can accurately serve packets by the strict order of their ranks (e.g., PIFO) and also support a large buffer size with low hardware resource consumption.

We present Sifter, an accurate and large-capacity programmable packet scheduler that operates free of packet inversions and supports a large buffer size with low implementation overhead. Sifter sorts packets by "Sift Sorting", a sorting method consisting of two parallel processes. Sifter sorts packets with larger ranks coarsely into a FIFO-based "Rotating Calendar Queue" (RCQ) and packets with smaller

*Corresponding author

ranks accurately using a mini PIFO. Sifter is intended for next-generation programmable switches and smart NICs. By taking advantage of the "Speed-up Factor" provided by the hardware architecture, Sifter combines the advantages of accurate scheduling of a PIFO with the large capacity provided by FIFO-based approximate schedulers. Based on our simulation results and our hardware prototype, Sifter eliminates packet inversions[1] while supporting a large buffer capacity and low implementation complexity. The contributions of this paper are summarized as follows:

- **Analysis of the impact of packet inversions on network performance.** Starting from the different scheduling goals of packet scheduling algorithms, this paper discusses the impact of packet inversion errors on network performance. Various scheduling algorithms have different sensitivities to packet inversions, and delay-guarantee scheduling algorithms are the most sensitive to the accuracy of the packet scheduling order.
- **Design of an accurate and large-capacity packet scheduler with inversion-free operation.** This paper presents Sifter, an accurate and large-capacity packet scheduler that operates free of packet inversions without sacrificing scalability or increasing implementation complexity using "Sift Sorting". By exploiting the "Speed-up Factor", Sifter combines the accuracy of PIFO and the large capacity of RCQ, a FIFO-based scheduler.
- **Definition of the conditions for achieving inversion-free operation.** This paper presents a quantitative analysis to determine the conditions under which Sifter guarantees operation with no packet inversions.
- **Comprehensive simulation of Sifter on NS3 and experiments on an FPGA-based Sifter prototype.** We implemented Sifter in the NS3 simulator and in VHDL on an AMD/Xilinx Alveo U250 FPGA card [4] with a mid-speed grade XCVU13P FPGA. In the NS3 [37] simulator, we conduct comprehensive evaluations of Sifter using multiple metrics, demonstrating that it provides packet scheduling performance that is close to the ideal PIFO. Our hardware testbed results show the FPGA-based Sifter prototype operates at 322 MHz and achieves a line rate of 100 Gbps for packets larger than 370 bytes[2] without any inversions.

The rest of the paper is organized as follows. *Section* 2 provides the background and motivation of our work. *Section* 3 presents the detailed architecture, scheduling processes of Sifter and the conditions for inversion-free scheduling. *Section* 4 provides a detailed Sifter prototype implementation and *Section* 5 presents the packet-based NS3 and the hardware testbed evaluations. We discuss related works in *Section* 6 and conclude the paper in *Section* 7.

[1] See *Section* 3.5 for the conditions for inversion-free scheduling.

[2] See *Appendix* H for more details.

2 Background and Motivation

2.1 Programmable Packet Scheduling

The concept of programmable packet scheduling is to enable network administrators to schedule packets using different packet scheduling algorithms. According to previous literature [48] [49] [35], different packet scheduling algorithms determine the packet scheduling order to achieve certain goals, such as max-min fairness in bandwidth allocation [20] [39] [40] [25] [26] [57], minimizing FCT [43] [3] as well as delay guarantee [34] [35]. When a generalized scheduler serves packets in ascending order of their rank values, it effectively implements the scheduling order dictated by a specific scheduling algorithm. Consequently, the abstraction of programmable packet scheduling involves two key steps: (1) establishing the sequence for scheduling each packet (*Rank Calculation*) and (2) enforcing this packet scheduling sequence (*Rank Sorting*).

The calculation of packet *rank* can be programmed at the end-hosts before the packet is sent out, or even in the programmable ingress and egress switch pipeline [9][8]. When the packet arrives at the scheduler, the switch needs to enforce the scheduling order by sorting the packets by their *rank* values. Based on this theory, multiple works have implemented generic programmable packet schedulers [45] [2] [24] [54].

2.2 Packet Inversions

To deliver a correct packet scheduling order, an ideal packet scheduler needs to serve packets strictly according to the increasing order of the *rank* of each packet. However, due to limited packet processing time and implementation complexity, many generic packet scheduler designs cannot serve packets in such an ideal order and instead, schedule packets in an approximate increasing order, which may introduce packet inversions. We formally define packet inversions[3].

Packet Inversions: When a packet with rank r departs from the scheduler, there exists packet(s) with a smaller rank r' (where smaller rank has higher priority, $r' < r$) in the scheduler.

2.2.1 Impact of Packet Inversions

Packet inversions have various impacts on different packet scheduling algorithms according to their logic and goals.

Fairness The impact of packet inversions on fairness-based scheduling algorithms varies from fluctuation in short-term bandwidth allocation to fairness impairment. Most of the fairness-based algorithms [20] [39] [40] [25] [47] [26] [57] are derived from GPS, which serves flows based on round-robin logic by assigning a departure time to each packet. Packet inversions can have several negative effects,

[3] We further extend *Packet Inversions* by *Inversion Magnitude* to quantify the severity of packet inversions in AppendixA.

including short-term fairness issues and fluctuations in both throughput and delay.

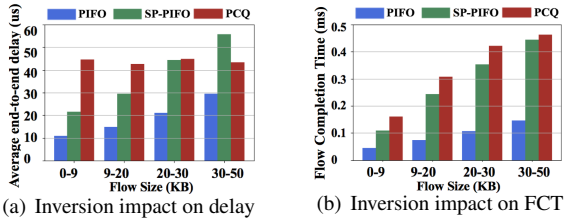
However, for the fairness-based scheduling algorithms that update the virtual clock according to the departure times of packets [25] [26], the consequence of packet inversions could be much more severe. When there is a packet inversion and a packet with a very large timestamp (rank) t' departs early, the scheduler updates its current virtual clock from t to a much larger value t' in advance. The skew on the virtual clock can prevent any newly arrived packet from getting a departure time between t and t'. As a consequence, existing packets with departure times smaller than t' would take up the bandwidth and lead to starvation of the newly arrived flows. The accumulation of such packet inversions can lead to performance degradation and impairment of network isolation [31] [6] [46] [30] [41] [50].

Minimizing FCT Other packet scheduling algorithms, such as those designed to minimize average FCT [43] [3] [28], can be significantly impacted by packet order inversions. The core logic of minimizing FCT is to always serve the flow with the minimum remaining flow size. However, if packet inversions occur, the scheduler may serve other flows with larger remaining sizes and may possibly serve multiple flows at the same time. For example, when packet inversions occur, the packet scheduler may fail to differentiate between multiple flows that have similar remaining sizes. As a result, the scheduler may start serving multiple flows in a round-robin fashion instead of serving the flow with the minimum size, which can dramatically increase the average FCT.

Delay Guarantee (Tail Packet Delay) Scheduling disciplines that are most affected by packet inversions are those that provide a delay guarantee [34] [52]. Typically, these scheduling algorithms cater to delay-sensitive applications that impose strict end-to-end delay requirements, such as Vehicle to Everything (V2X) and remote surgery. These algorithms usually assign a delay budget to each packet ("slack time"), which represents the end-to-end delay requirement. Upon the arrival of each packet, the packet scheduler sorts packets according to their urgency and always schedules the packet with the least "slack time". In this case, the accuracy of the packet scheduler is critical. A minor inversion in scheduling could result in a series of packets missing their deadlines and having a major negative impact on the performance of the application.

Evaluating Impacts of Packet Inversions We evaluated the impacts of packet inversions by running Start-Time Fair Queueing (SFQ) [26] on approximate schedulers and a PIFO. The evaluation setup is described in detail in *Appendix* F.

Figure 1(a) shows the impact of packet inversion on packet delay. When packet inversions occur, packets with larger ranks are scheduled prior to other packets with smaller ranks, which leads to extra queuing delay for the packets that should



(a) Inversion impact on delay (b) Inversion impact on FCT

Figure 1: Impacts of Packet Inversions

have been scheduled earlier. Furthermore, Figure 1(b) shows the impact of packet inversion on FCT. Approximate packet schedulers introduce extra delay and unfairness in throughput due to packet inversions, which impairs the FCT of small flows.

We have also noted that the introduction of randomness through packet inversion amplifies both throughput and delay uncertainties. Certain applications like autonomous vehicles and remote surgery heavily depend on consistent connection performance.

2.3 Achieving accurate packet scheduling

2.3.1 Causes of Packet Inversions

As discussed in *Section* 2.2, existing implementations of approximating packet schedulers [44] [45] [2] [24] [54] are subject to packet inversions. These approximating schedulers are implemented with one or more FIFO queues and packets from each FIFO queue are served on a first-come-first-served basis, which can lead to packet inversions between the packets from the same FIFO queue. Although such a FIFO-based structure has the advantages of low time-complexity, high scalability, and implementation simplicity, the potential packet inversions may result in various consequences as discussed in *Section* 2.2.1. The key to eliminating those packet inversions is to resolve any packet misordering within each FIFO queue.

2.3.2 Proactive and Reactive Packet Sorting

An intuitive solution to resolve packet misordering is to sort the packets in each FIFO queue before they are dequeued. The existing solutions of packet sorting can be categorized into two classes, proactive and reactive. The proactive approach sorts each packet upon arrival and maintains a sorted priority queue [19]. As Figure 2(a) shows, each packet is inserted into the correct place upon arrival. Such sorting algorithms are subject to a fairly large time complexity of $O(logN)$ for each packet enqueue [4]. This sorting is difficult to complete within the limited packet processing time in ultra-high-speed data planes. A number of works [13] [14] [15] [48] [49] overcome this challenge and provide proactive packet sorters operating at line rate, however, they usually cannot support a large buffer capacity. The reactive approach

[4] N is the total number of packets inside the packet sorter

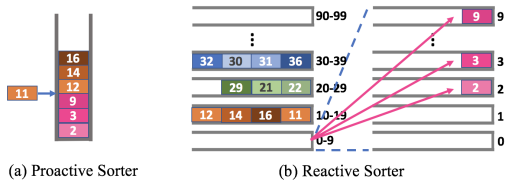


Figure 2: Proactive and Reactive Sorters

sorts packets as part of the dequeue process [45]. As shown in Figure 2(b), reactive solutions enqueue packets into a set of strict priority FIFO queues and sort them with finer granularity when the scheduler is about to serve these packets. Although these reactive sorters provide a fast enqueue process and large capacity, the sorting process is time-consuming and might lead to blocking in the dequeue process.

2.3.3 Speed-up Factor

We can achieve a perfect packet sorting solution that combines the advantages of non-blocking dequeue from proactive sorters and large capacity from reactive sorters. The key to our approach to accomplish this is sorting packets in parallel with the dequeue process.

Memory bandwidth is usually higher than the output links bandwidth on hardware switches and network interface cards (NIC). In addition, packet schedulers usually schedule descriptors[5] that represent packets. Packet descriptors are much smaller than the average size of a packet. Combining the above facts, hardware switches can access multiple packet descriptors during the time of dequeuing a single packet. We adopt the concept of "Speed-up Factor" K, a number that measures the relative speed between the descriptor access speed and packet dequeue speed.

$$K = \lfloor \frac{R_M}{R_o} \cdot \frac{\min L_P}{L_d} \rfloor \quad (1)$$

where K is the minimum Speed-up factor[6], R_M is the memory bandwidth, R_o is the output link line rate, $\min L_P$ is the minimum packet size and L_d is the byte-length of the packet descriptor. For illustration purposes, let us consider a system with R_M = 64 Gbps (64-bit memory running at 1 GHz), L_d = 64 bits, $\min L_P$ = 64 bytes, and R_o = 100 Gb/s, then

$$K = \lfloor \frac{64 \cdot 10^9}{100 \cdot 10^9} \cdot \frac{64 \cdot 8}{64} \rfloor = \lfloor 5.12 \rfloor = 5 \quad (2)$$

A speed-up factor K means the scheduler can access at least K packet descriptors during the time of dequeuing a packet. By taking advantage of the speed-up factor, the reactive FIFO-based sorter can sort packets with a proactive priority queue (PIFO) in parallel without causing blocking.

[5] A packet descriptor is also known as metadata.

[6] When packet sizes are larger than the minimum packet size $\min L_P$, the speed-up factor would be larger accordingly.

3 Sifter: An Accurate and Large-Capacity Programmable Packet Scheduler

3.1 Concept and Architecture

The previous sections establish a need for a programmable packet scheduler with accuracy and scalability, combining the advantages of both a perfect and an approximating PIFO.

We introduce Sifter, a scalable PIFO to support large-capacity programmable packet scheduling without packet inversions. As shown in Figure 3, Sifter consists of two major components: (1) A Mini-PIFO and (2) An RCQ. The Mini-PIFO stores the packets with relatively smaller ranks. It performs a strict sorting of packets and always outputs the packet with the smallest rank. The RCQ holds the rest of the packets with relatively larger ranks. It stores and sorts packets with coarse granularity[7] similarly to [12] [51] [44] [45] [2] [24] by a set of strict-priority FIFO queues and is therefore widely scalable in rank range and capacity.

Sifter uses an algorithm, "Sift Sorting", which consists of two parallel processes to schedule packets in strict order: The first process is Enqueue, in which packets are stored in different components according to their ranks. The packets with larger ranks enter the RCQs and are sorted with coarse granularity while the packets with smaller ranks enter the Mini-PIFO and are strictly sorted. The second process called "Sifting" operates in parallel. When packets dequeue from the Mini-PIFO and there is available space, the RCQ keeps migrating (sifting) the packets from FIFOs with smaller ranks to the Mini-PIFO. The Mini-PIFO therefore eliminates the potential misordering of ranks in a FIFO in the RCQ using only minimal-sized sifting registers. By combining the RCQ and Mini-PIFO, Sifter schedules packets in the correct order while supporting a large capacity.

The name Sifter is derived from its sorting method. If each packet is a particle in the sifter, the rank of each packet would represent the size of the particle. Sifter "filters" the larger particles from the upper layers and passes through the smaller particles, as shown in Figure 3. Since the Mini-PIFO always holds the packets with the smallest ranks in the scheduler, Sifter dequeues packets from the head of the Mini-PIFO. For a more in-depth look at Sifter's operation, we introduce its three major processes: Enqueue, Dequeue, and Sifting in *Section* 3.2.

Sifter only schedules packet descriptors[8]. In this paper, "packets" refer to the packet descriptor in the scheduler. To

[7] Granularity: the rank range that a FIFO in the RCQ accommodates.

[8] Packets are stored in a shared-memory buffer [1] [16] and the scheduler only deals with their descriptors. An example of packet descriptor is provided in Appendix E

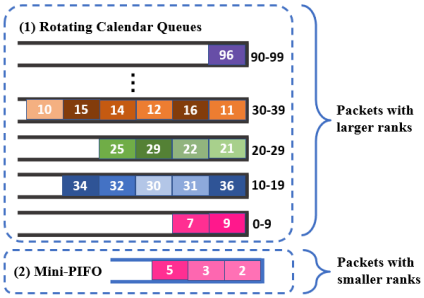


Figure 3: Sifter Architecture

Table 1: TERMS AND NOTATIONS

Notation	Description
$P_{(i,k)}$	k^{th} packet in flow i
$R_{(i,k)}$	Rank of packet $P_{(i,k)}$
K	Speed-up factor of Sifter
R_M	Memory bandwidth
R_o	Line rate of the output link
$\min L_P$	Minimal packet size
L_d	Byte length of a packet descriptor
S_F	Max size of a FIFO in the RCQ
F	Total number of FIFOs in the RCQ
g	Granularity of a FIFO in the RCQ
O_P	Current occupancy of the Mini-PIFO
S_P	Max size of the Mini-PIFO
Th_S	Sifting threshold of the Mini-PIFO
s	Rank value of the Sentinel

better illustrate the detailed schemes in Sifter, we summarize the related concepts and notations in Table 1.

3.2 Enqueue, Dequeue and Sifting Processes

Sifter has three major processes: Enqueue, Dequeue, and Sifting. Figure 4 shows the workflow of Sifter. When packets arrive at Sifter, those with larger ranks enter the RCQ and those with smaller ranks enqueue directly into the Mini-PIFO. When the Mini-PIFO's occupancy is below a given threshold described below, Sifter moves packets from the RCQ to the Mini-PIFO through "Sifting". If the Mini-PIFO becomes full, the packet with the largest rank is evicted from the tail of the Mini-PIFO back to the RCQ as it is replaced with a smaller-rank packet. In the dequeue process, Sifter dequeues the packet with the smallest rank from the head of the Mini-PIFO.

Enqueue The enqueue process decides whether the packet should be placed in the RCQ or the Mini-PIFO. As described in *Section* 3.1, the Mini-PIFO stores the packets with the smallest ranks in the scheduler. To preserve this property, if a newly arriving packet has a rank smaller than the smallest rank in the RCQ, it is stored in the Mini-PIFO. Here we introduce the concept of Sentinel, which represents the maximum rank value of a packet that can enter the Mini-PIFO. In most cases[9], Sentinel is equal to the smallest rank among the packets in the RCQ.

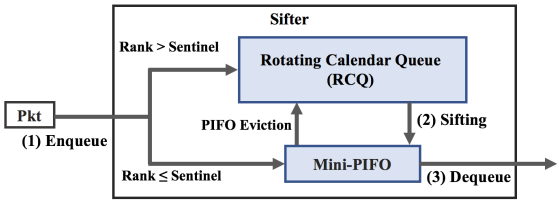


Figure 4: Sifter Workflow

The enqueue process consists of two steps: (1) Determine the target storage element (RCQ or Mini-PIFO) and (2) Enqueue into that storage element. The target storage element of a packet depends on the comparison between its rank and the Sentinel value. If a packet has a rank smaller than or equal to the Sentinel value, it is stored in the Mini-PIFO, otherwise, it is stored in the RCQ. For the second (enqueue) step, packets are enqueued in the Mini-PIFO and the RCQ differently. Enqueuing in the RCQ is straightforward: the packet is stored in the FIFO that accommodates its rank based on the FIFOs' granularity g. To enqueue in the Mini-PIFO, the packet's position in the PIFO is determined by its rank. If the Mini-PIFO is full when a new packet is enqueued, the packet with the largest rank is evicted back to the RCQ. Sifter updates the Sentinel to the evicted packet's rank and stores that packet in the RCQ as described above.

Dequeue Based on the property that the Mini-PIFO always holds the packets with the smallest rank in the scheduler, Sifter always dequeues the packet at the head of the Mini-PIFO. As long as Sifter maintains the above property and keeps sifting the packets with smaller ranks from the RCQ into the Mini-PIFO, the dequeue process will always output the packet with the smallest rank in the scheduler.

Sifting The sifting process is triggered by a "Sifting Threshold", denoted as Th_S. When the occupancy of the Mini-PIFO is lower than Th_S and the RCQ is not empty, Sifter finds the earliest[10] non-empty FIFO in the RCQ and transfers packets from it to the Mini-PIFO. The function and configuration of the Sifting Threshold Th_S are discussed in further detail in *Section* 3.4

The process of sifting the packets from the RCQ into the Mini-PIFO results in a strict ordering of the smallest ranks in the Mini-PIFO. Since the RCQ sorts packets with granularity g in each FIFO queue, the earliest non-empty FIFO queue contains the packets with the smallest rank in RCQ. Sifter needs to traverse all the packets in this FIFO to ensure that the packets it migrates to the Mini-PIFO have ranks smaller

[9] When outside of the Sifting process.

[10] Earliest FIFO: the FIFO that covers the lowest-valued rank range.

than all other packets remaining in the RCQ. This is accomplished by transferring all the packets from this FIFO to the Mini-PIFO such that the packets with the smaller ranks will remain in the Mini-PIFO while the others are evicted back to the RCQ if the Mini-PIFO overflows.

To start the sifting process, Sifter first updates the Sentinel value s to the maximum rank that the FIFO queue accommodates to guarantee that the Sentinel does not block that FIFO's packets from entering the Mini-PIFO. If the Mini-PIFO becomes full, as new packets enter the Mini-PIFO and the packets with the largest ranks are evicted back to the RCQ, Sifter updates the Sentinel s to the rank of the last evicted packet, as in the enqueue process. We further explain the Sentinel update mechanism in *Section* 3.3. When all packets from the earliest non-empty FIFO are transferred, the sifting process is completed and, as a result, the packets with the smallest ranks are held in the Mini-PIFO. If, following a sifting round, the Mini-PIFO occupancy is still lower than the sifting threshold Th_S, Sifter again finds the next earliest non-empty FIFO and repeats the sifting process until the occupancy of the Mini-PIFO is higher than Th_S.

We describe the Enqueue, Dequeue and Sifting processes in pseudo-code in Appendix I.

3.3 Sentinel Updates

The Sentinel s is a key element that controls the selection of storage elements (Mini-PIFO vs. RCQ) when a packet is enqueued. As described in *Section* 3.2, the sentinel represents the smallest rank in the RCQ and therefore guarantees that the packets that enqueue into the Mini-PIFO have ranks smaller than all packets in the RCQ. To maintain this property, the Sentinel needs to be updated with each eviction from the Mini-PIFO. Since the most recently evicted packet has the smallest rank among the evicted packets, it is always correct to update the Sentinel with the rank of the recently evicted packet[11].

Figure 5 illustrates an example of a sifting process with Sentinel updates. At stage (1), the occupancy of the Mini-PIFO is lower than the "Sifting Threshold" Th_S, which means Sifter needs to sift packets from the earliest non-empty FIFO in the RCQ into the Mini-PIFO. Sifter determines the FIFO that it needs to sift from, which covers the rank range from 20 to 29. To make sure all packets have the opportunity to enter the Mini-PIFO, the Sentinel value s is updated to 29, the highest possible rank in the FIFO. As Sifter transfers packets from the RCQ to the Mini-PIFO, the Mini-PIFO fills up and evicts the packet with the largest rank as shown in stage (2). In this example, the packet with a rank of 28 is evicted and sent back to the RCQ. Note that this packet with a rank of 28 will stay in the RCQ until the next sifting round, which means that other packets with ranks larger than 28 will not enter the Mini-PIFO until then. To achieve this, Sifter updates the Sentinel value s to 28 as described in *Section* 3.2. In stage (3), although the Mini-PIFO is not full, the packet with rank 29 cannot enqueue into the Mini-PIFO since its rank is larger than the Sentinel value s (28). The Sentinel guards the entrance of the Mini-PIFO to keep the packets with larger ranks out and thus guarantees all the packets in the Mini-PIFO have ranks smaller than any packet in the RCQ after each sifting round [12].

3.4 Sifting Threshold

The configuration of the Sifting Threshold Th_S is critical to the performance of Sifter. It serves as a watermark below which the Mini-PIFO is guaranteed to hold the packets with the smallest ranks. As stated in *Section* 3.2, Sifter replenishes the Mini-PIFO when its occupancy is below this threshold using the sifting process that must traverse *all* the packets in the earliest non-empty FIFO of the RCQ. Therefore, this threshold Th_S must be high enough, with enough packets below it to guarantee that the Mini-PIFO will not underrun before the sifting process is complete.

Although it would be reasonable to propose that the sifting threshold Th_S be large, at the limit as large as the size of the Mini-PIFO S_P, which would keep the Mini-PIFO fully utilized, a large sifting threshold would result in too-frequent sifting operations. If Th_S is set as large as S_P, Sifter would trigger the sifting process after each packet dequeue. Since the sifting process requires the transfer of all packets from the earliest non-empty FIFO, frequent sifting rounds would be unnecessarily wasteful, consuming a lot of energy with no performance gain. Therefore, the sifting threshold Th_S must be the smallest value that guarantees that the Mini-PIFO does not underrun.

Figure 6 shows the Mini-PIFO of size S_P and a sifting threshold, Th_S. According to Figure 6, the Mini-PIFO is bisected by the sifting threshold Th_S. The segment below (to the right of) Th_S is guaranteed to be occupied as long as packets are available in the RCQ. As mentioned above, this segment holds enough packets to dequeue before the sifting process is completed. The segment above (to the left of) Th_S determines the interval between two sifting rounds. Typically, the sizes of the FIFOs in the RCQ S_F are larger than the Mini-PIFO S_P, which means the Mini-PIFO would be filled up after each Sifting process and it would take $S_P - Th_S$ dequeues until the Mini-PIFO occupancy triggers the next Sifting. In other words, a larger segment above Th_S leads to less frequent sifting.

We provide the quantitative relationship between the FIFO size S_F, the Mini-PIFO size S_P and the sifting threshold Th_S that guarantees the condition for strict-order scheduling in *Section* 3.5. In *Appendix* C, we evaluate the average num-

[11] Before updating the Sentinel, Sifter compares the current Sentinel s with the rank of the evicted packet and updates s to the smaller rank

[12] Sentinel is initialized to ∞ and will be reset to ∞ if RCQ is empty.

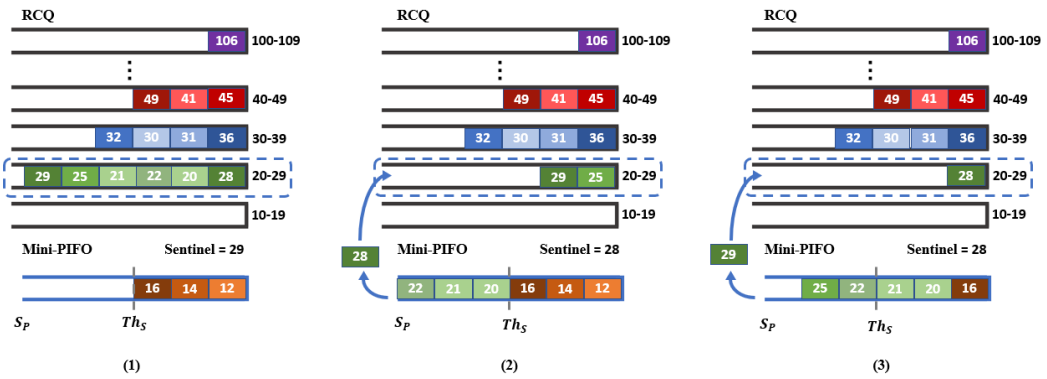


Figure 5: Sifting and Sentinel Updates

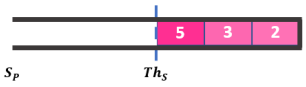


Figure 6: The Mini-PIFO and the Sifting Threshold

ber of extra memory accesses[13] per packet to show that a larger space above the sifting threshold Th_S would decrease the number of extra memory accesses introduced by the sifting process.

3.5 Condition for Inversion-free Scheduling

To guarantee there are no packet inversions in Sifter, we must ensure that each dequeue will output the packet with the smallest rank in the scheduler. According to the dequeue process described in *Section* 3.2, Sifter always dequeues the packet at the head of the Mini-PIFO. Therefore, Sifter needs to guarantee that the Mini-PIFO always holds the packet with the smallest rank at its head. We introduce **Property 1** of Sifter:

Property 1: The Mini PIFO always holds the packet with the smallest rank at its head.

To maintain **Property 1**, Sifter needs to guarantee that all the packets sifted into the Mini-PIFO have smaller ranks than the ones that remain in the RCQ after each sifting process. Thus Sifter needs to traverse all the packets in the earliest FIFO before the original packets in the Mini-PIFO drain out. That is the packets below the sifting threshold Th_S multiplied by the speed-up factor K should be larger than the maximum size of a FIFO S_F in the RCQ. We have:

$$Th_S * K \geq S_F \tag{3}$$

In addition to condition (3), Sifter must ensure that the Mini-PIFO holds at least Th_S packets before traversing through the earliest FIFO in the RCQ.

[13] Average number of extra memory accesses: the average number of extra memory access of a packet descriptor from the time it enters and until it leaves the packet scheduler when compared with a simple FIFO queue.

The size of the Mini-PIFO region between S_P and Th_S determines the number of packets transferred to the Mini-PIFO in each sifting process, assuming the source FIFO has enough packets. When Sifter sifts from a FIFO that has more than $(S_P - Th_S)$ packets, the Mini-PIFO will be loaded with $(S_P - Th_S)$ newly sifted elements at the end of the sifting process. If the earliest FIFO in the RCQ has fewer than $(S_P - Th_S)$ packets, Sifter will keep transferring from the next FIFOs until the Mini-PIFO fill level reaches Th_S as described in *Section* 3.2. Accordingly, Sifter guarantees a minimal Mini-PIFO fill level after each sifting as follows:

$$\min\{(S_P - Th_S), Th_S\} \tag{4}$$

Since Sifter needs to ensure there are at least Th_S packets at the beginning of the sifting process, we have the following additional condition:

$$S_P - Th_S \geq Th_S \tag{5}$$

Rearranging terms:

$$S_P \geq 2 * Th_S \tag{6}$$

Combining conditions (3) and (6), Sifter always guarantees **Property 1**, regardless of the state of the sifting process. Therefore, Sifter guarantees there are no packet inversions.

4 Sifter Hardware Prototype

Figure 7 shows an abstraction of the system-level application of a programmable packet scheduler.

Usually, the size of the packet descriptor[14] is much smaller than the packet itself. Thus, packet schedulers in high-speed data planes need to process only the descriptors rather than the packets themselves due to the limited packet processing time. According to Figure 7, there are two paths in the system: the *Packet Path* and the *Descriptor Path*. In the *Packet*

[14] In general, a packet descriptor in the data plane includes metadata related to the packet, including length, flow ID, memory pointer, etc.)

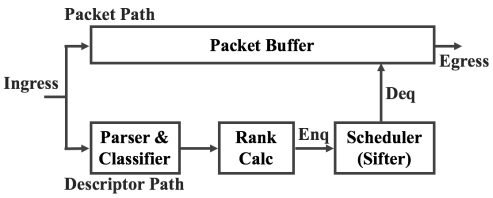


Figure 7: System-Level Packet Scheduler Application

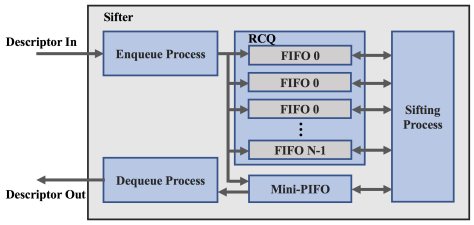


Figure 8: Sifter Block Diagram, VHDL Implementation

Path, the ingress packets are stored in the packet buffer and wait to be dequeued or dropped. The packet scheduling takes place in the *Descriptor Path*. The packet classifier and parser extract the packet information from the packet header and generate a packet descriptor by combining it with the memory address of the packet in the buffer. Next, the packet rank calculator computes the rank of the packet according to the applicable scheduling algorithm and updates the descriptor. Finally, the system enqueues the descriptor into the packet scheduler[15], which drops or dequeues descriptors according to their ranks. When the scheduler dequeues or drops a packet descriptor, the system accesses the associated packet in the buffer, referenced by its memory address, and dequeues or drops the corresponding packet. Since this paper focuses on programmable packet scheduling, the hardware prototype design section also targets the *Descriptor Path* in the system.

Hardware Prototype Architecture We implemented the Sifter hardware prototype in VHDL according to the algorithms in *Section* 3. Figure 8 shows a block diagram of the Sifter hardware prototype. The RCQ and Mini-PIFO hold descriptors during the enqueue and dequeue operations. The Enqueue block stores the packet in one of the FIFOs in the RCQ or in the Mini-PIFO according to the descriptor's rank. The Sifting process transfers the descriptors between the RCQ and the Mini-PIFO to ensure that those with the smallest ranks are held in the Mini-PIFO. Finally, the Dequeue block outputs the descriptors with the smallest rank.

There are multiple scaling parameters in the VHDL implementation of Sifter, including: (1) the number and size of FIFOs in the RCQ, (2) the size and sifting threshold of the Mini-PIFO, and (3) other sizing parameters for memories and logic. These parameters improve the flexibility and scalability of our design[16].

We evaluate the performance impact and the resource overhead of adjusting different design parameters in *Appendix* D.

We designed a 72-bit packet descriptor[17] for the Sifter hardware prototype, and Sifter schedules packets according to a 20-bit *Packet Rank* field.

[15] Sifter is a specific implementation of such a packet scheduler.

[16] The VHDL code, test bench, and FPGA implementation files are available at *https://github.com/Sifter-NSDI24/Sifter-NSDI24*

[17] The details of the packet descriptor are revealed in *Appendix* E.

Rank Computation We programmed SFQ[26], a virtual-clock-based fairness packet scheduling scheme, on the Sifter hardware prototype. The transmission time is pre-calculated by the packet classifier and stored in the *Packet Rank* field in the packet descriptor. The *Enqueue Process* module updates the *Packet Rank* by the system virtual clock and the last rank of the corresponding flow.

According to the definition of programmable packet scheduling in *Section* 2.1, most packet scheduling schemes compute the ranks in the end-hosts or flow tables prior to entering the packet scheduler[35] [48] [49]. Therefore, it would be straightforward to extend the Sifter prototype to support a wide variety of packet scheduling schemes with the pre-calculated ranks in the packet descriptors.

Fast RCQ FIFO Indexing For each packet descriptor that enqueues into the RCQ module, Sifter needs to find the FIFO associated with its rank. We call this operation FIFO indexing. RCQ performs FIFO indexing when a new packet descriptor arrives or when a packet descriptor is evicted from the Mini-PIFO, which occurs whenever the Mini-PIFO becomes full and additional descriptors need to be stored in the RCQ. Thus, FIFO indexing must be extremely fast in the hardware implementation.

Rather than scanning through the rank bounds of each FIFO, RCQ directly finds the corresponding FIFO index by extracting a range of bits in the *Packet Rank*. We configured the number of FIFOs F and the granularity g as powers of two to be able to use bit slice selection and shifting operations in the hardware implementation rather than slower math operations, such as multiplication[18].

Mini-PIFO The Mini-PIFO is implemented using a shift register structure and performs a push operation in two clock cycles. In the first clock cycle when a new descriptor is pushed, its rank is compared in parallel to those of all descriptors stored in the Mini-PIFO and an insertion position is determined. In the second clock cycle, the descriptors with higher ranks are shifted one position towards the tail, and the new descriptor is inserted in the newly created "hole". In the process of shifting towards the tail, one descriptor may be evicted and stored in the RCQ if the Mini-PIFO was previously full. The pop operation simply outputs the descriptor

[18] Additional details of fast FIFO indexing are provided in *Appendix* E

at the head of the Mini-PIFO and shifts all other descriptors toward the head. Once the Mini-PIFO fill level reaches the reload threshold, the sifting process starts a reload operation to replenish the PIFO from the RCQ.

The Mini-PIFO can also handle simultaneous push and pop operations, whether the arriving descriptor needs to be popped immediately or inserted in a Mini-PIFO location.

In addition, the Mini-PIFO has a logic "wrapper" that handles the resolution of race conditions that may occur if enqueue and reload operations generate simultaneous push operations toward the PIFO.

Search for The First Non-empty FIFO When the Mini-PIFO triggers a sifting process, RCQ finds the earliest non-empty FIFO queue to replenish the Mini-PIFO with packet descriptors. The FIFOs in the RCQ are organized as a circular list with an index indicating the current FIFO (earliest time range) corresponding to the current value of the virtual clock. The search for the first non-empty FIFO is performed in parallel on two groups of FIFOs: (1) First Group: From the current FIFO to the highest numbered FIFO in the range; (2) Second Group: From FIFO zero to the FIFO preceding the current FIFO.

Because of the circular structure, FIFOs in the First Group hold descriptors that have smaller ranks than those in the Second Group. Therefore, if the first non-empty FIFO is found in the First Group, it is selected as the next non-empty FIFO. Otherwise, the first non-empty FIFO in the Second Group is selected.

5 Evaluation

In *Section* 5.1 and 5.2, we evaluate Sifter using the NS3 [37] packet-based simulator and show the advantages of Sifter in fairness, FCT, and delay due to the elimination of packet scheduling inversions. Furthermore, we demonstrate the flexibility of Sifter with two use cases. Finally, in *Section* 5.3 we implement Sifter on an AMD/Xilinx Alveo U250 FPGA board and evaluate its functionality, performance as well as resource utilization.

5.1 Micro-benchmark Evaluation

We first set up a micro-benchmark to evaluate the performance of Sifter in detail at the packet level. The evaluated performance metrics include fairness in short-term bandwidth allocation shown in *Section* 5.1.2, the FCT and end-to-end delay given in *Appendix* F, as well as the average extra number of memory accesses shown in *Appendix* C.

5.1.1 Micro-benchmark setup

Topology Given that the micro-benchmark aims to assess the intricate packet-level performance of schedulers, it is advisable to maintain a compact topology scale. This approach facilitates a more straightforward observation of detailed metrics. In the micro-benchmark, we set up a single-node star topology with multiple end hosts connected via one switch with programmable packet schedulers. All links have a bandwidth of 10 Gbps and a delay of $3\mu s$.

Traffic We prepared two sets of traffic for different evaluation proposes: (1) A set with small-scale traffic that contains 8 flows to evaluate the convergence of fair bandwidth allocation and (2) A set with large-scale traffic that contains approximately $1k$ flows to evaluate the FCT and end-to-end delay. To create network congestion on the bottleneck link, we generate the TCP flows[19] in an in-cast traffic pattern.

Schedulers Setup We applied packet schedulers on the bottleneck link to evaluate their performance under congestion conditions. The evaluated schedulers include Sifter, Programmable Calendar Queues (PCQ) [45] and SP-PIFO [2]. Each of the schedulers has 16 FIFO queues with a depth of 64[20]. Sifter has a mini-PIFO with a capacity of 32 descriptors. In addition, we also introduce an ideal PIFO[48][49] with a capacity of 1024 and a standalone Mini-PIFO with a capacity of 32 to serve as benchmarks.

5.1.2 Convergence to fair bandwidth allocation

There are 8 TCP flows in the micro-benchmark test with the small-scale traffic, with each flow becoming active at successive 2 *ms* intervals. Flow 8 stops 2 *ms* after its arrival, and other flows stop progressively in reverse order of their arrival at intervals of 2 *ms*. To evaluate the fairness of short-term bandwidth allocation, we measure the throughput of each TCP flow in a window size of 120 μs, which is ten times the RTT. The evaluation results are shown in Figure 9.

Sifter converges to fairness in bandwidth allocation Figure 9 shows that Sifter has a fair bandwidth allocation close to that of an ideal PIFO. The 8 TCP connections in Sifter converge to the fair share of bandwidth very fast and with minimum fluctuations. On the other hand, PCQ and SP-PIFO experience packet inversions, which result in fluctuations in the short-term throughput as shown in Figure 9. The fluctuation is due to the fact that PCQ and SP-FIFO store packets in FIFOs. Packets in the same FIFO are scheduled on a first-come-first-served basis, which means that multiple packets from a flow could continuously enter the same FIFO in a short burst, and later leave the scheduler continuously in a burst. Such burstiness impairs the short-term fairness of bandwidth allocation and the steadiness of the throughput.

Sifter eliminates packet inversions In the simulation results, we observe no inversions in Sifter while PCQ and SP-PIFO exhibit a significant number of packet inversions with large magnitudes due to their FIFO-based architectures. The

[19] We use TCP New Reno in our micro-benchmark simulation

[20] An Insufficient FIFO size can result in sub-optimal performance due to overflows for both FIFO-based schedulers and Sifter. We will discuss the overflows in Sifter in Appendix B.

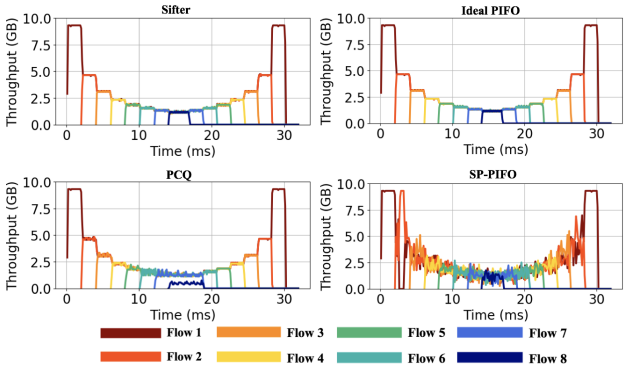


Figure 9: Convergence for fair bandwidth allocation

consequence of inversions is reflected in the fluctuations of flow throughput as shown in Figure 9.

5.2 Large-scale Packet-based Simulation

To evaluate the performance of packet schedulers in data-center environments, we use large-scale packet-based simulations with a fat-tree topology with a variety of larger-scaled empirical traffic loads.

We evaluate two use cases of programmable packet schedulers: (1) Weighted max-min fairness in bandwidth allocation with SFQ [26] in *Section* 5.2.2 and (2) Minimizing FCT with modified SRPT[3] in *Appendix* G.

5.2.1 Simulation Setup

Network topology We extend the topology to a 3-tier fat-tree consisting of 4 core switches, 8 aggregation switches, and 8 ToR switches. All the switch nodes are connected with $40Gbps$ links with $1\mu s$ delay. A total of 256 hosts are connected by the fat-tree topology. Each ToR switch connects with 32 hosts and the links between each host and the ToR switch each have a bandwidth of $10Gbps$ with $10ns$ delay.

Simulation Traffic We generate two types of empirical traffic with different patterns: (1) *random pattern* and (2) *incast pattern*. For the *random pattern*, each generated TCP flow randomly selects a source node and a destination node among the 256 hosts in the topology, while the flows in the *incast pattern* randomly select a source node but always have a fixed destination node to represent an in-cast traffic pattern. The arrival time of each TCP flow follows a Poisson distribution, and the flow size follows the Web-search distribution in data centers [3].

5.2.2 Use Case: Start-time Fair Queuing for weighted max-min fairness in bandwidth allocation

Weighted max-min fairness in bandwidth allocation is one of the most important goals of packet scheduling [46] [6] [41] [32] [36]. Among the algorithms derived from Weighted Fair Queueing (WFQ), Start-time Fair Queueing (SFQ) [26] is the most popular one due to its simplicity and accuracy. Therefore, we applied SFQ to all the schedulers in the simulation. We configured all three schedulers (Sifter, PCQ, and SP-PIFO) with 32 FIFO queues with a depth of 256 entries. In addition, Sifter has a Mini-PIFO with 64 entries[21]. Furthermore, we set two PIFO benchmarks with different sizes: an ideal PIFO with a large capacity of 1024 entries and a standalone Mini-PIFO with 64 entries to match the Mini-PIFO in Sifter.

Sifter is a close approximation to the ideal PIFO Figure 10 $(a)-(d)$ shows the Normalized FCT[22] performance of the different packet schedulers under *random pattern* and *in-cast pattern* respectively. As shown in Figure 10, Sifter has an FCT performance that is nearly identical to the ideal PIFO. Similarly, the end-to-end delay performance of Sifter matches closely that of the ideal PIFO. Figure 10 $(e)-(h)$ shows the 95th percentile end-to-end delay of the small flows ($\leq$50MB) with different setups. By providing an accurate scheduling order, Sifter guarantees that packets with smaller ranks are scheduled without the extra delay that would be caused by scheduling inversions.

Compared with the standalone Mini-PIFO, the RCQ of Sifter extends the buffer capacity and thus reduces the packet loss rate for mid-sized flows. As Figure 10 $(a)-(d)$ shows, Sifter outperforms the Mini-PIFO among the mid-sized flows by reducing packet loss and re-transmissions.

Sifter is stable under different traffic patterns Approximate packet schedulers are subject to the impacts of certain traffic patterns while Sifter achieves a stable performance close to that of the ideal PIFO.

As shown in Figure 10, PCQ has poor FCT and tail-latency performance for small-sized flows for both *random* and *in-cast* patterns. As illustrated in *Section* 2.3.1, the FIFO-based calendar queue structure of PCQ introduces inversions and extra delays for packets within one FIFO queue. The packets with small ranks from the short flows are significantly affected by the extra delay, which results in significant performance degradation for small flows between PCQ and other schedulers.

Although SP-PIFO provides decent performance with the *random pattern*, its performance degrades as the congestion level increases for the *in-cast pattern*. Queue-bound adjustments in SP-PIFO tend to put packets with larger ranks into the queues with lower priorities. Since SFQ is a virtual-clock-based scheduling algorithm where the ranks of incoming packets keep increasing as the congestion persists, SP-PIFO stores most of the packets in the last few FIFO queues. We observe that SP-PIFO does not fully utilize all of its

[21] In the Mini-PIFO of Sifter, we set the Sift threshold $Th_S = 32$ (half of the PIFO size), and used a speed-up factor $K = 4$ to fulfill the conditions in *Section* 3.5.

[22] Normalized FCT: The measured FCT normalized to the ideal FCT.

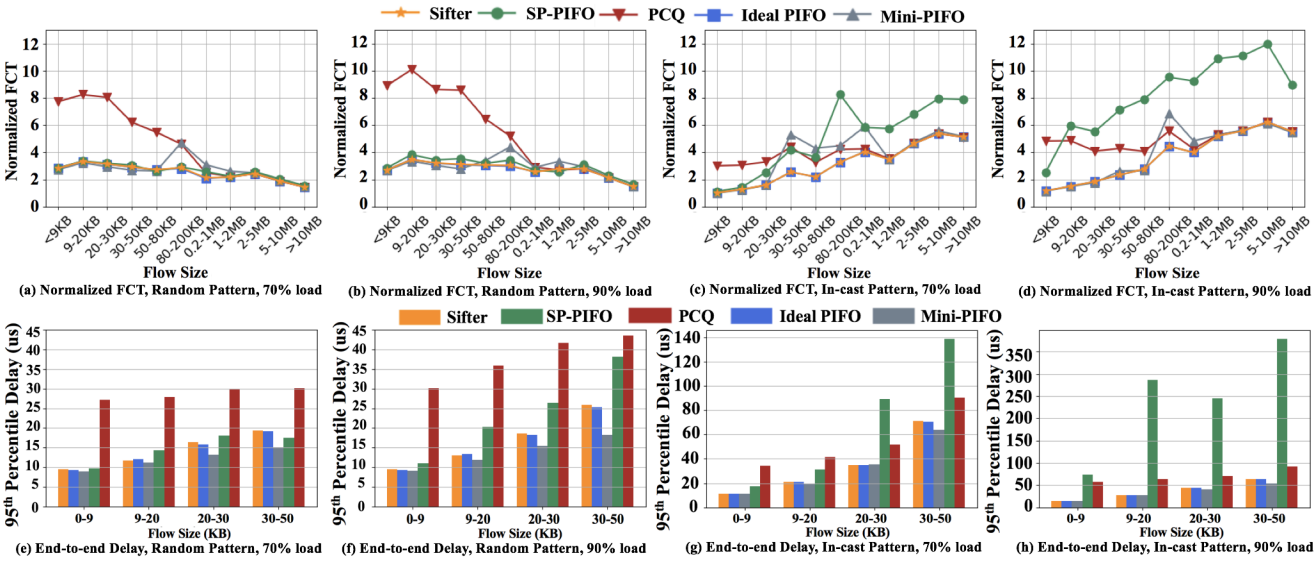


Figure 10: Normalized FCT and End-to-End Delay with SFQ

available FIFO queues, and that the last few FIFOs become particularly crowded under heavy congestion. This queue congestion increases the end-to-end delay as Figure 10 (g) and (h) shows and degrades the FCT of large flows as shown in Figure 10 (c) and (d).

5.3 FPGA Hardware Testbed

We built a hardware testbed for the Sifter prototype on an AMD/Xilinx Alveo U250 board [4], which uses an FPGA similar in size to an UltraScale+ VU13P with mid-speed grade. The hardware evaluation of the Sifter prototype covers three aspects: (1) Line rate measurement; (2) Verification of no inversions in the output of Sifter to validate error-free scheduling; (3) Evaluation of the hardware resources for different configurations by varying the design parameters.

Hardware Testbed Structure According to *Section* 4, we focus on the descriptor path in the system. Therefore, in our hardware testbed evaluation, we feed packet descriptors to our hardware prototype and scale their input and output rates to achieve a line rate of 100*Gbps*.

Figure 11 shows the structure of the hardware testbed. We use two sets of packet descriptors parsed from a PCAP file.

For simulation purposes, we implemented the testbed in Cocotb [21], a Python-based HDL simulation environment. For the hardware testing, we implemented the same testbed using Exegy's nxFramework [22], which provides the PCIe connectivity and C++ device drivers to access registers and memories within Sifter.

The input packet descriptors are originally generated from a PCAP packet capture [33]. We load the input packet descriptors in the *Input Buffer* module on the FPGA, along with time gaps between packets to set a line rate equivalent to 100*Gbps*. The *Enq Rate Ctl* module reads the descriptors from the input buffer and enqueues them into Sifter at the specified rate. After a software-programmable delay, the *Deq Rate Ctl* module starts dequeuing packet descriptor from Sifter at a rate equivalent to 100*Gbps*, by using the packet length information to control the descriptor dequeue rate. Dequeued descriptors are written to the Output Recorder block along with timestamps and read out and written to an output file by software at the end of the test. We analyze the packet descriptor output file from the *Output Recorder* to measure the line rate and validate that there are no packet inversions or missing descriptors.

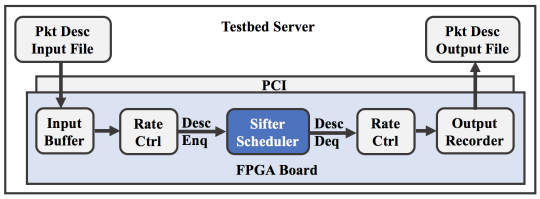
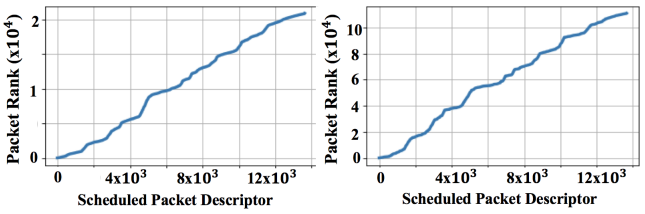


Figure 11: Sifter Hardware Testbed

Running Sifter on an FPGA We configured Sifter with 32 FIFOs with 32 locations each and a Mini-PIFO with 32 locations. We used Vivado 2021.2 to generate a bitstream of the test bed including the Exegy PCIe memory-mapped I/O (MMIO) logic and loaded it on an AMD/Xilinx Alveo U250 board [4]. With the above design parameters, the Sifter prototype achieved a frequency of 322MHz and used less than 1.25% of the FPGA. [23] We used C++ code and the Exegy MMIO device drivers to write the input data into the *Input Buffer* and read the output data from the *Output Recorder*.

[23] Additional information about the trade-off of different design parameters with FPGA resource overhead is given in *Appendix* D.2.



(a) Scheduling Order, fixed packet size (b) Scheduling Order, varying packet size

Figure 12: Inversion-free Scheduling Order of Sifter Prototype

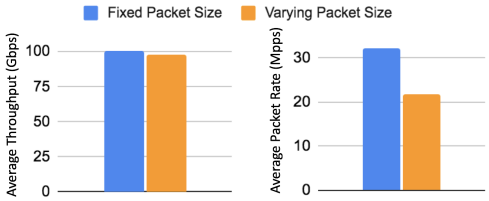


Figure 13: Sifter Hardware Throughput and Packet Rate

Traffic Trace We generated the packet descriptor inputs by parsing a PCAP packet capture, which includes 13708 packets and 941 flows. The average packet size of the input trace is 823 Bytes. With the max frequency of 322 MHz of its FPGA prototype, Sifter can achieve a line rate of 100Gbps with packets larger than 370 Bytes[24].Therefore, we modified the packet size of the PCAP file and generated two sets of packet descriptor inputs: (1) A fixed packet size input where all the packets have the same size of 370 Bytes and (2) A varying packet size input by modifying the small packets size to 370 Bytes while preserving the original packet sizes of other packets that are larger than 370 Bytes

Inversion-free Packet Scheduling Figure 12 shows the packet descriptor scheduling order from the output of our testbed. For both fixed and varying packet size traces, there are no inversions in the packet scheduling order and the packet loss rate[25] of both input traces is lower than 0.6%.

Scaling to 100Gbps Line Rate Figure 13 shows the throughput and the packet rate of Sifter's hardware prototype under fixed and varying packet size inputs. From the output of our hardware testbed, Sifter reaches the line rate of 100 Gbps for both types of input. For the fixed packet size input, the Sifter prototype operates at a constant packet rate of 32.2 Mpps, since the packet processing time of Sifter is 10 clock cycles. On the other hand, for the varying packet size input, the line rate is also 100*Gbps* with an average packet rate of 21.6 Mpps because the average packet size is larger.

[24]This packet length allows Sifter to dequeue at 100*Gbps* without inversions on our prototype. With a higher clock frequency, Sifter supports a 100*Gbps* line rate with 64-byte packets. See *Appendix* E and H for details

[25]Packet loss may occur if FIFOs become full and is not related to Sifter's operation. See *Appendix* B for more information on overflows in Sifter.

6 Related Work

In the 1990s, early implementations of general packet schedulers such as Sequencer [13] [14] [15] sort packets accurately by their ranks based on a shift-register design. However, the architecture of these ASIC-based packet schedulers limits their capacity. In the 2000s, pipeline-Heap (pHeap) [7][29] was proposed, which serves as an accurate packet scheduler that supports a large buffer capacity. However, the high implementation overhead of pHeap takes up a significant area on the switch chip. This issue would be compounded if a pHeap is used for each port on the switch.

In the 2010s, the advent of the programmable data plane brought to light the topic of implementing general packet schedulers in academia and industry. PIFO [48] [49] is a fundamental building block for programmable packet schedulers. However, while its design achieves accurate programmable packet scheduling with a small chip area overhead, it is still limited by its capacity and the requirement of special hardware support such as Ternary Content-Addressable Memory (TCAM).

PCQ [45] and SP-PIFO [2] are two approximate programmable packet schedulers based on strict-priority queues. While they greatly reduce implementation overhead and boost the capacity of the scheduler, the impact of the packet inversions they introduce is not negligible. Approximate Fair Queuing (AFQ) [44] and Gearbox [24] are two other approximate packet schedulers focused on fair queuing and weighted fair queuing, but are not generalized for programmable packet scheduling.

AIFO [54] is a programmable packet scheduler design, which only requires a single FIFO queue and filters the incoming packets using admission control. However, AIFO can cause packet inversions similar to the above-mentioned approximate packet schedulers, and scaling it to support a large buffer capacity would be challenging.

7 Conclusion

In this paper, we present Sifter, an accurate, programmable packet scheduler that operates free of packet inversions and supports a large buffer size with low implementation overhead. By taking advantage of the "Speed-up Factor", Sifter sorts packets by "Sift Sorting" and combines the advantages of accurate scheduling of a PIFO with the large capacity provided by FIFO-based approximate schedulers. Our simulations in NS3 show that Sifter achieves better fairness, lower FCT, and delay by eliminating packet inversions. We implemented Sifter in VHDL, targeting an AMD/Xilinx Alveo U250 FPGA card [4] with a mid-speed grade XCVU13P FPGA. Our FPGA-based hardware prototype operates at 322 MHz and reaches a line rate of 100 Gb/s for packets larger than 370 bytes. Sifter uses less than 1.25% of the FPGA resources.

References

[1] ADDANKI, V., APOSTOLAKI, M., GHOBADI, M., SCHMID, S., AND VANBEVER, L. Abm: active buffer management in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 36–52.

[2] ALCOZ, A. G., DIETMÜLLER, A., AND VANBEVER, L. Sp-pifo: Approximating push-in first-out behaviors using strict-priority queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 59–76.

[3] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 435–446.

[4] AMD/XILINX. AMD/Xilinx Alveo U250 Data Center Accelerator Card. `https://www.xilinx.com/products/boards-and-kits/alveo/u250.html`, 2021.

[5] AMD/XILINX. Vivado Design Suite, Integrated Design Environment. `https://www.xilinx.com/products/design-tools/vivado.html`, 2021.

[6] BALLANI, H., COSTA, P., KARAGIANNIS, T., AND ROWSTRON, A. Towards predictable datacenter networks. In *ACM SIGCOMM computer communication review* (2011), vol. 41, ACM, pp. 242–253.

[7] BHAGWAN, R., AND LIN, B. Fast and scalable priority queue architecture for high-speed network switches. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)* (2000), vol. 2, IEEE, pp. 538–547.

[8] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., ET AL. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review 44*, 3 (2014), 87–95.

[9] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review 43*, 4 (2013), 99–110.

[10] BROADCOM. Broadcom StrataDNX™ BCM88480 Traffic Management Architecture. `https://docs.broadcom.com/doc/88480-DG1-PUB`, 2021.

[11] BROADCOM. High Capacity StrataXGS®Trident II Ethernet Switch Series. `http://www:broadcom:com/products/Switching/Data-Center/BCM56850-Series.`, 2021.

[12] BROWN, R. Calendar queues: a fast 0 (1) priority queue implementation for the simulation event set problem. *Communications of the ACM 31*, 10 (1988), 1220–1227.

[13] CHAO, H. J. Architecture emersesign for regulating and scheduling user's traffic in atm networks. In *ACM SIGCOMM Computer Communication Review* (1992), vol. 22, ACM, pp. 77–87.

[14] CHAO, H. J., CHENG, H., JENQ, Y.-R., AND JEONG, D. Design of a generalized priority queue manager for atm switches. *IEEE Journal on Selected Areas in Communications 15*, 5 (1997), 867–880.

[15] CHAO, H. J., JENQ, Y.-R., GUO, X., AND LAM, C.-H. Design of packet-fair queuing schedulers using a ram-based searching engine. *IEEE Journal on Selected Areas in Communications 17*, 6 (1999), 1105–1126.

[16] CHOUDHURY, A. K., AND HAHNE, E. L. Buffer management in a hierarchical shared memory switch. In *Proceedings of INFOCOM'94 Conference on Computer Communications* (1994), IEEE, pp. 1410–1419.

[17] CISCO. Cisco Nexus 7700 F3-Series 12-Port 100 Gigabit Ethernet Module Data Sheet. `https://www.cisco.com/c/en/us/products/collateral/switches/nexus-7000-series-switches/data_sheet_c78-728423.html`, 2019.

[18] CISCO. Cisco Silicon One Product Family White Paper. `https://www.cisco.com/c/en/us/solutions/silicon-one.html`, 2021.

[19] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to algorithms*. MIT press, 2009.

[20] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM Computer Communication Review* (1989), vol. 19, ACM, pp. 1–12.

[21] DEVELOPMENT GROUP, C. Cocotb chip design testbench. In *https://www.cocotb.org/*. 2023, 2023, p. 1.

[22] EXEGY. Exegy nxframework. In *https://www.enyx.com/nxframework/*. 2022, 2022, p. 1.

[23] FENG, Y., CHEN, Z., SONG, H., XU, W., LI, J., ZHANG, Z., YUN, T., WAN, Y., AND LIU, B. Enabling in-situ programmability in network data plane: From architecture to language. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 635–649.

[24] GAO, P., DALLEGGIO, A., XU, Y., AND CHAO, H. J. Gearbox: A hierarchical packet scheduler for approximate weighted fair queuing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 551–565.

[25] GOLESTANI, S. J. A self-clocked fair queueing scheme for broadband applications. In *Proceedings of INFOCOM'94 Conference on Computer Communications* (1994), IEEE, pp. 636–646.

[26] GOYAL, P., VIN, H. M., AND CHEN, H. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. In *ACM SIGCOMM Computer Communication Review* (1996), vol. 26, ACM, pp. 157–168.

[27] HOGAN, M., LANDAU-FEIBISH, S., ARASHLOO, M. T., REXFORD, J., AND WALKER, D. Modular switch programming under resource constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 193–207.

[28] HONG, C.-Y., CAESAR, M., AND GODFREY, P. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication* (2012), ACM, pp. 127–138.

[29] IOANNOU, A., AND KATEVENIS, M. G. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Transactions on Networking (ToN) 15*, 2 (2007), 450–461.

[30] JANG, K., SHERRY, J., BALLANI, H., AND MONCASTER, T. Silo: Predictable message latency in the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), pp. 435–448.

[31] JEYAKUMAR, V., ALIZADEH, M., MAZIÈRES, D., PRABHAKAR, B., GREENBERG, A., AND KIM, C. {EyeQ}: Practical network performance isolation at the edge. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013), pp. 297–311.

[32] JEYAKUMAR, V., ALIZADEH, M., MAZIERES, D., PRABHAKAR, B., AND KIM, C. Eyeq: Practical network performance isolation for the multi-tenant cloud. In *Presented as part of the* (2012).

[33] KLASSEN, F., AND APPNETA. Tcpreplay. In *https://tcpreplay.appneta.com/wiki/captures.html*. 2023, 2023, p. 1.

[34] LEUNG, J. Y.-T. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica 4*, 1-4 (1989), 209.

[35] MITTAL, R., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Universal packet scheduling. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016), pp. 501–521.

[36] NAGARAJ, K., BHARADIA, D., MAO, H., CHINCHALI, S., ALIZADEH, M., AND KATTI, S. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 188–201.

[37] NETWORK SIMULATOR DEVELOPMENT GROUP, T. The network simulator 3. In *https://www.nsnam.org/*. 2022, 2022, p. 1.

[38] OPEN, H.-L. L. F. D. F.-R. S. F. P. N. P. Broadcom. np. In *https://nplang.org/*. 2019, 2019, p. 1.

[39] PAREKH, A. K., AND GALLAGER, R. G. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM transactions on networking*, 3 (1993), 344–357.

[40] PAREKH, A. K., AND GALLAGER, R. G. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM transactions on networking 2*, 2 (1994), 137–150.

[41] POPA, L., KUMAR, G., CHOWDHURY, M., KRISHNAMURTHY, A., RATNASAMY, S., AND STOICA, I. Faircloud: sharing the network in cloud computing. *ACM SIGCOMM Computer Communication Review 42*, 4 (2012), 187–198.

[42] RUFFY, F., WANG, T., AND SIVARAMAN, A. Gauntlet: Finding bugs in compilers for programmable packet processing. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), pp. 683–699.

[43] SCHRAGE, L. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research 16*, 3 (1968), 687–690.

[44] SHARMA, N. K., LIU, M., ATREYA, K., AND KRISHNAMURTHY, A. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (2018), pp. 1–16.

[45] SHARMA, N. K., ZHAO, C., LIU, M., KANNAN, P. G., KIM, C., KRISHNAMURTHY, A., AND SIVARAMAN, A. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 685–699.

[46] SHIEH, A., KANDULA, S., GREENBERG, A. G., KIM, C., AND SAHA, B. Sharing the data center network. In *NSDI* (2011), vol. 11, pp. 23–23.

[47] SHREEDHAR, M., AND VARGHESE, G. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking*, 3 (1996), 375–385.

[48] SIVARAMAN, A., SUBRAMANIAN, S., AGRAWAL, A., CHOLE, S., CHUANG, S.-T., EDSALL, T., ALIZADEH, M., KATTI, S., MCKEOWN, N., AND BALAKRISHNAN, H. Towards programmable packet scheduling. In *Proceedings of the 14th ACM workshop on hot topics in networks* (2015), ACM, p. 23.

[49] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND MCKEOWN, N. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 44–57.

[50] STEPHENS, B., SINGHVI, A., AKELLA, A., AND SWIFT, M. Titan: Fair packet scheduling for commodity multiqueue nics. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017), pp. 431–444.

[51] VARGHESE, G., AND LAUCK, A. Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility. *IEEE/ACM transactions on networking 5*, 6 (1997), 824–834.

[52] WILSON, C., BALLANI, H., KARAGIANNIS, T., AND ROWTRON, A. Better never than late: Meeting deadlines in datacenter networks. In *ACM SIGCOMM Computer Communication Review* (2011), vol. 41, ACM, pp. 50–61.

[53] XING, J., HSU, K.-F., KADOSH, M., LO, A., PIASETZKY, Y., KRISHNAMURTHY, A., AND CHEN, A. Runtime programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 651–665.

[54] YU, Z., HU, C., WU, J., SUN, X., BRAVERMAN, V., CHOWDHURY, M., LIU, Z., AND JIN, X. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (2021), pp. 179–193.

[55] YUAN, Y., ALAMA, O., FEI, J., NELSON, J., PORTS, D. R., SAPIO, A., CANINI, M., AND KIM, N. S. Unlocking the power of inline {Floating-Point} operations on programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 683–700.

[56] ZENO, L., PORTS, D. R., NELSON, J., KIM, D., LANDAU-FEIBISH, S., KEIDAR, I., RINBERG, A., RASHELBACH, A., DE-PAULA, I., AND SILBERSTEIN, M. {SwiSh}: Distributed shared state abstractions for programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 171–191.

[57] ZHANG, H., AND BENNETT, J. C. Wf2q: worst-case fair weighted fair queueing. In *IEEE INFOCOM* (1996), vol. 96, pp. 120–128.

Appendix

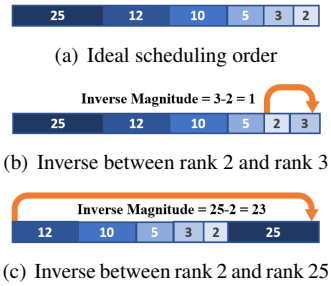
A Inversion Magnitude

In *Section* 2.2 we defined *Packet Inversions*. However, the number of packet inversions alone cannot fully reflect the severity of errors that an approximating packet scheduler may introduce. To quantify the severity of each packet inversion, we extend the concept of packet inversion with *Inversion Magnitude*.

Inversion Magnitude: When a packet with rank r departs from the scheduler and a packet with smaller rank r'[26] exists in the scheduler, the inversion magnitude is $r - r'$.

Take the example in Figure 14: There are packet inversions in both scenarios shown in Figure 14 (b) and (c), but the severities are different. In the case of Figure 14 (c), a large packet with rank = 25 is scheduled prior to a packet with a smaller rank = 2. When compared with the case of Figure 14 (b), packets with smaller ranks experience longer delays in Figure 14 (c).

As mentioned in *Section* 2.2.1, packet inversions with larger magnitudes lead to more severe consequences. For scheduling algorithms such as SCFQ [25] and SFQ [26], which update a virtual clock using packet departure times, a larger inversion magnitude leads to a worse skew of the virtual clock and results in worse throughput loss or additional delay on newly arrived flows. For the scheduling disciplines that aim to minimize tail packet delay, a larger inversion magnitude can cause packets with the least slack time to experience longer delays.



(a) Ideal scheduling order

(b) Inverse between rank 2 and rank 3

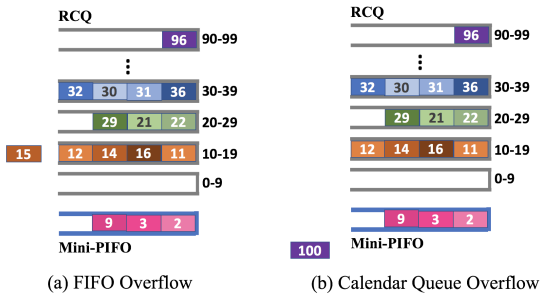
(c) Inverse between rank 2 and rank 25

Figure 14: Packet Inversion and Inversion Magnitude

B Overflows in Sifter

There are two types of overflows in Sifter: (1) *FIFO Overflow* and (2) *Calendar Queue Overflow*.

FIFO Overflow *FIFO overflows* are the cases where a packet is dropped when the associated FIFO is full. As shown in Figure 15 (a), a packet with rank 15 is dropped since the FIFO that covers the rank range from 10 to 19 is already full. Such overflows lead to a different outcome when compared with an ideal PIFO. PIFO always drops the packet with the largest rank when the capacity is full, while Sifter may drop packets with relatively smaller ranks in the cases of *FIFO overflow*.



(a) FIFO Overflow

(b) Calendar Queue Overflow

Figure 15: Overflows in Sifter

The solutions to mitigate *FIFO overflow* is straightforward: (1) extend the size of each FIFO in the RCQ or (2) reduce the FIFO granularity g so that each FIFO covers a narrower rank range, which reduces the number of packets in each FIFO. In *Appendix* D, we show that the above adjustments in the design parameters reduce the FIFO overflows.

Calendar Queue Overflow A *Calendar Queue Overflow*, occurs when an incoming packet has a rank value larger than the maximum rank that the scheduler covers. Figure 15 (b) presents an example of *Calendar Queue Overflow*, where the rank value of an incoming packet is 100 and the maximum rank value that the scheduler supports is 99. Such overflows have a similar effect as overflows in PIFOs, where packets with the largest ranks are dropped. *Calendar Queue Overflow* can be mitigated by increasing the rank range that the scheduler covers, however, if overflows are unavoidable, packets with the largest ranks are the ones to be dropped in most cases.

We further evaluate the impact of overflows and how the above solutions mitigate them in *Appendix* D.

C Average Number of Extra Memory Accesses

According to *Section* 3, the key idea for Sifter to eliminate packet inversions is to apply "Sift Sorting", which moves packet descriptors between the RCQ and the Mini-PIFO. While Sifter ensures accurate packet scheduling, the trade-off is the extra memory accesses introduced by the sifting process. In our evaluation, we quantify the number of extra memory accesses that Sifter introduces to guarantee inversion-free packet scheduling.

As defined in *Section* 3.4, the *average extra number of*

[26] The smaller rank has higher priority

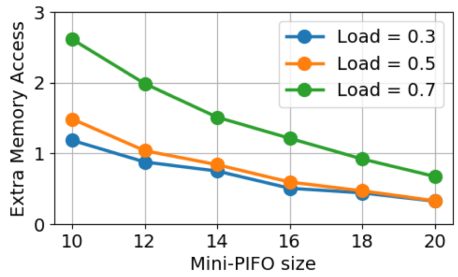


Figure 16: Average number of extra memory accesses per-packet.

memory accesses refers to the additional number of accesses to a packet descriptor between the times it is enqueued and dequeued from the packet scheduler when compared with a simple FIFO queue. For all FIFO-based packet schedulers, the number of memory accesses for each packet descriptor is equivalent to that of a simple FIFO queue, which is one access when writing the packet descriptor into the FIFO and another access when reading it out.

We conducted multiple tests with our single-node micro-benchmark in *Section* 5.1 to evaluate the *average number of extra memory accesses* with different Mini-PIFO sizes S_P. We used a configuration with 25 FIFOs with a depth of 40 and a Mini-PIFO size S_P between 10 and 20 in the evaluation.

Figure 16 shows the *average number of extra memory accesses* per packet scheduled by Sifter with different Mini-PIFO sizes S_P under different traffic loads[27]. From the evaluation results in Figure 16, Sifter has an *average number of extra memory accesses* of less than 2 in most cases, which, in total, is less than twice the baseline (the number of memory accesses for FIFO-based schedulers). This indicates under most circumstances that the packet descriptors in Sifter would at most experience one Sifting Process which introduces one extra read and one extra write operation before the dequeue operation. As the Mini-PIFO size S_P increases, Sifter can achieve nearly zero extra memory accesses since more packets would enqueue and dequeue directly from the Mini-PIFO, bypassing the RCQ FIFOs and the Sifting process.

D Evaluation of Design Parameters

As described in *Section* 3 and *Section* 4, there are multiple design parameters in Sifter including the number of FIFOs F, the FIFO size S_F, the PIFO size S_P, as well as the granularity g. In this section, we explore the performance and FPGA resource consumption as a function of the design parameters.

[27] We applied empirical traffic based on web-search flow distributions [3]. The configured traffic loads control the arrival rate of flows.

D.1 FCT Performance

We set up a series of evaluations on the single-node micro-benchmark[28] to test the performance with different design parameters.

FIFO size As stated in *Appendix* B, the *FIFO overflow* occurrence largely depends on the FIFO size S_F. Here we evaluate the impact of the FIFO size S_F on FCT. We set up four Sifter schedulers with 32 FIFO queues with different FIFO sizes S_F ranging from 32 to 256 locations. The Mini-PIFO size S_P of all four Sifter schedulers is set to 32. In order to meet the inversion-free condition (3) and (6), we set the speed-up factor K of the four Sifter schedulers to range between 2 and 16.

Figure 17(a) shows that when the FIFO size S_F is large enough to hold the incoming packets associated with each rank range, *FIFO overflows* are essentially eliminated and Sifter delivers an identical performance of an ideal PIFO.

PIFO size To investigate how the Mini-PIFO size S_P impacts performance, we start with a design parameter configuration to induce overflows with a specific traffic profile. We set up four Sifter schedulers with 32 FIFO queues, each with a FIFO size $S_F = 64$. We configured different Mini-PIFO sizes S_P as 32, 64, 128, and 256 to investigate whether a larger PIFO would reduce overflow events.

Figure 17(b) shows the FCT performance of Sifter schedulers with increasing Mini-PIFO sizes. From the results, a larger Mini-PIFO provides better FCT performance for the smaller flows (whose packets tend to have smaller ranks). In addition, according to *section* 3.4 and *Appendix* C, a larger Mini-PIFO decreases the frequency of Sifting operation, thereby reducing the number of extra memory accesses.

Number of FIFOs According to *Appendix* B, *FIFO overflow* occurs when the packets associated with a FIFO exceed the FIFO capacity. In addition to increasing the FIFO size S_F, another solution to reduce *FIFO overflows* is to decrease the rank range associated with each FIFO. To cover the same overall rank range when reducing the rank range associated with each FIFO, we increase the total number F of the FIFO queues.

We set the FIFO size $S_F = 64$ to observe the performance improvements due only to increasing the total number of FIFOs. All the Sifter schedulers cover the same total rank range of 512. When increasing the number F of FIFOs from 8 to 64, the granularity g (the rank range that each FIFO covers) of each FIFO is decreased from 64 to 8.

Figure 17(c) shows the FCT performance of Sifter schedulers with increasing the number of FIFO. Similarly to increasing the FIFO size S_F in the previous evaluations, in-

[28] Details of the micro-benchmark topology are given in *Section* 5.1.

creasing the number of FIFOs F also decreases the occurrence of *FIFO overflows*. When the number of FIFOs F is large enough to eliminate FIFO overflows, Sifter provides performance that matches that of an ideal PIFO.

D.2 Resource Overhead

In this section, we assess the logic resource utilization of the Sifter scheduler with different design parameters.

We synthesized the Sifter VHDL implementation with different parameter setups using AMD/Xilinx's Vivado software [5] targeting an AMD/Xilinx Alveo U250 FPGA board [4], which we used to implement the Sifter hardware prototype described in *Section* 5.3. Figure 18 shows the increase in resource consumption as we increase the FIFO size S_F, Mini-PIFO size S_P as well as the total number of FIFOs F. The evaluated resources on the FPGA device are *Look Up Tables* (LUT), *Flip Flops* (FF), and *Look Up Table RAM* (LUTRAM).

FIFO size To observe how the resource utilization grows as a function of the FIFO size S_F, we configured Sifter with 32 FIFOs and a Mini-PIFO of size 32 and increased the FIFO size S_F from 32 to 256.

Figure 18(a) shows the growth of the resource utilization as we increased the FIFO size S_F. The results indicate that the extra resources due to increasing the FIFO size S_F are not significant. From Figure 18(a), the resource consumption is around 1.5% when we apply a fairly large FIFO size $S_F = 256$. Combined with the performance evaluations in *Appendix* D.1, increasing the FIFO size S_F of Sifter is the most efficient way to reduce the occurrence of *FIFO overflows* and to match the performance of an ideal PIFO.

PIFO size According to *Section* 3.5, Sifter needs to fulfill condition (3) and (6) to guarantee inversion-free operation. To meet the above conditions, Sifter needs a larger Mini-PIFO to support larger FIFO sizes.

We increased the Mini-PIFO size S_P from 32 to 256 using the same parameter setup as in the previous evaluation, where we fixed the total number of FIFOs F to 32 and set the size of FIFOs S_F as 256.

Figure 18(b) shows how the resource utilization grows as the Mini-PIFO size S_P increases. From the results, we find that increasing the Mini-PIFO size S_P consumes more resources on the FPGA when compared with increasing the FIFO size S_F.

Number of FIFOs As stated in *Appendix* B and D, another way to reduce *FIFO overflows* is to increase the number of FIFOs F.

To evaluate the effect of increasing the number of FIFOs on resource utilization, we fixed the size of FIFOs S_F and the size of the Mini-PIFO S_P to 32 locations, while increasing the number of FIFOs F from 8 to 128. Figure 18(c) shows the growth of the resource utilization as we increase the number of FIFOs F. FIFOs consume fewer resources when compared with the Mini-PIFO. According to the results in Figure 18(c), a configuration with 64 FIFOs only consumes 2% of the resources on the FPGA while one with 128 FIFOs costs around 3.3% of the available resources.

Combining the results of resource consumption in Figure 18(a) and 18(c), both solutions of increasing FIFO size S_F and increasing the number of FIFOs F are efficient approaches to reduce *FIFO overflows*.

E Sifter Hardware Prototype Details

Packet Descriptor We used a 72-bit packet descriptor for the Sifter hardware prototype as shown in Figure 19. There are 5 fields in the descriptor: (1) a 15-bit *Packet Pointer* for the address of the packet in packet buffer, (2) an 11-bit *Packet Length* for the packet length in bytes, (3) a 20-bit *Packet Rank* for the assigned scheduling order of each packet, (4) a 10-bit *Flow ID* to identify the flow and (5) a 16-bit *Packet ID* for packet identification[29].

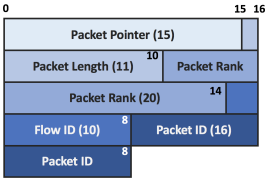
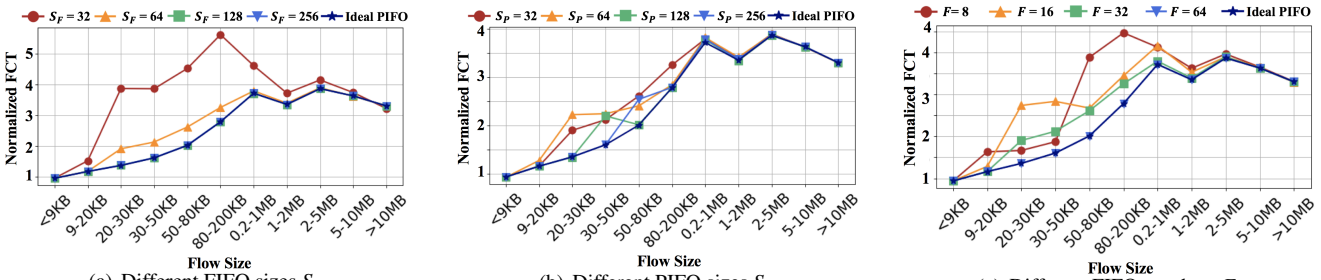


Figure 19: Packet Descriptor Structure

Packet Rank and FIFO indexing According to *Section* 4, we designed a fast RCQ FIFO indexing operation to find the index of the FIFO associated with the packet rank by bit slice selection and shifting. By configuring the number of FIFOs F and the granularity g as powers of two, the associated FIFO index is always equivalent to a grouping of bits in the packet rank.

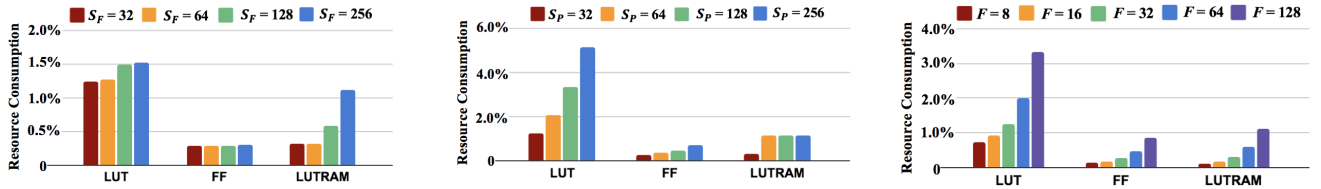
In the configuration shown in Figure 20, the scheduler has 32 FIFOs in the RCQ, and the granularity g of each FIFO is 32 (each FIFO covers a rank range of 32). We subdivide the 20-bit packet rank into three sections: 10 *Rotation bits*, 5 *Index bits* and 5 *Lower bits*. Since the granularity $g = 32$, the RCQ does not discriminate between descriptors that have equal (15) upper bits but different (5) *Lower bits*. Since the RCQ has 32 FIFOs in total, the FIFO index contains 5 bits. Sifter uses the *Index bits* (9th down to 5th bits, inclusive) in the packet rank to find the FIFO associated with the rank value. The *Rotation bits* in the packet rank only indicates

[29] The Packed ID may not be necessary in many applications but we added it to detect and identify missing packets



(a) Different FIFO sizes S_F (b) Different PIFO sizes S_P (c) Different FIFO numbers F

Figure 17: Normalized FCT with Different Design Parameters



(a) Different FIFO sizes S_F (b) Different PIFO sizes S_P (c) Different FIFO numbers F

Figure 18: Resource Overhead with Different Design Parameters

the number of times the RCQ has rotated and does not affect FIFO indexing. In the following example in Figure 20, Sifter needs to find the FIFO index associated with packet rank 996. After we convert 996 into binary and select the *Index bits*, the value is 11111 and matches index 31 of the associated FIFO. This approach was used to implement fast RCQ FIFO indexing to avoid using time-consuming math operations.

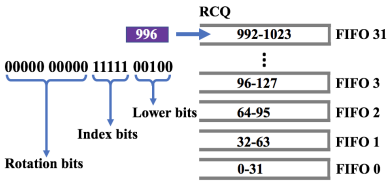


Figure 20: Fast RCQ FIFO Indexing

Speed-up Factor in Sifter Prototype In *Section* 3.5, we provided the conditions that Sifter must meet to guarantee inversion-free packet scheduling. Specifically, for a given FIFO size S_F and Sifting threshold Th_R, Sifter must provide a speed-up factor K that satisfies condition (3). In our Sifter prototype implementation, this speed-up factor K needs to be greater than 2.

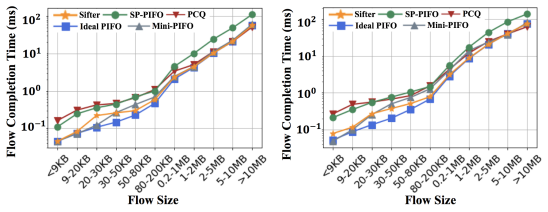
Our Sifter FPGA prototype implementation requires a minimum of 5 clock cycles for each Enqueue and Dequeue operation, as well as the operation to simultaneously "sift" a packet descriptor between the RCQ and the Mini-PIFO. However, to ensure inversion-free packet scheduling, we need to provide a speed-up factor $K = 2$, which means that the operation to sift a packet descriptor needs to be twice as fast as the dequeue operation. Therefore, our hardware prototype can sustain a dequeue interval of 10 clock cycles to achieve the necessary speed-up factor.

F Micro-benchmark Evaluation: FCT and end-to-end delay

We analyzed multiple flows using the micro-benchmark to evaluate the FCT and end-to-end delay performance of different packet schedulers. We extended the single-node star topology with 129 end-hosts to create a 128-to-1 in-cast pattern. We generated empirical traffic according to web-search flow size distributions in data centers [3]. The arrival pattern of the TCP flows follows a Poisson distribution, where the average arrival rate is determined by the configured traffic load. Upon the arrival of each TCP flow, the simulator randomly selects a sender host and starts a TCP connection[30] to the receiver host.

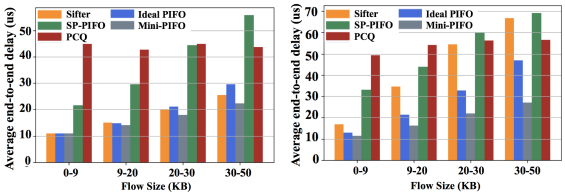
Sifter has the closest FCT to the ideal PIFO Figure 21 shows the FCT of Sifter, PCQ, SP-PIFO, the ideal PIFO, and the stand-alone Mini-PIFO in this single-node star topology. Overall, Sifter has the FCT performance that is closest to the ideal PIFO. For the small-sized flows, Sifter outperforms PCQ and SP-PIFO by eliminating packet inversions. When compared with the stand-alone Mini-PIFO, Sifter has a much

[30]For simplicity, all packets in our packet-based simulation have the same size of 1500 bytes and each TCP flow shares the same weight in SFQ.



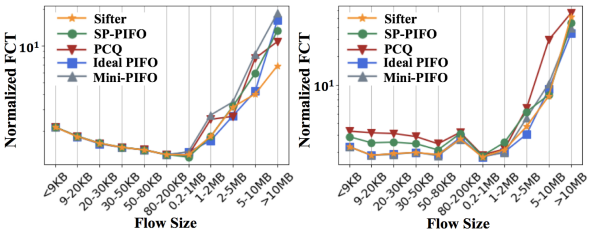
(a) Single-node FCT, 70% load (b) Single-node FCT, 90% load

Figure 21: FCT with single-node topology



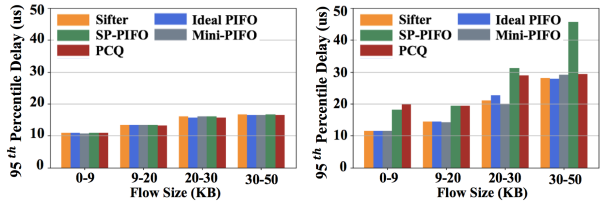
(a) Average delay, 70% load (b) Average delay, 90% load

Figure 22: Average end-to-end delay with the single-node topology



(a) Normalized FCT, Poisson arrival, 70% load (b) Normalized FCT, brust arrival

Figure 23: Normalized FCT with Modified SRPT



(a) End-to-end Delay, Poisson arrival, 70% load (b) End-to-end Delay, brust arrival

Figure 24: End-to-End Delay with Modified SRPT

better capacity to reduce packet losses, which leads to fewer re-transmissions and lower FCT for the mid-sized flows.

Sifter provides lower delay for short flows We examine the effect of scheduling inversions on short-flow delays because they are more susceptible to those inversions.

Figure 22 shows the end-to-end delay for packets of short flows. When compared with PCQ and SP-PIFO, Sifter has a lower end-to-end average delay in most cases. As discussed in *Section* 2.2.1, packet scheduling inversions have a larger impact on the delays of short-flow packets. When packets with larger timestamps depart before packets with smaller timestamps from the short flows, packets from the short flows experience longer delays.

G Use Case: Modified Shortest Remaining Processing Time to minimize FCT

Shortest Remaining Processing Time (SRPT) [43] is an effective scheduling algorithm for minimizing average FCT. However, SRPT may lead to starvation and packet misordering within each flow. There are two mainstream solutions for these issues: per-flow queues and pFabric's starvation prevention [3]. However, these solutions have costly implementations. The per-flow queue solution requires the scheduler to support a number of queues equal to the number of active flows ($> 10K$ flows [49]), which is impractical for current switches. The second solution, starvation prevention by bit-wise comparison among packets from the same flow, requires complex hardware implementation and is power-hungry.

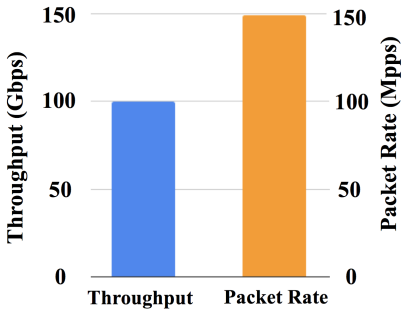
pFabric also proposed a modification to the SRPT scheduling algorithm to make it more practical with negligible impact on FCT performance. Instead of setting the rank of each packet using the remaining flow size, this modified SRPT algorithm simply sets the rank of all the packets from the same flow as the total flow size. Although the rank does not reflect the remaining size of each flow, the packet out-of-order and starvation issues are eliminated.

In this part of the simulation, we applied the modified SRPT algorithm for Sifter, PCQ, and SP-PIFO. Each of the schedulers has 16 FIFOs with a depth of 64 and Sifter has a Mini-PIFO with a capacity of 32. We also set up an ideal PIFO with a capacity of 1024 and a standalone Mini-PIFO with a capacity of 32, the same as Sifter.

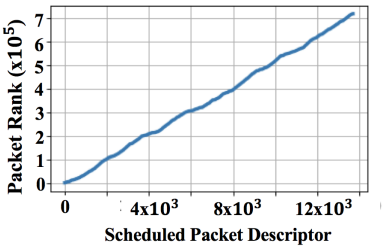
We applied two types of traffic in the evaluation: Poisson arrival traffic and bursty arrival traffic. We generated the Poisson arrival traffic following the procedure in *Appendix* F. The flow size follows empirical distributions of Web-search traffic [3] and the arrival pattern of the TCP flows follows a Poisson distribution. In addition, we added short bursts of TCP flows with Poisson arrivals to form bursty arrival traffic, aiming to create congestion on the bottleneck link.

Sifter achieves FCT and delay performance matching an ideal PIFO In the use case of the modified SRPT algorithm, each packet's rank is equivalent to the flow size it belongs to. As a result, the range of packet ranks is not larger than that of active flow sizes, which makes packet scheduling much easier. Therefore, all packet schedulers can achieve near-optimal FCT and delay performance, as shown in Figure 23 (a) and Figure 24 (a).

To increase the range of packet ranks, we introduced short bursts in flow arrivals. Figure 23 (b) and Figure 24 (b) show that with a wider range of packet ranks and heavier packet



(a) Throughput and Packet Rate



(b) Scheduling Order

Figure 25: ASIC Simulation

congestion, the performance of PCQ and SP-PIFO degrades while Sifter still closely approximates the benchmark of an ideal PIFO.

H Simulated ASIC Performance

As discussed in *Section* 5.3, the FPGA prototype sustains a 100 Gbps line rate for packets larger than 370 bytes, due to the limitation of the clock frequency of 322 MHz achievable in the target FPGA. The 370-byte value is derived as follows: The 322 MHz clock has a period of 3.1 ns. As explained in *Appendix* E it takes 10 clocks to achieve a K factor of 2. Sifter's FPGA prototype can dequeue a packet every 10 x 3.1 = 31 ns, which is equivalent to 3100 bits (387 bytes) at 100 Gbps. The 387-byte length includes 20 bytes of Ethernet overhead (Preamble, SFD, Interframe Gap), which makes the actual packet 367 bytes. We rounded up this value to 370 bytes so that we can easily control the dequeue rate in hardware by counting down 37 bytes per clock (10 x 37 = 370). To demonstrate that Sifter can achieve 100 Gbps with a faster clock rate (e.g., in an ASIC), we ran a simulation using our Python-based Cocotb test environment [21], where we set the clock frequency to 1.7 GHz. As shown in Figure 25 (a), Sifter can sustain a 100 Gbps line rate with the minimum size packets of 64 bytes. Figure 25 (b) shows that Sifter's VHDL implementation performs inversion-free scheduling for 64-byte packets when running at 1.7 GHz.

I Pseudo-code of Sifter

We summarize the *Enqueue*, *Dequeue*, and *Sifting* processes with pseudo-code in **Algorithm** 1.

Algorithm 1 Enqueue, Dequeue and Sifting process of Sifter

```
1:  function INITIALIZATION
2:      Initialize Sentinel s = ∞
1:  function ENQUEUE PACKET(P_(i,k))
2:      if R_(i,k) ≤ s then
3:          Mini-PIFO enqueue P_(i,k)
4:          if P_(i',k') evicted from Mini-PIFO then
5:              RCQ enqueue P_(i',k')
6:      else
7:          RCQ enqueue P_(i,k)
1:  function DEQUEUE PACKET( )
2:      Dequeue packet P_(i,k) from Mini-PIFO
3:      if O_P ≤ Th_S then
4:          SIFTING( )
1:  function SIFTING( )
2:      Find earliest non-empty FIFO f in RCQ
3:      Update s as the largest rank covered by FIFO f
4:      for Packet P_(i,k) in f do
5:          if R_(i,k) ≤ s then
6:              Mini-PIFO enqueue P_(i,k)
7:              if P_(i',k') evicted from Mini-PIFO then
8:                  RCQ enqueue P_(i',k')
9:                  s = MAX{R_(i',k'), s}
10:         else
11:             RCQ enqueue P_(i,k)
12:     if RCQ is empty then
13:         Reset Sentinel s = ∞
14:     else if O_P ≤ Th_S then
15:         SIFTING( )
```

For variable names and definitions, please refer to Table 1.

Acknowledgements

We acknowledge AMD/Xilinx for providing the Alveo U250 board and Exegy for providing the nxFramework, both of which enabled the FPGA prototyping of Sifter. We extend our sincere gratitude to our shepherd, Brent Stephens, and the anonymous NSDI reviewers for their valuable feedback.

Flow Scheduling with Imprecise Knowledge

Wenxin Li[1] Xin He[1] Yuan Liu[1] Keqiu Li[1] Kai Chen[2,3] Zhao Ge[1] Zewei Guan[1]
Heng Qi[4] Song Zhang[1] Guyue Liu[5]
[1]*Tianjin Key Laboratory of Advanced Networking, Tianjin University*
[2]*Hong Kong University of Science of Technology* [3]*University of Science and Technology of China*
[4]*Dalian University of Technology* [5]*New York University Shanghai*

Abstract

Most existing data center network (DCN) flow scheduling solutions aim to minimize flow completion times (FCT). However, these solutions either require precise flow information (e.g., per-flow size), which is challenging to implement on commodity switches (e.g., pFabric [7]), or no prior flow information at all, which is at the cost of performance (e.g., PIAS [10]). In this work, we present QCLIMB, a new flow scheduling solution designed to minimize FCT by utilizing *imprecise flow information*. Our key observation is that although obtaining precise flow information can be challenging, it is possible to accurately estimate each flow's lower and upper bounds with machine learning techniques.

QCLIMB has two key parts: i) a novel scheduling algorithm that leverages the lower bounds of different flows to prioritize small flow over large flows from the beginning of transmission, rather than at later stages; and ii) an efficient out-of-order handling mechanism that addresses practical reordering issues resulting from the algorithm. We show that QCLIMB significantly outperforms PIAS (88% lower average FCT of small flows) and is surprisingly close to pFabric (around 9% gap) while not requiring any switch modifications.

1 Introduction

Flow scheduling is an effective scheme for low latency data center network (DCN) transport design [7, 10, 24, 41, 27, 37, 33]. One of the most important goals of flow scheduling is to minimize flow completion times (FCT), which is essential for many critical DCN applications, such as web search [5], key value store [2, 40, 3], and machine learning training [23]. These applications are dominated by small messages and have stringent latency requirements, as a result, even a very small delay can significantly degrade application performance [37].

There are two lines of prior work aimed at minimizing FCT, known as clairvoyant and non-clairvoyant scheduling. The clairvoyant scheduling [7, 41, 37, 24] assumes prior knowledge of precise flow size information and uses it to approximate the Shortest Remaining Processing Time (SRPT). This approach can theoretically achieve optimal performance, but is very challenging to be deployed in current DCNs, e.g., requiring too many priority queues [7] or re-factor the entire TCP/IP stack [41, 37, 24]. The non-clairvoyant scheduling [10, 11, 21, 48] requires no prior flow information and dynamically estimates flow size (e.g., based on the bytes the flow has sent [10]). While this approach is easy to implement in practice, it cannot precisely distinguish between large and small flows at the beginning, thus failing to minimize FCTs for latency sensitive short flows.

To minimize FCT and be practical, we explore a new design space that lies between existing clairvoyant and non-clairvoyant scheduling solutions. Rather than relying on precise flow information or no prior flow information at all, we ask: *Is it possible to use imprecise flow information to minimize FCT with commodity switches*?

Answering this question requires identifying useful, yet imprecise, flow information and incorporating it into flow scheduling. Some existing work [23, 42] employ machine learning (ML) techniques to estimate per-flow size but fail to get high accuracy (§2.2). Utilizing imprecise flow information is also very challenging. Simply feeding imprecise information to existing clairvoyant schedulers significantly degrades their performance [23].

We address these challenges with QCLIMB, a practical flow scheduling solution that uses imprecise flow information to minimize FCT in DCNs. QCLIMB is based on a key observation that although determining per-flow size of DCN applications can be difficult, it is possible to accurately estimate each flow's lower and upper bounds. Our experiments (§2.3) on realistic DCN workloads have shown that the actual sizes of a majority of flows ($> 99.9\%$) fall within their lower and upper bounds estimated by random forest (RF) model. Moreover, we found the actual sizes of small flows are generally close to their lower bounds, while medium and large flows may

have a larger gaps between their actual sizes and lower bounds. These findings provide an opportunity to precisely differentiate small flows from large flows based on their *lower bounds*, making it possible to prioritize small flows over large ones from the start, rather than in later stages of transmission.

Based on these findings, we develop a novel scheduling algorithm consisting of two main phases: *queue-climbing-up* and *queue-climbing-down*. Each flow is initially mapped to a priority queue based on its lower bound. During the *queue-climbing-up* phase, the flow is gradually promoted to higher-priority queues based on its remaining data size relative to the lower bound. If the flow is not yet completed after the first phase, it enters the *queue-climbing-down* phase. During this phase, QCLIMB gradually demotes the flow to lower-priority queues based on its bytes sent, and once its upper bound finishes, it is pulled directly to the lowest priority queue.

QCLIMB's algorithm is effective at prioritizing small flows over large ones for two reasons: i) small flows are close to their lower bounds and thus can finish in the first few higher priority queues during the queue-climbing-up phase; and ii) medium and large flows will be transmitted behind small flows during the queue-climbing-up phase because their lower bounds are relatively larger. They will also be penalized to the last few lower priority queues during the queue-climbing-down phase.

In addition to designing the scheduling algorithm, we must also tackle practical out-of-order (OOO) issues. This is because during the queue-climbing-up phase, the later packets of a flow can enter higher-priority queues than the earlier ones. The default TCP OOO handling mechanism considers this event as a packet loss and triggers unnecessary retransmissions. This results in serious performance degradation, especially for small flows (details in §5.2).

Addressing the OOO issue needs to overcome two practical challenges: First, how to differentiate reorderings caused by QCLIMB's scheduling algorithm from actual packet loss? Second, for packets reordered by QCLIMB, how to efficiently reorder them at the receiver side? Simply relying on the default TCP reordering mechanism will cause redundant retransmissions.

To tackle the first challenge, we present a priority-based loss detection mechanism at the receiver (§3.2.1). The idea is to leverage the fact that the packets carrying the same priority belonging to the same flow should be in order, and a gap within the same priority is identified as a loss. For the second challenge, we take a non-intrusive approach by customizing ACKs for OOO packets at the receiver to bypass the default TCP ACKing mechanism. Through this way, normal TCP ACKs will not be sent for OOO packets, and the senders have no chance to trigger redundant retransmissions (§3.2.2). Moreover, with the customized ACK, the sender can quickly retransmit the lost packet without waiting for timeout (§3.2.3).

We have implemented a QCLIMB prototype (§4), which only requires end host implementation and the built-in function (e.g., strict priority queuing) in existing commodity switches. We implement the scheduling logic and RF model inferencing as Linux kernel modules, which reside between the NIC driver and the TCP/IP stack as a shim layer. Further, OOO handling is implemented within the TCP and IP layers and requires minimal modifications to the kernel source code but does not touch the core TCP congestion control code.

We build a small-scale 25G testbed with eight servers and a Barefoot Tofino switch[1], together with large-scale simulations at 40/100G network, to evaluate the performance of QCLIMB (§5). We find that:

- Compared to PIAS [10] that requires no flow size, QCLIMB reduces the overall average FCT and the average/tail FCT of small flows by up to 49.5% and 88%/97%, respectively. It also improves the query performance by 70%∼97% in a Memcached application.
- Compared to pFabric [7] that assumes precise flow information, QCLIMB can deliver an average gap of 9% for overall performance and even show a 6.8% lower tail FCT of small flows in a PageRank workload.
- QCLIMB's design components are effective for the performance. QCLIMB is resilient to extreme cases with model-application mismatching, coexisting applications, and tiny workload. Yet, it outperforms PIAS even with two queues and low model accuracy.

2 Background and Motivation

2.1 Limitations of Existing Approaches

As shown in Table 1, existing DCN flow scheduling solutions can be classified into two categories:

Clairvoyant solutions: Clairvoyant solutions attempt to approximate the Shortest Remaining Processing Time (SRPT) scheduling based on prior knowledge of precise flow size information. In general, they can provide good performance but have significant limitations in deployability. First, pFabric [7] uses millions of priorities to implement SRPT, whereas such fine-grained priority queues require clean-slate switches and are not supported by existing commodity switches. Second, pHost [24], Homa [37] and EPN [34] use limited number of priority queues, but need to re-factor the network stack or rely on programmable switches. Finally, PDQ [28] and FastPass [41] do not use priorities but are impractical as well. For instance, PDQ [28] requires non-trivial switch modification to adjust the flow rate to implement flow preemption. FastPass [41], as a centralized scheduling design, is challenging to be deployed in a large cluster.

[1] Though the switch is P4-programmable, we use it as a commodity hardware in our evaluation.

Schemes		Requiring no switch changes or advanced hardware	Using limited number of priority queues	Retaining existing TCP/IP network stacks	Using lower bounds for flow scheduling
Clairvoyant	pFabric [7]	No	No	No	No
	PDQ [28]	No	× (priority queues are not required)	No	No
	EPN [34]	No	Yes	Yes	No
	FastPass [41]	Yes (but centralized arbiter is required)	× (priority queues are not required)	No	No
	pHost [24]	Yes	Yes	No	No
	Homa [37]	Yes	Yes	No	No
Non-Clairvoyant	PIAS [10]	Yes	Yes	Yes	No
QCLIMB		Yes	Yes	Yes	Yes

Table 1: Summary of main DCN flow scheduling schemes in literature and comparison to QCLIMB.

Non-clairvoyant solutions: Non-clairvoyant schedulers are agnostic about flow size information and generally use the idea of flow aging to estimate the pending data with the bytes a flow has already sent. For example, PIAS [10] gives the highest priority to new flows and then gradually demotes their priorities as they send more data. It is a readily-deployable solution that works with multiple priority queues available in commodity switches and is compatible with legacy TCP/IP stacks. However, PIAS has limited ability in minimizing FCT (§5).

In short, this short analysis inspires us to think about if there is a middle-point design between clairvoyant and non-clairvoyant schedulers, i.e., scheduling flows with imprecise knowledge and with commodity switches.

2.2 Imprecise Flow Information

Indeed, researchers have shown the possibility to learn flow size information from past traces using prevailing ML techniques [23, 42, 35, 30]. Nevertheless, the estimated flow sizes of ML models are often imprecise. To validate this point, we use a widespread ML technique—RF, over three workloads[2]: K-Means, PageRank, and SGD, provided by [23]. Fig. 1 plots the gap between actual and estimated flow sizes for different-scale RF models we trained (i.e., different *maximum tree depth d*). In general, a larger d leads to higher prediction accuracy. However, the gap between actual and estimated flow sizes always exists. Under $d = 10$, the mean/maximum gap can reach 856KB/296811KB, 820KB/333271KB, and 55KB/54446KB, for the K-Means, PageRank and SGD workloads, respectively. In particular, for PageRank under $d = 10$ RF model, 34% of flows have a gap of over 100KB to their estimated sizes.

Little work can utilize imprecise knowledge well. The only work, FLUX [23], directly takes imprecisely estimated flow sizes as input for clairvoyant schedulers (e.g., pFabric [7]), which, however, suffers degraded FCT performance. Indeed, we did a simulation and the results show that the average FCT of small flows achieved by pFabric with imprecise knowledge can be slowed down by up to 22.8× (appendix A). Even though the estimated flow sizes are imprecise, in what follows we demonstrate that the lower bound part is highly accurate.

[2]We did not use traditional DCN workloads like web search [5] and data mining [25] because they only provide flow size distribution. The workloads of [23] are collected from university clusters running real ML applications and contain enough information for learning.

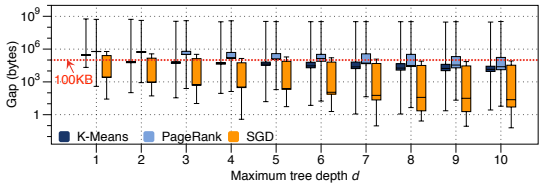


Figure 1: The gap between predicted and actual flow sizes.

2.3 Motivation: Lower Bound on Flow Size

2.3.1 Application Examples

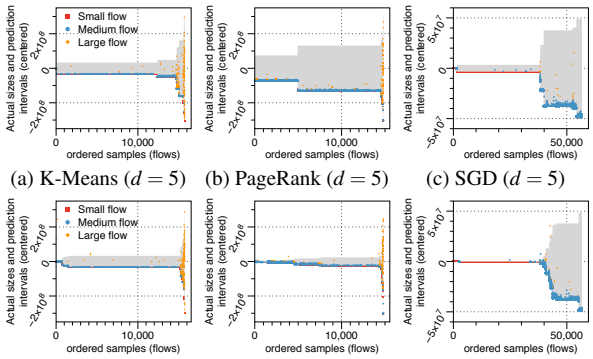
We begin by introducing a few application examples in which lower bounds on flow sizes indeed exist:

Distributed ML: Training ML models in parallel is an increasingly important workload in datacenters. During the model training, each generated flow needs to transfer at least one parameter update (e.g., a 32-bit integer). This means that the flow is at least ∼40 bytes, considering the length of various headers including TCP and IP. As a concrete example, we inspect the flow size distribution of a SGD workload [23] and find that all flows are more than 44 bytes (see §5.1).

Web Search: Large-scale web search application is another example, where a query might be sent to many aggregators and workers, each responsible for a different part of the index. From a networking perspective, it contains query traffic, delay-sensitive short messages for cluster coordination, and background traffic for response-quality-oriented massive data transfer. The smallest flow comes from the query traffic, e.g., transferring the index of at least one page between workers and aggregators, which is typically lager than 1.6KB [5].

2.3.2 Experimental Observations

The above examples only provide application-level identical lower bounds for all flows. These bounds are loose for individual flows. That said, they may be far away from the actual flow sizes, thus limiting the effect of flow scheduling. To provide a tight lower bound for each individual flow at its start, we use the RF technique [15]. Specifically, we keep the full conditional distribution of each RF tree's decisions and use a quantile regression forest (QRF) method [36] to build a prediction interval for each flow's size (see appendix B). Fig. 2 gives an in-



(a) K-Means ($d = 5$) (b) PageRank ($d = 5$) (c) SGD ($d = 5$)

(d) K-Means ($d = 10$) (e) PageRank ($d = 10$) (f) SGD ($d = 10$)

Figure 2: Relationship between actual flow sizes and prediction intervals. For better visualization, flows are ordered according to their prediction intervals.

tuitive visual feeling[3] for the prediction intervals. We have the following observations that inform our design.

Observation 1 (O1): *With RF model, each flow can have a bounded interval on its data size, and the vast majority of flows can be accurately bounded.*

A flow is bounded if its actual size falls exactly within its lower and upper bounds; otherwise, it is an out-of-bound flow. As shown in Table 2, the ratio of bounded flows accounts for 99.9055%∼100%. This result stems from the fact that QRF keeps the full conditional distribution of the estimated size; thus, the obtained prediction intervals can cover each flow with high probability.

Observation 2 (O2): *For small flows, their actual sizes are mostly close to their lower bounds.*

From Fig. 2, we see that small flows are mostly scattered around their lower bounds. The average gap of small flows' actual sizes to their lower bounds can be only 12983B, 12630B, and 213B, for K-Means, PageRank, and SGD workloads, respectively. For the root cause of this phenomenon, an intuitive conjecture is that these workloads exhibit a long-tail distribution where most flows are short. This makes an RF model have many small values in its leaf nodes, thus predicting a lower bound close to the small flow's size.

Observation 3 (O3): *Medium and large flows' lower bounds are relatively larger than those of small ones.*

Across all the cases we examined, the average lower bound of small flows is 52B∼12KB, 1002B∼1198B, and 80B∼664B, for K-Means, PageRank, and SGD workloads, respectively, For medium (large) flows, the average lower bound is 52B∼124KB (52B∼26899KB), 1002B∼2090B (1002B∼19060KB), and 255KB∼ 260KB (260KB∼1109KB) for the three workloads, respectively.

Observation 4 (O4): *For out-of-lower-bound flows, the small flows have a smaller gap to relevant lower bounds than medium and large ones.*

[3]Fig. 2 only shows two settings (i.e., the maximum tree depth $d = 5$ and 10), but our observations are condensed from all settings we tested.

Workloads		The ratio of bounded flows	The ratio of out-of-bound flows
K-Means	$d = 5$	99.9936%	0.0064%
	$d = 10$	99.9301%	0.0699%
PageRank	$d = 5$	100%	0
	$d = 10$	99.9055%	0.0945%
SGD	$d = 5$	100%	0
	$d = 10$	99.9877%	0.0123%

Table 2: The ratio of bounded and out-of-bound flows.

We gather the out-of-lower-bound flows and explore how far their actual flow sizes are from their lower bounds. We observe that this differs in different types of flows. For small flows, the average gap between their actual sizes and lower bounds is only 2.8B∼25KB, across the cases we tested. For medium and large flows, this gap can reach 68KB∼743KB and 6863KB∼77072KB, respectively.

Observation 5 (O5): *For out-of-upper-bound flows, they are primarily medium and large flows and may go beyond their relevant upper bounds a lot.*

We barely see any small flows going beyond their upper bounds in our experiments. The out-of-upper-bound flows are dominated by medium and large ones; yet, their actual flow sizes have a substantial gap to relevant upper bounds, i.e., 54KB∼2322KB and 262KB∼43252KB for medium and large flows, respectively.

Observations **O4** and **O5** can be intuitively caused by different densities of flow size between small and medium/large flows. Compared to small flows, medium/large flows have a more sparse distribution of flow size value; thus, if being out-of-bound, they are more likely to have a larger gap to their bounds.

Discussion: The observations above might not manifest for applications without long-tail flow size distributions, e.g., local file system [22] and archival data [43]. This is expected because regular RF shows high predictability for majority class samples, while small flows in these applications are minority classes. A possible way would be to modify the weighting strategy in regular RF [45] to bias small-value samples and consequently make better predictions for small flows than medium/large ones.

3 QCLIMB Design

3.1 Lower-bound-based scheduling

The above findings offer an opportunity to differentiate small flows from large flows precisely, based on their *lower bounds*. This makes it possible to prioritize small flows over large ones from the start, rather than in later stages of transmission. Based on this insight, we develop a novel flow scheduling algorithm consisting of two main phases: *queue-climbing-up* and *queue-climbing-down*.

Phase 1: Queue-Climbing-Up. The observations **O1** in §2.3 indicate that the lower bound of each flow is highly

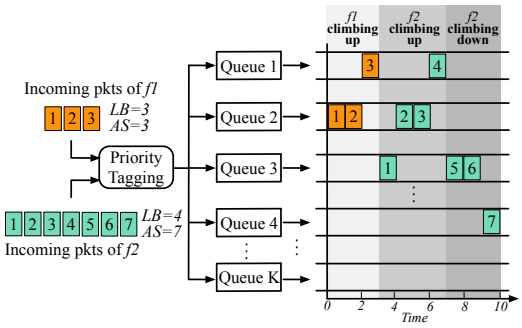


Figure 3: A queue-climbing-up&-down example with two flows: *f1* and *f2*. AS: actual size; LB: lower bound.

credible. That said, each flow's actual size is, with a high probability, larger than its lower bound. So, QCLIMB hypothetically believes that each flow's size equals its lower bound. In particular, QCLIMB lets each flow enter a lower-bound-matched initial priority queue first and then gradually promotes it from this initial queue to the higher-priority ones based on the remaining bytes to its lower bound (see Fig. 3). In this phase, flows with smaller lower bounds but the same bytes sent or larger bytes sent but the same lower bounds are likely to have higher priority.

Phase 2: Queue-Climbing-Down. The hypothetical judgment made above can be wrong for the flow not yet completed after its lower bound part finishes. QCLIMB thus takes an additional queue-climbing-down phase as a remedy. Specifically, QCLIMB pulls back the flow to its initially entered priority queue and punishes it by demoting it to lower-priority queues gradually based on its bytes sent. Moreover, if the bytes sent exceed the upper bound, QCLIMB takes a greater punishment by dragging the flow down to the lowest priority queue.

Why this works. Our scheduling algorithm is able to effectively prioritize small flows over large ones for the following reasons. *First*, if a flow is a small one, its size is near the lower bound (**O2**); thus, it can finish with the first few higher priorities. *Second*, small flows can keep relatively higher priorities than medium and large flows during the queue-climbing-up phase since their lower bounds are somewhat smaller (**O3**). *Third*, despite very few small out-of-lower-bound flows, they will not enter a priority queue lower than the one mapping their actual sizes because of their small gap to lower bounds (**O4**) and the relatively large flow size range of each priority queue. *Fourth*, a medium or large flow will eventually be penalized to lower-priority queues or even the lowest priority queue (**O5**).

In comparison to the non-clairvoyant solutions such as PIAS [10], which demotes flows as they transmit more data, medium/large flows can only be detected and separated into low-priority queues after they coexist with short flows in higher-priority queues for a short period. In contrast, QCLIMB separates flows into different priority queues once they start, using their lower bounds.

Putting it all together: QCLIMB adopts multiple priority queues available in commodity switches. Packets carrying different priorities will enter into different priority queues. Packets in different queues are scheduled with strict priority, while packets in the same queue follow FIFO scheduling. Packet priority tagging (appendix C) is distributed at each end-host, which is triggered whenever a new packet p arrives. It first gets the bytes sent B, lower bound L, and upper bound U for the parent flow of the arriving packet p. It then relies on the following three steps working cooperatively.

1. **Priority Promoting:** If $B < L$, we invoke *get_priority* for determining packet p's priority, which scans queue thresholds bottom-up and returns the first priority having a threshold smaller or equal to $L - B$. Meaning, a flow is initially mapped to a priority queue according to its lower bound. Then, it climbs up from this initial queue to higher-priority queues gradually based on the remaining bytes to its lower bound.
2. **Delay Demoting:** A flow may only contain minor errors: its lower/upper bounds map to the same priority. Such flow can only remain a small amount of data after the first phase. So, we take a delay demoting approach here, i.e., we keep each flow's priority (mostly the highest one) for a short while to transmit a slack size S more data to hope it can finish.
3. **Priority Demoting:** At this point, if a flow is not yet completed, we will change the strategy and gradually demote its packets to lower-priority queues according to its bytes sent. If the flow is not end even after the upper bound portion finishes, we move it to the lowest priority queue directly.

Choice of Slack Size S: A larger slack size S may allow more flows to complete before switching to lower-priority queues while leading more large flows to coexist with small flows. The optimal value of S in terms of average FCT is closely related to the gap between the actual sizes and lower bounds of flows, and varies under different traffic patterns and different RF models. However, as mentioned earlier, almost all small flows are close to their lower bounds and can finish within the queue-climbing-up phase, making QCLIMB beneficial under a wide range of S (see §5.2).

3.2 Out-of-Order Handling

In addition to designing the scheduling algorithm, we must also tackle practical out-of-order (OOO) issues. This is because during the queue-climbing-up phase, the later packets of a flow can enter higher-priority queues than the earlier ones. As a result, the subsequent packets of a flow may arrive at the receiver before previous ones, i.e., out-of-order (OOO). The default TCP OOO

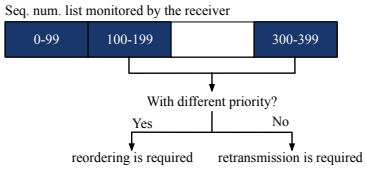


Figure 4: Priority-based loss detection.

handling may treat this event as loss, thus triggering unnecessary retransmissions and significantly hurting the performance, especially for the tail FCT of small flows (see Fig. 15 in §5.2). Addressing the OOO issue needs to overcome two practical challenges: (i) how to differentiate reorderings caused by QCLIMB's scheduling algorithm from actual packet loss? and (ii) for packets reordered by QCLIMB, how to efficiently reorder them at the receiver side?

3.2.1 Loss detection

To detect if an OOO event is due to loss, QCLIMB adopts a priority-based loss detection mechanism at the receiver. This mechanism leverages the fact that for each flow, the packets carrying the same priority traverse the same priority queue in each switch along the transmission path[4] and should be in-order. A gap within the same priority is assumed to be a loss. For a better intuition of this point, we consider a concrete example in Fig. 4. As we can see, there appears a sequence number gap. We check if the left-hand packet of this gap has different priority as the right-hand packet. If yes, the packet may be in-flight, and reordering is required (§3.2.2); Otherwise, the relevant packet must be lost, and the retransmission logic will be invoked (§3.2.3). Note that because a flow may transmit multiple packets using the same priority, there could be a corner case. More specifically, when the gap occurs at the boundary of a priority, it will not be identified as a loss with the above mechanism, but it is still likely to be a loss. In this paper, we leave such loss detection to timeout.

3.2.2 Packet reordering

Packet reordering is conducted by separating OOO packets into slow path, as detailed below. Starting from the bottom to up, packets arrive at the NIC and are read into the kernel as `skbs`. Once a packet is copied to `skb`, it will be pushed up to the TCP layer, where QCLIMB checks if it is in-ordered. If yes, this packet will be sent along the fast path that directly connects to the application layer receiver buffer. Otherwise, this packet will be identified as an OOO packet. As such, QCLIMB will send this packet along the slow path and store it in an OOO receive queue. After the missing packets arrive, QCLIMB will forward it up to the application layer receive buffer.

While the reordering is simple, the OOO event caused by QCLIMB's scheduling may mislead the default TCP stack to trigger duplicate ACKs from receiver to sender, thus leading to redundant retransmissions. To avoid this phenomenon, we let the receiver bypass the default TCP ACKing mechanism by directly sending a customized ACK to the sender on receiving any OOO packet. Meanwhile, we let the sender maintain a scoreboard to track which packets have been cumulatively and selectively acknowledged. Here, each customized ACK carries the cumulative acknowledgment (indicating its expected sequence number), a scoreboard update flag, and a SACK tag (indicating the OOO packet received). Upon receiving such ACK, the QCLIMB sender checks if this ACK contains an update flag. If yes, the sender will update its scoreboard to allow TCP to continue as usual. As such, the sender will not have a chance to trigger redundant retransmissions because the TCP's normal ACKs for OOO packets will not be sent.

3.2.3 Packet retransmission

When a loss is detected, the receiver will not discard the OOO packet, and the sender will retransmit the lost packet. More precisely, the receiver also sends a customized ACK, which has the same format as that in §3.2.2 but has a different (retransmission) flag. Upon receiving such an ACK, the QCLIMB sender checks if this ACK contains a retransmission flag. If yes, it enters loss recovery mode, where the sender retransmits the first packet that corresponds to the cumulative acknowledgment value. Any subsequent packet will be retransmitted if the sender receives another ACK carrying a higher cumulative acknowledgment value and a retransmission signal. Through this way, we can quickly retransmit the lost packets without waiting for the timeout.

Note that to guarantee the delivery of all ACKs, we transmit them at the highest priority. Moreover, we also give the highest priority to retransmitted packets because we want to fill the gap as soon as possible to minimize the resequencing buffer's memory footprint.

4 Implementation

We have implemented a QCLIMB prototype under the Linux 4.15 kernel, as detailed below.

4.1 Sender

Flow monitoring: We implement this module in Linux kernel space and integrate it with the RF inferencing and packet tagging modules. It collects flow information using the Netfilter hook [1]. For each new flow, we collect its start time, flow gap (time since the end of the previous flow), flow sizes for the last 5 flows, and TCP 5-tuple (i.e., source/destination IPs and ports, protocol ID) and feed them into the RF inference module. During the flow lifetime, we also record its bytes sent to guide the packet tagging. For each finished flow, we append its flow size to the features and transfer the collected information to

[4] We consider single-path routing for each flow, i.e., ECMP [29].

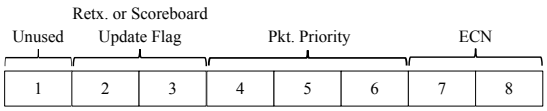


Figure 5: The TOS field format used in QCLIMB.

RF training using the `netlink` channel.

RF training: This module is in user-space. It receives flow information and performs RF training. We transfer the trained RF model to the RF inferencing module, again using the `netlink` channel.

RF inferencing: We implement this module in the kernel space to avoid frequent cross-space communication between packet tagging and RF inferencing. Specifically, we use C to implement the RF model in the kernel. This module performs online inference to obtain the lower/upper bounds for each flow and passes the inferred information to the packet tagging module for flow scheduling.

Packet tagging: This module enforces the scheduling policies of QCLIMB algorithms by marking packets with priorities at end hosts. We implement it as a Linux kernel module, locating between TCP/IP stack and Linux TC. For each outgoing packet, this module has three key operations. *First*, it maintains a hash-based flow table that stores the 5-tuple and bytes sent. *Second*, it identifies the flow the packet belonging to and updates the bytes sent of the relevant entry in the flow table. *Third*, based on the lower/upper bounds and bytes sent of the flow, it calculates a priority for this packet and invokes the `ipv4_change_dsfield` function to tag the priority into this packet's IP header using three bits of the TOS field (as shown in Fig. 5). Note that three bits can represent 8 priorities at most, which can match the number of priority queues in most commodity switches.

Packet retransmission: This module is mainly responsible for retransmitting the lost packet identified by the priority-based loss detection mechanism. Specifically, we have two operations. First, using some newly added codes in the `tcp_rcv_established` function, we check if an arriving ACK contains a flag (the 2nd and 3rd bit in TOS field). If no, the TCP/IP stack continues as usual. Otherwise, we go to the second operation, which first further checks the value of this flag. If the flag equals to 01, retransmission is needed; if it is 10, only scoreboard update is required. For retransmission, we get the sequence number of the lost packet according to the ACK's `ack_seq` field and then invokes the `tcp_transmit_skb` function to retransmit the lost packet.

Rate control: Since our implementation does not touch the congestion control code, QCLIMB is compatible with any TCP-like congestion control implementations (e.g., TCP, DCTCP). Note that when a loss is detected by the priority-based detection mechanism (§3.2.1), we halve the window size of the relevant flow to avoid injecting too many traffic in the network.

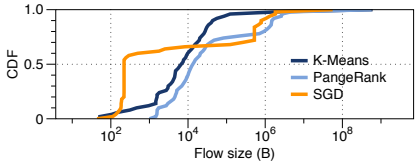


Figure 6: Workloads used for evaluation.

4.2 Receiver

Loss detection: This module contains three operations to enforce the loss detection mechanism described in §3.2.1. First, as packets are read in off the wire and converted to `skbs`, the packet priority is copied to the `skb->priority` field. Second, we compare the `skb->priority` of the first segment in the OOO receive queue with that of the last segment in the in-order TCP receive buffer to check if there is a missing gap. Third, QCLIMB adds a new `skb->flag` field in `skb_buff`. The `skb->flag` equals to 0 by default and `skb->flag=1` means that the lost packet needs retransmission. Note that `skb->flag` is in the receiver's kernel, we need to send it over the network to the relevant sender. To do this, we call a newly added function, `tcp_send_ack_qclimb`, to send a customized ACK and tag the `skb->flag` in this ACK packet. Specifically, we copy the `skb->flag` value to the 2nd&3rd bits of the TOS field in this ACK's IP header. Note here, when an OOO event is not due to loss, we will also send a customized ACK, but set the `skb->flag` value to 2 and copy it to the ACK's header to notify the sender to update the scoreboard solely.

Packet reordering: As OOO packets and in-order packets are separated into different queues, the reordering module is quite simple. Whenever there is a data packet, the receiver scans the OOO receive queue to move any in-sequence packets to the in-order receive buffer. After that, users leverage `recv` function to copy data from the TCP receive buffer to application layer receive buffer.

Remark: On the switch side, QCLIMB only needs to configure strict priority queuing (SP); if TCP/ECN transport is in use, ECN is also required. Both SP and ECN are standard features in existing commodity switches.

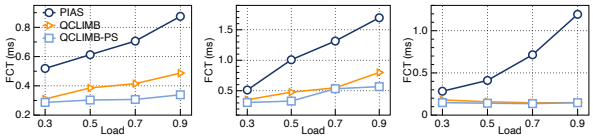
5 Evaluation

We evaluate QCLIMB through a combination of testbed experiments and large-scale simulations and show that

- QCLIMB achieves lower FCTs than PIAS (§5.1).
- QCLIMB's design components are effective (§5.2).
- QCLIMB works well in large datacenters (§5.3).

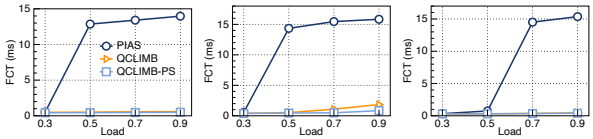
5.1 Testbed Experiments

Testbed: We build a small-scale testbed consisting of 8 servers connected to a Barefoot Tofino switch using 25Gbps links. Each server is equipped with a 16-core CPU (Intel Xeon Silver 4314@2.4GHz), 64G memory, a 25G NIC (Mellonax CX5), and installed with Ubuntu (kernel version 4.15.1). The switch runs SONIC and is



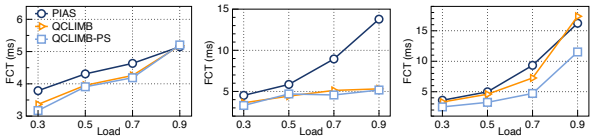
(a) K-Means (b) PageRank (c) SGD

Figure 7: Average FCT of 0-100KB small flows.



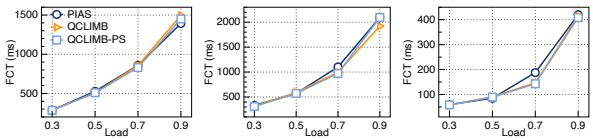
(a) K-Means (b) PageRank (c) SGD

Figure 8: 99th percentile tail FCT of 0-100KB small flows.



(a) K-Means (b) PageRank (c) SGD

Figure 9: Average FCT of 100KB-10MB medium flows.



(a) K-Means (b) PageRank (c) SGD

Figure 10: Average FCT of >10MB large flows.

configured with strict priority queuing with at most 8 queues. We use the DCTCP as the rate control.

Comparisons: We compare QCLIMB with PIAS [10] and QCLIMB-PS. We chose PIAS because it has the same practical features as QCLIMB but falls short in FCT minimization. QCLIMB-PS is a variant of QCLIMB that takes precise flow sizes as input and promotes a flow to higher-priority queues based on its remaining bytes. For OOO handling, it uses the same component as in §3.2. QCLIMB-PS is essentially a clairvoyant scheduler and is used for quantifying how far QCLIMB is from the scheme that has precise knowledge.

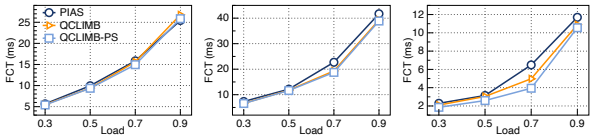
Workloads: As mentioned in §2, we mainly use three workloads provided by the paper [23]: K-Means, PageRank and SGD. Their distributions are shown in Fig. 6. All the three workloads exhibit a heavy-tailed distribution, where most flows are short and most of the traffic are dominated by a small percent of large flows. To replay these workloads in our testbed, we strictly keep the message arrival order and let message sizes follow the original testing traces. We reset the inter-arrival time of requests to match a particular network load.

Setup: For each workload, we first train an RF model (with the maximum tree depth $d = 10$ by default) for $\sim$3 minutes using 80% of the dataset and then use the trained RF model for testing with the remaining dataset to conduct our evaluation. We use 8 priority queues by default. We use the same queue thresholds as the PIAS paper [10] for all workloads. We set slack size S to 100KB by default. We set the RTO_{min} to 10ms and use a per-port buffer of 350KB at each switch.

Performance of small flows: Fig. 7 and Fig. 8 first show the average and tail FCT of small flows, respectively, for the K-Means, PageRank and SGD workloads with the network load varying from 0.3 to 0.9. We have the following two observations. First, for all workloads, QCLIMB outperforms PIAS in both the average and tail FCT of small flows. Compared to PIAS, it reduces the average/tail FCT of small flows by up to 44%/96%, 58%/96%, and 88%/97%, for the K-Means, PageRank, and SGD workloads, respectively. This is because QCLIMB promotes a flow's priority until its lower bound part finishes, while small flows are mostly scattered around their lower bounds. Therefore, unlike PIAS, QCLIMB will not let small flows suffer from the queue-climbing-down phase (see §5.2). Second, QCLIMB shows a comparable average and tail FCT of small flows with QCLIMB-PS. Particularly, compared to QCLIMB-PS at 0.7 network load, QCLIMB has a 3% gap for the small average FCT in PageRank workload and a 5% gap for the small tail FCT in SGD workload.

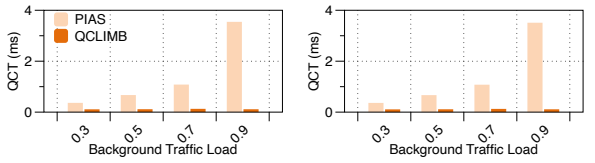
Performance of other flows: Fig. 9, Fig. 10 and Fig. 11 depict the FCT statistics for the other flows. The first observation is that QCLIMB outperforms PIAS for medium flows. Under a moderate 0.5 load, QCLIMB reduces the average FCT of medium flows over PIAS by 8%, 24%, and 8% for the K-Means, PageRank, and SGD workloads, respectively. This is expected because medium flows are more likely to leave a small amount of data after their lower bound parts finish (see Fig. 2), thus are more likely to complete in QCLIMB's delay demoting step. As the second observation, we find that QCLIMB even performs slightly better than PIAS for large flows in some cases. For instance, compared to PIAS, QCLIMB reduces the average FCT of large flows by 5% on average across all network loads in the SGD workloads. The last observation is for the average FCT of all flows. More precisely, QCLIMB shows up to 6%, 15%, and 23% reductions in overall average FCT over PIAS for the K-Means, PageRank, and SGD workloads, respectively.

Results with the Memcached application: We further build a Memcached application to evaluate QCLIMB's performance. We use one host as the client and the remaining 7 hosts as servers. Before sending each GET query (i.e., *memcached_get()*), the client will first send a SET (i.e., *memcached_set()*) request to pre-populate each server with a key-value pair. The key is flow id. The value is flow size which is randomly chosen according to a Facebook Memcached workload [8]. Once the SET request is completed, the client immediately sends a GET query to each server to get the value just populated by the SET request. We repeat this process

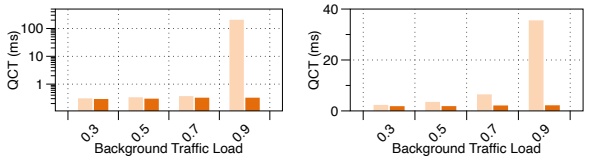


(a) K-Means (b) PageRank (c) SGD

Figure 11: Average FCT of all flows.



(a) Overall average QCT (b) Avg. QCT of <100KB queries

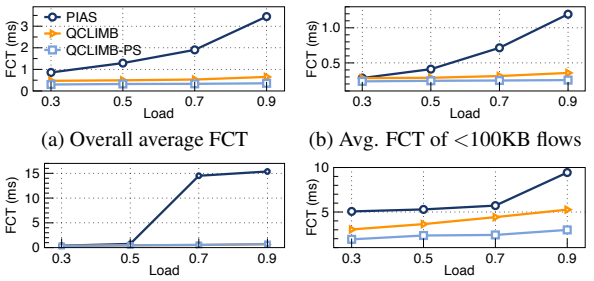


(c) Tail QCT of <100KB queries (d) Avg. QCT of >100KB queries

Figure 12: QCT statistics across different query sizes.

roughly 140000 times and thus generate around 140000 GET queries. A GET query completes only when the client receives all servers' responses. We only consider GET queries and measure the query completion time (QCT) as the application performance metric. We also inject background traffic, which is a mix of small and large flows generated according to a Web Search workload [5]. The background traffic load varies from 0.3 to 0.9. We calculate each query's size as the sum of its individual flows' size. Fig. 12 compares the QCT of QCLIMB with that of PIAS, across different query sizes and different background traffic loads. Compared to PIAS, QCLIMB reduces the overall average QCT by 70%∼97%, the average/tail QCT of 0-100KB small queries by 70%∼97%/6%∼99.8%, and the average QCT of >100KB queries by 21%∼94%. Note that QCLIMB's QCT does not increase with background traffic load. This is because background traffic is transmitted with the lowest priority, while all queries' traffic is in flows less than 100MB and will not overlap with background traffic in the same queue under QCLIMB's scheduling logic.

Tiny workload: We further evaluate QCLIMB in a Tensorflow workload from the Flux dataset [23]. This workload contains a significant proportion (i.e., 77%) of very tiny flows (i.e., with less than 100B) and thus is more likely to cause congestion than the other workloads. Despite this, we can observe from Fig. 13 that QCLIMB still shows superior performance. Compared to PIAS, it reduces the overall average FCT by up to 81.1%. For the average/tail FCT of small flows, the improvement of QCLIMB over PIAS is up to 70%/96.4%. The reductions of QCLIMB in small flows do not penalize other flows. In fact, QCLIMB even reduces the average FCT of >100KB flows by 22.7%∼44.2%, compared to PIAS.



(a) Overall average FCT (b) Avg. FCT of <100KB flows

(c) Tail FCT of <100KB flows (d) Avg. FCT of >100KB flows

Figure 13: FCT statistics across different flow sizes under Tensorflow workload.

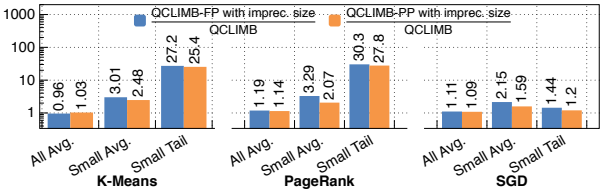


Figure 14: Effectiveness of error-tolerant scheduling.

5.2 QCLIMB Deep Dive

5.2.1 Effectiveness of the design components

Effect of lower-bound-based scheduling: We first check the effectiveness of QCLIMB's scheduling design. We construct two baselines, both of which directly use the estimated flow sizes for scheduling. The first one, QCLIMB-FP, determines a flow's priority only once by comparing its estimated flow size with queue thresholds. The second one, QCLIMB-PP, has two more steps than QCLIMB-FP. More precisely, it gradually promotes a flow's priority according to its remaining bytes to the estimated size. Afterward, if the flow is not yet ended and the sent bytes exceeds the estimated size, it directly pulls this flow to the lowest priority queue. Fig. 14 shows the results for different workloads under load 0.5. We observe that compared to QCLIMB-FP and QCLIMB-PP, respectively, QCLIMB brings up to 16% and 12% improvements in overall average FCT, and especially, it brings up to 70%/97% and 60%/96% improvements in average/tail FCT of small flows. Since QCLIMB is at most 1.63× slower than QCLIMB-PS for the average FCT of small flows (see 0.7 load in Fig. 7a), the 3.29× speedup of QCLIMB over QCLIMB-PP means that QCLIMB reduces the impact of prediction size errors on small flows by $\frac{3.29-1.63}{3.29-1} = 72\%$.

Effect of priority-based loss recovery: To quantify the effectiveness of our priority-based loss recovery mechanism, we construct a variant of QCLIMB, which abandons the proposed OOO design (§3.2) and uses the default TCP loss recovery mechanism. Fig. 15 compares the FCT performance achieved by this variant and QCLIMB for the K-Means, PageRank and SGD workloads under load 0.7. As we can see from this figure, QCLIMB outperforms the constructed variant significantly, with the overall average FCT and the average/tail

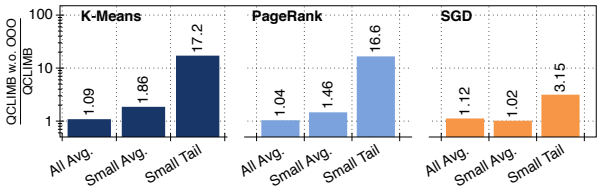
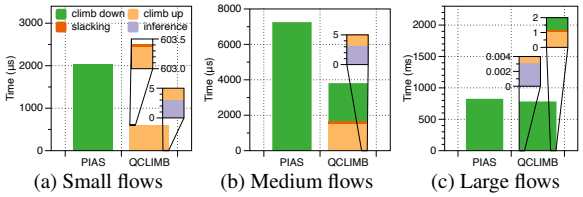


Figure 15: Effectiveness of priority-based loss recovery.



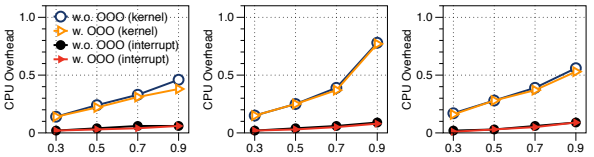
(a) Small flows (b) Medium flows (c) Large flows

Figure 16: Performance breakdown.

FCT of small flows being reduced by up to 1.12× and 1.86×/17.2×, respectively. Meaning, the priority-based loss recovery design brings up to 11% and 46%/94% improvements for the two flow types, respectively.

Performance breakdown: QCLIMB differs from PIAS because it has an additional scheduling step—queue-climbing-up, making it complete small flows with the first few higher-priority queues while separating medium/large flows into relatively low-priority queues. To confirm this, we conduct a performance breakdown evaluation for QCLIMB and PIAS under the PageRank workload at load 0.7. As we can see from Fig. 16, though QCLIMB has more processing steps for flow scheduling, it delivers significantly lower FCTs than PIAS. We further observe that for small flows in QCLIMB, only 0.52% of the FCT is spent for RF model inference, and the queue-climb-up step accounts for 99.47%, the remaining 0.01% is for the slacking step. By contrast, PIAS only contains a queue-climbing-down step and will keep demoting small flows to lower-priority queues, thus delivering poor performance. Medium and large flows in QCLIMB first go through model inference step and then climb up to higher-priority queues. They will stay in the highest-priority queue for short while using the slacking step, and finally will be penalized to lower-priority queues using the queue-climbing-down step. Owing to the queue-climbing-up step, QCLIMB achieves lower FCTs of medium/large flows than PIAS. Note that across all flow types, the model inference time is ∼3.1 μs, which is negligible as compared to the FCT of flows (603μs∼784 ms). This means QCLIMB can quickly obtain the lower/upper bounds for each flow and has the potential to apply to high-speed datacenter networks.

OOO handling Overhead: We conduct a testbed evaluation and consider two types of OOO handling overheads at receivers: the average CPU utilization (including software interrupts) and the OOO buffer space. Fig. 17 first compares the CPU overheads of QCLIMB with that of a QCLIMB variant using default TCP OOO. We observe that QCLIMB incurs up to 17.4% less CPU overhead



(a) K-Means (b) PageRank (c) SGD

Figure 17: CPU overhead and software interrupts for kernel packet processing.

Load	K-Means		PageRank		SGD	
	QCLIMB (TCP OOO)	QCLIMB	QCLIMB (TCP OOO)	QCLIMB	QCLIMB (TCP OOO)	QCLIMB
0.3	1338.70	591.66	1204.25	604.39	1793.06	936.81
0.5	2776.05	1548.54	1924.27	769.42	2006.02	1812.35
0.7	5752.96	4979.13	3133.81	2620.44	2316.89	1888.72
0.9	8077.62	6096.89	3436.34	2787.49	3333.34	2016.14

Table 3: Buffer length (bytes) for storing OOO packets.

than the baseline. Yet, the software interrupts only occupy 6.2%∼17% of the total CPU overhead. Table 3 further shows the buffer length for OOO packet storing. We observe that QCLIMB incurs 194B∼623B less OOO buffer than the default TCP OOO, equaling a 10%∼60% reduction. The reason why QCLIMB incurs less CPU overhead and OOO buffer space is that QCLIMB can quickly detect the loss and retransmit the lost packets to fill the gap in the receiver's sequence number list. Thus, OOO packets do not need to stay too long in the OOO buffer and can be quickly removed from this buffer.

5.2.2 Never-seen-before flow handling

One can either use QCLIMB or simply fall-back to PIAS for handling never-seen-before flows.

Using QCLIMB: Using QCLIMB for never-seen-before flow scheduling forms a model-application mismatching scenario. To test QCLIMB in this case, we evaluate the performance of the PageRank workload at 0.7 network load while using the RF models trained based on the K-Means and SGD workloads. Fig. 18 shows the FCT statistics. We observe that when using the SGD model for the PageRank workload, the average FCT of all flows and small flows can be prolonged by 3.63× and 2.33×, respectively. This is expected because the PageRank and SGD workloads have very different flow size distributions (see Fig. 6). By contrast, when using the RF model derived from a workload having a similar flow size distribution, the FCTs will not be affected too much. For example, the average FCT of small flows is slowed down by only 0.5% when using the K-Means RF model for the PageRank workload.

Using PIAS: Using PIAS for never-seen-before flows leads to QCLIMB flows coexisting with PIAS ones. To show how these flows affect each other, we conduct an experiment where PIAS traffic uses the WebSearch workload [5] at load 0.3, and QCLIMB traffic is the PageRank workload at load 0.5. We compare this mixing solution with a baseline that uses PIAS for all traffic. Fig. 19 shows the results. We see that for both workloads,

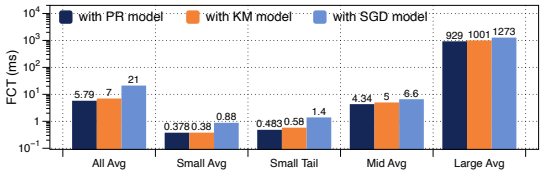


Figure 18: PageRank workload with mismatched models.

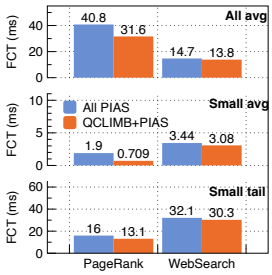


Figure 19: Mixing PIAS with QCLIMB.

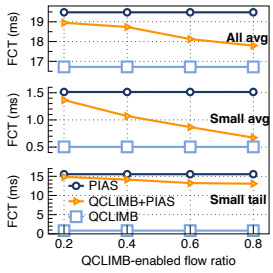


Figure 20: Impact of QCLIMB flow ratio.

the mixing solution achieves lower FCTs than the baseline. Particularly, the average FCT of small flows can be reduced by 62.7% and 10.1% for the PageRank and WebSearch workloads, respectively. This implies that mixing QCLIMB traffic in the network helps improve the performance of non-QCLIMB traffic. Moreover, as the PIAS flows does not use QCLIMB, the reduction of their FCTs under the mixing solution is less than that of QCLIMB-enabled PageRank flows.

Varying QCLIMB-enabled flow ratio: We proceed to investigate the impact of the QCLIMB-enabled flow ratio. We use PageRank workload at network load 0.7 and divide the traffic into two parts. The first part is non-QCLIMB flows that use PIAS for scheduling, while the second is QCLIMB-enabled flows. We vary the QCLIMB-enabled flow ratio from 0.2 to 0.8. We compare PIAS, QCLIMB, and the QCLIMB+PIAS mixing solution. Fig. 20 depicts the results. As expected, the performance of the mixing solution is between PIAS and QCLIMB. Even when there are 20% QCLIMB flows, the mix solution delivers ∼10% lower average FCT of small flows than PIAS. We further observe that the higher the QCLIMB-enabled flow ratio, the more flows can depart quicker, thus leaving more room for non-QCLIMB flows and consequently delivering lower FCTs.

5.2.3 Sensitivity to parameter settings

Impact of number of queues: To validate the impact of the number of queues on QCLIMB, we further conduct the testbed evaluation with 2 and 4 priority queues under the PageRank workload at 0.7 load. Fig. 21 shows the results. We observe that in general, QCLIMB outperforms PIAS under all shown settings. Even with two queues, QCLIMB works well and reduces the overall average FCT and the average/tail FCT of small flows by 28.6% and 74.3%/96.3%, respectively, compared to PIAS.

Impact of model accuracy: To understand the impact of the flow size predictor, we tested QCLIMB in the PageRank workload at 0.5 load, under different RF models with varying maximum tree depth d. Fig. 15 shows the results. It is clear that the overall average FCT decreases as d grows. This means that the higher the RF model's accuracy, the better performance QCLIMB achieves. We can further find that even under $d = 5$, QCLIMB still performs significantly better than PIAS, with the overall average FCT reduced by 16.3% and the average/tail FCT of small flows by 50%/93.8%. Note that for the extreme case of $d = 1$, the RF model may predict a wide size range for each flow, thus making QCLIMB achieve a similar overall average FCT with PIAS. But the predicted lower bounds of this $d = 1$ model might help QCLIMB complete most small flows during the queue-climbing-up phase. This makes QCLIMB achieve a dramatically lower average/tail FCT of small flows than PIAS (see Fig. 22b and Fig. 22c).

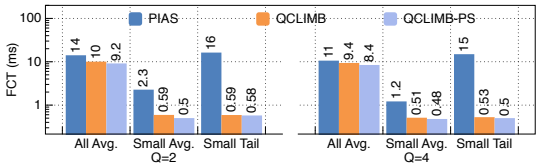
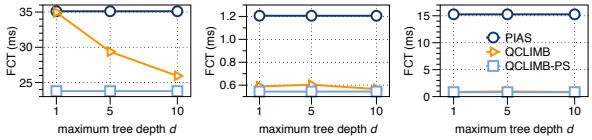


Figure 21: Impact of number of queues.



(a) Overall Avg. (b) Small Avg. (c) Small Tail

Figure 22: PageRank workload with different RF models.

Impact of slack size S: QCLIMB allows each flow to transmit a slack size S more data after its lower bound part finishes. QCLIMB outperforms PIAS under a wide range of S, and setting S to 100KB achieves the best performance for QCLIMB (appendix D).

5.3 Large-scale Simulations

Settings: In our simulations, we use the same topology as prior evaluations of PIAS [10] and pFabric [7], consisting of 144 hosts divided among 9 racks with a 2-level switching fabric; the difference is that we use 40Gbps host links and 100Gbps core links. Again, we use the K-Means, PageRank, and SGD workloads as above. We vary the load from 0.3 to 0.9.

Comparison with PIAS: Fig. 23 compares the FCT statistics of QCLIMB with that of PIAS. Compared to PIAS, QCLIMB reduces the overall average FCT by 17.4%∼47.9%, 9.1%∼38.3%, and 37.9%∼49.5%, for the K-Means, PageRank and SGD workloads, respectively. QCLIMB also achieves 1%∼30.1%/1%∼62%, 1%∼35.1%/1%∼55%, and 3%∼46.5%/2%∼42.5% lower average/tail FCT of small flows than PIAS, in the K-Means, PageRank and SGD workloads, respectively. Such good results demonstrate the effectiveness of QCLIMB's scheduling. Note

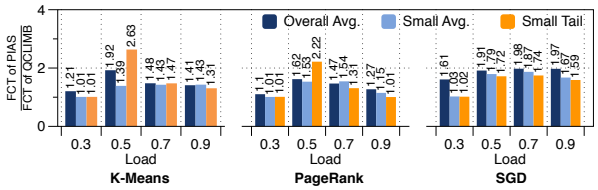
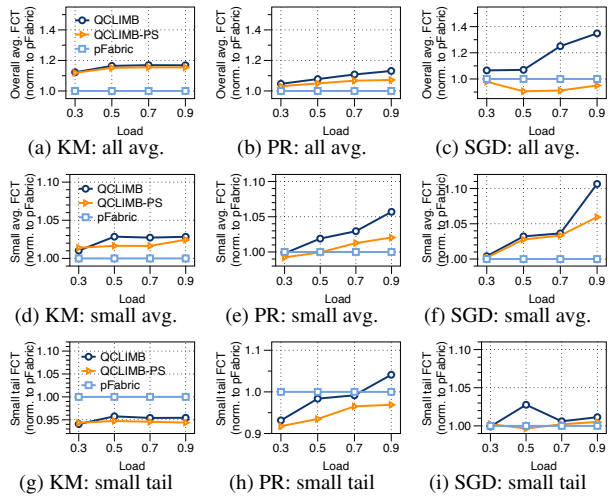


Figure 23: Comparison with PIAS in simulations.



(a) KM: all avg. (b) PR: all avg. (c) SGD: all avg.

(d) KM: small avg. (e) PR: small avg. (f) SGD: small avg.

(g) KM: small tail (h) PR: small tail (i) SGD: small tail

Figure 24: Comparison with QCLIMB-PS and pFabric in simulations (KM: K-Means; PR: PageRank).

that QCLIMB's overall improvement over PIAS in SGD workload is relatively larger than those in other workloads. The reason is that the SGD workload is more skewed. Around ∼95% and ∼74% flows in the K-Means and PageRank workloads, respectively, are smaller than 100KB, while only ∼66% flows in the SGD workload, are smaller than 100KB. In such case, large flows are more likely to coexist with short flows under PIAS. QCLIMB is less likely to be affected by such problems, as medium and large flows have relatively higher lower bounds and take less time in queue-climbing-up phase, thus exhibiting less time to coexist with small flows.

Comparison with precise-knowledge-aware schemes: We compare QCLIMB, QCLIMB-PS (§5.1) and pFabric [7]. Note that QCLIMB-PS and pFabric take precise flow sizes as scheduling input. Fig. 24 depicts the results and reveals the following findings. First, QCLIMB does not show severely worse overall performance in the K-Means and PageRank workloads. Specifically, under K-Means, QCLIMB achieves 0.4%∼1.4% and 12%∼17% higher overall average FCT than QCLIMB-PS and pFabric, respectively. For PageRank, this gap is 1.6%∼5.9% to QCLIMB-PS and 4.6%∼13.1% to pFabric. The average gap of QCLIMB to pFabric is 9% in PageRank workload. Second, for the SGD workload, QCLIMB's overall performance has a relatively large gap to QCLIMB-PS and pFabric. This is because that the SGD workload is more skewed, leading large flows in QCLIMB to be more likely to coexist with small flows. Third, QCLIMB can achieve comparable performance

Flow bins	Small flows	Medium flows	Large flows	All flows
QCLIMB	0	0	124	124 (83 by loss detection)
pFaric	10	25	414	414 (all by timeout)

Table 4: Loss events comparison (K-Means, 0.7 load).

with QCLIMB-PS for small flows.For the average FCT of small flows, QCLIMB has 1%∼2.8%, −0.2%∼5.7%, and 0.4%∼10.5%, for the K-Means, PageRank and SGD workloads, respectively. Fourth, we find that the performance gap between QCLIMB and QCLIMB-PS in the K-Means workload is smaller than that in the other workloads. This is because the RF model trained over K-Means has a higher prediction accuracy than that over the PageRank and SGD workloads. Note that QCLIMB does not severely penalize large flows. In fact, Compared to PIAS, it achieves up to 48.2% lower average FCT of large flows. Further, it has at most 15% gap to pFabric in the average FCT of large flows. For more details on the performance of large flows, please see appendix D.

Remark: Fig. 24g and Fig. 24h reveal that QCLIMB can even deliver a slightly lower tail FCT of small flows than pFabric. This is because QCLIMB incurs fewer packet loss events than pFabric. As a concrete example, Table 4 shows that QCLIMB has 70% fewer loss events than pFabric and incurs no packet loss for small and medium flows, under the K-Means workload at network load 0.7. Moreover, 83 of these 124 loss events can be quickly detected by QCLIMB's priority-based loss detection mechanism (§3.2.1), whereas the packet loss events in pFabric can only be detected by timeout.

6 Conclusion

Scheduling flows with imprecise knowledge is an important and practical problem that has been neglected by prior work in this field. QCLIMB bridges this gap with a key observation that it is possible to accurately estimate each flow's lower bounds with ML techniques. QCLIMB employs a lower-bound-based flow scheduling scheme to ensure small flows can be prioritized over medium/large ones from the beginning of transmission. It also contains fast priority-based lost recovery and packet reordering mechanisms to handle OOO issues resulting from the scheduling. We have implemented a QCLIMB prototype using all commodity hardware, and evaluated it through small-scale testbed experiments and large-scale simulations. The results show that QCLIMB is a viable solution for scheduling flows with imprecise size information.

Acknowledgments

We thank the anonymous NSDI reviewers and our shepherd Prof. Rachit Agarwal for their constructive feedback and suggestions. This work was supported in part by the NSFC Grant (62202325, 62062005), the Hong Kong RGC TRS T41-603/20-R. Keqiu Li is the corresponding author: keqiu@tju.edu.cn.

References

[1] Linux netfilter. http://www.netfilter.org/.

[2] memcached: a distributed memory object caching system. http://www.memcached.org/, 2011.

[3] Redis. https://redis.io/, 2021.

[4] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proc. of ACM SIGCOMM*, 2014.

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

[6] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proc. of USENIX NSDI*, 2012.

[7] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013.

[8] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proc. of ACM SIGMETRICS*, 2012.

[9] Wei Bai, Kai Chen, Shuihai Hu, Kun Tan, and Yongqiang Xiong. Congestion control for high-speed extremely shallow-buffered datacenter networks. In *Proc. of the Asia-Pacific Workshop on Networking (APNet)*, 2017.

[10] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *Proc. of USENIX NSDI*, 2015.

[11] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Pias: Practical information-agnostic flow scheduling for commodity data centers. *IEEE/ACM Transactions on Networking*, 2017.

[12] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. Enabling {ECN} in multi-service multi-queue data centers. In *Proc. of USENIX NSDI*, 2016.

[13] Wei Bai, Shuihai Hu, Kai Chen, Kun Tan, and Yongqiang Xiong. One more config is enough: Saving (dc) tcp for high-speed extremely shallow-buffered datacenters. In *Proc. of IEEE INFOCOM*, 2020.

[14] Steven Blake, David Black, Mark Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An architecture for differentiated services. 1998.

[15] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[16] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. dcpim: Near-optimal proactive datacenter transport. In *Proc. of ACM SIGCOMM*, 2022.

[17] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. Scheduling mix-flows in commodity datacenters with karuna. In *Proc. of ACM SIGCOMM*, 2016.

[18] Li Chen, Shuihai Hu, Kai Chen, Haitao Wu, and Danny HK Tsang. Towards minimal-delay deadline-driven data center tcp. In *Procs. of ACM HotNets*, 2013.

[19] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proc. of ACM SIGCOMM*, 2018.

[20] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proc. of ACM SIGCOMM*, 2017.

[21] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *Proc. of ACM SIGCOMM*, 2015.

[22] Allen B Downey. The structural cause of file size distributions. In *Proceedings of the 2001 ACM SIGMETRICS*, 2001.

[23] Vojislav Dukic, Sangeetha Abdu Jyothi, Bojan Karlas, Muhsen Owaida, Ce Zhang, and Ankit Singla. Is advance knowledge of flow sizes a plausible assumption? In *Proc. of USENIX NSDI*, 2019.

[24] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proc. of ACM CoNext*, 2015.

[25] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta

Sengupta. Vl2: a scalable and flexible data center network. In *Proc. of ACM SIGCOMM*, 2009.

[26] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues dont matter when you can {JUMP} them! In *Proc. of USENIX NSDI*, 2015.

[27] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proc. of ACM SIGCOMM*, 2017.

[28] Chi-Yao Hong, Matthew Caesar, and P Brighten Godfrey. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM Computer Communication Review*, 42(4):127–138, 2012.

[29] Christian Hopps et al. Analysis of an equal-cost multi-path algorithm. Technical report, RFC 2992, November, 2000.

[30] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[31] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A building block for proactive transport in datacenters. In *Proc. of ACM SIGCOMM*, 2020.

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

[33] Ziyang Li, Wei Bai, Kai Chen, Dongsu Han, Yiming Zhang, Dongsheng Li, and Hongfang Yu. Rate-aware flow scheduling for commodity data center networks. In *Proc. of IEEE INFOCOM*, 2017.

[34] Yuanwei Lu, Guo Chen, Larry Luo, Kun Tan, Yongqiang Xiong, Xiaoliang Wang, and Enhong Chen. One more queue is enough: Minimizing flow completion time with explicit priority notification. In *Proc. of IEEE INFOCOM*, 2017.

[35] Larry Medsker and Lakhmi C Jain. *Recurrent neural networks: design and applications*. CRC press, 1999.

[36] Nicolai Meinshausen and Greg Ridgeway. Quantile regression forests. *Journal of Machine Learning Research*, 7(6), 2006.

[37] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proc. of ACM SIGCOMM*, 2018.

[38] A. Munir, I. A. Qazi, Z. A. Uzmi, A. Mushtaq, S. N. Ismail, M. S. Iqbal, and B. Khan. Minimizing flow completion times in data centers. In *Proc. of IEEE INFOCOM*, 2013.

[39] Ali Munir, Ghufran Baig, Syed M Irteza, Ihsan A Qazi, Alex X Liu, and Fahad R Dogar. Friends, not foes: synthesizing existing transport strategies for data center networks. In *Proc. of ACM SIGCOMM*, 2014.

[40] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015.

[41] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: a centralized" zero-queue" datacenter network. In *Proc. of ACM SIGCOMM*, 2014.

[42] Pascal Poupart, Zhitang Chen, Priyank Jaini, Fred Fung, Hengky Susanto, Yanhui Geng, Li Chen, Kai Chen, and Hao Jin. Online flow size prediction for improved network routing. In *Proc. of IEEE ICNP*, 2016.

[43] Vaidyanathan Ramaswami, Kaustubh Jain, Rittwik Jana, and Vaneet Aggarwal. Modeling heavy tails in traffic sources for network performance evaluation. In *Proceedings of ICC3*, 2014.

[44] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. A computational approach to packet classification. In *Proc. of ACM SIGCOMM*, 2020.

[45] Marko Robnik-Šikonja. Improving random forests. In *Proceedings of Springer ECML*, 2004.

[46] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *Proc. of USENIX NSDI*, 2017.

[47] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 41(4):50–61, 2011.

[48] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. Coda: Toward automatically identifying and scheduling coflows in the dark. In *Proc. of ACM SIGCOMM*, 2016.

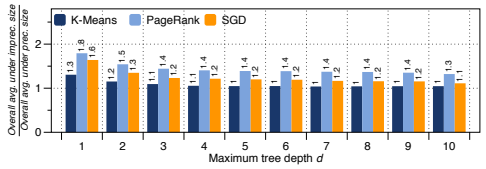
[49] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. Resilient datacenter load balancing in the wild. In *Proc. of ACM SIGCOMM*, 2017.

[50] Junxue Zhang, Wei Bai, and Kai Chen. Enabling ecn for datacenter networks with rtt variations. In *Proc. of ACM CoNEXT*, 2019.

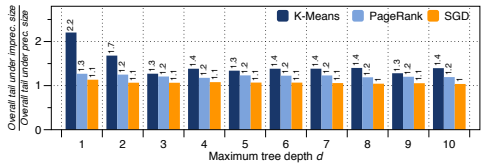
[51] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. Aequitas: admission control for performance-critical rpcs in datacenters. In *Proceedings of ACM SIGCOMM*, 2022.

A The Performance of pFabric under Imprecise Knowledge

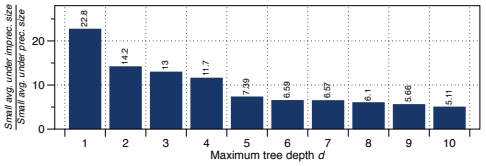
In this section, we want to know how pFabric [7] performs when scheduling flows with imprecise flow sizes. We use the same topology and workloads as §5.3. The load is 0.7. Fig. 25 shows the FCT statics across different flow sizes when using imprecise flow sizes for pFabric [7] as compared to using precise knowledge. From this figure, we observe that due to the errors in flow sizes, the FCT achieved by pFabric can be slowed down significantly. More specifically, the overall average/tail FCT achieved by pFabric with imprecise flow sizes can be increased by up to 1.8×/2.2×. Moreover, the average FCT slowdown of small flows can even reach 22.8×.



(a) Slowdown in overall average FCT



(b) Slowdown in overall tail FCT



(c) Slowdown in avg. FCT of small flows with PageRank

Figure 25: FCT slowdown across different flow sizes when using imprecise flow sizes for pFabric [7] as compared to using precise knowledge.

B Obtaining Prediction Intervals

For brevity, let us consider an RF model with T trees, trained over a set of flows, i.e., $\mathcal{N} = \{1, \ldots, n\}$. Each flow i is identified by (X_i, y_i), where X_i contains the flow features and y_i is the flow size. For a new (testing) flow with its $X = x$, each tree t will drop it down to a specific leaf node l. Let $\mathcal{N}_l$ denote the set of flows falling in l. The tree t will predict a size for this flow as the weighted average over the sizes of all flows in $\mathcal{N}_l$, i.e., $y_t = \sum_{i=1}^{n} w_i(x,t) \cdot y_i$, where $w_i(x,t) = 1/|\mathcal{N}_l|$ if flow $i \in \mathcal{N}_l$ and 0 otherwise. For the entire RF, the predicted size is thus the average prediction of all trees, i.e., $y = \frac{1}{T}\sum_{t=1}^{T} y_t = \sum_{i=1}^{n} w_i(x) \cdot y_i$, where $w_i(x) = \frac{1}{T}\sum_{t=1}^{T} w_i(x,t)$ represents the weight of flow i in predicting the new flow's size. This implies that the conditional mean of y, given $X = x$, is approximated by a weighted mean over the sizes of all flows in $\mathcal{N}$, namely, $E(y|X = x) = \sum_{i=1}^{n} w_i(x) y_i$.

Unlike RF focusing on the conditional mean, QRF considers the conditional distribution of y under $X = x$ by defining it as the probability that for $X = x$, y is smaller than y', i.e., $F(y'|X = x) = P(y \leq y'|X = x) = E(1_{\{y \leq y'\}}|X = x)$. Just as the way approximating $E(y|X = x)$, $E(1_{\{y \leq y'\}}|X = x)$ can be approximated by the weighted mean over the observations of $1_{\{y \leq y'\}}$. This leads to $F(y'|X = x) = \sum_{i=1}^{n} w_i(x) \cdot 1_{\{y \leq y'\}}$. With this function, the α-quantile $Q_\alpha(x)$ is defined such that the probability of y being smaller than $Q_\alpha(x)$ is, for a given $X = x$, exactly equal to α, i.e., $Q_\alpha(x) = \inf\{y' : F(y'|X = x) \geq \alpha\}$. Using $Q_\alpha(x)$, one can build a prediction interval. For instance, a β $(0 \leq \beta \leq 1)$ prediction interval for the value of y is $[Q_{(1-\beta)/2}(x), Q_{(1+\beta)/2}(x)]$. Thus, $Q_{(1-\beta)/2}(x)$ and $Q_{(1+\beta)/2}(x)$ are taken as the lower and upper bounds, respectively, for the estimated size of the new flow. We set β to 1 unless otherwise specified.

C QCLIMB's Scheduling Algorithm

Algorithm 1 shows the pseudocode of QCLIMB's lower-bound-based scheduling. To understand this procedure, we use an illustrating example in Fig. 26. In this example, there are two flows (i.e., f1 and f2) and one switch port capable of transmitting 1 unit of data at each time. This port has three priority queues (i.e., Q1, Q2, and Q3), with queue thresholds being ≤ 2, $[3,5]$, and ≥ 6, respectively. We consider $S = 0$ for ease of presentation. As shown in Fig. 26(a), f1 is actually larger than f2, but its predicted size is smaller than f2's. Purely performing queue-climbing-up according to predicted size will make

Load	K-Means			PageRank			SGD		
	PIAS	QCLIMB-PS	pFabric	PIAS	QCLIMB-PS	pFabric	PIAS	QCLIMB-PS	pFabric
0.3	1.21/1.67	0.99/1.0	0.89/1.07	1.11/1.41	0.99/1.0	0.96/1.09	1.25/1.11	1.0/1.01	1.06/1.06
0.5	1.93/1.53	0.99/1.0	0.86/0.95	1.60/1.84	0.99/1.0	0.95/1.07	1.22/1.16	1.02/1.05	1.07/1.08
0.7	1.48/1.37	0.99/0.99	0.85/1.07	1.47/1.56	0.99/1.0	0.92/1.10	1.29/1.23	1.05/1.04	1.10/1.10
0.9	1.41/1.25	0.99/1.01	0.86/1.08	1.34/1.58	0.99/1.01	0.91/1.09	1.36/1.27	1.06/1.05	1.08/1.07

Table 5: Average/tail FCT of large flows achieved by different schemes in simulation (norm. to QCLIMB).

Algorithm 1 QCLIMB's Priority Tagging Algorithm

Require: An incoming packet p; A slack size S; Queue priorities $P = \{P_1, \ldots, P_k\}$ and thresholds $\alpha = \{\alpha_1, \ldots, \alpha_k\}$

1: **procedure** PRIORITYTAGGING
2: $B \leftarrow bytes_sent_of_parent_flow(p)$
3: $L \leftarrow lower_bound_of_parent_flow(p)$
4: $U \leftarrow upper_bound_of_parent_flow(p)$
5: **if** $B < L$ **then** ▷ priority promoting
6: $v \leftarrow get_priority(L-B, P, \alpha)$
7: **else if** $B < L+S$ **then** ▷ delay demoting
8: $v \leftarrow$ the value of the highest priority P_1
9: **else if** $B < U$ **then** ▷ priority demoting
10: $v \leftarrow get_priority(B, P, \alpha)$
11: **else**
12: $v \leftarrow$ the value of the lowest priority P_K
13: **end if**
14: Tag v into the packet p's header
15: **end procedure**

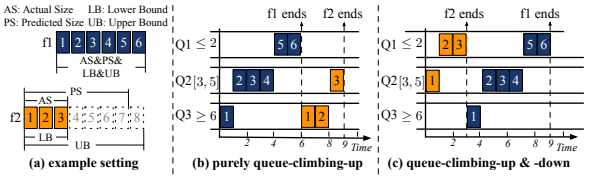


Figure 26: An example for illustrating QCLIMB's error-tolerant scheduling.

f1 complete faster than f2 and result in an average FCT of $\frac{6+9}{2} = 7.5$. Conversely, QCLIMB lets f1 and f2 initially enter Q3 and Q2, respectively, according to their lower bounds. As a result, f2 has a higher priority than f1 to transmit data and will be promoted to Q1 when its lower bound portion reduces to 2. At time 3, since its lower bound portion has finished, f2 will be moved back to the queue it initially entered, i.e., Q2, where it finishes its last packet transmission. After that, f1 starts data transmission until time 10. Thus, the average FCT is reduced to $\frac{3+9}{2} = 6$.

D Supplementary Experiment Results

Impact of slack size S [Testbed]: QCLIMB allows each flow to transmit a slack size S more data after its lower bound part finishes. To understand the impact of S, we measure the FCT statistics of QCLIMB with different S in the PageRank workload at load 0.7. As we can see from Fig. 27, with various settings of S, QCLIMB reduces the overall average FCT by 4.1%∼18.3% and the average/tail FCT of small flows by 48.6%∼62.2%/15.6%∼93.4%, compared to PIAS. We further observe that the overall average FCT of QCLIMB reduces as S increases at the beginning (e.g., before S reaches 100KB). This is reasonable because a larger S makes more flows (especially small ones) complete at the delay demoting step. However, as S keeps increasing, this step will introduce more medium and large flows to coexist with small flows in high-priority queues. That's why the FCTs of QCLIMB go up after S=100KB. In general, QCLIMB shows performance improvements over PIAS under a wide range of S, and we consider setting S to 100KB (§5.1) is a feasible choice.

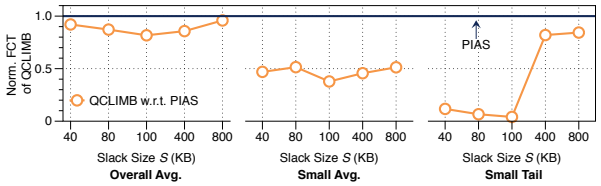
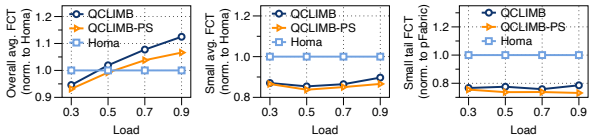


Figure 27: Sensitivity to the parameter of slack size S.

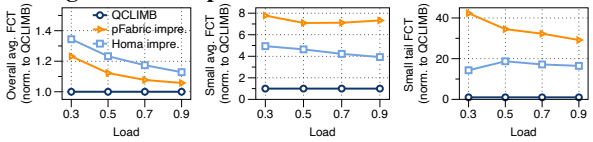
Large flow performance [Simulation]: Table 5 further summarizes the average and tail FCT of large flows achieved by different schemes in simulations. We have the following observations. First, QCLIMB achieves 10%∼48.2%/10%∼45.7% lower average/tail FCT of large flows than PIAS, across all workloads at all loads. Second, for the average FCT of large flows, QCLIMB is worse than pFabric by 14%∼4% across the K-Means and PageRank workloads, and outperforms pFabric by 6%∼10% in the SGD workload. Third, QCLIMB maintains a lower tail FCT of large flows than pFabric for most of the tested cases. In particular, it cuts the tail FCT of large flows by 10% in both PageRank and SGD workloads at 0.7 load, compared to pFabric. The reason is again that pFabric detects packet loss too late with timeout, deferring the retransmission of lost packets and prolonging tail FCTs accordingly.

Comparison with Homa [Simulation]: We compare QCLIMB with Homa [37] under the PageRank workload. The results are shown in Fig. 28. We observe that QCLIMB has up to a 12.5% gap to Homa



(a) All avg. (b) Small avg. (c) Small tail

Figure 28: Comparison with Homa in simulation.



(a) All avg. (b) Small avg. (c) Small tail

Figure 29: Comparison with pFabric and Homa using imprecise knowledge in simulation.

for the overall average FCT. This is expected because QCLIMB uses the reactive DCTCP for rate control. By contrast, Homa can proactively allocate bandwidth as "grants" to senders, who can then send scheduled packets at the correct rate to realize high utilization. Homa's improvements mainly come from medium/large flows. In fact, our experiments show that Homa can achieve up to 24.6%/6.7% lower average FCT of medium/large flows (for brevity, we omit the detailed figures). For small flows, Homa performs worse than our QCLIMB. As shown in Fig. 28b and Fig. 28c, the reduction of QCLIMB over Homa in the average/tail FCT of small flows is up to 14.6%/24.2%. This is mainly because of Homa's proactive nature, which enforces Homa to blindly transmit unscheduled packets in the first RTT, causing traffic bursts, non-trivial queuing delay, and eventually, packet losses. Such first RTT issue will seriously affect the performance of small flows [31].

Comparison with pFabric and Homa using imprecise knowledge [Simulation]: Readers may wonder if QCLIMB can outperform pFabric and Homa with imprecise knowledge as scheduling input. To answer this, we conduct an experiment over PageRank workload with varying network load. The results are shown in Fig. 29. Compared to pFabric and Homa with imprecise knowledge, QCLIMB can reduce the average FCT of all flows by up to 18.8% and 25.6%, respectively. For the average/tail FCT of small flows, the reductions of QCLIMB are even larger: 85.9%∼87.1%/96.6%∼97.6% and 74.5%∼79.8%/93%∼94.7% over pFabric and Homa with imprecise knowledge, respectively.

E Discussion

The QCLIMB testbed measurements in this paper were based on 25 Gbps link speeds, but QCLIMB may still work in higher-speed networks. The reason is that the dominant overhead of an end-to-end QCLIMB flow is the end-host processing delay. Such processing delay is independent of the line-rates and is mainly caused by kernel network stack processing. In our evaluation, we observe that a 100KB flow requires roughly 200μs for the end-host processing on both sender and receiver sides. By contrast, QCLIMB's model inference takes only 3μs, which is negligible as compared to such processing delay. One can further reduce this inference latency to 1μs with advanced FPGA hardware [23] or may even reduce it to tens of nanoseconds by carefully pipelining RF's decision tree on hardware [44].

F Other Related Work

We have discussed the closely related works [7, 28, 10, 34, 41, 24, 37, 23] extensively in §2.1. Here, we only review some other DCN flow scheduling ideas that have not been discussed elsewhere. For example, NDP [27] uses receiver-driven scheduling but can only account for fair sharing rather than SRPT. Moreover, it relies on special hardware support from switches to provide receivers a full view of the traffic demand. Aeolus [31] adds selective dropping and loss recovery on top of receiver-driven transports (e.g., Home [37], NDP [27], ExpressPass[20]), suffering the same issues as above. dcPIM [16] takes multiple rounds of matching to compute conflict-free pairings of sender and receiver to achieve high utilization, but relies on precise flow size information and also needs to refactor the network stack. QJUMP [26] applies Internet QoS-like techniques (e.g., DiffServ [14]) to schedule flows of datacenter applications but requires the application itself to specify the priorities. Karuna [17] schedules a mix of flows with and without deadlines. It uses PIAS-like mechanisms for non-deadline flows and hence suffers from the same effectiveness problems as PIAS. PASE [39] synthesizes exiting transport designs to provide good performance, but requires non-trivial switch modification or complex control plane for arbitration. Auto [19] applies deep reinforcement learning (DRL) techniques to flow scheduling and traffic optimization. Aequitas [51] uses weighted fair queuing (WFQ) to guarantee RPC-level service-level objectives (SLOs), while QCLIMB employs strict priority queuing for prioritizing small flows to minimize FCTs. Above all, there exist no existing schedulers in the literature that can address the flow scheduling problems under imprecise flow sizes.

There are other DCN research efforts such as congestion control (e.g., DCTCP [5], D3 [47], MCP [18], L2DCT [38], HULL [6], HPCC [32], BCC [13, 9], MQECN [12], and ECN$^\sharp$ [50]) and multi-path load balancing (e.g., CONGA[4], Flowlet [46], and Hermes [49]). These designs are insufficient for FCT minimization as in-network priority queues are not used.

Pudica: Toward Near-Zero Queuing Delay in Congestion Control for Cloud Gaming

Shibo Wang[†‡], Shusen Yang[†], Xiao Kong[‡], Chenglei Wu[‡], Longwei Jiang[‡], Chenren Xu[*], Cong Zhao[†], Xuesong Yang[•], Jianjun Xiao[‡], Xin Liu[‡], Changxi Zheng[◦], Jing Wang[‡], Honghao Liu[‡]

[†]*Xi'an Jiaotong University*, [‡]*Tencent Inc.*, [*]*Peking University*, [•]*Bonree*,
[◦]*Pixel Lab, Tencent America, and Columbia University*

Abstract

Congestion control (CC) plays a pivotal role in cloud gaming services. However, existing CC methods often cause self-induced bottleneck queuing. As a result, they may largely delay game frame transmission and undermine the player's gaming experience. We present a new end-to-end CC algorithm named *Pudica* that strives to achieve near-zero queuing delay and high link utilization while respecting cross-flow fairness. Pudica introduces several judicious approaches to utilize the paced frame to probe the bandwidth utilization ratio (BUR) instead of bandwidth itself. By leveraging BUR estimations, Pudica designs a holistic bitrate adjustment policy to balance low queuing, efficiency, and fairness. We conducted thorough and comprehensive evaluations in real networks. In comparison to baseline methods, Pudica reduces the average and tailed frame delay by 3.1× and 4.9× respectively, and cuts down the stall rate by 10.3×. Meanwhile, it increases the frame bitrate by 12.1%. Pudica has been deployed in a large-scale cloud gaming platform, serving millions of players.

1 Introduction

The digital world has been largely on the "cloud", a metaphor suggesting that storage- and computation-heavy applications should be in the ether, available through the Internet whenever and wherever we need them. The recent emergence of next generation internet infrastructures such as WiFi-7, 5G, and full fibre broadband catalyzes more cloud-based applications, and one of the most notable is *cloud gaming*. Thanks to the academic research effort [1–5] and industrial investment [6–9], cloud gaming has already gained worldwide popularity while still under rapid growth [10].

In the cloud gaming setup (see Fig. 1), the end device collects user operations (e.g., mouse clicks and finger taps) and immediately sends them to the cloud server, which runs the entire game engine. Then, the corresponding game frame is rendered on the server, streamed to the end device, and displayed locally. In this way, the often heavy-duty game engine is moved to the cloud; the end device is only responsible for user interactions, and thus, it can be lightweight, portable, and low-cost while still providing engaging gaming experiences.

This is a promising paradigm for gaming, but a key premise of this paradigm is that the game frames rendered on the cloud must be transmitted to the end device at a *consistent ultra-low* delay, typically within 50 *ms*. High frame delays, even if occasional, would severely undermine the gaming interactivity and in turn the player's quality of experience (QoE) (see our studies in §2.3). While many congestion control (CC) algorithms [11–17] have been proposed to restrict the inflation of bottleneck queuing, none of these solutions actually offers ultra-low frame delays in both average and tail (§5.2).

Prior delay-aware CC methods [11, 12, 18, 19] strive for approaching an *ideal* state with high link utilization, zero queuing, and fair allocation, which is indeed a state we long for as well. However, these methods often require a substantial queue buildup *initially* to probe the link status before making adjustments toward the ideal state. The periodic nature of overshooting-based probing actions leads to frequent *self-induced* queuing. Recently, some CC methods [13,20] achieve low latency in single-flow or isolated environments, but they fail to maintain low queuing when multiple flows coexist.

To achieve both stringent delay control and the highest possible bitrate, a cloud gaming system must be equipped with a *carefully* designed CC algorithm that meets the following three requirements. Firstly, it should be able to reach high link utilization (i.e., efficiency) without resorting to overshoot-based network probing. Secondly, heavy queuing should be also prevented when multiple homogeneous flows run concurrently. Thirdly, when the link condition suddenly degrades, the CC agent must promptly react to it and drain the bottleneck queue (if any) as soon as possible. The last is crucial because there exists a large amount of urgent bandwidth degradation on the Internet (see our measurements in §2.2).

With the above principles in mind, we design a new CC algorithm, which we call *Pudica*[1], for large-scale cloud gaming services. Under the constraint of consistent low *frame* delay, Pudica achieves high bandwidth utilization and cross-flow fairness (if applicable) in both single-flow cases and scenarios where it contends with other homogeneous flows. Pudica has been successfully deployed on a commercial cloud gaming platform (Tencent START [9]), serving millions of players.

Unlike existing CC frameworks [11, 12, 18], which intentionally trigger queue buildups to probe the link condition, we estimate the link's bandwidth utilization ratio (BUR) rather than the bandwidth *per se*. This signal has been exploited in [22, 23] with explicit notifications, but in the context of end-to-end CC design, we propose a new probing method with

[1] Pudica is a plant known for its sensitivity and rapid movement [21].

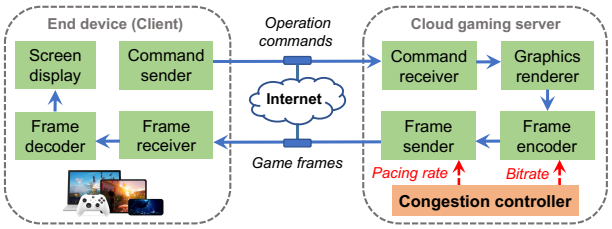


Figure 1: A typical cloud gaming system. The congestion controller adjusts both frame bitrates and packet pacing rates before transmitting game frames from the gaming server to the end device.

adaptive pacing control and complementary probe packets to estimate the BUR based on end-side feedback.

Based on the BUR estimation, Pudica multiplicatively increases the bitrate to exponentially ramp up the link utilization *while* ensuring that it remains below 100%. Then, Pudica can sustain the state without heavy queuing, and at the same time, quickly converge toward cross-flow fairness. This is achieved by applying additive increase (AI) and multiplicative decrease (MD) *simultaneously* and *adaptively*. In addition to smoothed BUR estimations, *short-term* BUR signals are also utilized in Pudica to make prompt reactions to any overshoots or urgent link degradation. The proposed mechanisms, including temporary bitrate fallback and active queue draining, facilitate the minimization of queuing delay.

We implemented Pudica and several baselines, including Copa [11], Salsify[2] [13], and SQP [20], on Tencent START cloud gaming production [9] for A/B tests. The final evaluation involved real-world wired and wireless networks, and more than 57,000 gaming sessions. In comparison to the baselines, our experiments confirm Pudica's advantages: **i)** It reduces the average frame delay, the 95%-tailed frame delay, and the 99%-tailed frame delay by 3.1×, 5.1×, and 4.7×, respectively. **ii)** It reduces the percentages of frames whose transmission to the end device exceeds 100 *ms* and 200 *ms* by 6.2× and 14.4×, respectively. And **iii)** it increases the frame bitrate by 12.1%, better utilizing networks' capacity.

Furthermore, we used a large-scale, in-the-wild network testing platform [24] to assess Pudica's efficiency and fairness. It shows that Pudica offers a superior balance between link utilization, cross-flow fairness, and convergence speed. Additionally, we conducted comprehensive microbenchmarks to justify the gains offered by Pudica.

2 Background and Related Work

In this section, we introduce the control variables, network characteristics, and control objectives of cloud gaming systems, which necessitates a new CC algorithm. We also review related CC algorithms and discuss their limitations.

2.1 Congestion Control for Cloud Gaming

Unlike the traditional transport-layer CC, the cloud gaming CC agent operates on the application layer, and controls two factors on the fly, namely the *frame bitrate* and *packet sending pace* (see Fig. 1). The bitrate is a parameter to the image compression process that encodes the stream of game frames before sending them to the end device. A lower bitrate reduces the bandwidth consumption but also sacrifices displayed quality. The game frames after compression are chopped into network packets, and the sending pace controls how hastily the packets are sent to the network. Typically, the sending interval of packets within a frame is the same. Thus, we measure the pace based on the sending behavior of the last packet for each frame. We quantify the pace using a *pace multiplier* ρ, a scalar indicates that the last packet is sent to the network within the time duration of L/ρ, where L is the interval of frame sending (16.67 *ms* for the frame rate of 60). For example, a pace with a multiplier of two means that all packets of a frame are sent over within 8.34 *ms* after encoding. *Given* a pace multiplier, the exact packet sending interval (a.k.a., packet pacing rate) depends on the number of packets within a frame, which is in turn up to the frame bitrate, or more precisely, the frame size.

2.2 Network Measurement in Real Gaming Systems

We present two observations from real cloud gaming systems, which greatly influence the rationale behind our CC design.

Short base RTTs. Edge computing has become the de facto infrastructure for cloud gaming services in industry [2, 25], which dramatically shortens the feedback loop of CC agents. We conducted measurements of the base RTT (a.k.a, minimal RTT) for over one million real cloud gaming sessions. As depicted in Fig. 3, the base RTT for 50% and 90% gaming sessions is below 10 *ms* and 20 *ms*, respectively, for both Ethernet and WiFi networks. Such *unique* network characteristics play an important role in our CC design. It not only helps us narrow down the scope of issues to be addressed but also opens up new possibilities for achieving a more suitable CC solution for cloud gaming systems.

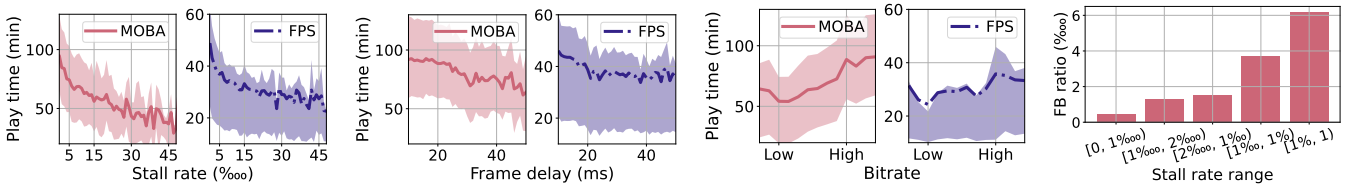
Frequent urgent bandwidth decreases. The available bandwidth frequently encounters urgent, large reductions over the Internet, e.g., due to channel degradation. We carried out the consistent bandwidth measurement by mildly flooding-based probing methods. The bandwidth was estimated and recorded with a granularity of 100 *ms* interval. As shown in Fig. 4, approximately 5.6% of Ethernet users and 35.5% of WiFi users encounter at least five occurrences of dramatic bandwidth reduction (i.e. reduction by more than 50% within 100 *ms*) per minute, which poses a non-trivial challenge to achieving consistent low latency in real-world networks.

2.3 Control Goals through Real-World Case Studies

In this section, we crystalize our congestion control goals through real-world case studies of user engagement on Tencent START cloud gaming app [9].

We conducted case studies on two popular games, a multi-player online battle arena (MOBA) game and a first-person

[2] In this paper, *Salsify* refers to its frame size control part solely.



(a) Stall rate v.s. Playing time (b) Avg. frame delay v.s. Playing time (c) Avg. bitrate v.s. Playing time (d) Stall rate v.s. Feedback ratio

Figure 2: A case study on cloud gaming QoE analysis: the effects of stall rate (i.e., the frame ratio with delay >100ms), average frame delay, and average bitrate on playing time (as a proxy of QoE). The dataset involves two hot games (i.e., MOBA and FPS) and one million gaming sessions from a production cloud gaming service. Transmission-unrelated factors, such as decoding delay, are excluded from these metrics.

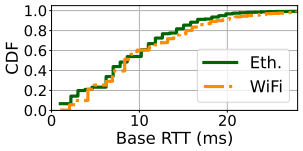


Figure 3: Base RTTs in edge-based cloud gaming services.

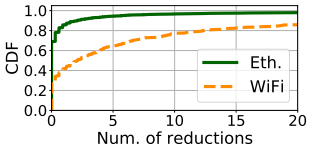


Figure 4: Num. of bandwidth reductions ($\geq 50\%$) per minute.

shooting (FPS) game. We collected logs of more than a million gaming sessions to study how network performance affects user engagement in gaming. Following [26], we use the user playing time (i.e., the length of a gaming session) to measure user engagement. We consider three metrics for network performance: the average delay of game frames, the average bitrate, and the stall rate. The last measures how often a player experiences a *stall*, defined as the percentage of frames whose delivery to the user device is delayed by more than 100 *ms*.

The results of our studies are plotted in Fig. 2. The stall rate has a significant impact on user engagement (Fig. 2a): as the stall rate increases linearly, the playing time rapidly decreases in a superlinear manner. Average frame delay also affects gaming engagement (Fig. 2b). But when the average delay is sufficiently small (e.g., <40 *ms*), the negative impact caused by an increased average delay is not as significant as the stall rate, which reflects the long-tailed portion on the distribution of frame delays (as opposed to the average delay). Not surprisingly, the bitrate is also crucial to the user engagement, since it determines the image display quality (Fig. 2c). Lastly, Fig. 2d illustrates the percentage of sessions that received user complaints with respect to different ranges of stall rates. When the stall rate is in $[1‰, 1\%)$, the percentage of user-indicated unappealing gaming sessions is $10\times$ more than the percentage when the stall rate is within $[0, 1‰)$.

Our CC goals. To reduce the frame delay and stall rate, we aim to achieve ultra-low (or *nearly zero*) queuing delay at the bottleneck. To achieve the highest possible bitrate, we seek to maximize bandwidth utilization; however, it should be pursued only *after* the achievement of low frame delays. A high bitrate with a high frame delay or stall rate is meaningless for cloud gaming. We also value fairness among homogeneous flows. It is noteworthy that, we specifically focus on achieving *frame-level* zero queuing, which directly impacts the QoE. This means that the queue induced by a frame should be drained by the time the next frame arrives at the bottleneck, rather than aiming for no queuing for every packet, although these two levels of queuing are correlated.

2.4 Existing CC Solutions and Their Limitations

Many CC algorithms have been proposed with the aim of reducing end-to-end delay, but they cannot simultaneously fulfill the three specific requirements we mentioned in §1.

Delay-bounding CC. In previous studies, various *delay-bounding* CC algorithms [11, 18, 19, 27–34] has been proposed. Most of these algorithms, such as BBR [18] and PCC Vivace [19], *periodically* increase packet sending rate beyond the estimated bandwidth in order to dynamically probe network bandwidth. This strategy, however, leads to frequent queue buildup, and thus may cause the delay to oscillate persistently even when the bandwidth stays stable. Other algorithms (e.g., Vegas [27], Fast [28], and Copa [11]) use packet delay as the congestion signal to avoid bufferbloat. When the bottleneck queue is nearly empty, these methods tend to *recklessly* raise the sending rate (even with an increasing step size) until the queue is constructed, since these delay-based methods can only accurately assess link utilization when there is sufficient queuing in the network. Thus, their CC frameworks are fundamentally unable to maintain zero queuing.

RTC-oriented CC. Recent years have also witnessed the development of CC algorithms specifically for real-time communication (RTC) applications [12–17,35]. GCC [12] reduces the bitrate only when the delay variation, rather than the delay *per se*, becomes high. So, GCC is unable to empty the queue promptly when an urgent congestion occurs. Similar to packet delay-based methods, the network estimator in GCC works only when there exists enough queuing in the network. As a consequence, GCC has to often introduce queue buildups by itself (as illustrated in Fig. 16, Appendix).

Salsify [13] (built upon Sprout [29]) adjusts frame sizes based on the estimation of packet inter-arrival time. Using samples in a short time scale for estimation, Salsify is sensible to network changes and can respond to congestion quickly. However, in real networks, the inter-arrival time at the packet level suffers from significant fluctuations, which raise the chances of estimation mistakes. Additionally, Salsify, which is designed based on the assumption of independent queues, is unable to detect other competing flows unless they intersect. Furthermore, the aggressiveness of Salsify magnifies the im-

pact of estimation errors stemming from the aforementioned issues, ultimately leading to suboptimal performance.

SQP [20] utilizes frame-based packet trains to estimate available bandwidth and adjusts the frame bitrate with AIMD-style updates. This approach may allow SQP to achieve high link utilization and avoid heavy queuing in an isolated environment. However, when multiple flows coexist, SQP faces challenges in accurately estimating the available bandwidth, especially when the flows are staggered. This may result in overshoots and queue buildups. Furthermore, SQP tries to achieve competitive bandwidth shares when competing with queue-building flows and may incorrectly identify a network degradation as the competition of buffer-filling flows (as explained in Fig. 12). Consequently, SQP fails to promptly drain the bottleneck queue in such situations.

3 Challenges and Rationale

Before diving into Pudica's algorithmic details (in §4), we describe the challenges that Pudica aims to address for clouding gaming applications and the rationale that leads to its design.

3.1 Network Probing and Estimation

Existing bandwidth estimation methods are unable to sustain a nearly empty bottleneck queue at the frame level. In Pudica, we opt to estimate the BUR (i.e., bandwidth utilization ratio) rather than the bandwidth itself, and to achieve this, we design a new network probing method.

Limitations of traditional methods. Many CC algorithms aim to estimate the available bandwidth or capacity between two ends of a connection [18–20]. Here, to eliminate ambiguity, we refer to [36] to define the term "available bandwidth" as the maximum rate that the path can provide to a flow, without reducing the rate of the rest of the traffic. Similarly, "available capacity" is defined as the amount of data that can be inserted into a network path at a certain time, so that the transit delay of these packets remains within a specified maximum permissible delay (for further details, refer to [36]).

For estimating bandwidth or capacity, most of the existing frameworks require the intentional and periodic introduction of heavy queuing at the bottleneck to assess whether the link is fully utilized. This network probing method is not at our disposal, because we aim for a *consistently* empty queue. Recently, a few low-latency CC methods try to probe the link condition without resorting to heavy queuing [13, 20]. However, these designs often begin with a potential assumption that the sender runs in a single-flow environment, without giving due consideration to the collective dynamics of multiple flows. This oversight may result in an *overestimation* of available network resources when multiple streams coexist.

In fact, even with exact knowledge of the remaining bandwidth or capacity, we cannot guarantee a rate increase without surpassing the available resources. This is due to the fact that, as an end-to-end CC agent, we lack information regarding the number of flows utilizing the identical bottleneck link, and consequently, an appropriate allocation for aggregated resources remains unattainable. This limitation is inherent to the signal of available bandwidth/capacity.

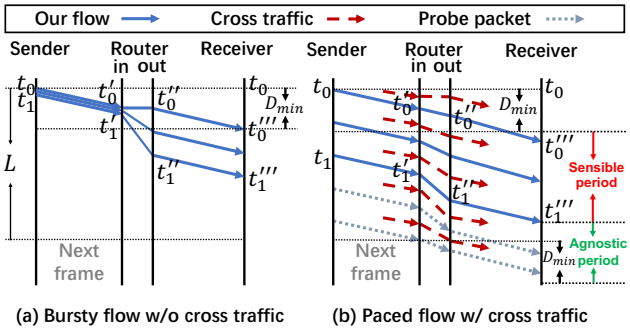


Figure 5: An illustration of our BUR probing and estimation method with the router acting as the bottleneck.

What to estimate? As will be mentioned repeatedly, it is the requirement of nearly zero queuing delays for game frames that profoundly shapes Pudica's design in many aspects. As the prerequisite for control actions, not only should the *probing process* keep no queuing, but the chosen signal after probing must also enable the rate controller to avoid queue buildups during the *efficiency-convergence process*, particularly when multiple streams coexist.

Inspired by VCP [23], Pudica probes the BUR as a feedback signal for CC, rather than the bandwidth *per se*. The BUR, a scalar, is defined as the ratio of current bandwidth usage to the link capacity, which provides an indicator of the precise level of link utilization. It not only indicates whether congestion is occurring (congested when BUR >1), but also distinguishes between different utilization regions. As proved by [23], by leveraging the BUR information, we can achieve efficiency and fairness under the constraint of consistently low queuing, regardless of the number of concurrent flows or link capacity.

How to probe? Our goal is to estimate BUR by relying solely on end-to-end feedback. Basically, a queuing delay greater than zero serves as a reliable indicator of the full-load duration at the bottleneck. Since we aim for nearly zero queuing at the frame level, for probing BUR, packet-level queuing delay must be introduced judiciously.

To illustrate this, we present an example in Fig. 5, where the router acts as the bottleneck. In the absence of cross traffic (refer to Fig. 5a), all packets of a frame are transmitted in a bursty manner by the sender between time t_0 and t_1, and they reach the router between time t_0' and t_1'. Due to the limitation of network bandwidth, the queuing time of the last packet can be denoted as $(t_1'' - t_0'')$, during which the link utilization is 100%. In this case, the bandwidth utilization can be calculated as $(t_1'' - t_0'')/L = (t_1''' - t_0''')/L = (t_1''' - t_0 - D_{min})/L$, where D_{min} is the minimum one-way delay from the sender to the receiver, and L is the interval of frame sending.

However, when cross traffic exists, traffic arriving at the router after t_1' cannot be detected by the receiver. To address

this issue, we intentionally slow down the frame transmission slightly, allowing a portion of the cross traffic to arrive between t_0' and t_1', as depicted in Fig. 5b. This enables the receiver to observe the presence of such traffic. Although extending the time period between t_0 and t_1 allows for more cross traffic to arrive in this period, once the instantaneous utilization drops below 100%, the packet-level queuing delay becomes zero, and then we lose the ability to determine the level of bandwidth utilization accurately. Therefore, our goal is to send each frame as slowly as possible while maintaining a packet-level queuing delay greater than zero (at least for the last packet of each frame). To achieve this, we dynamically and adaptively adjust the pace multiplier. As a result, we can still *basically* estimate the BUR as $(t_1''' - t_0 - D_{min})/L$, which remains consistent with the no-cross-traffic cases.

Nonetheless, we are actually uncertain about the cross traffic arriving after t_1'. To complement this, we employ a small number of extra probe packets beyond frame data to measure the queuing time and robustify the BUR estimation. Even though a competing flow that consistently sends at a low rate and causes zero queuing delay may still go undetected, this complementary probing mechanism can greatly reduce the occurrence of BUR underestimation (see Fig. 7). Furthermore, the lack of awareness of such low-rate flows diminishes as the link utilization approaches 100%.

3.2 Bitrate Adaptation based on BUR Estimations

Basically, Pudica utilizes BUR estimation to dynamically adjust the frame bitrate. When BUR is low, we multiplicatively increase the rate for efficiency; when BUR is high (but less than one), we operate AI and MD for fairness; when BUR exceeds one, we reduce the bitrate to quickly drain the bottleneck queue. Within this framework, Pudica introduces several meticulously designed methods to minimize queuing delay.

Coping with queuing caused by possible estimation errors. Despite significant improvements in the accuracy of BUR estimation, we acknowledge that there is still a gap between the predicted value and the actual value. Therefore, we recognize the need for adaptive steps in bitrate upgrading, i.e., the step should be smaller (applied more cautiously) as the BUR increases. Furthermore, we design a temporary *fallback mechanism* to minimize the adverse effects of potential estimation errors. Specifically, Pudica timely and moderately reduces the bitrate whenever it detects a significant delay in any individual frame. To enhance resilience against jitters, this reduction is limited to the subsequent frame only, ensuring its transient nature. This fallback strategy is particularly useful for cloud gaming due to the smaller RTT (§2.2). With shorter feedback loops, we can effectively manage the effects of minor overshoots through prompt corrections.

Balancing fairness and queuing. A common way to promote fairness in CC design is through additive increase and multiplicative decrease (referred to as *AI/MD*) of the bitrate. It additively increases the bitrate until *a high queuing delay* (or overflow-induced packet loss) is detected, and then it switches to multiplicatively decrease the bitrate.

Again, for cloud gaming this strategy falls short, because we wish to minimize queue buildups. We therefore propose a different strategy: While we additively increase the bitrate, we do not wait until the queue is built up to begin the MD operation. Instead, we apply MD earlier, concurrently with each AI action, to proactively free up a portion of bandwidth before the bitrate causes any queue buildup. In this way, MD-based de-allocation occurs simultaneously with AI-based allocation, executed more frequently as well. Thereby, fairness across competing flows can be reached in a faster way while avoiding excessive queuing. To distinguish it from the traditional AI/MD, this strategy is referred to as *AI-MD*.

Yet, a naïve implementation of AI-MD with fixed MD amplitudes and AI steps may hinder link utilization. For instance, in high-bandwidth scenarios, the de-allocation amount from the MD operation may exceed AI's allocation, leading to a bitrate decrease even with a low utilization of overall bandwidth. Our remedy to this limitation is by applying bitrate AI in an adaptive manner. The step of AI gradually increases from an initial small value until the link is close to being fully utilized. This approach ensures high utilization and low queuing during the convergence process toward fairness.

Handling bandwidth fluctuations. As a commercially deployed system, Pudica must be able to cope with frequent network fluctuations, especially urgent bandwidth degradation (which has been shown in §2.2). When the link degrades, a belated bitrate reduction would result in severe queuing; conversely, a timely response to this degradation would significantly reduce the tailed delay and stall rate.

To achieve a timely response to bandwidth reduction, Pudica relies on not only the smoothed BUR estimation but also the BUR feedback from the latest frames, to judge if congestion has occurred. When the BURs of three consecutive frames both exceed one, Pudica will reduce the bitrate to a value below the packet receiving rate. The underlying insight is that when the link utilization exceeds 100%, the short-term receiving rate can serve as a dependable indicator of the achievable maximum throughput. At this time, if every flow in the network can set its bitrate below the receiving rate, the queue at the bottleneck can be drained quickly.

4 Pudica Design

We propose a new probing framework that achieves high-accuracy BUR estimation. By leveraging both smoothed (from long-term history) and short-term BUR estimations, we design multiple bitrate control strategies to enhance efficiency and fairness while minimizing the queuing delay. We provide a brief overview of our control policy:

- MI for efficiency when smoothed BUR (or $\widetilde{R}$) is low;
- AI-MD for fairness when $\widetilde{R}$ is high but not more than one;

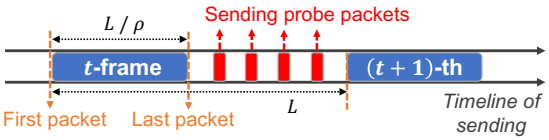


Figure 6: Pudica sends several payload-free probe packets beyond frame data during the intermission of frame sending to detect the existing competing flows during the agnostic period.

- MD for queue draining when short-term BUR exceeds one.

4.1 BUR Probing and Estimation

Pudica utilizes all packets within each game frame as a short burst (namely a packet train) to probe the link utilization (i.e. BUR). To improve the accuracy of BUR estimation, Pudica introduces an innovative adaptive pacing control algorithm, accompanied by a complementary approach that uses a small number of probe packets to detect competing flows.

As illustrated in Fig. 5 and §3.1, during the period of $[t_0 + D_{\min}, t_1''']$, the queuing delay keeps non-zero, and in turn, the bottleneck link is fully utilized. We refer to this period as the *sensible period*. On the contrary, $[t_1''', t_0 + L + D_{\min}]$ is the *agnostic period*, during which there is no feedback from the network. We adaptively adjust the pace multiplier such that it introduces just a slight amount of packet-level queuing for effective probing while maximizing the duration of the sensible period. In the agnostic period, we insert several additional probe packets without payload to measure the potential queuing time during this period.

BUR estimation. We estimate the BUR of a frame (denoted by R) by calculating the difference between the transmission delay of this frame and the physically minimal delay:

$$R = \frac{D - D_{min}}{L}. \tag{1}$$

Here, D is the one-way frame delay (in *sec*), i.e., the duration from the sent time of the first packet to the received time of the last packet for this frame. D_{min} (in *sec*) is estimated as the smallest packet one-way delay over the period of ten seconds. L is the interval of frame sending (in *sec*). There is no clock synchronization issue here, as the same errors for both D and D_{min} cancel each other out.

Adaptive pacing control. To extend the sensible period while sustaining a slight burstiness, Pudica dynamically sets the pacing multiplier ρ as:

$$\rho = \frac{\gamma_\rho}{\min(\, R,\ 1\,)}, \tag{2}$$

where γ_p is a constant greater than one, empirically assigned a value of 1.25. Using this method, the sending duration between the first packet and the last packet in a frame (i.e., $t_1 - t_0$ in Fig. 5) is slightly shorter than the queuing delay (i.e., $D - D_{min}$), which introduces just a slight packet-level queuing in the bottleneck buffer. By bounding the denominator up to one, all packets of each frame are sent within the frame interval, to avoid superfluous waiting time at the sender.

Competing flow detection with probe packets. To detect potential competing flows after the sensible period, Pudica sends N_{packet} (set as four) payload-free probe packets during the agnostic period, as illustrated in Fig. 6. These probe packets are sent out evenly, i.e., the sending interval T_{packet} (in *sec*) between probe packets is set to be $\frac{(1-1/\rho)\times L}{N_{packet}+1}$.

Then, Pudica uses the feedback of probe packets to robustify the BUR estimation within this frame interval. Let D_i denote the one-way delay (in *sec*) of the i-th probe packet ($i = 1, \ldots, N_{packet}$). If the competing flows that occupy the network resource during the agnostic period block the probe packets, the queuing delay T_i, i.e., $D_i - D_{min}$, for the i-th probe packet will be non-zero and should be taken into account.

However, the queuing delay also occurs when probe packets are blocked by the game frame data sent by Pudica itself. Such queuing delay needs to be removed as it has already been counted in the sensible period. If the frame data delays the i-th probe packet, the actual queuing delay caused by competing flows is the duration (in *sec*) between the arrival of the frame's last packet to the arrival of the i-th probe packet. Such delay is smaller than $D_i - D_{min}$ in this case, defined as H_i. So far, a reasonable queuing delay can be expressed as:

$$T_i = \min(\, D_i - D_{min},\ H_i). \tag{3}$$

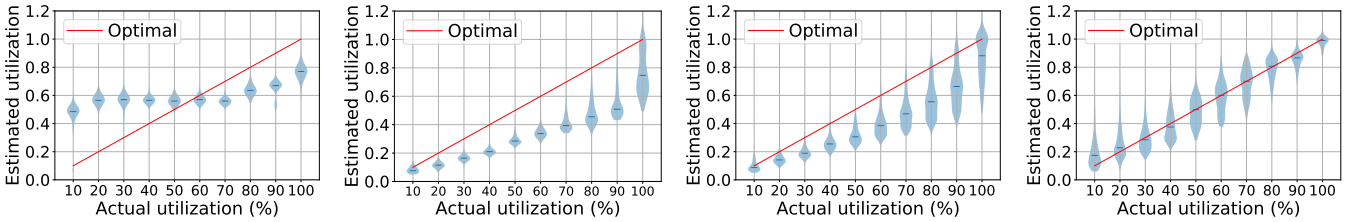
Furthermore, if the queuing delay of the i-th probe packet (i.e., T_i) is larger than the packet sending interval (i.e., T_{packet}), it will also affect the queuing delay of the following probe packet (i.e., T_{i+1}). To avoid repetitively counting the queuing delay that has been already experienced by the previous probe packets, T_i should be bounded by T_{packet} in computation. The rectified queuing delay is finally defined as:

$$T_i = \min(\, D_i - D_{min},\ H_i,\ T_{packet}\ ). \tag{4}$$

With the above analysis, Pudica can leverage the cross-traffic-induced queuing delay of probe packets to obtain a more accurate BUR estimation, i.e., $R^{corrected}$:

$$R^{corrected} = R + \Sigma_{i=1}^{N_{packet}} \frac{T_i}{L}. \tag{5}$$

Validation of usefulness. We validated the effectiveness of our BUR estimation method through a large-scale Internet test. The details of the validation methodology are explained in Appendix A. Fig. 7 depicts the BUR estimation performance of different probing methods. Our probing approach (see Fig. 7d), which incorporates both adaptive paces and extra probe packets, outperforms other alternatives in accurately estimating BUR. Furthermore, as the actual BUR increases, the estimation distributions exhibit reduced variance, indicating enhanced reliability. Fixed 2× pacing (used in SQP [20], see Fig. 7a) and burst sending (see Fig. 7b) are prone to unawareness of competing flows and consequently lead to BUR underestimation, particularly in high-utilization scenarios. The use of probe packets not only corrects unawareness-induced BUR underestimation, but also guides the pacer toward smaller multipliers. This, in turn, extends the sensible period and enhances the precision of BUR estimation.



(a) Fixed paces (2× pacing). (b) Fixed paces (burst sending). (c) Adaptive paces (w/o probe packet). (d) Pudica (complete).

Figure 7: Validation of various BUR estimation methods. Each method involves more than 2,000 groups of test data in real networks. Our method, which combines adaptive paces and probe packets, significantly improves the accuracy of BUR estimation over the alternatives.

4.2 Bitrate Adaptation over the Smoothed BUR

In this section, we introduce how Pudica leverages a *smoothed BUR estimation* as an indicator to dynamically adjust its bitrate for optimizing link utilization and cross-flow fairness while sustaining near-zero bottleneck queuing.

Smoothed BUR computation. To obtain the smoothed BUR (a scalar, denoted by $\widetilde{R}$), we utilize historical BUR estimation samples from N_{sample} frames over the past T_{wd} *ms*. T_{wd} is set to 200, as most flows have base RTTs less than 200 *ms*. This choice strikes a balance between the likelihood of awareness for competing flows and the freshness of the data [23]. We employ the weighted average method to calculate $\widetilde{R}$ as:

$$\widetilde{R} = \sum_{k=1}^{N_{sample}} \omega_k \times R_k \times \frac{B}{B_k}, \tag{6}$$

with weight coefficients ω_k subject to $\sum_{k=1}^{N_{sample}} \omega_k = 1$ (detailed in Appendix B). Here, R_k and B_k represent the BUR feedback and bitrate of the k-th frame, respectively. B is the current bitrate for frame encoding, which is dynamically updated on the fly. To address possible differences between B and B_k (both in units of *Mbps*), we use the ratio of the two values to rectify any deviation and ensure that the smoothed BUR estimation aligns with the current state.

Multiplicative increase (MI) for efficiency. Similar to VCP [23], Pudica decouples efficiency control and fairness control. When the smoothed BUR estimation (i.e., $\widetilde{R}$) is low, Pudica prioritizes efficiency over fairness, by adjusting bitrate with MI operations. Then, when $\widetilde{R}$ is high but less than one, the goal of Pudica is to converge toward fairness while maintaining nearly full utilization (will be introduced later).

Pudica employs a threshold, denoted by α, to determine whether to activate the efficiency control or fairness control phase. The value of α is empirically determined to be 0.85. When $\widetilde{R} \leq \alpha$, the bitrate is multiplied to rapidly converge toward efficiency, which can be expressed as:

$$B^{new} = B \times (1+\xi), \tag{7}$$

$$\xi = \gamma_{MI} \times \frac{(\alpha+1)/2 - \widetilde{R}}{\widetilde{R}}, \tag{8}$$

where γ_{MI} is a discounting coefficient, empirically set as 0.3. Note that the next adjustment is postponed until the feedback regarding the current adjustment is received, to prevent over-aggressive rate increases and mitigate bitrate oscillation.

Simultaneous AI and MD for fairness. When $\widetilde{R} > \alpha$, we do not apply the traditional AI technique which linearly increases the bitrate until the queue is built up. Instead, Pudica operates AI and MD bitrate adjustment *simultaneously* (namely AI-MD), by replacing the *fixed* AI step (used in classical AI/MD frameworks) with an *adaptive* step $\mathcal{A}$:

$$B^{new} = B + \mathcal{A}, \tag{9}$$

$$\mathcal{A} = I - \gamma_{MD} \times B, \tag{10}$$

where I is the linear part of the adjustment step (in *Mbps*), and γ_{MD} is the MD parameter of the multiplicative part (set to be 0.05, empirically). Simultaneously performing AI and MD in one step significantly augments the prospects for bandwidth reallocation, thereby accelerating the convergence to fairness. In practice, we enforce both upper and lower bounds on $\mathcal{A}$ to prevent excessive bitrate oscillation during this stage.

Given an MD ratio (i.e., γ_{MD}), a fixed linear increase (i.e., I) may be too small to fully utilize bandwidth for a high link rate, or conversely, it may cause rate oscillation and delay spikes for low link bandwidth. To solve this dilemma, we design an AI step adaptation mechanism to match the link rate. Specifically, the value of I is determined by the equation $(B_{max} + \frac{2^{\tau}}{log(B)}) \times (\frac{\gamma_{MD}}{2})$. Here, B_{max} is the maximum bitrate limited by the application (set as 50 *Mbps* in our implementation). τ denotes the accumulated number of received frames after initialization (discussed later). This equation is designed to be inversely related to the current bitrate B, to enhance fairness convergence by increasing the value of I more slowly for flows with higher bitrates.

Since the value of I keeps increasing, Pudica resets τ to be zero when $\widetilde{R} > 1$, in order to prevent a heavy queuing caused by an overlarge I. This initialization method relies on BUR estimations, which may not be flawless. Thus, Pudica further implements a *time-driven* initialization mechanism to avoid severe unfairness. To be specific, τ is initialized every five seconds. This periodical and highly synchronized τ initialization enhances the robustness of our AI-MD scheme. By applying AI-MD with an adaptive AI step, Pudica can achieve faster convergence to fairness while avoiding frequent queue buildups (further illustrated in Appendix C).

4.3 Bitrate Adaptation over the Short-Term BUR

Using a smoothed BUR estimation that combines multiple BUR samples from the past can provide more robustness to

network noises, compared to relying solely on a single-frame BUR. However, due to its lagging nature, the smoothed BUR may not be able to respond promptly to improper control decisions or sudden network changes. To address this issue and achieve more timely adaptations, we propose two additional bitrate adjustment approaches that utilize the short-term BUR, particularly the BUR of recently received frames. By combining the responsiveness of short-term BUR with the robustness of smoothed BUR, Pudica strikes a superior balance between timeliness and stability in bitrate control.

Temporary bitrate fallback. Upon detecting a potential overshoot or congestion *for the first time*, Pudica instantly takes action by temporarily reducing the bitrate. Specifically, when the BUR feedback of any single frame exceeds one, Pudica will reduce the bitrate by ζ (set to be 15%, empirically) for the subsequent frame to be encoded. This bitrate reduction is temporary, meaning that after adjusting the bitrate for the next to-be-encoded frame, the encoder will revert back to the previous bitrate setting.

It is important to highlight that in cases of sudden congestion, the sender may experience delays in receiving feedback regarding network degradation, which can lead to delayed bitrate fallback. To address this issue, we monitor the frames that have been sent to the network but have not yet been acknowledged. We introduce the concept of *next delay*, which represents the elapsed time *from* the moment the next to-be-received frame (i.e., the earliest sent frame among the in-flight frames) was sent *until* the current time. When the next delay is significant, Pudica performs the bitrate fallback as well. The introduction of the next delay signal enables Pudica to respond to congestion more swiftly and timely.

This fallback scheme could reduce queue buildups caused by potential overshoots since we acknowledge the inherent difficulty in achieving flawless BUR predictions. The transient nature of the bitrate reduction helps minimize the impact of incorrect fallback due to jitter-induced delay variations.

Active queue draining. When the BUR feedback of three recently received frames *both* exceeds one, Pudica enters the queue-draining phase with active undershooting. At this time, the bitrate is set to be:

$$B^{new} = \alpha \times receiving_rate - draining_rate, \quad (11)$$

where *receiving_rate* is the average data receiving rate from the onset of congestion to the present moment. It serves as a reliable estimation of the maximum throughput achievable during this period. Additionally, Pudica employs a calculation to determine the required additional throughput rates (referred to as *draining_rate*) needed to clear the self-induced queuing at the bottleneck within the next 200 *ms*. Here, the volume of self-induced queued data is quantified by measuring the number of in-flight packets. By carrying out an active queue draining over the Eq. 11, Pudica is able to efficiently and swiftly empty the existing queue at the bottleneck.

Algo.	Avg. delay	95%/99% -tile delay	Stall rate >100/200ms	Avg. bitrate
Copa	26.9ms	47.2/79.9ms	1.68%/0.77%	42.9Mbps
Salsify	21.5ms	30.0/67.0ms	0.48%/0.09%	42.4Mbps
SQP	41.8ms	101.7/147.8ms	1.27%/1.03%	43.7Mbps
Pudica	**19.5ms**	**25.0/30.6ms**	**0.07%/0.004%**	**47.5Mbps**

Table 1: Overall system-level performance at scale (**Ethernet**).

Algo.	Avg. delay	95%/99% -tile delay	Stall rate >100/200ms	Avg. bitrate
Copa	76.5ms	305.2/672.8ms	7.7%/3.9%	22.6Mbps
Salsify	195.5ms	631.7/982.5ms	22.9%/13.5%	**32.6Mbps**
SQP	302.2ms	815.2/1218.5ms	10.3%/8.7%	23.2Mbps
Pudica	**33.7ms**	**74.9/175.6ms**	**2.5%/0.72%**	31.2Mbps

Table 2: Overall system-level performance at scale (**WiFi**).

On the other hand, when the congestion vanishes, gradually increasing the bitrate from a low value can lead to suboptimal utilization due to the delayed nature of smoothed BUR estimation. Therefore, when the BUR of a recently arrived frame is less than one, Pudica recalculates the current *receiving_rate* and directly restores the bitrate to match it. Considering the delay sensitivity of the queue draining mechanism, this one-step recovery scheme improves resilience to network jitters and consequently enhances link utilization, by promptly recovering when the delay returns to normal.

5 Evaluation

We deployed Pudica and the state-of-the-art CC algorithms on a commercial cloud gaming platform, and conducted extensive evaluations in real-world wired and wireless networks. We also evaluated the ability of convergence to efficiency and fairness through a large-scale, in-the-wild network testing platform. We demonstrated the reason for Pudica's gain by miscellaneous emulation experiments. Finally, we proved the effects of several selected algorithm components on end-to-end performances by microbenchmarks. All timeline-based line charts in this section are presented at a frame-wise level.

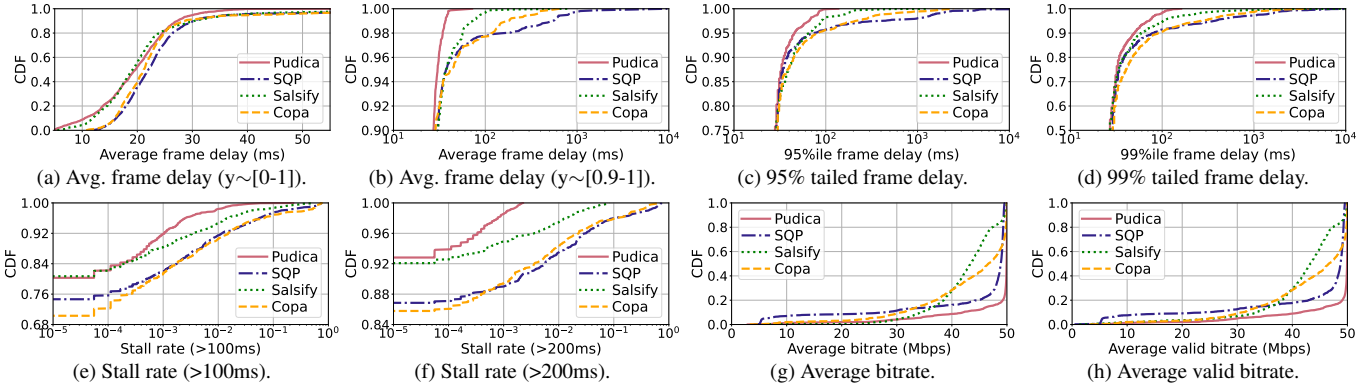
5.1 Methodology for Large-Scale Algorithm Evaluation

We deployed four CC approaches, including Pudica, Salsify [13], Copa [11], and SQP [20] on Tencent START cloud gaming platform [9] for large-scale A/B tests[3]. The START system integrates the customized network protocol stack and video codec, supporting submillisecond-class pacing, packet-level acknowledging, and frame-level size control.

Experiment setups. Our commercial system operates on fully public networks over the Internet. The evaluation finally involved more than 57,000 gaming sessions across 15 cities, two network types (Ethernet and WiFi), and three ISPs over five weeks. We randomly chose one of the CC algorithms for each session and kept the other system modules the same for a fair comparison. We set the frame rate as 60 and the maximal bitrate as 50 Mbps[4] for all algorithms. We set

[3] A preliminary emulation experiment is conducted in Appendix D.

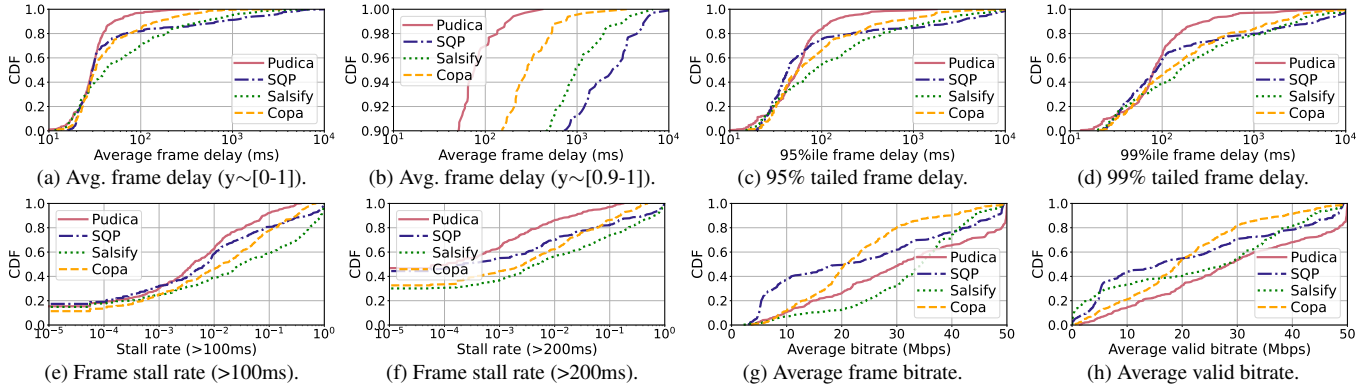
[4] In production cloud gaming services, the gaming bitrate above 50 Mbps is bandwidth-costly with low marginal benefits on player QoE.



(a) Avg. frame delay (y∼[0-1]). (b) Avg. frame delay (y∼[0.9-1]). (c) 95% tailed frame delay. (d) 99% tailed frame delay.

(e) Stall rate (>100ms). (f) Stall rate (>200ms). (g) Average bitrate. (h) Average valid bitrate.

Figure 8: Performance at scale on a commercial cloud gaming platform over the Internet (**Ethernet**). X-axis in some figures is in log scale.



(a) Avg. frame delay (y∼[0-1]). (b) Avg. frame delay (y∼[0.9-1]). (c) 95% tailed frame delay. (d) 99% tailed frame delay.

(e) Frame stall rate (>100ms). (f) Frame stall rate (>200ms). (g) Average frame bitrate. (h) Average valid bitrate.

Figure 9: Performance at scale on a commercial cloud gaming platform over the Internet (**WiFi**). X-axis in some figures is in log scale.

the delay constraint as $(L+D_{min})$ in Salsify. Following [20], Copa was implemented with a setting of $\delta = 0.1$ and without mode switching. For CWND-based methods, the frame bitrate was determined based on the ratio of the CWND size to the smoothed RTT.

Evaluation metrics. To evaluate operation interactivity, we computed several metrics for each gaming *session*, including the average, 95%ile, and 99%ile round-trip frame delay, and the stall rates for frame delays exceeding 100 *ms* and 200 *ms*. For display quality, we computed the average values of frame bitrate and valid bitrate for each session. The *valid bitrate* refers to the average bitrate of frames that have a low delay, specifically less than 50 *ms*. While a delayed frame may have a high bitrate, the information it carries could be outdated and therefore useless to the player experience.

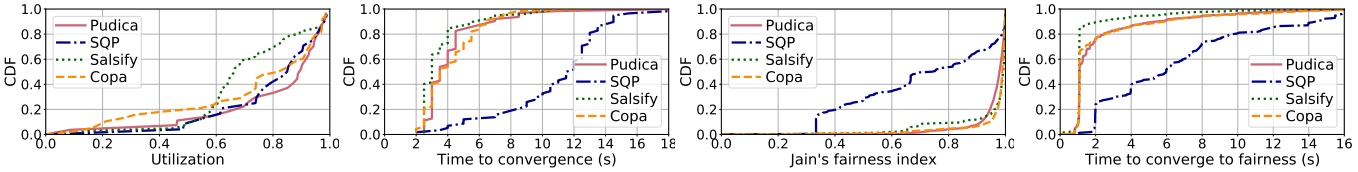
5.2 System-Level Performance at Scale

We list evaluation results by averaging across all sessions in Tab. 1 and Tab. 2, and we plot the CDF curves of the various metrics in Fig. 8 and Fig. 9. In summary: *i)* Pudica significantly reduces frame delay and stall rate, and cuts the delay tail. *ii)* Pudica achieves an equivalent or higher bitrate. *iii)* Pudica strikes a better balance between delay and bitrate.

Average and tailed frame delay. Fig. 8a–8d illustrates that compared to the baselines, Pudica reduces the frame-level average delay, 95%ile delay, and 99%ile delay by 1.5×, 2.4×, and 3.2×, respectively, over Ethernet networks. Similarly, over WiFi networks (Fig. 9a–9d), the corresponding reductions are 5.7×, 7.8×, and 5.5×, respectively. We observed that Salsify experienced lower delays in situations of highly underutilized bandwidth (especially on wired networks) due to its relatively bursty sending, as compared to methods with conservative pacing control. However, Salsify exhibited excessive sensitivity to intensive packet arrival or jitters, which are common occurrences in WiFi networks, leading to dramatic overshoots, high queuing delays, and frequent stalls.

Stall rate. Compared to the alternatives, Pudica reduces the stall rate with thresholds of 100 *ms* and 200 *ms* by 16.3× and 22.5×, respectively, over Ethernet networks (Fig. 8e–8f). On WiFi networks (Fig. 9e–9f), the reductions are 5.5×–12.1×. We noticed that many users had access to sufficiently high bandwidth, and as a result, experienced negligible stall rates regardless of the CC algorithm employed, due to the upper limit of 50 *Mbps* set in our experiments. Nonetheless, there is still a notable proportion of sessions that can derive benefits from the implementation of Pudica.

Frame bitrate. As depicted in Fig. 8g and 9g, Pudica achieves a 1.10× and 1.19× bitrate enhancement on average over the Ethernet and WiFi networks, respectively. While Salsify outperforms Pudica in average bitrate for WiFi scenarios, its average delay and stall rate is 5.8× and 9.2× higher than Pudica, respectively. From Fig. 8h and Fig. 9h, we can



(a) Avg. utilization. (b) Convergence speed (efficiency). (c) Fairness. (d) Convergence speed (fairness).

Figure 10: Convergence ability evaluation for efficiency and fairness through a large-scale, in-the-wild network testing platform [24].

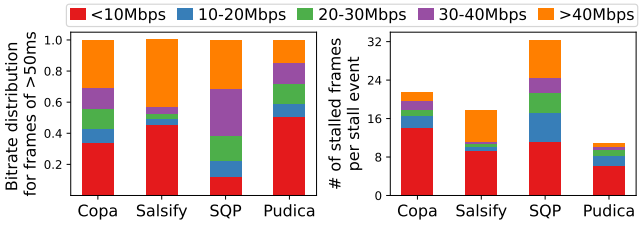


Figure 11: The proportion of different bitrates for frames with a delay greater than 50 *ms* (**Left**), and the average number of consecutively stalled frames per occurrence of stall event (**Right**).

see that Pudica achieves the highest valid bitrate over both Ethernet and WiFi networks.

Deeper analysis. To gain a better understanding of how Pudica reduces the stall rate and cuts the delay tail, we focused on frames with large delays and observed how these CC schemes respond to stalls. In Fig. 11, we present the bitrate distribution for frames with a delay exceeding 50 *ms* (*left-side*), and the number/bitrate of consecutively stalled frames per occurrence of stall event (*right-side*). Here, we set a threshold of 100 *ms* for judgment of stall events/frames. Our analysis shows that for Pudica, low-bitrate (<10 *Mbps*) frames make up more than half of frames with a delay exceeding 50 *ms* (similar results for consecutively stalled frames), indicating Pudica's superior performance in reducing CC-induced queuing. When a stall event occurs, Pudica can quickly respond to it by reducing the bitrate, resulting in a significant reduction in the average stall count to 10.9 frames.

5.3 In-the-Wild Evaluation for Efficiency and Fairness

We used a large-scale network test platform [24] (detailed in Appendix E) to evaluate the efficiency and fairness of various CC algorithms, as well as the speed at which they reached "steady state" and "fair state". A flow is deemed to be in a steady state if it does not experience a bitrate change exceeding 2% within the last two seconds. The fair state is defined as a situation wherein the mean throughput of any two flows does not diverge by more than 20% within a two-second interval, or both flows have reached steady states.

Efficiency. Efficiency and its convergence are measured by bandwidth utilization and the duration necessitated for each algorithm to attain the steady state. Each test involved measuring available bandwidth over a 90-second span, followed by the random selection and execution of one CC algorithm for 60 seconds. The average bandwidth utilization of the last 30 seconds of each test was calculated, as shown in Fig. 10a. The findings indicate that Pudica achieved an average utilization of 77.6%, while SQP, Copa, and Salsify achieved 76.8%, 70.8%, and 68.7%, respectively. Fig. 10b plots the time to reach a steady state. The majority of flows (>80%) in Pudica and Salsify converged within 6 seconds, whereas SQP required an average of 9.7 seconds to reach a steady state.

Fairness. Fairness and its convergence are measured using Jain's fairness index and the time necessitated for each algorithm to reach the fair state. In each test, we initiated one flow, followed by the second and third flows with the same algorithm after 20 and 40 seconds, respectively. Jain's fairness index was calculated using the throughput data collected during the 45-60 second interval, as plotted in Fig.10c. Pudica achieved an average fairness index of 0.95, while SQP, Copa, and Salsify achieved 0.735, 0.965, and 0.947, respectively. Fig. 10d illustrates the time to reach the fair state. We can see that, Salsify achieved the fastest fairness convergence, while SQP required the longest time to converge to a fair state.

Summary. Pudica provides high utilization and good fairness, while also achieving rapid convergence to both. Consequently, Pudica strikes a better balance between efficiency, fairness, and low latency, compared to the baselines.

5.4 Pudica Deep Dive over Emulation

We assessed Pudica's convergence capability over the Mahimahi emulation [37], elucidating the reason for improvement.

Consistent convergence to low-queuing efficiency. As depicted in Fig. 12, Pudica exhibits a faster and more stable adaptation to network variations, both in terms of bandwidth decrease and increase, compared to the baselines. When the bandwidth abruptly reduces, Pudica employs the active queue draining mechanism (§4.3) to trigger a dramatic bitrate fallback in response to congestion. This allows Pudica to quickly empty the queue, a process that typically takes only 200 *ms*. Once the queue has been emptied, Pudica recovers the bitrate by one step, ensuring a reasonable level of link utilization. This strategy also provides Pudica with robustness against network jitters, as demonstrated in Appendix F. As a consequence, Pudica achieves a superior balance between low queuing and efficiency. On the other hand, SQP exhibits slow adaptation to reduced bandwidth, taking nearly five seconds to drain the queue, resulting in high delay spikes and prolonged stall duration. Salsify and Copa are badly sensitive to both transient packet queuing and empty queues, causing choppy rate decisions and persistent delay oscillations.

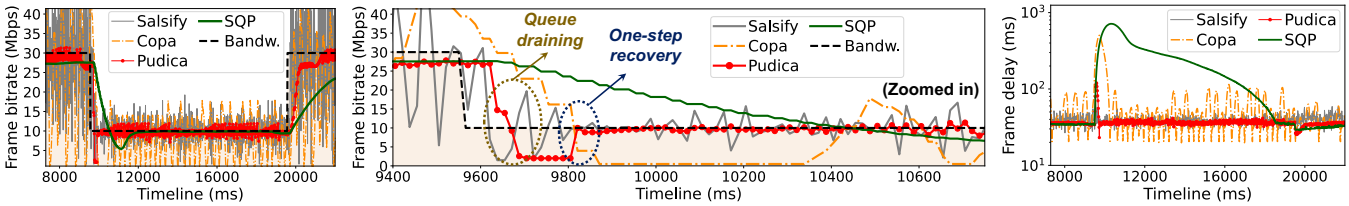
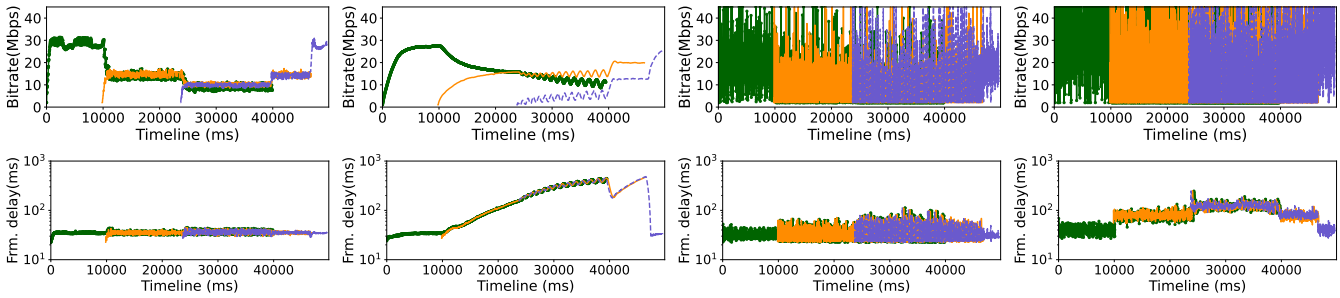


Figure 12: Comparison of convergence to low-queuing efficiency: bitrate and delay of each frame as the bandwidth alters. Pudica reconciles the low queuing delay and high bandwidth utilization by merging responsive queue draining and rapid, stable bandwidth approaching.



(a) Pudica (b) SQP (c) Copa (d) Salsify

Figure 13: Comparison of fairness convergence: bitrate and delay of each frame for multiple competing flows as they gradually enter and quit. Up to three flows share a 30 *Mbps* bottleneck. Pudica achieves rapid, stable convergence to fairness with sustaining low frame delays.

Rapid and stable convergence to fairness. As shown in Fig. 13, Pudica achieves rapid and consistent fairness convergence for both throughput and delay whenever a new stream enters or leaves the same link. Pudica also prevents the formation of significant queues in the presence of competing flows. SQP utilizes an essentially MIMD-based rate adjustment approach, leading to poor fairness, especially when a slow-start newcomer encounters multiple existing high-bitrate flows. Further, multiple concurrent SQP flows would mistakenly regard each other as the buffer-filling flow and enter the competing mode by continually raising the estimate of minimum delay, resulting in an increasing delay. While Salsify and Copa achieve decent fairness, they suffer from significant bitrate and delay oscillation when multiple flows are present.

5.5 Microbenchmark

We conducted a microbenchmark in real networks to quantify the contribution of different design elements to overall performance. We plot the key evaluation results in Fig. 14 and provide more detailed information in Appendix G.

Different BUR estimation methods. We replaced the BUR estimator in Pudica with the network estimation methods introduced in SQP [20] and Salsify [13]. SQP uses the frame as a burst to probe the available bandwidth, while Salsify utilizes packet trains to probe the available capacity. Therefore, these methods do not directly estimate the BUR indicator. That being said, we evaluated the performance of their probing methods within the Pudica framework. To accomplish this, we considered the ratio of the current bitrate to the estimated available bandwidth/capacity as the BUR estimation in the Pudica variants. As depicted in Fig. 14a, Pudica achieves lower stall rates and tailed delay, particularly in multi-flow scenarios, when compared to the variants. This confirms the advantages of our BUR estimation approach.

AI-MD v.s. AI/MD. Fig. 14a also compares the strategy of simultaneous AI and MD in Pudica (i.e., AI-MD) with the traditional AI/MD method. For AI/MD, we additively increase the bitrate when the BUR is between α and one; we multiplicatively decrease the bitrate when the BUR exceeds one. Our findings demonstrate that Pudica with AI/MD achieves comparable performance to AI-MD in single-flow scenarios. However, when three flows operate simultaneously on the same link, AI-MD exhibits the advantage of reducing stall rate and tailed delay (as well as fairness, see Appendix G).

Temporary bitrate fallback and active queue draining. Fig. 14b evaluates the contributions of two bitrate adjustment methods driven by short-term BUR signals (§4.3). Evidently, the introduction of temporary fallback and active queue draining significantly improves the stall rate and tailed delay.

Sensitivity analysis for algorithm parameters. We assessed the impacts of several parameter choices in Pudica. Tab. 6 in the Appendix shows that Pudica maintains consistent performance across various parameter configurations, indicating its lesser dependence on specific parameter settings.

6 Discussion

Granularity for periodic bursts. Pudica uses a network probing approach where all packets within a frame are treated as a burst, minimizing sender-side delay and aligning with our focus on frame-level end-to-end delay. This granularity is suitable for our context. However, periodic burst transmission at different granularities could also be effective, especially for applications beyond frame-level performance assessment.

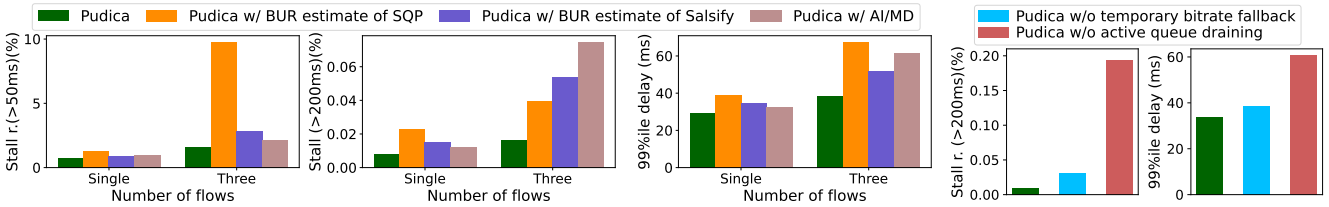


Figure 14: Key microbenchmark results: (a) Different BUR estimation; AI-MD v.s. AI/MD. (b) Responsive reaction.

Balancing feedback usefulness and cross-traffic detection. Pudica may intentionally slow the pace multiplier to boost the chance of detecting competing flows. While this packet pacing may reduce feedback usefulness by eliminating packet queuing, it shortens the agnostic period and lessens BUR underestimation due to cross flow unawareness. This is a trade-off between feedback precision and flow detection. Our method may overrate BUR compared to fully bursty transmission, impacting efficiency convergence. However, it improves delay performance, a more fitting trade-off for cloud gaming.

Handling packet losses. Since Pudica estimates available bandwidth based on the actual arrived size of frames, it would inherently respond to packet losses, triggering bitrate decreases, due to the reduced amount of arrived packets when losses occur. However, the nature of the frame-as-a-whole sending pattern in video streaming, coupled with various types of network bottlenecks, can still result in losses despite BUR being below 100% (e.g. in shallow buffer scenarios), causing additional frame delays and unnecessary bitrate downgrades. Moving forward, our research aims to explore practical strategies for mitigating the impact of packet losses through both proactive and reactive ways.

Competitiveness with buffer-filling flows. Pudica achieves a decent fairness for *homogeneous* flows. However, unlike mode-switched CC methods (e.g., Copa [11], Nimbus [33]), Pudica does not explicitly compete with buffer-filling or *inelastic* flows. Instead, Pudica concedes to them by lowering the bitrate whenever the queue becomes full. This approach is adopted because sacrificing delay for high bitrate is meaningless for cloud gaming. Actually, Pudica keeps fair competitiveness with loss-based flows under the circumstance of *shallow* bottleneck buffers (see Appendix H).

User-centric QoE optimization. Pudica primarily aims to achieve the highest possible bitrate while maintaining low latency. However, it may not necessarily result in the optimal user QoE. For instance, excessively high bitrates may only marginally enhance video quality but can make the streaming session more susceptible to bandwidth fluctuations. In the future, we will explore the opportunity that leverages cross-layer and user-centric approaches (e.g., introducing visual quality instead of bitrate) to further enhance player QoE.

Potential gains for other low-latency applications. While Pudica is designed for cloud gaming, we believe that the concepts and principles embedded within our CC framework could be beneficial for other networked applications that require both low latency and high throughput. We intend to investigate its potential across various domains.

7 Other Related Work

Prior works [22, 23, 35, 38] utilize explicit signals to judge the precise level of utilization or congestion, enabling convergence to both low queuing, efficiency, and fairness. However, it remains challenging to deploy these methods at scale. Nimbus [33] tries to detect the elasticity of cross traffic and dynamically switches between throughput-competitive and delay-controlling modes based on its detection. It still prioritizes throughput over delay, which is unable to satisfy the need for consistent low delay. Several works explore the use of deep reinforcement learning on bitrate adaptation for video telephony [14–16]. They partly imitate the behavior of GCC agents or take GCC as the protective backup, thereby with similar limitations as GCC. Moreover, cross-flow fairness is not carefully handled by these methods. A detailed comparison between our network probing method and traditional packet-train techniques is included in Appendix I.

8 Conclusion

We present Pudica, a practical congestion control (CC) algorithm designed for cloud gaming systems. Given a constraint of near-empty bottleneck queues, we rethink how to achieve efficiency and fairness in end-to-end CC design, regardless of single-flow or multiple-flow scenarios. To reach this goal, we propose a novel network probing method to estimate the bandwidth utilization ratio (BUR) of the bottleneck link. By leveraging both long-term and short-term BUR estimations, we design several intuitive but effective control strategies to minimize the queuing delay while maintaining efficiency and fairness. By conducting large-scale experiments on Tencent START cloud gaming services in both wired and wireless networks, we demonstrate that Pudica considerably reduces the frame delay and stall rate while preserving high bitrate and decent fairness, compared to the alternatives.

Acknowledgements: We sincerely thank our anonymous reviewers for their valuable comments. This work was supported by Tencent Rhino-Bird Research Elite Program, National Key Research and Development Program of China under Grant 2021YFB2401300, and National Natural Science Foundation of China under Grant U21A6005. Shusen Yang, Jing Wang, and Honghao Liu are the corresponding authors.

References

[1] Hao Chen, Xu Zhang, Yiling Xu, Ju Ren, Jingtao Fan, Zhan Ma, and Wenjun Zhang. T-gaming: A cost-efficient cloud gaming system at scale. *IEEE Transactions on Parallel and Distributed Systems*, 30(12):2849–2865, 2019.

[2] Xu Zhang, Hao Chen, Yangchao Zhao, Zhan Ma, Yiling Xu, Haojun Huang, Hao Yin, and Dapeng Oliver Wu. Improving cloud gaming experience through mobile edge computing. *IEEE Wireless Communications*, 26(4):178–183, 2019.

[3] Pouya Hamadanian, Doug Gallatin, Mohammad Alizadeh, and Krishna Chintalapudi. Ekho: Synchronizing cloud gaming media across multiple endpoints. In *SIGCOMM*, pages 533–549, 2023.

[4] Jiangkai Wu, Yu Guan, Qi Mao, Yong Cui, Zongming Guo, and Xinggong Zhang. Zgaming: Zero-latency 3d cloud gaming by image prediction. In *SIGCOMM*, pages 710–723, 2023.

[5] Sandeepa Bhuyan, Shulin Zhao, Ziyu Ying, Mahmut T Kandemir, and Chita R Das. End-to-end characterization of game streaming applications on mobile platforms. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(1):1–25, 2022.

[6] Amazon. Amazon luna: Amazon's cloud gaming service. https://luna.amazon.com/.

[7] Microsoft. Xbox cloud gaming (beta) on xbox.com. https://www.xbox.com/en-us/play.

[8] NVIDIA. Your games. your devices. play anywhere | nvidia geforce now. https://www.nvidia.com/en-us/geforce-now/.

[9] Tencent. START Cloud Gaming. https://start.qq.com/.

[10] Grand View Research. Cloud gaming market size, share & trends analysis report. https://www.grandviewresearch.com/industry-analysis/cloud-gaming-market.

[11] Venkat Arun and Hari Balakrishnan. Copa: Practical delay-based congestion control for the internet. In *NSDI*, pages 329–342, 2018.

[12] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Congestion control for web real-time communication. *IEEE/ACM Transactions on Networking*, 25(5):2629–2642, 2017.

[13] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *NSDI*, pages 267–282, 2018.

[14] Anfu Zhou, Huanhuan Zhang, Guangyuan Su, Leilei Wu, Ruoxuan Ma, Zhen Meng, Xinyu Zhang, Xiufeng Xie, Huadong Ma, and Xiaojiang Chen. Learning to coordinate video codec with transport protocol for mobile video telephony. In *MOBICOM*, pages 1–16, 2019.

[15] Huanhuan Zhang, Anfu Zhou, Jiamin Lu, Ruoxuan Ma, Yuhan Hu, Cong Li, Xinyu Zhang, Huadong Ma, and Xiaojiang Chen. Onrl: improving mobile video telephony via online reinforcement learning. In *MOBICOM*, pages 1–14, 2020.

[16] Huanhuan Zhang, Anfu Zhou, Yuhan Hu, Chaoyue Li, Guangping Wang, Xinyu Zhang, Huadong Ma, Leilei Wu, Aiyun Chen, and Changhui Wu. Loki: improving long tail performance of learning-based real-time video adaptation by fusing rule-based models. In *MOBICOM*, pages 775–788, 2021.

[17] Xiaoqing Zhu and Rong Pan. Nada: A unified congestion control scheme for low-latency interactive video. In *International Packet Video Workshop*, pages 1–8. IEEE, 2013.

[18] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 14(5):20–53, 2016.

[19] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. Pcc vivace: Online-learning congestion control. In *NSDI*, pages 343–356, 2018.

[20] Devdeep Ray, Connor Smith, Teng Wei, David Chu, and Srinivasan Seshan. Sqp: Congestion control for low-latency interactive video streaming. *arXiv preprint arXiv:2207.11857*, 2022.

[21] Wikipedia. Mimosa pudica. https://en.wikipedia.org/wiki/Mimosa_pudica.

[22] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *SIGCOMM*, pages 89–102, 2002.

[23] Yong Xia, Lakshminarayanan Subramanian, Ion Stoica, and Shivkumar Kalyanaraman. One more bit is enough. In *SIGCOMM*, 2005.

[24] Bonree. Data - bonree one. https://www.bonree.com/.

[25] Zili Meng, Tingfeng Wang, Yixin Shen, Bo Wang, Mingwei Xu, Rui Han, Honghao Liu, Venkat Arun, Hongxin Hu, and Xue Wei. Enabling high quality real-time communications with adaptive frame-rate. In *NSDI*, 2023.

[26] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. *SIGCOMM CCR*, 41(4):362–373, 2011.

[27] Lawrence S Brakmo, Sean W O'Malley, and Larry L Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *SIGCOMM*, pages 24–35, 1994.

[28] David X Wei, Cheng Jin, Steven H Low, and Sanjay Hegde. Fast tcp: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking*, 14(6):1246–1259, 2006.

[29] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *NSDI*, pages 459–471, 2013.

[30] Keith Winstein and Hari Balakrishnan. Tcp ex machina: Computer-generated congestion control. *SIGCOMM*, 43(4):123–134, 2013.

[31] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. Pcc: Re-architecting congestion control for consistent high performance. In *NSDI*, pages 395–408, 2015.

[32] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive congestion control for unpredictable cellular networks. In *SIGCOMM*, pages 509–522, 2015.

[33] Prateesh Goyal, Akshay Narayan, Frank Cangialosi, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. Elasticity detection: A building block for internet congestion control. In *SIGCOMM*, pages 158–176, 2022.

[34] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. Classic meets modern: A pragmatic learning-based congestion control for the internet. In *SIGCOMM*, pages 632–647, 2020.

[35] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. Achieving consistent low latency for wireless real-time communications with the shortest control loop. In *SIGCOMM*, pages 193–206, 2022.

[36] Manish Jain and Constantinos Dovrolis. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with tcp throughput. *SIGCOMM CCR*, 32(4):295–308, 2002.

[37] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for http. In *ATC*, pages 417–429, 2015.

[38] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. ABC: A simple explicit congestion controller for wireless networks. In *NSDI*, pages 353–372, 2020.

[39] Xinlei Yang, Xianlong Wang, Zhenhua Li, Yunhao Liu, Feng Qian, Liangyi Gong, Rui Miao, and Tianyin Xu. Fast and light bandwidth testing for internet users. In *NSDI*, pages 1011–1026, 2021.

[40] SpeedTest. The global broadband speed test. https://www.speedtest.net/.

[41] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems*, 17(1):1–14, 1989.

[42] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

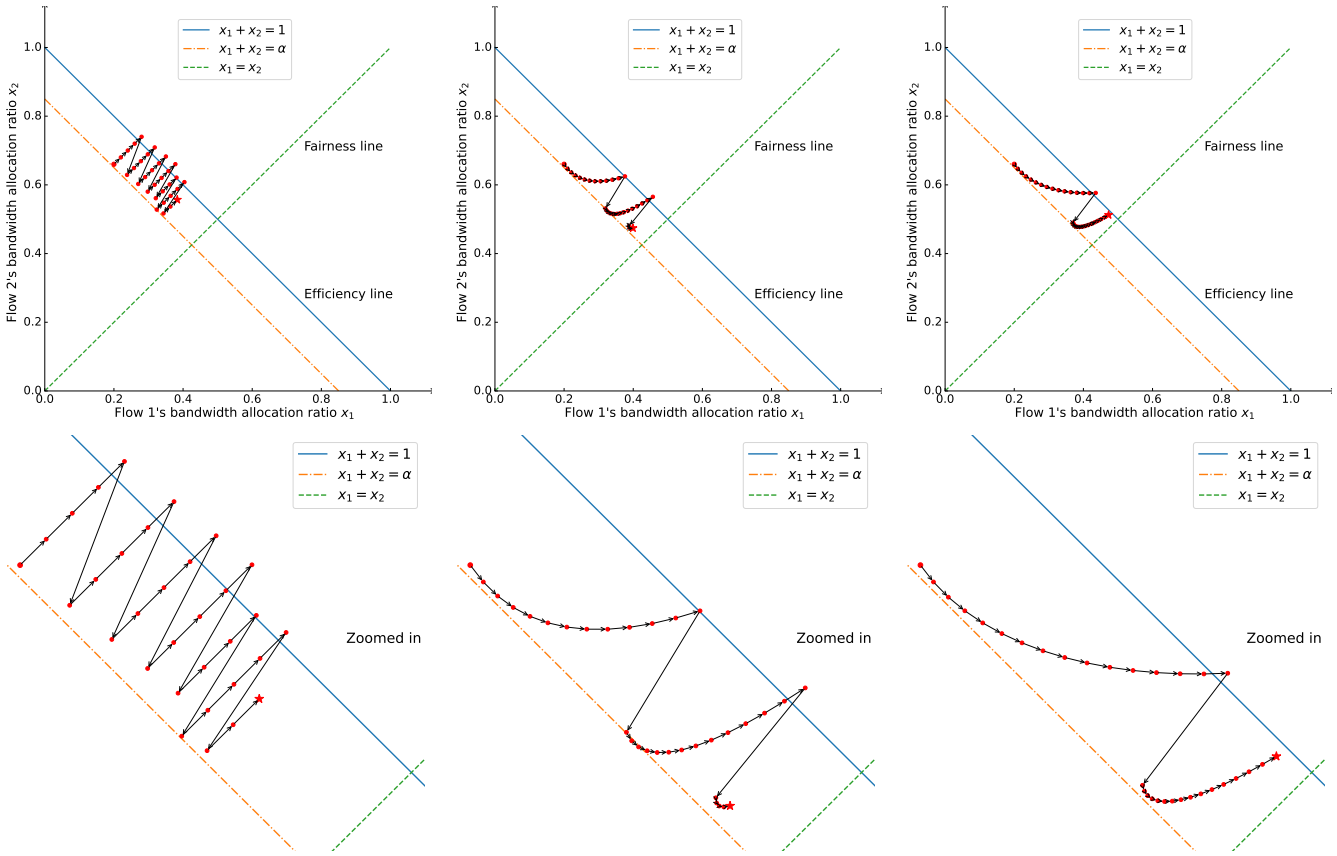
[43] Constantinos Dovrolis, Parameswaran Ramanathan, and David Moore. Packet-dispersion techniques and a capacity-estimation methodology. *IEEE/ACM Transactions on Networking*, 12(6):963–977, 2004.

[44] Ravi Prasad, Constantine Dovrolis, Margaret Murray, and Kimberly Claffy. Bandwidth estimation: metrics, measurement techniques, and tools. *IEEE Network*, 17(6):27–35, 2003.

[45] Constantinos Dovrolis, Parameswaran Ramanathan, and David Moore. What do packet dispersion techniques measure? In *INFOCOM*, volume 2, pages 905–914. IEEE, 2001.

A Validation Methodology for BUR Estimation Methods

In this section, we present the methodology used to validate the effectiveness of our BUR estimation method. Following [39], we regard the estimated bandwidth obtained through a flooding-based probing approach (similar to SpeedTest [40]) as the ground truth of bottleneck bandwidth, denoted by C. For each round of tests, after a short period of probing to judge C, we conducted two simultaneous flows on the same connection.



(a) Traditional AI/MD (b) AI-MD w/ bitrate-independent adaptive AI (c) AI-MD w/ bitrate-dependent adaptive AI

Figure 15: Fairness convergence trajectory for the traditional AI/MD paradigm and our AI-MD method (i.e., simultaneous AI and MD). Here, these diagrams are presented for illustration purposes and do not exactly align with the proposed algorithms in Pudica.

One flow adopted our probing strategies and was set to a fixed bitrate B. The other flow acted as the competing flow with a *random* pace multiplier (to simulate complex traffic patterns) and the same bitrate B. To evaluate the performance of various BUR estimation methods, we use the ratio $(2 \times B)/C$ as the actual BUR. This allowed us to assess how well the estimation methods performed in estimating the true BUR.

B Sample Weights for Smoothed BUR Computation

Due to the interdependence of sending behaviors and network feedback, we assign different weights to the BUR samples to enhance estimation robustness. Specifically, we set:

$$\omega_k^I = min\big(\, R_k + 1,\ 2\,\big), \tag{12}$$

$$\omega_k^{II} = min\big(\, B_k + 10,\ 50\,\big), \tag{13}$$

$$\omega_k^{III} = k + 20, \tag{14}$$

$$\omega_k = \frac{\omega_k^I \times \omega_k^{II} \times \omega_k^{III}}{\sum_{j=1}^{N_{packet}} \omega_j^I \times \omega_j^{II} \times \omega_j^{III}}. \tag{15}$$

ω_k symbolizes the importance weight given to the k-th BUR sample, comprising three elements. ω_k^I assigns higher importance to samples with longer frame delays, as they spend more time in flight. ω_k^{II} assigns higher importance to samples with larger frame sizes, as they exhibit greater estimation robustness against delay jitters. ω_k^{III} assigns higher importance to more recent samples, as they provide fresher feedback.

C Fairness Convergence Trajectory for Different AI-and-MD Schemes

As referenced in [41], we plotted the trajectories of a two-flow system starting from the same point using various AI-and-MD policies via numerical simulation, which is depicted in Fig. 15. On the figures, the horizontal axis represents the bandwidth allocation ratio of Flow 1 (denoted by x_1), while the vertical axis corresponds to Flow 2 (denoted by x_2). Allocations satisfying $x_1 + x_2 = 1$ indicate efficient allocations, representing 100% bandwidth utilization (refer to *Efficiency line*). Allocations satisfying $x_1 = x_2$ represent fair allocations (refer to *Fairness line*). The optimal point is the intersection of these two lines. The objective of control schemes should be to converge the system to this optimal point, irrespective of the initial position.

In the AI/MD method (see Fig. 15a), we additively increase the bitrate when the BUR exceeds α but remains below one; we multiplicatively decrease the bitrate by 15% when the BUR surpasses one. For AI-MD methods, Fig. 15b illustrates

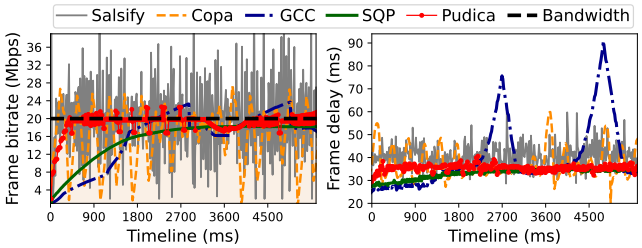


Figure 16: Preliminary validation: the performance of various baseline algorithms using Mahimahi emulations with a plain trace featuring a constant bandwidth of 20 *Mbps*.

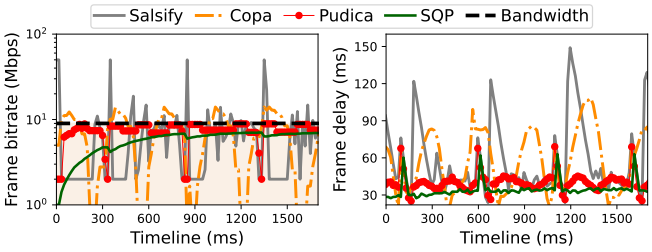


Figure 17: Robustness test with a 40 *ms* jitter per period of 500 *ms*. Pudica achieves a good balance between fast responsiveness to potential congestion and jitter resilience.

the trajectory of AI-MD with a bitrate-independent adaptive AI scheme, where the AI step (i.e., I) is linearly increased. In contrast, the AI-MD method in Pudica (see Fig. 15c) adopts a bitrate-dependent adaptive AI scheme, where the magnitude of increased AI steps (i.e., ΔI) is inversely proportional to the bitrate. Compared to the AI/MD method, AI-MD reduces the occurrences of over-sending and thus lowers the average queuing delay, by executing MD before queue construction. At the same time, AI-MD speeds up fairness convergence by increasing the frequency of throughput reallocation. Moreover, the implementation of bitrate-dependent AI step adaptation further enhances the convergence speed toward fairness.

D Preliminary Evaluation for Various CC Algorithms through Emulation

Prior to conducting large-scale experiments over the Internet, we evaluated Pudica and the other potential solutions (i.e., GCC [12], Copa [11], Salsify [13], and SQP [20]) on the Mahimahi network emulator [37]. These evaluations were performed using a trace that maintained a constant bandwidth of 20 *Mbps* and RTTs of 20 *ms*. As depicted in Fig. 16, Pudica and SQP achieve consistent bandwidth convergence with negligible queuing delays. Salsify and Copa maintain low frame delays in spite of significant oscillations in bitrates. In contrast, GCC fails to consistently achieve low latency, displaying periodic delay spikes even under a constant bandwidth. Therefore, we selected Pudica, Salsify, Copa, and SQP for further performance testing at scale (shown in §5.2).

E A Large-Scale Dummy Client Platform

In our study, we utilized a specialized dummy client platform called Bonree [24], instead of the real-user platform (i.e., START [9]), to evaluate the convergence ability of efficiency and fairness in the wild. We opted for the Bonree platform due to the additional client privileges required for testing the aforementioned metrics, such as the permission to run multiple streams simultaneously. These privileges cannot be accommodated by our profit-oriented cloud gaming services.

Bonree offers millions of end devices (e.g., PCs) distributed globally to emulate real user clients. These end devices possess computing capacities and network resources similar to those commonly used by consumers. They can install customized software, e.g., the cloud gaming client app, and establish connections with the cloud gaming servers designated by us over the Internet. As a result, the gaming server can transmit frame streams to these end devices and receive their feedback. Additionally, these devices can record the necessary logs as per our requirements. Unlike real users, we can impose more demands on these devices without concerns about user experience. Thus, we employed Bonree to conduct experiments regarding efficiency and fairness convergence. Note that the end devices on Bonree communicate with servers through fully public networks. Consequently, the information within the network remains unknown.

F Pudica Robustness to Network Jitters

In Fig. 17, we assessed Pudica's robustness to delay jitters by introducing a 40*ms* jitter every 500*ms* period. While this jitter delays some packet arrivals, it does not impact the bandwidth. These delayed packets eventually reach the receiver with subsequent packets, leading to a burst arrival. The results show that Pudica effectively balances sensitivity to high delay and resilience against jitters. While SQP shows strong robustness to jitters, it struggles with slow bandwidth adaptation (Fig. 12). The burst packet arrival can mislead Salsify and Copa to momentarily overrate network conditions, which causes bitrate overshoots and high delay spikes.

G Additional Microbenchmark Results

This section presents more statistics from our microbenchmark experiment. Tab. 3 and Tab. 4 showcase the performances of multiple Pudica variants under a single flow and three simultaneous flows, respectively. These experimental results confirm the efficacy of Pudica's BUR estimation method and AI-MD technique. For Pudica with AI/MD, we substitute our AI-MD scheme with the sole additive increase when $\alpha < BUR \leq 1$. MD operations will be triggered by our queue-draining scheme when $BUR > 1$. Tab. 5 shows the effectiveness of Pudica's control schemes driven by short-term BUR signals in reducing tail delays and stall rates. Tab. 6 presents Pudica's performance under various parameter settings.

H Competitiveness with Buffer-Filling Flows

Figure 18 depicts the Pudica performance when it encounters a buffer-filling flow. Upon initiating a Pudica flow, we subsequently launched and terminated a Cubic flow [42] at 5s and

Algo.	Avg. delay	95%/99% -tile delay	Stall rate >50/100/200ms	Avg. bitrate
Vanilla Pudica	18.0ms	22.3/29.2ms	0.73%/0.09%/0.0083%	46.940Mbps
Pudica w/ BUR estimation of SQP	20.1ms	26.1/39.0ms	1.25%/0.21%/0.0229%	47.103Mbps
Pudica w/ BUR estimation of Salsify	18.4ms	22.9/34.2ms	0.84%/0.13%/0.015%	41.164Mbps
Pudica w/ AI/MD	18.3ms	22.7/32.2ms	0.91%/0.10%/0.0117%	46.181Mbps

Table 3: Microbenchmarks: the performance of different Pudica variants when launching a single flow.

Algo.	Avg. delay	95%/99% -tile delay	Stall rate >50/100/200ms	Avg. bitrate	Fairness index
Vanilla Pudica	21.5ms	29.2/38.6ms	1.60%/0.11%/0.0164%	31.962Mbps	0.958
Pudica w/ BUR estimation of SQP	31.9ms	49.2/67.5ms	9.77%/0.42%/0.0391%	33.132Mbps	0.952
Pudica w/ BUR estimation of Salsify	21.7ms	34.2/51.8ms	2.81%/0.25%/0.0538%	33.185Mbps	0.941
Pudica w/ AI/MD	22.3ms	32.8/61.2ms	2.10%/0.29%/0.0747%	32.209Mbps	0.944

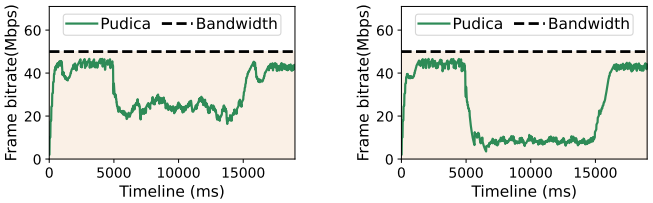
Table 4: Microbenchmarks: the performance of different Pudica variants when simultaneously launching three flows on the same link.

Algo.	Avg. delay	95%/99% -tile delay	Stall rate >100/200ms	Avg. bitrate
Vanilla Pudica	20.6ms	27.3/33.6ms	0.17%/0.009%	46.21Mbps
Pudica w/o temporary bitrate fallback	20.9ms	28.4/38.3ms	0.29%/0.031%	46.19Mbps
Pudica w/o active queue draining	23.7ms	36.0/60.5ms	1.40%/0.193%	47.16Mbps

Table 5: Microbenchmarks: performances of multiple Pudica variants to explain the contribution of our short-term BUR-driven control schemes.

Algo.	Avg. delay	95%/99% -tile delay	Stall rate >100ms	Avg. bitrate
Pudica	18ms	21.3/28.5ms	0.061%	47.86Mbps
α=0.8	17.9ms	21.3/29.1ms	0.075%	47.55Mbps
α=0.9	18.8ms	21.9/28.0ms	0.049%	48.28Mbps
γ_{MI}=0.20	18.5ms	22.1/30.7ms	0.084%	47.26Mbps
γ_{MI}=0.25	17.8ms	21.4/28.1ms	0.08%	48.14Mbps
ζ=10%	18.5ms	22.4/30.4ms	0.066%	47.99Mbps
ζ=20%	18.9ms	21.5/27.9ms	0.054%	47.42Mbps
T_{wd}=150	18.5ms	22.0/30.6ms	0.058%	47.96Mbps
T_{wd}=250	18.4ms	21.6/28.6ms	0.066%	47.82Mbps

Table 6: Sensitivity analysis for algorithm parameters.



(a) Buffer size: 10,000 bytes. (b) Buffer size: 50,000 bytes.

Figure 18: Pudica performance in the presence of a Cubic flow, with different buffer size settings of the bottleneck queue.

10s, respectively. We can see that Pudica exhibits a certain level of competitiveness when the buffer size of the bottleneck queue is small. However, when the buffer size increases, Pudica fails to compete effectively against Cubic.

I Difference Between Our BUR Estimation and Traditional Packet-Train Methods

Pudica leverages all packets within a frame as a short burst, which can be regarded as a packet train, to probe the link condition. The difference between our method and traditional packet-train network estimation approaches lies in what exactly to estimate and how to probe.

Traditional packet-train methods [43, 44] utilize the dispersion of a packet train, i.e., the gap in arrival time between the first and last packets, to estimate the available bandwidth or capacity. For a detailed definition of packet train and its estimation method, see [43]. In a single-flow environment, packet-train dispersion indicates the bottleneck link capacity, which also represents the available bandwidth. However, for a multi-flow scenario, the estimated value by these methods is actually neither available bandwidth nor capacity [45].

By contrast, Pudica leverages the queuing delay of a packet train to estimate the link utilization (i.e., BUR) during a frame interval. As per [43], a link is either transmitting at full capacity or idle at any given moment, meaning its instantaneous utilization is either zero or one. Therefore, BUR is defined as the time-averaged instantaneous utilization over a specific interval. In our case, this interval is the frame sending interval (i.e., L), typically 16.67 ms for a 60 frame rate. Then, the BUR of the time period $(t, t+L)$ can be expressed as:

$$R(t, t+L) = \frac{1}{L}\int_{t}^{t+L} r(x)dx \tag{16}$$

where $r(x)$ is the instantaneous link utilization at time x.

Traditional packet-train probing methods typically implement a fixed *pace multiplier* (as defined in §2.1). These methods can only adjust the *pacing rate* by resorting to bitrate adjustments. In contrast, Pudica introduces an adaptive pace multiplier algorithm that enables appropriate queuing even when the bitrate remains unchanged. Another difference is that Pudica proposes a supplementary probing technique via a small number of probe packets beyond application data. This method enables Pudica to effectively probe the link condition even during the agnostic period.

Note that in Pudica, the BUR probing and estimation method introduced in §4.1 is only used when the link utilization is below one. When the link utilization exceeds one, the agnostic period is eliminated, and we rely on the packet receiving rate to estimate the available bandwidth (see §4.3), which is similar to traditional packet-train techniques.

Revisiting Congestion Control for Lossless Ethernet

Yiran Zhang[1,2], Qingkai Meng[1,3], Chaolei Hu[1], Fengyuan Ren[1]
[1]*Tsinghua University,* [2]*Beijing University of Posts and Telecommunication,* [3]*Beihang University*

Abstract

Congestion control is a key enabler for lossless Ethernet at scale. In this paper, we revisit this classic topic from a new perspective, i.e., understanding and exploiting the intrinsic properties of the underlying lossless network. We experimentally and analytically find that the intrinsic properties of lossless networks, such as *packet conservation*, can indeed provide valuable implications in estimating pipe capacity and the precise number of excessive packets. Besides, we derive principles on how to treat congested flows and victim flows individually to handle HoL blocking efficiently. Then, we propose ACK-driven congestion control (ACC) for lossless Ethernet, which simply resorts to the knowledge of ACK time series to exert a temporary halt to exactly drain out excessive packets of congested flows and then match its rate to pipe capacity. Testbed and large-scale simulations demonstrate that ACC ameliorates fundamental issues in lossless Ethernet (e.g., congestion spreading, HoL blocking, and deadlock) and achieves excellent low latency and high throughput performance. For instance, compared with existing schemes, ACC improves the average and 99th percentile FCT performance of small flows by 1.3~3.3× and 1.4~11.5×, respectively.

1 Introduction

As the adoption of RDMA continues to grow in Ethernet-based data centers, there is a wave of the deployment of lossless networks. A lossless network can harness the full potential of RDMA and benefit application latency performance that used to be affected by packet loss [6, 15, 17].

Lossless Ethernet employs hop-by-hop flow control, i.e., Priority-based Flow Control (PFC) [1], to ensure that packets are not dropped due to buffer overflow. However, when persistent congestion occurs, PFC may impose a backpressure effect on upstream ports, and the cascade reaction can even spread to remote switches. Flows not destined to the congestion point are also paused, which is well-known as the Head-of-Line (HoL) blocking issue [42, 49]. The frequent trigger of PFC may also be along with deadlock and unfairness issues [21, 25, 26, 34, 42, 49]. The root cause of the above side effects is that PFC itself can not allocate appropriate bandwidth for each flow, so network congestion can not be eliminated but only spreads. As a result, end-to-end congestion control becomes a key enabler for high-performance lossless Ethernet at scale [21, 49].

Many efforts have been devoted to developing congestion control to facilitate the deployment of lossless Ethernet. DCQCN [49] employs widely-used ECN in switches to detect congestion and heuristically throttles congested flows. Nevertheless, ECN solely based on queue length may not provide the correct congestion indicator once PFC takes effect [45]. HPCC [29] relies on high-precision INT [2] to guide congestion control but incurs high per-packet overhead and sacrifices throughput performance. TIMELY [33] advocates RTT-based congestion control. Still, the RTT samples may mislead congestion judgment when RTT suddenly increases owing to HoL blocking. Recently, TCD [45] makes a new attempt to accurately detect congested flows but still follows the heuristic rate control rules as traditional congestion controls, which makes it essentially hard to achieve low latency and high throughput simultaneously.

Reflecting on these efforts, we notice that instead of designing congestion control for lossless Ethernet from scratch, almost all existing works weave unfitted pieces into their schemes thus resulting in sub-optimal performance. In particular, congestion detection is the foundation while rate control is indeed the core of achieving appropriate bandwidth allocation for each flow. Both of them should consider essential features of lossless Ethernet and work harmoniously together to realize high-performance congestion control. Therefore, we ask this question: *can we take a step back and rethink congestion control for lossless Ethernet by taking full advantage of its intrinsic properties?*

The crux to understanding the intrinsic properties of lossless Ethernet is to recognize the profound impact of hop-by-hop flow control (i.e., PFC). For instance, the ON-OFF regulation of hop-by-hop flow control introduces a new ON-

OFF style port state in switches, changing the foundation of congestion detection. More importantly, no packet dropping at intermediate switches in turn enables an end-to-end lossless path, which alters the view of the network pipe and opens new opportunities for flow-level bandwidth allocation. In fact, previous efforts unconsciously neglect one intrinsic yet powerful property, i.e., **packet conservation**: packets are not lost in the lossless network[1]. All injected packets are eventually delivered to receivers and acknowledged by ACKs, before which they fly in the network pipe or queue at switches.

In this paper, based on the understanding of above intrinsic properties and in-depth experimental investigations, we derive two key principles to build a desirable congestion control for lossless Ethernet: (1) Since packet conservation property implies that the number of ACKs equals to the number of packets injected into the network, the ACK-driven paradigm should gain renewed emphasis in lossless Ethernet to infer network pipe capacity and the exact number of excessive packets (§ 3.2). (2) To handle HoL blocking efficiently, congested flows should be suppressed sufficiently to eliminate accumulated buffers as soon as possible; victim flows should adapt to the severity of congestion to balance HoL blocking alleviation and throughput performance (§ 3.3).

Armed with the above principles, we propose a new congestion control for lossless Ethernet called ACC (ACK-Driven Congestion Control). ACC utilizes TCD [45] to accurately detect congested flows and victim flows. For congested flows, ACC gracefully employs a two-step strategy: senders first wait for the excessive packets to drain out via a temporary halt and then match the rate to network pipe capacity. In detail, by correlating *ACK sequences* across multiple periods and *ACK arrival rate*, ACC senders figure out the number of excessive packets and the exact time to be drained at switches. For victim flows, ACC senders collect the duration pattern of *ACK markings* to perceive the severity of congestion and only adjust the rate when congestion spreading lasts long.

Our key contributions are summarized as follows:

• Understanding the implications of intrinsic properties of lossless Ethernet (e.g., packet conservation) on congestion control and deriving basic principles for handling congested flows and victim flows. Our perspective strikes out a new path of exploiting intrinsic properties to guide the precise rate control for lossless networks.

• Developing a new congestion control scheme called ACC for lossless Ethernet. As ACK can correspond to each injected packet precisely, ACC utilizes ACK time series to derive exact backlogged packets in switches and network pipe capacity. ACC can quickly converge to the proper rate in one RTT and empty accumulated queues rapidly.

[1]Note that in lossy networks where packets can be dropped due to buffer overflow, the number of packets injected into the network is not equal to the number of ejected packets (i.e., acknowledged by receivers). So packet conservation property does not hold in lossy networks. In addition, we do not consider packet loss due to packet corruption or device failure in this paper.

• Implementing ACC in SoftRoCE [36] and evaluating ACC via extensive experiments. Results show that ACC well alleviates congestion spreading and HoL blocking issues and can achieve low latency and high throughput. For instance, compared with state-of-the-art schemes, ACC improves the average and 99th percentile FCT performance of small flows by 1.3~3.3× and 1.4~11.5×, respectively.

2 Design Space

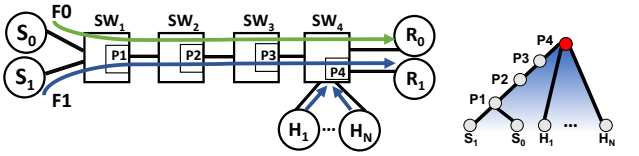
2.1 Desirable Properties

We begin with reexamining the desirable properties of congestion control for lossless Ethernet. Specifically, we target the scenario of a single RDMA domain deployed with a single CC scheme, which is common in Ethernet-based datacenters.

(1) **Fast convergence.** The primary goal of congestion control is allocating the proper rate for each flow thus the aggregation rate converges to the bottleneck capacity. However, in lossless Ethernet, fast convergence is particularly crucial to the following issues:

• *Restricting the spreading of congestion and alleviating HoL blocking.* Without fast convergence, a congestion point is likely to spread into a congestion tree with cascading accumulated queues along branches. Take Figure 1(a) as an example. After congestion happens at port P4, P4 becomes the root of the congestion tree. Once the PFC pause propagates to upstream switches, the paths of congested flows (e.g., <S_1-R_1>) become the main branch of the congestion tree. When other flows (e.g., F0) destined for non-congestion points pass through the main branch, the backpressure of PFC may further induce the secondary branch. Thus, HoL blocking occurs. Flows not destined for congestion points innocently suffer from PFC pause. The accordingly secondary branch in Figure 1(b) is <P1-S_0>. When the congestion tree grows, the input and output of the switches along the branches are prone to be modulated into an ON-OFF pattern by PFC (i.e., alternating between sending and pausing), thus disturbing congestion detection and subsequent rate regulation. If multiple congestion points are in the same congestion tree (i.e., a larger congestion tree covers a smaller congestion tree), congestion point in the larger congestion tree will be first eliminated then the smaller congestion tree.

• *Lowering the risk of deadlock*. Deadlock is a silent killer in the lossless network [21, 25, 26, 38]. The typical causes of deadlock in today's data centers are as follows: Initially, owing to reasons such as routing misconfiguration, switch firmware failure, or link failure, a Cyclic Buffer Dependency (CBD) emerges in the topology. Then if congestion happens and the congestion control fails to converge quickly, the frequent trigger of PFC may be induced. Eventually, PFC pause propagates to the whole cycle and all switches wait for its upstream to send PFC resume, thus a deadlock occurs. From another perspective, even if CBD exists, the fast convergence of congestion control can make it resilient against deadlock.



(a) Topology and flows ($N > 1$) (b) Congestion tree

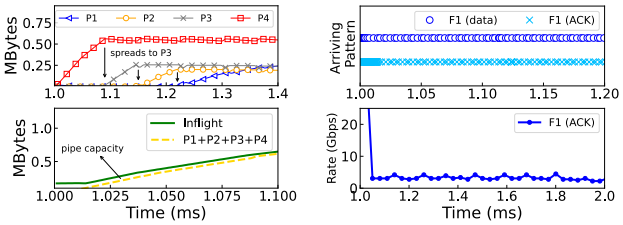
Figure 1: Typical scenario.

(2) **Low latency and high throughput.** Today's applications increasingly pursue stringent latency and throughput performance [23,27]. In lossless Ethernet, offloading network stacks into hardware (i.e., RDMA) helps sustain the critical latency and bandwidth requirements at hosts but leads to the network becoming the bottleneck. On the one hand, for increasingly dominant storage workloads with NVMe-over-Fabrics technology [10,27], the overall completion latency for a single storage operation is determined by the latency of the slowest network operation. Each storage operation involves messages with a size of only several KB and may require microsecond-scale latency [3, 19, 47, 48]. On the other hand, data processing applications [16, 44] and AI applications increasingly involve communicating among high computation speed devices at scale [29, 39, 51]. These applications periodically transfer a large volume of data and the average transfer time of each round directly impacts the overall processing/training time and cost, which imposes high requirements on throughput performance. To sum up, congestion control should be able to achieve low latency for short flows and predictable high throughput for longer flows.

2.2 Ternary Flow States in Lossless Ethernet

The foundation of congestion control in lossless Ethernet is understanding the network state space under the influence of PFC. We notice that a recent work TCD [45] develops a ternary congestion signal tailored to lossless networks, which can distinguish between switch ports that are roots of congestion trees (i.e., congestion ports) and switch ports on the branches that are only affected by PFC (i.e., undetermined ports). TCD terms the new state as "undetermined" because the real states of these ports are masked due to intermittent ON-OFF sending pattern when PFC triggers. TCD-enabled switches mark packets passing through a congested port with CE and mark packets passing through an undetermined port with UE, which indicates "congestion encountered" and "undetermined encountered", respectively. Besides, UE can only be marked when the packet is not marked with CE. As a result, TCD can notify end hosts of ***ternary*** flow states: congested flows that pass through congestion ports, undetermined flows [2] that *only* pass through undetermined ports, and uncongested flows. As shown in Figure 1, port P4 is the congestion port, while all other nodes of the congestion tree are undetermined ports. F1 is the congested flow, while F0 is the victim flow.

[2]We interchangeably use "victim flows" and "undetermined flows" throughout the paper.



(a) F1: inflight packets and packets backlogged in switch buffers (b) Packets arriving pattern

Figure 2: [$N = 20$] F1: packets backlogged in switches and data/ACK arriving pattern.

However, although TCD can be useful for perceiving network states, it remains an essential issue on how to deal with these refined flows to fulfill the desirable properties for congestion control. Concretely, for congested flows that are actual contributors to congestion, congestion control should regulate its rate to converge quickly to the proper value and stay near it. By doing so, emerging congestion can be eliminated rapidly and prevents the potential risk of congestion spreading. Still, chances are that congestion spreads (e.g., caused by bursty traffic or first RTT traffic) and thus may accompany the emergence of HoL blocking and victim flows.

3 Principles

In this section, we aim to present design principles to answer key questions for congestion control in lossless Ethernet:

(1) How to adjust the rate of congested flows to achieve fast convergence? (§ 3.2)

(2) How to treat congested flows and victim flows individually to handle HoL blocking? (§ 3.3)

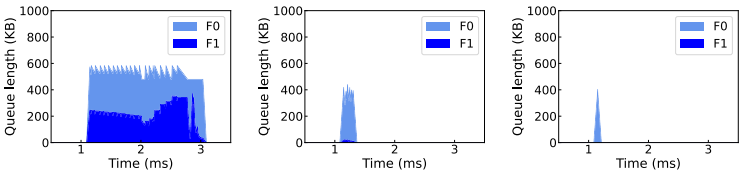
3.1 Experiment Setup

To provide vivid illustration, we conduct detailed investigations via ns-3 simulations. The network topology is shown in Figure 1(a). The link capacity is 100Gbps with 2 μs propagation delay. F0 and F1 are long-lived flows. $H_1 \sim H_N$ send concurrent 64KB bursts lasting for about 2ms and can hardly be regulated by end-to-end congestion control as the size is smaller than bandwidth-delay-product (BDP). For rate decrease algorithms at hosts, we adopt the widely-used DCQCN [49]. The PFC threshold is 512KB.

The switches support TCD so hosts are aware of ternary states of flows. Specifically, assume F0 and F1 start simultaneously and achieve their fair bandwidth allocation, then bursts start. After congestion occurs at port P4, F0 becomes the victim flow in the branch <P4-S_1>. We introduce different congestion degrees by changing the value of N, which indicates the number of concurrent senders. A larger N induces more severe congestion at port P4, resulting in a deeper congestion tree.

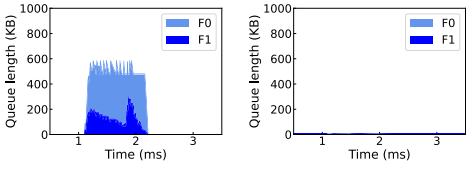
3.2 The Power of ACK-Driven

To explore the potential principle to throttle congested flows, we first take a close look by combining the in-network per-



(a) F1:DCQCN (b) F1 stops 20μs and sends at PC_{rate} (c) F1 stops 180μs and sends at PC_{rate}

Figure 3: [$N = 20$] Queue occupancy at Port P3. PC_{rate}: the rate corresponding to the pipe capacity of the flow.



(a) F1: DCQCN (b) F1 stops 90μs and sends at PC_{rate}

Figure 4: [$N = 10$] Queue occupancy at Port P3.

spective (e.g., congested packets backlogged in switches) and the end-host perspective (e.g., inflight bytes seen by senders).

Figure 2(a) shows the evolution of F1 packets backlogged in each switch port (the upper subfigure) and the inflight bytes maintained at the sender H_1 (the lower subfigure) under Setting I ($N = 20$). The inflight bytes are derived from the difference between the next sending byte sequence and the unacknowledged byte sequence. After congestion occurs at 1ms, F1 packets are only queued at P4 till 1.1ms. As PFC takes effect and congestion gradually spreads to upstream switches, the number of backlogged bytes at ports P3, P2, and P1 rise one by one. For congested flow F1, the maximum amount of traffic that can be delivered per RTT is $C \times RTT/(N+1)$, also called as the network pipe capacity of this flow (around 0.012MB in this scenario). As shown in Figure 2(a), we notice that the number of *inflight bytes* closely tracks the sum of *total backlogged packets* (i.e., packets queued in P4, P3, P2, and P1) and the *network pipe capacity* of F1.

From the end-host perspective, Figure 2(b) depicts the packets arriving pattern of congested flow F1 at R_1 and its ACKs at S_1. After congestion occurs at 1ms, the arriving pattern of F1 data packets is continuous for a long time period at first because excessive packets would eventually go through the continuously-ON congested port P4. The arriving pattern of ACKs corresponds well to data packets. We also record the arriving rate of F1 ACKs. For congested flow F1, the arriving rate of ACKs (around 4.7Gbps) is exactly the available bandwidth of F1, which can also deduce the capacity of network pipe (i.e., available bandwidth $\times$ RTT). In summary, we have ***Observation 1****: For congested flows: (1) the number of excessive packets backlogged in switches is exactly the difference between inflight packets and the network pipe capacity; and (2) the ACK arrival rate can imply the available bandwidth.*

Indeed, the above observations corroborate an intrinsic property of lossless Ethernet: **packet conservation**, i.e., sent packets are never lost. Different from traditional lossy networks where packets may be dropped due to buffer overflow, in lossless Ethernet, excessive packets will backlog in switch buffers along the branch after filling the pipe. Thus all of them are eventually delivered and acknowledged by ACKs.

The implications behind packet conservation property lie in that: the precise number of excessive packets and the network pipe capacity of congested flow can both be inferred from the ACK time series. Thus, the *ACK-driven* paradigm really comes into its own in a lossless network with packet conservation property. To this end, we derive **Principle 1: In lossless Ethernet, the ACK-driven paradigm is a powerful knob to infer the proper throttled rate and the precise number of excessive packets for congested flows.**

3.3 Handling HoL Blocking

Once PFC takes effect, victim flows may emerge and suffer from HoL blocking. The following question is how to treat congested flows and victim flows individually to handle HoL blocking. Ideally, HoL blocking should be eliminated as soon as possible without incurring unnecessary performance loss.

Next, we focus on the effect of rate control strategies of congested flows on HoL blocking alleviation and victim flows under different degrees of congestion (e.g., $N = 20$ and $N = 10$). We enable TCD in switches thus flows with packets marked as UE are identified as victim flows. The default rate control strategy is that only congested flows will be throttled once detected, and victim flows will adjust the sending rate the same as uncongested flows according to DCQCN.

Figure 3(a) reports the queue occupancy of individual flows at the switch port P3 when $N = 20$ with DCQCN. After bursts start at 1ms, congestion emerges at port P4. Then congestion spreads to upstream switches and packets accumulate at ports P3 and P2 (not shown in the figure). The HoL blocking process lasts for about 2ms. We notice that the queue accumulation of F0 accompanies with the queue buildup of F1, which confirms that HoL blocking originates from queue buildup of congested flows along the branch.

Further, to understand how congested flows with precise rate adjustment may impact HoL blocking, we let congested flow F1 first empty the accumulated packets along switches by stopping for a while (after congestion is detected) and then send at the rate corresponding to the network pipe capacity (i.e., PC_{rate}). As shown in Figure 3(b), congestion spreading at P3 is largely alleviated. The queue occupancy of F1 is quickly suppressed and the total blocking time of F0 is reduced. We also estimate an ideal stopping time (i.e., 180*us*) and Figure 3(c) depicts the queue occupancy. With sufficiently long time to empty the accumulated packets along switches, the duration of HoL blocking is much shorter than in Figure 3(b). However, F0 is still blocked at port P3 and has queue buildup. This is because although congested flow F1 are stopped precisely,

pause frames may still be triggered due to large ingress queue previously accumulated at the downstream port [3].

When $N = 10$, we see a more desired impact on alleviating HoL blocking. As shown in Figure 4(a), the congestion degree is smaller when $N = 10$, where the duration time of HoL blocking is short. With DCQCN, the queue occupancy of F0 is even much larger than F1 at port P3. However, as depicted in Figure 4(b), once congested flow F1 stops for a sufficient time to empty the queue and then sends at the rate corresponding to the network pipe capacity, HoL blocking can even be eliminated. With fewer accumulated packets and a faster draining rate at the downstream congested port, the corresponding ingress queue falls below the PFC threshold faster. In this scenario, victim flows can safely pass without causing pause frames and HoL blocking. In summary, we have ***Observation 2***: *Stopping congested flows sufficiently long can eliminate associated buffers as soon as possible. However, under different congestion situations, HoL blocking may still occur and victim flows have the risk of inducing further congestion spreading.*

The above observations tell that although stopping congested flows does not necessarily stop pause frames from affecting victim flows, it does help get rid of such blocking states as soon as the accumulated buffer of congested flow is sufficiently drained. During this process, victim flows may still emerge and have the risk of congestion spreading. Thus, we advocate treating victim flows dynamically to adapt to different congestion situations, rather than treat them identically. We derive **Principle 2: Stopping congested flows sufficiently long is the foremost means to suppressing HoL blocking. While congested flows are stopping, victim flows should balance HoL blocking alleviation and throughput by adapting to the severity of congestion.**

Incorporating **Principle 1** and **Principle 2**, we come to the following strategies for congested flows and victim flows in lossless Ethernet: Congested flows should first stop to wait for accumulated congested queues to drain out, and then send at the rate of network pipe capacity. While victim flows should sacrifice throughput as little as possible and benefit HoL blocking alleviation. Such collaborative strategies can rapidly suppress congestion spreading without link underutilization, and benefit latency and throughput performance.

4 ACK-Driven Congestion Control

Following the above principles, we present the design of ACK-driven congestion control (ACC). The switch supports TCD and marks ternary congestion notification in data packets. A CE marked packet indicates a congested flow that passes through a congestion port. A UE marked packet indicates that the flow only passes through ports affected by PFC. The flow is uncongested if neither CE nor UE is marked (denoted by NO). The receiver copies the TCD marking to the corresponding ACK and sends it back to the sender.

[3] PFC is triggered based on the ingress queue length. In today's commodity shared-buffer switches, the ingress queue length is a counter which is updated when packets enqueue the ingress and dequeue the egress [21].

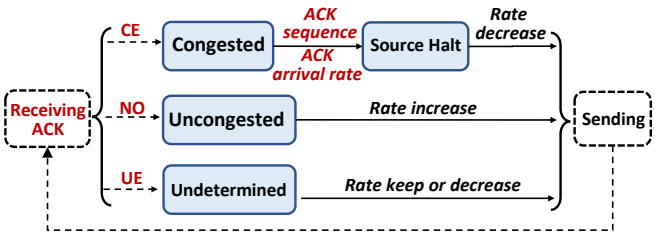


Figure 5: ACC state machine.

At its core, ACC conducts ACK-driven rate adjustment including enforcing a source halt according to ACK sequences (§ 4.2), and referring to ACK arrival rate to guide rate decrease for congested flows (§ 4.3). The source halt state involves a halt time which guarantees draining out accumulated packets in the network while avoiding link under-utilization. For victim flows, ACC adaptively adjusts the rate according to the feature of duration time of ACK markings (§ 4.4).

4.1 State Machine Overview

The ACC sender makes rate adjustment decisions every period T (e.g., base RTT, see § 5). At the end of each period T, the sender identifies the current state of the flow according to TCD marking and conducts corresponding rate adjustment.

Figure 5 illustrates the state machine of ACK-driven rate adjustment at ACC senders. TCD marking is aggregated in each T with the priority order CE > UE > NO. If there is ACK with CE received, the flow may experience congestion. The ACC sender records the number of CE marked ACKs of each flow. If the marking fraction of CE is above a threshold (i.e., 90%) during a period T, it is considered a steadily congested flow. If there is no ACK with CE received but with UE marking received, the flow is regarded as a victim flow that experiences an undetermined state. Only when neither UE nor CE marking is received, the flow is uncongested. Specifically, when the flow is identified as congested, it will first enter into a source halt state where the connection ceases transmission and the ACK arrival rate is recorded for later sending. After leaving the source halt state, congested flows are throttled at the proper rate derived from the ACK arrival rate. For uncongested flows, the sender directly increases the sending rate. For victim flows, the sender adjusts the sending rate according to the duration pattern of UE markings.

4.2 Halting Congested Flows

For congested flows, ACC introduces a source halt state which actively stops the transmission of congested flows before sending at the appropriate rate (§ 3.2 **Principle 2**). The crux of the source halt is the halt time. An appropriate halt time should ensure that accumulated packets are drained without under-utilizing the link. Indeed, a proper number of packets should fill the network pipe without causing queue buildup in the switch buffer. Specifically, ACC senders calculate the

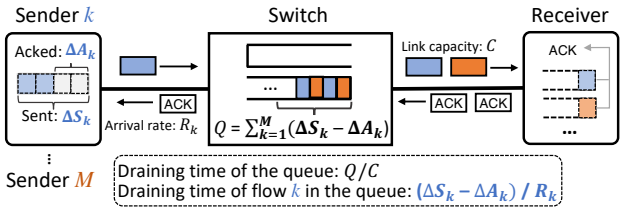


Figure 6: The rationale of calculating t_{Halt}.

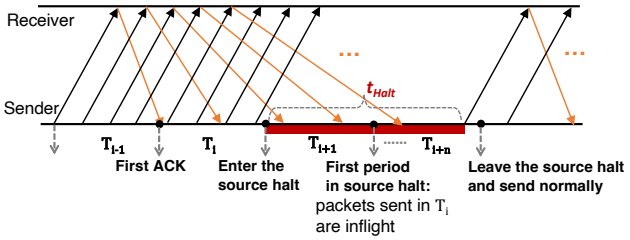


Figure 7: The timeline of entering and exiting source halt.

duration of the source halt state according to the time series of ACK sequences (§ 3.2 **Principle 1**).

Concretely, for each flow, the sender maintains a value called $t_{NextPkt}$ to represent when the next packet is transmitted. Initially, $t_{NextPkt}$ is set to 0 and is assigned by the congestion control algorithm. With ACC, $t_{NextPkt}$ is updated as follows when the sender enters into the source halt state:

$$t_{NextPkt} \leftarrow t_{NextPkt} + t_{Halt} \tag{1}$$

where t_{Halt} is calculated based on the estimated draining time. After t_{Halt}, a sender considers that the accumulated queues have drained out. Figure 6 illustrates the rationale under the typical incast scenario with M senders. Assume for flow k, ΔS_k is the total sent out bytes in one period T (i.e., base RTT). ΔA_k is the total acknowledged bytes in the next period. Ideally, all packets sent in one period should be acknowledged in the next period when there is no congestion (i.e., $\Delta A_k = \Delta S_k$). Once congestion happens, the excessive packets build up the queue. The total excessive sent packets are $\sum_{k=1}^{M}(\Delta S_k - \Delta A_k)$. Then the draining time of the bottleneck queue is $\sum_{k=1}^{M}(\Delta S_k - \Delta A_k)/C$, where C is link capacity. Since flow k can only pass through the bottleneck with the rate R_k, the draining time of total packets belonging to flow k in the bottleneck queue can be calculated as $(\Delta S_k - \Delta A_k)/R_k$, which is the expected t_{Halt} of flow k. For clarity, in the following, we use ΔS and ΔA to denote ΔS_k and ΔA_k for flow k, respectively.

Each flow maintains a sequence space to calculate ΔS and ΔA. Assume *snd_una* denotes the first sequence that has been sent but not acknowledged, and *snd_nxt* denotes the next sequence to be sent. *snd_una* is updated every time an ACK is received. When a packet is sent, *snd_nxt* is also updated. Then ΔS is the difference between *snd_nxt* at the end of the current period and *snd_nxt* at the beginning of the current period. ΔA is the difference between *snd_una* at the end of the next period and *snd_una* at the beginning of the next period. When severe congestion causes no ACK to arrive during a period, ΔS should accumulate until an ACK is received.

In detail, ACC senders manipulate ΔS and ΔA among several periods to calculate t_{Halt} and exert source halt as shown in Figure 7: (1) Period T_{i-1} in normal transmission; (2) Period T_i starts receiving ACKs of packets sent in T_{i-1} and sends at the same rate as T_{i-1}; (3) The first period T_{i+1} enters the source halt state and stops transmission; (4) Period T_{i+n} leaves the source halt state and starts transmission. Assume the sender starts a flow and then experiences congestion during T_{i-1}. The available bandwidth is derived from ACK arrival rate, and T_{i-1} period directly uses ACK sequences received in T_i period to obtain the excessive packets in T_{i-1}. Indeed only T_i period will use the excessive rate to infer the excessive packets. The detailed process of exerting source halt is as follows:

- At the end of T_i, the sender identifies the flow as congested and prepares to exert source halt. t_{Halt} is calculated by $(\Delta S - \Delta A)/R$ and the flow halts until over period T_{i+1}. ΔS is obtained during T_{i-1}, while ΔA and R is obtained in T_i. Note that the flow still sends packets at the previous rate during T_i, where excessive packets sent in T_i may also lead to queue buildup.
- At the end of T_{i+1}, the sender already enters the source halt state and stops transmission. The sender should estimate excessive packets sent in T_i and extend the insufficient halt time calculated at the end of T_i. We notice that the excessive packets sent in T_i can be inferred according to how much the sending rate exceeds the available bandwidth, without the need to wait for the arrival of ACKs. Concretely, based on the sending rate R_s in the period before source halt (i.e., sending rate of T_{i-1}) and the current available bandwidth R (i.e., the ACK arrival rate in T_i), the number of excessive sent packets r in T_i equals to ΔS subtracting the estimated pipe capacity (line 7 in Algorithm 1). Finally, t_{Halt} is extended by t_{extend} (line 8-9 in Algorithm 1).
- During T_{i+n}, the sender leaves the source halt state and starts transmission. The sender should only consider ACKs of new packets sent after leaving the source halt state for following ACK-driven rate adjustment. To this end, the sender records *snd_nxt* before entering into the source halt state. For each received ACK, only when the current *snd_una* is greater than the recorded *snd_nxt*, the ACK is considered eligible (line 2 in Algorithm 1).

4.3 Throttling Congested Flows

For congested flows, after leaving the source halt state, the sender utilizes the ACK arrival rate to guide rate decrease (§ 3.2 **Principle 1**). ACK arrival rate (line 3 in Algorithm 1) can reflect the data receiving rate at the receiver [4]. The aggregate receiving rate is the capacity of the bottleneck link. To reduce the disturbance due to congestion in the reverse path, ACKs are sent with higher priority than data packets to avoid a significant queueing delay. The effectiveness of this method has been confirmed in [33]. Besides, the arrival

[4]Currently, we only consider per-packet ACKs for accurately estimating excessive packets and network pipe capacity. However, if there is ACK coalescing, the precise number of excessive packets can still be inferred from cumulative ACK sequences.

Variables	Description
snd_una	The current unacknowledged byte
snd_nxt_start	The next sending byte when the sender is allowed to send, initialized as 0
halt	Whether the sender has halted
R_s	The current sending rate
RI	The rate increment value
$t_{NextPkt}$	The next sending time
N_u	The number of consecutive uncongested periods
N_v	The number of consecutive periods receiving UE
P_{thresh}	The threshold of N_v to throttle victim flows

Table 1: Variables of ACC sender algorithm.

rate of ACKs may be so small that just one ACK arrives in several periods. To address this corner case, the sender also records the inter-arrival time of ACKs. When the inter-arrival time is larger than T, the sender estimates the arrival rate by replacing T with the inter-arrival time.

4.4 Adapting Undetermined Flows

If ACKs marked with UE are received in a period T, the corresponding flow is an undetermined flow (i.e., victim flow). ACC treats victim flows adaptively according to the severity of congestion (§ 3.3 **Principle 2**). Indeed, the duration of receiving UE marks can indicate the severity of congestion. If congestion is not severe, ACC senders will receive few UE marks (e.g., only within one period) as congestion may not spread or can be quickly suppressed due to source halt of congested flows. However, if congestion is severe, congestion spreading may last long and ACC senders are likely to observe UE marks stretching across multiple periods. Thus, after identifying victim flows, ACC senders continuously observe the arriving pattern of UE marks to adjust the sending rate.

Concretely, ACC senders first keep the rate and then reduce the sending rate according to the number of consecutive periods with UE marks. As shown in Algorithm 1 (lines 23-24), if the number of consecutive periods with UE marks exceeds a period threshold P_{thresh}, victim flows will decrease the current rate by half to reduce the injection rate and prevent further congestion spreading. In this way, victim flows avoids blindly decreasing the rate with the side effect of losing throughput, but help alleviate HoL blocking when congestion spreading lasts long.

4.5 Rate Increase for Uncongested Flows

If neither CE nor UE marked ACKs are received during period T, the flow is uncongested and should attempt to increase the sending rate. The principles are as follows:

(1) The increase step of uncongested flows should consider the current sending rate. A flow with a small sending rate should increase more, while a flow with a large sending rate should increase less.

(2) The rate increase process should be gradual at first to avoid causing congestion immediately while aggressive after available bandwidth is considered adequate.

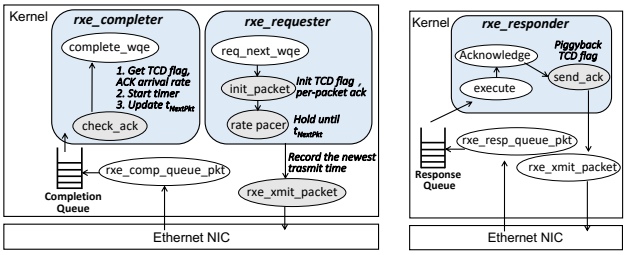
Algorithm 1 ACC sender algorithm.

```
1:  function CONGESTED(recvNum, snd_nxt_start, snd_una)
2:      if halt == false and snd_una > snd_nxt_start then
3:          R = recvNum * MTU/T; //calculating ACK arrival rate
4:          R_s = R;
5:          SourceHalt(R, ΔS, ΔA); //calculating the halt time
6:      else if Halt == true then
7:          r = ΔS − ΔS * R/R_s;
8:          t_extend = r/R;
9:          t_Halt ← t_Halt + t_extend;
10:         t_NextPkt ← t_NextPkt + t_extend;
11:     end if
12: end function
13: function UNCONGESTED
14:     if N_u < 2 then
15:         RI = RI_Low;
16:     else
17:         RI = RI_High;
18:     end if
19:     R_s = R_s + ((lineRate − R_s)/lineRate) * RI;//increasing the sending rate
20:     N_u = N_u + 1;
21: end function
22: function UNDETERMINED
23:     if N_v > P_thresh then
24:         R_s = R_s/2;
25:     end if
26: end function
27: function SOURCEHALT(R, ΔS, ΔA)
28:     if ΔS > ΔA then
29:         t_Halt = (ΔS − ΔA)/R;
30:         t_NextPkt ← t_NextPkt + t_Halt;
31:         halt = true;
32:         snd_nxt_start = snd_nxt;
33:     end if
34: end function
```

ACC senders make rate adjustments following the law in lines 13-21 in Algorithm 1. Generally, the sender increases the sending rate by adding an amount proportional to the difference between $lineRate$ and the current sending rate. Note that $lineRate$ means the maximum rate of NIC which may vary if NIC speed changes. There are two stages for the rate increase. *RI* is the maximum increase amount per period with different values for two stages and is proportional to $lineRate$. For the early stage, *RI* is set as a small value RI_{Low}. After two consecutive increase periods, the maximum increase amount per period *RI* is set as a large value RI_{High}. RI_{Low} and RI_{High} are both proportional to *RI*. In this way, uncongested flows increase gradually at first, then aggressively with a large increase step and finally increase less per period as the sending rate gets closer to the line rate.

4.6 Theoretical Analysis

We build a fluid model of ACC and analyze its performance, including convergence and fairness. The main conclusions are summarized in the following propositions, and the detailed proof is listed in Appendix A.



(a) Sending data and receiving ACKs

(b) Receiving data and sending ACKs

Figure 8: ACC implementation in SoftRoCE.

Proposition 1: When the aggregate sending rate is larger than the bottleneck link bandwidth, ACC can converge the aggregate sending rate towards the bottleneck link bandwidth within one control period T.

Proposition 2: ACC can always guarantee fair bandwidth allocation regardless of the initial sending rate of the flow, i.e.,

$$S_i \to \frac{C}{N}$$

S_i is the sending rate of flow i, C is the bottleneck link bandwidth, N is the number of flows sharing the bottleneck link.

5 Implementation

In this section, we describe the Linux implementation of ACC using SoftRoCE, which is a software implementation of RDMA and fulfills the RoCE NIC function on Ethernet NICs. Figure 8 illustrates the kernel SoftRoCE architecture with our ACC extensions shaded in gray. ACC involves modifications in three main modules, as described below.

`rxe_completer` This module is responsible for processing received ACKs at the sender. We modify the `check_ack` function to implement the ACC algorithm and maintain the state machine of ACC. For the first ACK of each flow, we start a timer for periodically checking ACKs in ACC. The ACC algorithm determines the sending rate of each flow according to current state (congested, undetermined or uncongested) and converts the rate to pacing delay among packets. Based on the latest sending time and pacing delay, the function can calculate $t_{NextPkt}$. $t_{NextPkt}$ is also updated once the flow enters the source halt state (Equation 1).

`rxe_requester` This module is responsible for sending data at the sender. Since the original SoftRoCE does not support congestion control, we add a rate pacer to controlling the transmission time of packets according to $t_{NextPkt}$. The rate pacer continues comparing current time with $t_{NextPkt}$, and source halt can be enforced by holding back the next packet until $t_{NextPkt}$.

`rxe_responder` This module receives data packets and triggers the generation of ACKs. For ACC, after processing the data packets and prepare to generate ACK, it will piggyback the TCD marking to ACK packets.

Discussion on NIC implementation. In RDMA NICs, the QP Context (QPC) maintains for a QP all its contexts, including the DMA states and connection states (e.g., expected packet sequence numbers and sending rate). For ACC, the additional QPC size required for each QP is less than 40B (including ternary states and variables in Table 1), so the total required size for 1K QPs can be less than 40KB, which is acceptable for on-chip memory consumption. Besides, current commercial RDMA NICs already support around 1K rate limiters based on timers [43].

6 Evaluation

6.1 Evaluation Setup

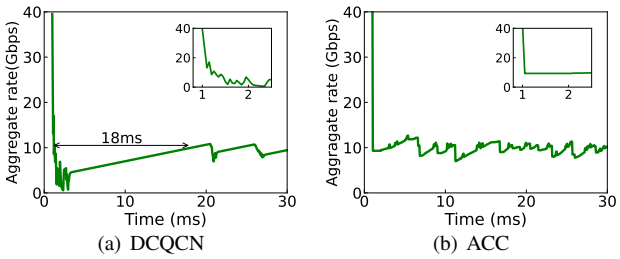
Testbed. We build a testbed consisting of 5 servers connected to one switch. Each server is equipped with AMD Ryzen 9 3950X CPU@3.5GHz, 64GB RAM, an Intel 82599ES 10GbE NIC and runs Ubuntu 20.04 with Linux kernel 5.4.127. All hosts are connected via a 32×100Gbps Tofino switch. We use the default shared buffer setting in the Tofino switch, and also implement ACC and DCQCN [49] in SoftRoCE [36].

Schemes compared. We focus on the comparison among ACC, ECN-based DCQCN [49], RTT-based TIMELY [33] and INT-based HPCC [29]. We use the open-source code of DCQCN, TIMELY and HPCC provided in the HPCC simulator [28] and implement ACC (including the support of TCD in switches). HPCC and DCQCN are state-of-the-art and state-of-the-practice in data centers, respectively. Note that HPCC requires INT support from switches to obtain precise link load. In large-scale simulations, we also compare ACC with DCQCN+TCD and TIMELY+TCD [45], which are congestion control schemes enhanced by accurate congestion detection in TCD. TCD preliminarily proposes to reduce the rate of congested flows aggressively and adjust victim flows gently via heuristically modifying the parameters of existing algorithms.

Network topology. We adopt a fat-tree [5] topology in large-scale simulations. There are 320 servers in 20 racks, 20 aggregation switches and 16 core switches. Each server has a 100 Gbps link connected to ToR switches. All links between core, aggregation and ToR switches are 400 Gbps. Each link has 1μs delay. The switch buffer size is set to 32MB according to real device configurations. PFC is enabled by default and X_{OFF} is set to 512KB.

Workloads. We generate flows according to *Web Search* [6] and *Cache Follower* [40] workloads. The overall load is 80%. We also use *Hadoop* [40] workload (load 50%) adding incast traffic (load 20%). For each incast, 64 randomly selected senders send 20KB to one receiver. These workloads contain typical traffic patterns in data centers with most traffic constituted by few but large flows [6, 11]. *Cache Follower* and *Web Search* workloads contain more heavy flows. *Hadoop* workload is the lightest, around 70% flows smaller than 10KB.

Parameter setting. The core parameters of ACC include the period T, RI (i.e., RI_{Low} and RI_{High}) and P_{thresh}. Since ACC relies on the difference between sent bytes and acknowledged bytes in a period T to figure out excessive data, T must be



(a) DCQCN (b) ACC

Figure 9: Testbed experiment.

the base RTT to ensure that ACKs can arrive during the next period when there is no congestion. RI_{Low} and RI_{High} are the maximum increase amount in each period. As the aggressive source halt state would directly cease transmission, RI_{Low} and RI_{High} can be set large to rapidly occupy available bandwidth. We recommend setting RI_{Low} as 1/250 of the line rate while $RI_{High} = 10RI_{Low}$. For example, RI_{Low} is 400Mbps and RI_{High} is 4Gbps when link capacity is 100Gbps. We recommend setting the queue length threshold in TCD as a small value, i.e., 2MTU. A small but non-zero threshold ensures that congestion can be sensed at the onset while not too sensitive to transient jitter. By default, we set $RI_{High} = 4Gbps$, $RI_{Low} = 400Mbps$ and $P_{thresh} = 1$. The experiments also indicate that our default parameters are proper (Appendix B).

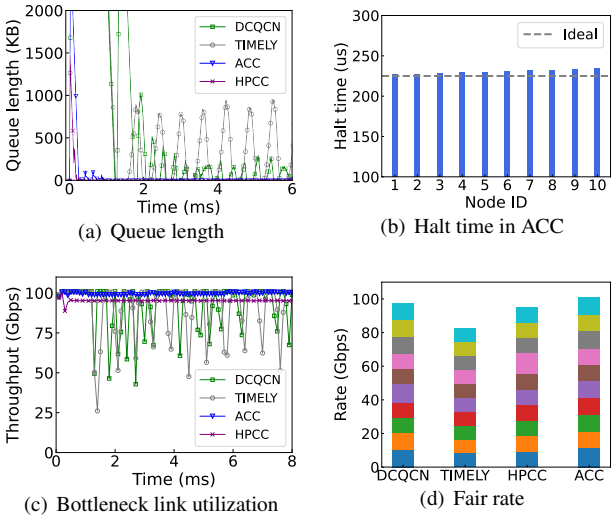
For congestion detection at switches, TCD relies on $max(T_{on})$ to determine the maximum duration of a continuous ON period to detect the ternary state transitions. According to the equation in [45], $max(T_{on})$ is set to 24μs in our large-scale simulations. For HPCC, we use the default parameter with $\eta = 0.95$ and $maxStage = 5$. For DCQCN, we set $K_{min} = 5KB$ and $K_{max} = 200KB$ following the parameter setting in [31,49]. For TIMELY, $\alpha = 0.875$ and $\beta = 0.8$, as suggested in [50].

6.2 Testbed

We implement ACC and DCQCN in SoftRoCE referring to [49]. In our testbed, RTT is around 20μs. Since the period of generating CNPs in DCQCN is 50μs, we set the period T in ACC also to 50μs. The congestion detection results of TCD are equal with ECN in this incast scenario. We configure SoftRoCE such that ACK is generated for every packet[5].

We let each of the four servers start a long flow and record the aggregated sending rates calculated by the congestion control algorithm. Note that in ACC, the sender maintains a sending rate driven by ACK arrival rate and the rate increase algorithm. As shown in Figure 9(a) and Figure 9(b), ACC outperforms DCQCN in terms of fast convergence and stability. After the congestion occurs, DCQCN gradually reduces the sending rate. Due to the heuristic rate regulation, it takes about 0.5ms (~25RTT) to reduce to 10Gbps, and then the aggregation sending rate is reduced to 1Gbps. Finally, it takes around 18ms to converge to the fair share. By contrast, ACC can directly adjust the rate to the proper rate (i.e., 10Gbps) after one period. The subgraph in each figure shows the aggregate sending rate just after the congestion occurs. The rate determined by the ACC algorithm is maintained as 10Gbps under the guidance of the ACK arrival rate. The actual sending is halted because the flows firstly enter the source halt.

[5]We set MTU to 4000B due to the performance issue of SoftRoCE.



(a) Queue length (b) Halt time in ACC

(c) Bottleneck link utilization (d) Fair rate

Figure 10: Convergence, link utilization and fairness.

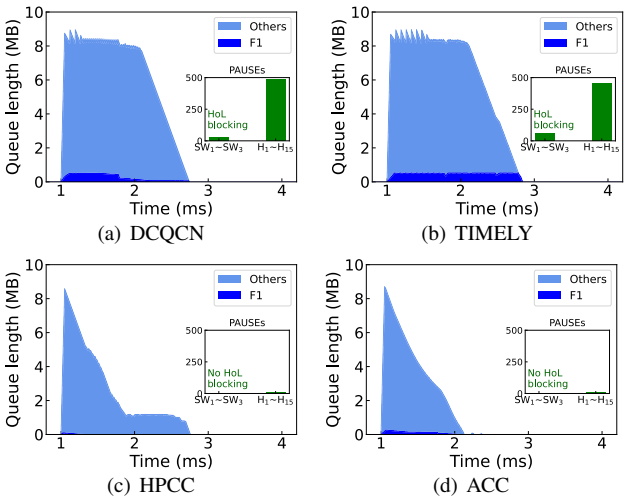
6.3 Microbenchmarks

We first conduct fine-grained simulations to evaluate the basic performance of ACC (e.g., convergence, link utilization and fairness). Then we focus on its ability in dealing with typical issues in lossless Ethernet such as congestion spreading, HoL blocking and deadlock.

ACC can quickly eliminate congestion, maintain near full link utilization and attain fair rate allocation: We let ten source nodes send long-lived traffic to a single destination node through a switch. All links are 100Gbps. Figure 10(a) shows the bottleneck queue length evolution. ACC quickly suppresses the deep queue. Thus congestion is eliminated rapidly. This benefits from the source halt state based on precise excessive packet information in ACC. As depicted in Figure 10(b), the actual halt time of each flow after entering the source halt state is close to the ideal halt time (i.e., the maximum bottleneck queue length divided by link bandwidth), but DCQCN and TIMELY fail to rapidly eliminate the congestion and maintain a low-standing queue.

As for link utilization, both DCQCN and TIMELY incur unstable utilization and underutilize the link due to step-by-step rate adjustment rules. Since HPCC trades high throughput for low latency, it maintains a 95% link utilization. Overall, ACC can achieve steady and near-full link utilization.

As illustrated in Figure 10(d), each flow attains the expected average fair rate (i.e., 10Gbps) with ACC. DCQCN, TIMELY, and HPCC also achieves reasonable fairness. The aggregate flow rate attained in DCQCN, TIMELY, and HPCC are lower than expected. The reasons are that HPCC reserves 5% link bandwidth headroom and the heuristic rate adjustment in



(a) DCQCN (b) TIMELY (c) HPCC (d) ACC

Figure 11: Buffer occupancy at the congested port P4/SW_4 and the number of received PAUSEs.

DCQCN and TIMELY.

ACC can effectively suppress congestion spreading and alleviate HoL blocking under challenging bursty traffic: We adopt the topology in Figure 1 to further evaluate the ability to deal with congestion spreading and HoL blocking under bursty traffic in simulation (N=15). Note that uncongested flow F0 shares links with congested flow F1 in $SW_1 \sim SW_3$. Figure 11 depicts the individual buffer occupancy of F1 and burst at the congested port P4, as well as the number of received PAUSEs in switches and hosts. DCQCN and TIMELY both receive PAUSEs at switches and hosts, indicating that congestion spreading is severe and the congestion tree grows many branches. The backlogged packets of F1 can not be drained rapidly due to sluggish rate decrease in DCQCN and TIMELY. HoL blocking (i.e., F0 is blocked) also occurs because PAUSEs are received at $SW_1 \sim SW_3$ where uncongested flow F0 sharing ports with F1.

With ACC and also HPCC, congestion spreading is suppressed rapidly with many fewer PAUSEs received at both switches and hosts. ACC and HPCC receive no PAUSEs in $SW_1 \sim SW_3$, indicating that there is no HoL blocking and F0 avoids becoming a victim. Only a few PAUSEs are received due to uncontrollable bursts from $H_1 \sim H_{15}$. Besides, HPCC drastically drains the queue at first, then suffers a standing queue for a long period, i.e., from 2ms to 2.7ms, as illustrated in Figure 11(c). This is because the total sending rate has matched the bottleneck bandwidth while there are still backlogged packets. As shown in Figure 11(d), ACC drains backlogged packets at the congested port at the maximum rate, with 2ms to empty the long-standing queue. This is because ACC senders timely enter the source halt state and refer to the precise information of excessive packets.

ACC is resilient against deadlock: To validate the significance of congestion control in preventing deadlock in lossless Ethernet, we adopt a typical topology as illustrated in Figure 12, which is a common unit in Clos topology. To emulate the typical scenarios where deadlock is prone to appear, we artificially let link L0-T3 and link L1-T0 fail to form a CBD. There are eight servers under each ToR switch. We simulate ACC, DCQCN, TIMELY, and HPCC with *Web Search* workload [6]. The load is 60%. We run simulations 50 times with different traffic traces, and every simulation lasts for 100ms. Table 2 shows the fraction of deadlock runs. We identify a deadlock by finding whether there are flows with infinite completion times. Among 50 simulations, there is no deadlock in ACC and HPCC. However, DCQCN and TIMELY encounter deadlocks 3 and 37 times, respectively. With fast convergence, ACC is resilient against deadlock.

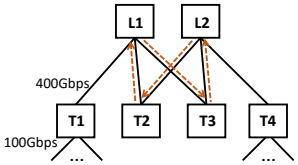


Figure 12: Topology

Scheme	Fraction
DCQCN	6%
TIMELY	74%
HPCC	0%
ACC	0%

Table 2: Fraction of deadlock

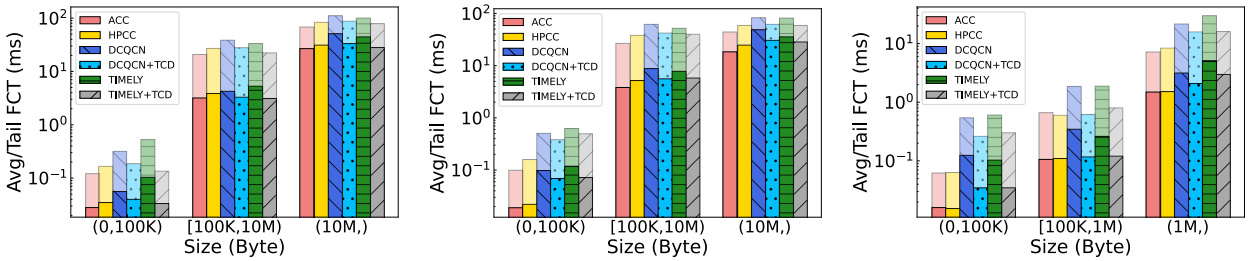
6.4 Large-Scale Simulations

We conduct large-scale simulations to evaluate the overall FCT performance of ACC. For each workload, we classify flows into small (<100KB), medium (≥100KB and < 10MB), and large (>10MB) flows. Note that in *Hadoop* workload, the size of large flows is larger than 1MB.

ACC achieves low latency for small and medium flows: Figure 13(a) exhibits the average FCT and 99th percentile FCT under *Web Search* workload. On the whole, DCQCN and TIMELY have undesirable performance. For DCQCN, the average FCT for small flows is 3.3× of ACC, and the 99th percentile FCT is 11.5× of ACC. This is because ACC can quickly converge to the appropriate rate through the ACK arrival rate. Besides, the FCT performance of ACC is better than HPCC. For small flows, ACC reduces the average FCT by 29% compared to HPCC, and the 99th percentile FCT drops by 40%. The performance gain on small flows comes from the source halt that directly stops the transmission of the congested flow so that the queues can be rapidly drained.

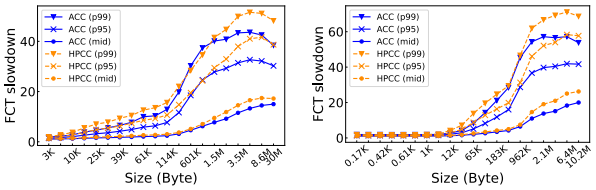
Under *Cache Follower* workload, ACC achieves better FCT performance than HPCC, with lower FCT even for small and medium flows. For medium flows, ACC reduces the 99th percentile FCT by 41% compared with HPCC. For DCQCN, the average FCT and 99th percentile FCT are 2.4× and 1.7× larger than ACC, respectively.

Under *Hadoop* workload, the FCT performance of ACC is close to HPCC. As shown in Figure 13(c), ACC achieves comparable FCT performance with HPCC for small and medium flows. The flow sizes in *Hadoop* workload are smaller, with 55% of flows being less than 1KB. Such flow size distribution induces less persistent congestion than the other two workloads. As a result, the queuing delay dominates the FCT for



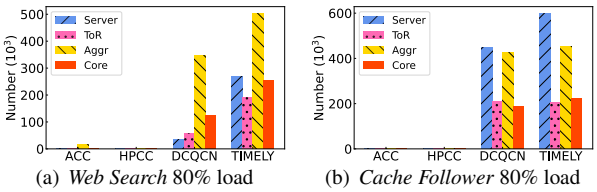
(a) *Web Search* 80% load (b) *Cache Follower* 80% load (c) *Hadoop* (50% load)+*incast* (20% load)

Figure 13: Average and tail (99th percentile) FCT performance under various workloads.



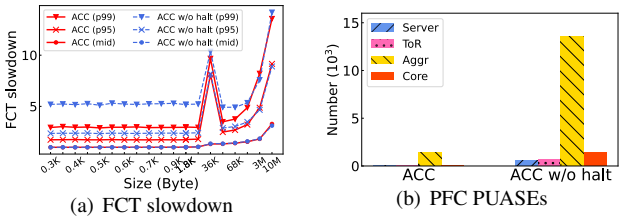
(a) *Web Search* 80% load (b) *Cache Follower* 80% load

Figure 14: FCT slowdown performance



(a) *Web Search* 80% load (b) *Cache Follower* 80% load

Figure 15: Number of PFC PAUSEs



(a) FCT slowdown (b) PFC PUASEs

Figure 16: Impact of source halt

small flows. As HPCC sacrifices link utilization leading to queue underflow, thus benefiting FCT performance for small and medium flows.

ACC does not sacrifice throughput of large flows: Figure 14(a) and Figure 14(b) depict the detailed FCT slowdown under *Cache Follower* and *Web Search* workload. For numerous small and medium flows, ACC achieves comparable or even better 99th percentile FCT slowdown than HPCC under both workloads while always attaining better performance than HPCC for relatively large flows. For example, for flows larger than 30MB/10MB, the 99th percentile FCT slowdown with ACC is 26% and 27% lower than HPCC, respectively. In Figure 13(b), ACC also outperforms HPCC by 30% for the 99th percentile FCT of large flows. ACC does not sacrifice the throughput of large flows because it precisely drains out excessive packets and does not underflow the queues to waste link bandwidth.

ACC outperforms DCQCN+TCD and TIMELY+TCD enhanced with heuristic rules: Figure 13 also demonstrates that with accurate congestion detection and heuristic rules on adjusting the rate of congested/victim flows, both DCQCN+TCD and TIMELY+TCD improve the average and tail FCT performance compared with DCQCN and TIMELY. However, since DCQCN+TCD and TIMELY+TCD still heuristically decrease the rate of congested flows following the original paradigm, they can not drain congested queues rapidly and allocate the proper rate precisely under various workloads. Thus, DCQCN+TCD and TIMELY+TCD may suffer from slow convergence and long-standing queues. For instance, under *Cache Follower* workload, ACC improves tail FCT performance of small flows by 3.9× and 5.1× compared with DCQCN+TCD and TIMELY+TCD, respectively.

ACC greatly reduces PFC PAUSE generation: The number of received PAUSEs in different layers under *Web Search* and *Cache Follower* workload is drawn in Figure 15. The results show that ACC almost has no trigger of PFC messages because of fast convergence and the ability to maintain low-standing queues. Thus there is little HoL blocking. In *Web Search* workload, for DCQCN and TIMELY, most PAUSEs are received in the aggregation layer because the challenging incast pattern happens in the ToR layer. In *Cache Follower* workload, most PAUSEs are received at the aggregation layer and servers, indicating that congestion spreading is severe hence many innocent senders are affected.

Source halt benefits low latency for small flows and reduces the risk of PFC PAUSEs: Aiming at understanding the significance of source halt, we take a closer look at the comparison between ACC and ACC w/o halt. ACC w/o halt removes the source halt state and directly decreases the sending rate of congested flows according to the ACK arrival rate. Figure 16(a) manifests that ACC w/o halt can achieve similar performance for medium and large flows, while small flows suffer from larger FCT. Small flows experience undesirable queueing delays as congested queues can not be rapidly drained out. ACC w/o halt can still fulfill fast convergence owing to the guidance of ACK arrival rate. Besides, as illustrated in Figure 16(b), since there are more long-standing queues, the risk of triggering PFC PAUSEs increases. ACC significantly reduces the PFC PAUSEs at each layer compared with ACC w/o halt. The congestion spreading is also restricted

effectively, with many fewer PAUSEs received at the servers, ToR, and the core layer.

7 Related Work

Lossless Ethernet vs. Lossy Ethernet. There has been an ongoing discussion about lossless and lossy Ethernet in RDMA-based datacenter. Lossless Ethernet can benefit application performance but comes with side effects of deploying PFC (i.e., HoL blocking, congestion spreading, deadlock, etc). However, ACC reveals that besides its side effects, PFC also gives a unique packet conservation property which can facilitate precise congestion control and in turn mitigate the side effects of PFC. In contrast, although lossy Ethernet can essentially avoid the side effects of PFC, it may suffer from latency degradation due to packet losses and imposes higher implementation requirements on RDMA NICs (e.g., more complex loss recovery and reorder logic). Thus, existing efforts can be mainly classified into two lines: *(i) Developing congestion control (or flow control) to minimize the side effects of PFC*: For example, CC schemes including DCQCN [49], PCN [13], TCD/TCD-MQ [45,46], ACC, etc, and new link-layer flow control like deadlock-free GFC [38]. *(ii) Developing advanced loss recovery to improve RDMA performance*: For example, IRN [34] first introduces selective retransmission into RDMA NICs. TLT [30] proposes to reduce timeouts by protecting important packets. SLR [32] utilizes switches to send loss notifications to request fast retransmissions and is compatible with default Go-back-N retransmission.

Congestion control in lossless interconnects. Besides lossless Ethernet, there are other lossless interconnects, e.g., InfiniBand [9] and Fibre Channel [8]. For example, InfiniBand deploys credit-based flow control to guarantee no packet loss, which also suffers from HoL blocking, congestion spreading, and deadlock problems [9,22,41]. Despite the fact that ACC is designed for lossless Ethernet employing PFC, we think our key insights (e.g., utilizing the ACK-driven paradigm to infer excessive packets and throttled rate) are also suitable for congestion control in other lossless interconnects with packet conservation property.

Receiver-driven and other advanced congestion controls. Recently several receiver-driven congestion controls have been proposed, such as ExpressPass [14], pHost [18], NDP [23], and Homa [35]. The core of receiver-driven congestion control is “request and allocation” style transport [24]. These receiver-driven schemes are first proposed in lossy Ethernet, where new flows blindly transmit unscheduled packets in the first RTT. Hence congestion may also occur with the risk of packet loss. Besides, NDP and ExpressPass utilize CP [12] or explicitly drop credits to enforce a bounded queue. PCN [13] is a receiver-driven congestion control scheme designed in lossless Ethernet while only utilizing the receiving rate at the receiver to guide rate decrease, thus accumulated queues may still be eliminated slowly and HoL blocking can not be suppressed rapidly. There are also other advanced congestion controls designed for datacenters. For instance, PowerTCP [4] proposes to fulfill fine-grained congestion control by adapting to *Power* (i.e., bandwidth-window product) based on INT. Bolt [7] leverages sub-RTT signals provided by programmable switches to enable reacting to congestion faster than the RTT control loop, which realizes low queueing and benefits minimal packet loss under bursty workloads. Besides, Fasspass [37] first proposes to delegate all control (i.e., when each packet should be transmitted and what path it should follow) to a centralized arbiter.

ACC stands out as it fully builds on the simple yet powerful packet conservation law in lossless Ethernet while achieving low latency and high throughput simultaneously.

Source halt. On-Ramp [31] proposes to pause the source actively to deal with transient and equilibrium tension in traditional lossy Ethernet. The source PAUSE technique in On-Ramp is similar to the source halt state in ACC. However, On-Ramp relies on clock synchronization [20] to obtain accurate one-way delay measurement and roughly estimates the pause time without knowing network pipe capacity. The source halt state in ACC precisely drains out accumulated packets of congested flows while only relying on the ACK time series. To the best of our knowledge, none of the existing congestion control schemes in lossless Ethernet proposes to accurately and rapidly drain out backlogged packets.

8 Conclusion

In this paper, we revisit congestion control for lossless Ethernet from the perspective of unlocking its intrinsic properties, e.g., *packet conservation*. We propose a new congestion control called ACC, which essentially treats the ACKs as accurate "tokens" of the network pipe to guide the traffic injection rate and timing, i.e., waiting for backlogged packets to drain out and then sending with the rate corresponding to network pipe capacity. Extensive experiments show that ACC alleviates thorny issues in lossless Ethernet, such as congestion spreading, HoL blocking, and deadlock. Besides, ACC achieves low latency and high throughput simultaneously under various workloads and even outperforms state-of-the-art congestion control schemes. We believe that the philosophy of exploiting intrinsic properties opens new avenues for rethinking congestion control and other traffic management mechanisms in lossless Ethernet.

Acknowledgements

We thank the anonymous reviewers and our shepherd Naveen Kr. Sharma, for their valuable comments and helpful suggestions. This work is supported in part by the National Key Research and Development Program of China (No. 2022YFB2901404), and by National Natural Science Foundation of China (NSFC) under Grant No. 62132007, No. 62221003 and No. 62302055, and by the China Post-Doctoral Science Foundation under Grant No. 2023M730172, as well as gifts from Huawei. Corresponding author: Qingkai Meng and Fengyuan Ren.

References

[1] Ieee 802.1 qbb - priority-based flow control. http://www.ieee802.org/1/pages/802.1bb.html, 2010.

[2] In-band network telemetry (int) dataplane specification. https://p4.org/p4-spec/docs/INT_v2_1.pdf, 2020.

[3] Nvm express over fabrics revision 1.1a. https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1a-2021.07.12-Ratified.pdf, 2021.

[4] Vamsi Addanki, Oliver Michel, and Stefan Schmid. PowerTCP: Pushing the performance limits of datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 51–70, Renton, WA, April 2022. USENIX Association.

[5] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, page 63–74, New York, NY, USA, 2008. Association for Computing Machinery.

[6] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[7] Serhat Arslan, Yuliang Li, Gautam Kumar, and Nandita Dukkipati. Bolt: Sub-RTT congestion control for Ultra-Low latency. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 219–236, Boston, MA, April 2023. USENIX Association.

[8] Fibre Channel Industry Association. Fibre channel. https://fibrechannel.org/, 2022.

[9] InfiniBand Trade Association. Infiniband architecture specification volume 2 release 1.3.1. https://cw.infinibandta.org/document/dl/8125, 2016.

[10] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, mar 2017.

[11] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, page 267–280, New York, NY, USA, 2010. Association for Computing Machinery.

[12] Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. Catch the whole lot in an action: Rapid precise packet loss notification in data center. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 17–28, 2014.

[13] Wenxue Cheng, Kun Qian, Wanchun Jiang, Tong Zhang, and Fengyuan Ren. Re-architecting congestion management in lossless ethernet. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 19–36, 2020.

[14] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 239–252, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[16] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, jan 2008.

[17] Nandita Dukkipati, Neal Cardwell, Yuchung Cheng, and Matt Mathis. Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses. Internet-Draft draft-dukkipati-tcpm-tcp-loss-probe-01, Internet Engineering Task Force, February 2013. Work in Progress.

[18] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery.

[19] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533. USENIX Association, April 2021.

[20] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 81–94, Renton, WA, April 2018. USENIX Association.

[21] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[22] M. Gusat, D. Craddock, W. Denzel, T. Engbersen, N. Ni, G. Pfister, W. Rooney, and J. Duato. Congestion control in infiniband networks. In *13th Symposium on High Performance Interconnects (HOTI'05)*, pages 158–159, 2005.

[23] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 29–42, New York, NY, USA, 2017. Association for Computing Machinery.

[24] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A building block for proactive transport in datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 422–434, New York, NY, USA, 2020. Association for Computing Machinery.

[25] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Deadlocks in datacenter networks: Why do they form, and how to avoid them. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, page 92–98, New York, NY, USA, 2016. Association for Computing Machinery.

[26] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Tagger: Practical pfc deadlock prevention in data center networks. *IEEE/ACM Trans. Netw.*, 27(2):889–902, apr 2019.

[27] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *ACM SIGCOMM*, pages 514–528, 2020.

[28] Yuliang Li. HPCC NS-3 simulator. https://github.com/alibaba-edu/High-Precision-Congestion-Control, 2019.

[29] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Hwijoon Lim, Wei Bai, Yibo Zhu, Youngmok Jung, and Dongsu Han. Towards timeout-less transport in commodity datacenter networks. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 33–48, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Shiyu Liu, Ahmad Ghalayini, Mohammad Alizadeh, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. Breaking the Transience-Equilibrium nexus: A new approach to datacenter packet transport. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 47–63. USENIX Association, April 2021.

[32] Qingkai Meng, Yiran Zhang, Shan Zhang, Zhiyuan Wang, Tong Zhang, Hongbin Luo, and Fengyuan Ren. Switch-assistant loss recovery for rdma transport control. *IEEE/ACM Transactions on Networking*, pages 1–16, 2023.

[33] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *SIGCOMM Comput. Commun. Rev.*, 45(4):537–550, aug 2015.

[34] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 313–326, New York, NY, USA, 2018. Association for Computing Machinery.

[35] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery.

[36] NVIDIA. How To Configure Soft-RoCE. https://enterprise-support.nvidia.com/s/article/howto-configure-soft-roce, 2023.

[37] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: a centralized "zero-queue" datacenter network. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 307–318, New York, NY, USA, 2014. Association for Computing Machinery.

[38] Kun Qian, Wenxue Cheng, Tong Zhang, and Fengyuan Ren. Gentle flow control: Avoiding deadlock in lossless networks. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 75–89, New York, NY, USA, 2019. Association for Computing Machinery.

[39] Sudarsanan Rajasekaran, Manya Ghobadi, Gautam Kumar, and Aditya Akella. Congestion control in machine learning clusters. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 235–242, 2022.

[40] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. *SIGCOMM Comput. Commun. Rev.*, 45(4):123–137, aug 2015.

[41] J.R. Santos, Y. Turner, and G. Janakiraman. End-to-end congestion control for infiniband. In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428)*, volume 2, pages 1123–1133 vol.2, 2003.

[42] Brent Stephens, Alan L. Cox, Ankit Singla, John Carter, Colin Dixon, and Wesley Felter. Practical dcb for improved data center networks. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, pages 1824–1832, 2014.

[43] Zilong Wang, Xinchen Wan, Chaoliang Zeng, and Kai Chen. Accurate and scalable rate limiter for rdma nics. In *Proceedings of the 7th Asia-Pacific Workshop on Networking*, APNET '23, page 15–20, New York, NY, USA, 2023. Association for Computing Machinery.

[44] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, oct 2016.

[45] Yiran Zhang, Yifan Liu, Qingkai Meng, and Fengyuan Ren. Congestion detection in lossless networks. In *Proceedings of the ACM SIGCOMM Conference*, SIGCOMM '21, page 370–383, New York, NY, USA, 2021. Association for Computing Machinery.

[46] Yiran Zhang, Qingkai Meng, Yifan Liu, and Fengyuan Ren. Revisiting congestion detection in lossless networks. *IEEE/ACM Transactions on Networking*, 31(5):2361–2375, 2023.

[47] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. Aequitas: admission control for performance-critical rpcs in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 1–18, 2022.

[48] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2015.

[49] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery.

[50] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. Ecn or delay: Lessons learnt from analysis of dcqcn and timely. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 313–327, New York, NY, USA, 2016. Association for Computing Machinery.

[51] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. Hoplite: Efficient and fault-tolerant collective communication for task-based distributed systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 641–656, New York, NY, USA, 2021. Association for Computing Machinery.

Parameter	Definition
C	Bottleneck link bandwidth
N	The number of flows
T	Control period
k	The kth period
RI	Rate increment
$S_i(k)$	Sending rate of flow i at kth period
$R_i(k)$	Actual sending rate of flow i considering the source halt state at kth period
$Q(k)$	Bottleneck queue length at kth period
$\varphi_i(k)$	Allocated bandwidth fraction of flow i at kth period, $\varphi_i(k) = \frac{S_i(k)}{\sum S_i(k)}$

Table 3: Parameter list

APPEDNDIX

A. Theoretical Analysis

In this section, we analyze the convergence and fairness of ACC in theory. The key parameters are listed in Table 3. We assume that the receiver will generate ACK for each received packet and the delay of generating ACK is negligible.

Convergence: Without loss of generality, assuming that the switch queue corresponding to the bottleneck link is initially empty, the sending rate of N flows is arbitrary. There are following two cases:

(1) $\sum S_i(0) > C$: If the aggregate sending rate of the initial N flows exceeds the bottleneck link bandwidth, congestion occurs and queues start to accumulate. All congested flows will enter the source halt state:

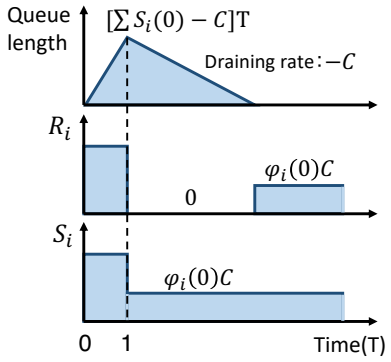
$$\begin{cases} Q(1) = (\sum S_i(0) - C)T \\ R_i(1) = 0 \\ S_i(1) = \varphi_i(0)C \end{cases} \tag{2}$$

As shown in Figure 17(a), after each flow enters the source halt state, the actual sending rate $R_i(1) = 0$. Flow i stops transmission to wait for the queue to be emptied. During this period, the sending rate $S_i(1)$ derived from ACK arrival rate is $\varphi_i(0)C$, and is recorded for following transmission. The congested queue drains at the rate C, and needs $\lceil \frac{\sum S_i(0) - C}{C} \rceil$ periods to approach 0:

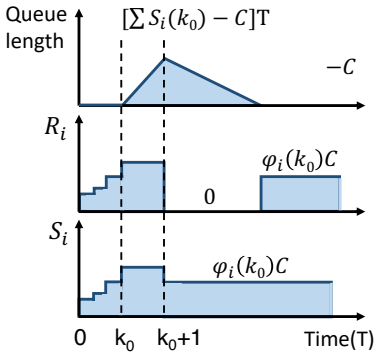
$$\begin{cases} Q(k) = (\sum S_i(0) - C)T - (k-1)CT \\ R_i(k) = \varphi_i(0)C \\ S_i(k) = \varphi_i(0)C \end{cases} \tag{3}$$

Thus, we derive $\sum S_i(k) = \sum \varphi_i(0)C = \sum \frac{S_i(0)}{\sum S_i(0)} C = C$, indicating that the aggregate sending rate $\sum S_i(k)$ converges to C within one period. After leaving the source halt state, $R_i(k) = S_i(k)$ and flow i starts normal transmission.

(2) $\sum S_i(0) \leq C$: If the initial aggregate sending rate of N flows is less than or equal to the bottleneck link bandwidth,



(a) $\sum S_i(0) > C$



(b) $\sum S_i(0) \leq C$

Figure 17: Convergence of ACC

all flows will try to increase the sending rate. Eventually, the aggregate sending rate of N flows will exceed the bottleneck bandwidth after k_0 periods, then the evolution behavior of the congested queue and aggregate sending rate of N flows is consistent with those in the previous case. Figure 17(b) shows the evolution of bottleneck queue length, R_i and S_i.

To sum up, when the aggregate sending rate is larger than the bottleneck link bandwidth, ACC can converge to the bottleneck link bandwidth within one control period T and eliminate queue accumulation at the fastest speed of the bottleneck link bandwidth.

Fairness: Assuming that the above convergence process ends at the beginning of the k_1 period, when the switch queue has been emptied, and the sending rate of the flow i starts to increase from $\varphi_i(k_0)C$. Let $r = \frac{RI}{C}$, we have

$$\begin{cases} Q(k_1) = 0 \\ S_i(k_1) = (1-r)\varphi_i(k_0)C + rC \\ R_i(k_1) = S_i(k_1) \end{cases} \tag{4}$$

We have

$$\varphi_i(k_1) = \frac{S_i(k_1)}{\sum S_i(k_1)} = \frac{(1-r)\varphi_i(k_0) + r}{1 - r + rN} \tag{5}$$

Since $N > 1$, it is obvious that $\sum S_i(k_1) = (1 + rN - r)C > C$. The link is congested, and the flow i will slow down in the next period and enter the source halt state:

$$\begin{cases} Q(k_1+1) = (rN-r)CT \\ S_i(k_1+1) = \varphi_i(k_1)C \\ R_i(k_1+1) = 0 \end{cases} \tag{6}$$

The switch queue will be drained at a rate of C, during which the sending rate S_i is kept unchanged, we have

$$\begin{cases} Q(k_1+m) = (rN-r-m+1)CT \\ S_i(k_1+m) = \varphi_i(k_1)C \\ R_i(k_1+m) = 0 \end{cases} \tag{7}$$

Let $M = \lceil rN - r + 1 \rceil$. Finally, at the beginning of the $k_1 + M$ period, the queue length is 0, and the flow i will start to increase the sending rate again. We have

$$\begin{cases} Q(k_1+M) = 0 \\ S_i(k_1+M) = (1-r)\varphi_i(k_1)C + rC \\ R_i(k_1+M) = S_i(k_1+M) \end{cases} \tag{8}$$

We have

$$\varphi_i(k_1+M) = \frac{S_i(k_1+M)}{\sum S_i(k_1+M)} = \frac{(1-r)\varphi_i(k_1)+r}{1-r+rN} \tag{9}$$

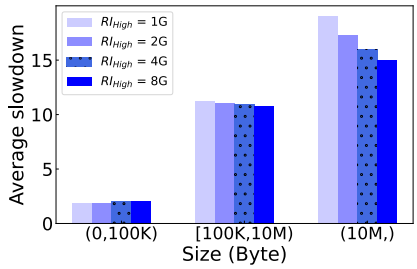
Comparing Equation (5) and Equation (9), it can be seen that ACC will repeat the process of rate increase for one period and rate decrease for $M-1$ periods. At the same time, according to Equation (5) and Equation (9), we can calculate $\varphi_i(k_1+kM)$ as follows:

$$\begin{aligned} \varphi_i(k_1+kM) &= \frac{(1-r)\varphi_i(k_1+(k-1)M)+r}{1-r+rN} \\ &= \frac{1-r}{1-r+rN}\varphi_i(k_1+(k-1)M) + \frac{r}{1-r+rN} \\ &\cdots \\ &= \left(\frac{1-r}{1-r+rN}\right)^{k+1}\varphi_i(k_0) + \\ &\sum_{j=0}^{k}\left(\frac{1-r}{1-r+rN}\right)^{j}\left(\frac{r}{1-r+rN}\right) \end{aligned} \tag{10}$$

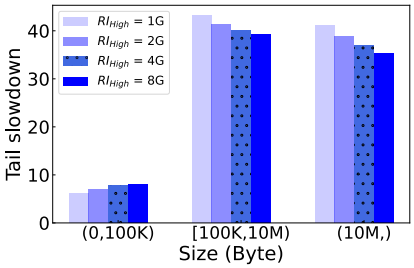
When $k \to \infty$, as $\frac{1-r}{1-r+rN} < 1$, so $\left(\frac{1-r}{1-r+rN}\right)^{k+1}$ approaches zero. We have:

$$\varphi_i(k_1+kM) \to \frac{\frac{r}{1-r+rN}}{1-\left(\frac{1-r}{1-r+rN}\right)} = \frac{1}{N} \tag{11}$$

Equation (11) indicates that ACC can achieve fair rate allocation.



(a) Average slowdown



(b) The 99th percentile slowdown

Figure 18: Parameter sensitivity of *RI*, 80% load (*WebSearch*)

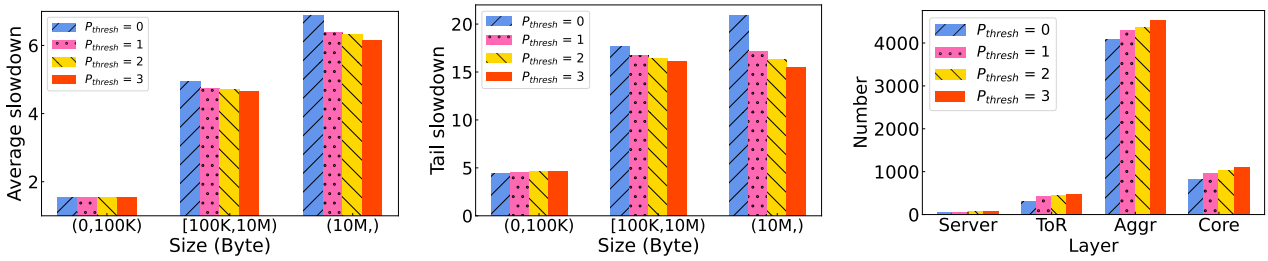
B. Supplementary of Evaluations

To better understand the parameter sensitivity and link utilization performance of ACC, we supplement experiment results in this section. In §B.1, we present the parameter sensitivity of *RI* and P_{thresh}. After that, we present the simulation results under typical bursty traffic (§B.2).

B.1 Parameter Sensitivity of *RI* and P_{thresh}

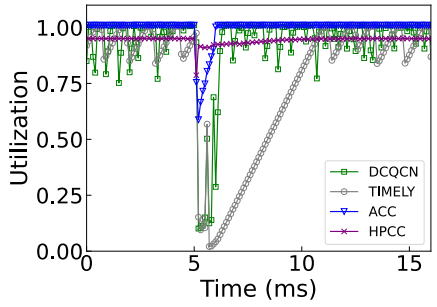
Parameter *RI*: In ACC, the key parameter in the rate increase process is *RI*. A larger *RI* indicates a more aggressive rate increase. To explore the performance sensitivity of ACC on *RI*, we use the same *Web Search* workload and topology as in Figure 14(a) but vary *RI* between 1G and 8G. Figure 18 shows the average FCT slowdown of small, medium, and large flows. On the whole, a larger *RI* yields higher throughput for large flows but may introduce larger FCT for small and medium flows. Due to the source halt state and ACK arrival rate guidance in ACC, the sending rate can quickly drop once the aggressive rate increase induces congestion. As a result, the average FCT slowdown of small and medium flows is not sensitive to *RI*. The overall results indicate that our recommended value of *RI* is reasonable.

Parameter P_{thresh}: In ACC, P_{thresh} controls the rate regulation of victim flows. Victim flows will keep its rate until P_{thresh} consecutive periods. A smaller P_{thresh} will enforce the victim flow to throttle its rate earlier to help mitigate the congestion spreading. To explore the performance sensitivity of ACC on P_{thresh} especially when HoL blocking happen frequently, we run a challenging workload consisting of foreground traffic

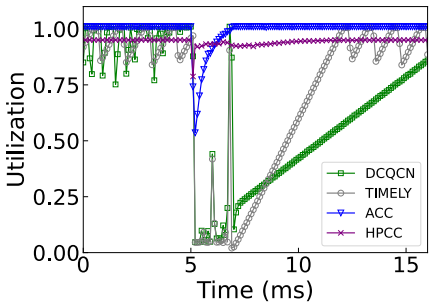


(a) Average slowdown (b) The 99th percentile slowdown (c) The number of PFC PAUSEs

Figure 19: Parameter sensitivity of P_{thresh}, 80% load (*WebSearch* + incast)



(a) $N = 10$



(b) $N = 20$

Figure 20: Utilization of link <SW_3-SW_4>. Bursts start at time 5ms. Topology is Figure 1(a).

and background traffic: 60% *Web Search* workload as the foreground traffic with 20% load of 64-1 incast workload as the background traffic. For each incast, 64 randomly selected senders send 50KB to one receiver. We vary P_{thresh} from 0 to 3. Note that $P_{thresh} = 0$ represents that victim flows will always reduce half of the sending rate per T.

Figure 19(a) and Figure 19(b) show the average and 99th percentile FCT slowdown of the foreground *WebSearch* workload. As P_{thresh} increases, medium flows and large flows have better FCT slowdown performance. For example, for large flows, the 99th percentile FCT slowdown with $P_{thresh} = 0$ is 1.22× of when $P_{thresh} = 1$. For medium and large victim flows, by keeping the sending rate until P_{thresh} periods, they lose less throughput with a larger P_{thresh}. While the FCT slowdown performance of small flows with size less than 10KB are mainly determined by queueing delay. As a result, the average and tail FCT slowdown of small flows are not sensitive to P_{thresh}.

However, as shown in Figure 19(c), a larger P_{thresh} may trigger more PFC PAUSEs at each layer, indicating heavier HoL blocking and more risks of PFC. This is because with a large P_{thresh}, victim flows react very late to congestion spreading, at which HoL blocking already occurs for a long time and propagations to multiple switches. Indeed, throttling the rate of victim flows early can aggressively reduce the injected traffic around the blocked switch ports, thus speeding up the alleviation of HoL blocking. On the whole, the results indicate that our recommended value of P_{thresh} (i.e., 1) is proper, which can balance the FCT performance and alleviation of HoL blocking.

B.2 Link Utilization

We adopt the same topology and traffic pattern as in Figure 1(a) to evaluate the link utilization performance especially under bursty traffic. Note that uncongested flow F0 shares links with congested flow F1 in SW_1∼SW_3. At time 5ms, concurrent bursts with size of 64KB start. Then the congested flow F1 should be throttled while uncongested flow F0 should be unaffected and occupy available bandwidth. We use $N = 10$ and $N = 20$ to introduce different degree of burstiness.

As illustrated in Figure 20(a), after time 5ms, both DCQCN and TIMELY incur low link utilization lasting for around 1.5ms and 6ms, respectively. Figure 20(b) shows that when $N = 20$, DCQCN and TIMELY get worse burst tolerance performance, with around 7ms and 12ms to recover to full link utilization. Due to step-by-step rate adjustment rules of DCQCN and TIMELY, they can not eliminate congestion at SW_4 quickly and congestion spreads to upstream SW_3, thus link <SW_3-SW_4> is blocked. The low link utilization is also attributed to the uncongested flow F0 being throttled mistakenly due to misleading queue-based (or delay-based) congestion signals. ACC incurs not full link utilization lasting for about 1.5ms ($N = 10$) and 2ms ($N = 20$), which is the time required for uncongested flow F0 to increase the sending rate. HPCC can ramp up quickly after bursts start but only maintains a 95% link utilization in steady state.

Autothrottle: A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices

Zibo Wang[†§], Pinghe Li[¶], Chieh-Jan Mike Liang[§], Feng Wu[†], Francis Y. Yan[§]
[†]University of Science and Technology of China, [¶]ETH Zurich, [§]Microsoft Research

Abstract

Achieving resource efficiency while preserving end-user experience is non-trivial for cloud application operators. As cloud applications progressively adopt microservices, resource managers are faced with two distinct levels of system behavior: end-to-end application latency and per-service resource usage. Translating between the two levels, however, is challenging because user requests traverse heterogeneous services that collectively (but unevenly) contribute to the end-to-end latency. We present *Autothrottle*, a bi-level resource management framework for microservices with latency SLOs (service-level objectives). It architecturally decouples application SLO feedback from service resource control, and bridges them through the notion of performance targets. Specifically, an application-wide learning-based controller is employed to periodically set performance targets—expressed as CPU throttle ratios—for per-service heuristic controllers to attain. We evaluate Autothrottle on three microservice applications, with workload traces from production scenarios. Results show superior CPU savings, up to 26.21% over the best-performing baseline and up to 93.84% over all baselines.

1 Introduction

To ensure a seamless end-user experience, many user-facing latency-sensitive applications impose an SLO (service-level objective) on the end-to-end latency. Traditionally, cloud application operators resort to resource over-provisioning to avoid SLO violations, yet doing so unnecessarily wastes resources [21, 32]. Previous efforts have demonstrated significant savings if the excess resources could be harvested or reclaimed for co-located applications in a multi-tenant environment [13, 29, 30, 39, 57, 62].

A key enabler for such resource saving is SLO-targeted resource management. Its goal is to continuously minimize the total resources allocated, while still satisfying the end-to-end latency SLO. Unfortunately, modern cloud applications can be beyond current resource managers, due to the progressive shift from monolithic to distributed architecture [11, 25, 31, 40, 64]. They are a topology of cloud-native services or microservices[1], and user requests traverse a chain of execution dependencies among services of logic, databases, and machine learning (ML) model serving. Notably, this creates distinct levels of system behavior—the macro perspective reveals the end-to-end performance (e.g., user request latencies) and SLO, and the micro perspective is scoped to local measurements (e.g., service CPU usage) and resource control.

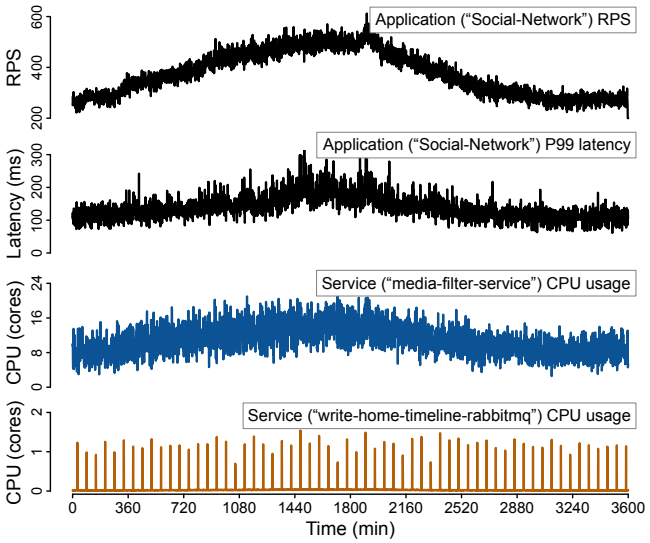


Figure 1: Individual microservices (bottom two panels) can exhibit vastly different resource usage patterns and short-term fluctuations. In addition, they do not necessarily have a strong correlation with the end-to-end application-level measurements (top two panels).

The distributed nature of microservices brings unique implications to resource management. First, heterogeneous services can exhibit vastly different resource usage patterns, due to how various user requests stress each service. The bottom two panels in Figure 1 contrast the CPU usage of two services in an application, Social-Network [22]. Second, application performance and per-service resource usage are measurements at different levels, without necessarily exhibiting a strong correlation (top two vs. bottom two panels in Figure 1). Translating between them requires knowing each user request's actual resource requirements and its service-to-service execution flow. Moreover, this execution chain incurs undesirable delays in observing effects of allocation changes on the end-to-end performance, further complicating resource management.

At first glance, it appears that resource managers could implicitly address the distributed nature by either considering application-wide dependencies [17, 28, 43, 44, 63] or employing heuristics with operator-defined rules on individual services [5]. The former centralizes resource control with a

[1] In this paper, we use "services" and "microservices" interchangeably.

global view of the service topology, while the latter delegates control to each service that acts on locally observed resource usage. Nevertheless, maintaining a global view is susceptible to topology changes and evolution [50, 64], and relying solely on local speculations may not achieve global optimality.

Instead, we embrace the distinct levels of distributed system behavior, and architecturally decouple mechanisms of application-level SLO feedback and service-level resource control. We design *Autothrottle*, a bi-level learning-assisted resource management framework for SLO-targeted microservices. The goal is to better use the visibility into application performance and SLO, to assist services in autonomously adjusting their own resource allocations. Autothrottle conveys this bridging "assistance" through *performance targets*, which translate the desired application performance to local proxy metrics measurable by services. Doing so hides low-level resource control details from the SLO feedback mechanism. In this paper, we use CPUs to discuss the framework design—not only is the CPU harder to manage due to its higher usage fluctuation over time [13, 22, 40], but it also has an immense impact on microservice response time [38, 57, 63].

At each microservice, Autothrottle locally runs a lightweight resource controller called *Captain*. Captain swiftly adjusts CPU allocations through OS APIs (e.g., CPU quota in Linux's cgroups), to ensure its governed microservice reaches the given performance target. Autothrottle represents this target using an unconventional metric—*CPU throttles*, namely the number of times a service exhausts its CPU quota in a time period. Not only are CPU throttles sufficiently cheap to sample at high frequency to enable Captains' timely adjustments, but we also observe that they have higher correlation with latencies than other proxy metrics such as CPU utilization (§5.3). These characteristics make CPU throttles an indicative target to track locally, for maintaining an end-to-end SLO. At the application level, Autothrottle employs a centralized SLO feedback controller called *Tower*. It observes the application workload measured by RPS (requests per second) and learns to determine the most cost-effective performance targets that maintain the SLO, using a lightweight online algorithm known as contextual bandits [14].

This paper makes the following key contributions:

- We examine unique implications that SLO-targeted microservices introduce to resource management (§2). Since there is no strong correlation between the end-to-end application performance and per-service resource usage, directly computing the optimal resource allocations is non-trivial.
- Autothrottle is a bi-level learning-assisted framework (§3), to embrace distinct levels of distributed system behavior. It separately designs mechanisms of application-level SLO feedback and service-level resource control, and introduces CPU-throttle-based performance targets to bridge them.
- Comprehensive experiments (§5) demonstrate Autothrottle's superior CPU savings over state-of-the-art heuristics and ML-based baselines, in three SLO-targeted applications: Train-Ticket [52], Social-Network [63], and Hotel-Reservation [22]. Compared with the best-performing baseline in each application, Autothrottle maintains the SLO for the 99th percentile latency while saving up to 26.21% CPU cores for Train-Ticket, up to 25.93% for Social-Network, and up to 7.34% for Hotel-Reservation, across four real-world workload patterns. Finally, running Social-Network over a 21-day period with production workloads from a global cloud provider, Autothrottle saves up to 35.2 CPU cores while reducing hourly SLO violations by 13.2×.

2 Background and Motivation

Our goal of SLO-targeted resource management is to minimize the total CPU allocations to microservice-based applications, while avoiding SLO violations on the user request latency. Following real-world findings [19], our SLO is an upper limit on tail latencies, specifically the 99th percentile (P99) request latencies unless otherwise noted.

2.1 Implications of microservices

Unlike monolithic applications, the distributed nature of microservices implies that multiple services collectively contribute to the end-to-end latency. This section presents observations, to motivate its implications on computing per-service resource allocations from the end-to-end latency SLO.

2.1.1 Service execution dependencies

As user requests traverse services, their end-to-end latency is a function of per-service performance (and hence resource usage). Being functionally different, services can consume resources differently. Moreover, service execution dependencies can introduce complex correlations to this function—not only are there various patterns such as parallelism, but services can also exhibit unexpected increases in resource demand.

An illustrative example is backpressure [22]—as an underprovisioned service undergoes performance degradation during request processing, the resource manager can misinterpret its idling parent's longer response time as the culprit. Simply identifying all parent-child relations does not fully solve the problem, as backpressure can vary subtly depending on service implementations. In one case we encountered, the CPU usage of a waiting parent unexpectedly increased with the number of requests, which was counterintuitive as waiting for child services should result in idle CPUs. Further investigations revealed that the parent service spawned a separate thread for each outstanding request (i.e., Thrift's `TThreadedServer` RPC model), leading to excessive thread maintenance and spurious context switching. An alternative implementation with non-blocking or asynchronous I/O (e.g., Thrift's `TNonBlockingServer`) eliminated the problem.

To grapple with the complexities arising from service dependencies, prior work considers how the end-to-end performance *directly* correlates with application-wide execution dependencies. However, maintaining an accurate and up-to-date global view of these dependencies is both challenging and costly. First, the interdependencies among services are constantly evolving during development, often with multiple versions of the same service coexisting [50], requiring frequent updates to the global view. Second, although ML-based resource management strategies [43, 63] have the potential to comprehend complex and large-scale service dependencies, they may entail substantial training and retraining expenses. Third, fine-grained distributed tracing (beyond the basic monitoring through sampling-based tracing) may be necessary for resource managers to observe and analyze service dependencies [23, 44], resulting in additional system overheads from increased instrumentation and telemetry collection.

Observation #1: *Maintaining an up-to-date global view of service dependencies can be impractical.*

2.1.2 Delayed end-to-end performance feedback

The chain of service execution dependencies brings about the *delayed effect*, i.e., a time delay for the impact of any changes in resource allocations or workloads to be fully observed in the end-to-end performance. This prevents resource managers from immediately responding to misallocations. The delayed effect is often amplified. One source is service queues—underprovisioning of resources will cause requests to accumulate in queues, and thus SLO violations are not detected until all queued requests are eventually processed or timed out. Even if resources are scaled at this point, it takes time to flush queues [22, 63]. Another amplification is that SLO is typically defined on aggregated performance data (e.g., percentiles), which require a sufficient number of requests to be profiled.

In light of the delayed effect, prior work [63] proposes to proactively predict the long-term impact of resource changes on the end-to-end performance. Such performance predictions are theoretically possible but they usually involve expensive data collection and model training. On the other hand, prematurely deploying ML models can result in a high percentage of mispredictions. For instance, our efforts to fully train Sinan's neural networks [63] for a 28-microservice application took 14+ hours, plus ∼6 hours to collect 20,000 training data points. Despite reproducing the published prediction accuracy, we observed that mispredictions can trick resource managers to overallocate at least 40.75% more CPU cores (§5).

Observation #2: *Predicting end-to-end application performance under the delayed effect can be unreliable.*

2.2 A practical approach

In light of the observations in §2.1, a more promising approach for SLO-targeted resource managers is to embrace the distributed nature of microservices by taking into account of the distinct levels of system behavior—the macro perspective reveals the end-to-end performance (e.g., user request latencies) and SLO, whereas the micro perspective is scoped to local measurements (e.g., service CPU usage) and control.

Naturally, these two levels can map to: ***(1)*** *application-level SLO feedback*, which compares the end-to-end performance and SLOs, and ***(2)*** *service-level resource control*, which computes resource allocations based on local measurements. In fact, if we architecturally decouple these mechanisms, it becomes feasible to position them close to their required inputs. Doing so brings the benefit of fast reaction, which opens up opportunities for resource managers to relax the requirement of computing the optimal resource allocations. Rather than striving to accurately model service dependencies (§2.1.1) or predict long-term application-wide behavior (§2.1.2), we can now employ lightweight service-level controllers that autonomously and swiftly adjust resource allocations, assisted by periodic guidance computed at the application level through a lightweight online learning approach.

In summary, SLO-targeted resource managers for microservices should incorporate the following design principles.

1. Decouple mechanisms of application-level SLO feedback and service-level resource control.
2. Rapidly drive per-service resource control with local performance targets and near-term prospects.
3. Achieve practicality through lightweight solutions.

3 The Autothrottle Framework

Following the design principles laid out in §2.2, we present Autothrottle, a practical and readily deployable resource management framework for SLO-targeted microservices.

3.1 Overview

Autothrottle is a bi-level learning-assisted framework, consisting of an application-wide global controller and per-service local controllers. The application-level controller is based on online learning, periodically assisting local resource control with its visibility into application workloads, end-to-end latencies, and SLO violations. The service-level controllers, on the other hand, are heuristic-based, continuously performing fast and fine-grained CPU scaling using local metrics as well as the assistance from the global controller.

The "assistance" bridging the two levels is based on the notion of *performance target*, a target performance level set by the application-wide controller for per-service controllers to attain. Autothrottle implements the performance target with *CPU throttle ratio*—the fraction of time a microservice is stopped by the underlying CPU scheduler. This design is motivated by the strong correlation between CPU throttles and service latencies revealed by our correlation test (§5.3).

Maintaining a CPU throttle ratio locally also allows tolerating a certain range of workload fluctuations (§5.3). When per-service controllers fail to rein in end-to-end latencies (e.g., the workload exceeds the tolerable range), the application-wide controller issues lower throttle targets to guide local controllers to allocate more CPUs. Conversely, higher throttle targets are assigned in the event of CPU over-provisioning.

Acting as a broker, the performance target allows Autothrottle to decouple the mechanisms of application-level SLO feedback and service-level resource control. Consequently, we are able to simplify the learning process of the application-level controller by concealing low-level resource details and avoiding the overhead of aggregating them, while enabling per-service controllers to focus on a self-contained in-situ task—reaching a given performance target using locally available information. Our bi-level design sets us apart from approaches that directly infer resource demands with proxy metrics (e.g., [5]) or machine learning (e.g., [63]).

Figure 2 depicts the architecture of Autothrottle. We refer to the per-service controllers as *Captains* (§3.2), and the application-wide controller as the *Tower* (§3.3).[2]

Autothrottle Captains. At the local level, each microservice runs a Captain instance, which periodically receives performance targets—CPU throttle ratios—from the Tower and strives to realize these targets using heuristic control. The heuristic control algorithm collects statistics on CPU usage and throttles, and employs two feedback control loops to scale CPUs up and down, respectively. This lightweight design ensures swift and fine-grained CPU autoscaling of the Captain even amid rapidly fluctuating workloads.

Autothrottle Tower. At the global level, the Tower leverages *contextual bandits* [14], a lightweight class of online reinforcement learning (RL), to dynamically determine suitable performance targets that maintain the SLO. It monitors application workload (e.g., RPS) and observes CPU allocations and end-to-end latencies (along with associated SLO violations) as feedback for its output targets. This online learning approach is directly applicable to any microservices, eliminating the need for extensive offline profiling or training.

Overall, Autothrottle takes a pragmatic stance and provides a resource management framework that is readily deployable across diverse latency-sensitive microservice applications. Next, we elaborate on Autothrottle from the bottom up, starting off with Captains (§3.2), followed by the Tower (§3.3).

3.2 Per-service controllers—*Captains*

Each Captain periodically (e.g., every minute) receives a target CPU throttle ratio from the Tower. Given a throttle target, Captain focuses on a self-contained, in-situ task—scaling up and down the CPUs made available to its governed service

[2]The air traffic control "tower" (application-wide controller) assigns "routes" (performance targets) to flights, while each "captain" (per-service controller) follows the assigned route by actually steering the aircraft.

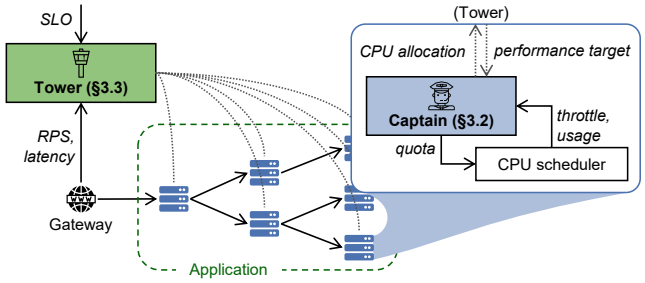


Figure 2: Autothrottle features bi-level resource management: The application-level learning-based controller (Tower), observing end-to-end latencies and workloads, periodically sets performance targets, expressed as CPU throttle ratios, for per-service heuristic controllers (Captains) to meet.

in order to meet the throttle target upon changing demand. Algorithms 1 and 2 present the pseudocode of Captain's main components, which we describe in detail below.

3.2.1 Resource metrics and knobs

Common CPU schedulers in the OS, such as the Linux CFS scheduler we use as a running example in this paper, assign each microservice a CPU quota (e.g., `cpu.cfs_quota_us`) to limit and isolate resource usage. To accomplish the task of maintaining a target CPU throttle ratio, Captain continuously collects two statistics exposed by the OS in each time window—CPU throttle count and CPU usage.

CPU throttle count. The Linux CFS scheduler maintains a CPU throttle count for each microservice in the variable `cpu.stat.nr_throttled`, which represents the cumulative number of CFS periods (100 ms by default) during which the CPU quota has been exhausted. Intuitively, if the CPU quota is used up early in a CFS period before a request can be fulfilled, the request will be approximately delayed by the remaining period, underscoring the importance of avoiding CPU throttles when maintaining latency SLOs. Anecdotal evidence in blog posts [15, 34] corroborates our intuition. To calculate the CPU throttle ratio over a time window, we divide the increase in the CPU throttle count (`cpu.stat.nr_throttled`) by the number of elapsed CFS periods.

CPU usage. The Linux CFS scheduler also reports the total CPU time consumed by a microservice as `cpuacct.usage`. This metric is particularly useful when the CPU is *over-provisioned*, as it reveals the actual (lower) CPU demand. Otherwise, this actual demand would be capped by the allocated CPUs if under-provisioned and thus remain unknown.

3.2.2 Multiplicative scale-up

In every time window of N ($N = 10$ by default) CFS periods, each Captain compares the measured CPU throttle ratio at its microservice with the target ratio. When the measured ratio exceeds the target, it indicates the CPU is under-provisioned,

Algorithm 1: Captain: scaling up and down

```
1  /* executes every N periods */
2  throttleCount = throttle count during last N periods;
3  throttleRatio = throttleCount/N;
4  margin = max(0, margin + throttleRatio − throttleTarget);
5  if throttleRatio > α × throttleTarget then
6  |  /* multiplicatively scale up */
7  |  quota = quota × (1 + throttleRatio − α × throttleTarget);
8  else
9  |  /* instantaneously scale down */
10 |  history = CPU usage history in the last M periods;
11 |  proposed = max(history) + margin × stdev(history);
12 |  if proposed ≤ β_max × quota then
13 |  |  quota = max(β_min × quota, proposed);
14 |  end
15 end
```

Algorithm 2: Captain: rollback mechanism

```
1  /* executes every period for N periods after each scale-down */
2  lastQuota = CPU quota before scale-down;
3  throttleCount = throttle count since scale-down;
4  throttleRatio = throttleCount/N;
5  if throttleRatio > α × throttleTarget then
6  |  /* revert to the previous (higher) quota before scale-down
7  |     with an additional allocation equal to the quota difference */
8  |  quota = lastQuota + (lastQuota − quota);
9  |  margin = margin + throttleRatio − throttleTarget;
10 end
```

demanding a prompt increase in the CPU quota to prevent imminent SLO violations at the application level.

To ensure that any desired target can be reached quickly within several steps, Captain increases the current CPU quota *multiplicatively*. We further make the size of the increase proportional to the difference between the measured CPU throttle ratio and the target ratio. This represents a form of proportional control, where a larger difference results in a larger stride in the CPU quota increase. The rationale is that when the difference is significant, a queue of requests is likely to have built up, thus requiring more CPUs to drain.

In practice, we find that the local workload arriving at a microservice is naturally bursty and irregular—regardless of the pattern of end-to-end requests—tricking Captains into spurious scale-ups. Hence, we execute the scale-up only when the CPU throttle ratio surpasses "$\alpha \times$ target ratio" ($\alpha \geq 1$), where α is a customizable weight that controls the sensitivity to transient load spikes. Correspondingly, the CPU quota is also multiplied by "1 + throttle ratio − $\alpha \times$ target ratio" in each step. The pseudocode is in Line 5–7 of Algorithm 1.

3.2.3 Instantaneous scale-down

Under frequent CPU throttling, Captain is forced to incrementally probe the actual CPU demand of the service. In contrast, when the measured throttle ratio is below the target ratio, the service's CPU demand has been adequately met. Consequently, historical CPU usage begins to more accurately reflect the actual (less throttled) CPU demand and help *instantaneously* determine the desired CPU quota.

Motivated by this characteristic of over-provisioning, the Captain maintains a sliding window of CPU usage over the most recent M ($M = 50$ by default) CFS periods, and calculates a new CPU quota based on two statistics from the sliding window: the maximum and the standard deviation of CPU usage. Specifically, the proposed quota is "max CPU usage + *margin* $\times$ standard deviation of CPU usage," where *margin* ≥ 0 is a dynamically tuned parameter that generally increases when the CPU throttle ratio exceeds the target ratio and decreases otherwise. To avoid unnecessary fluctuations in CPU allocation, the proposed quota is put into action only when it represents a significant-yet-moderate change. The details are described in Line 9–14 of Algorithm 1.

Our scale-down design draws inspiration from prior work [45,46], but differs in the carefully maintained parameter *margin* that depends on CPU throttles. Intuitively, if the CPU is recently throttled more often than desired, we should be more conservative by using a larger *margin* in the subsequent scale-down to avoid overreacting to momentary tranquility amid workload spikes; and vice versa. In summary, historical CPU usage in the sliding window allows for instantaneous scale-down, reclaiming extra CPU allocations in a single step.

3.2.4 Rollback mechanism after scaling down

Accidentally scaling up CPUs only leads to resource waste (and existing cloud applications tend to be over-provisioned); however, mistakenly scaling down the CPU allocation to any microservice may cause SLO violations at the application level. Thus, we introduce a fast rollback mechanism to the Captain to revert "reckless" scale-downs as follows.

After each scale-down, we continuously check whether it is "reckless"—if it has caused the CPU throttle ratio to exceed $\alpha \times$ target ratio—during *every* CFS period within the next N periods. We note that the triggering condition is the same as that used for scaling up, but due to the urgency of initiating a rollback, this check is performed more frequently, without waiting for the Captain's regular decision-making interval (N periods). After a rollback is triggered, the current CPU quota is restored to the previous (higher) quota used before the scale-down, plus an additional allocation equal to the difference between the two quotas. We grant slightly more CPUs to account for the potential processing delays that may have occurred since the erroneous scale-down. Details of the rollback mechanism are presented in Algorithm 2.

3.3 Application-level controller—*Tower*

In Autothrottle, Tower delegates the in-situ resource control to per-service Captains and only provides periodic assistance by dispatching the target CPU throttle ratios for Captains to

meet. This indirection via throttle targets effectively avoids the latency overhead associated with distributed tracing and logging, while retaining Tower's global perspective on end-to-end requests and SLO feedback.

We design Tower to compute a new target throttle ratio infrequently, e.g., once a minute, leaving ample time for tail request latencies and average CPU usage to stabilize as the new target settles in. Importantly, doing so minimizes the influence of Tower's previous decisions, simplifying the problem into a "one-step" decision-making process: Tower only needs to determine the optimal CPU throttle targets for the current step, without considering their long-term consequences.

This "one-step" nature motivates us to employ *contextual bandits*, a lightweight class of online reinforcement learning (RL) algorithms. In the taxonomy of RL, contextual bandits can be viewed as one-step RL and are well suited for real-time online scenarios in which the algorithm is required to learn efficiently from a limited amount of sample data.

3.3.1 Primer on contextual bandits

Recent work has modeled resource management with sequential decision-making paradigms and seen the application of multi-armed bandits [46, 47] and reinforcement learning [41, 44, 59]. Contextual bandits are intermediate between multi-armed bandits and the full-fledged RL [54].

Contextual bandits are like multi-armed bandits in that they are well suited to problems where an *action* (e.g., CPU throttle targets) taken at a step (e.g., one-minute interval) does not have long-term impact beyond that step. They receive a *cost* (negative reward) as feedback for the chosen action, and aim to minimize the cumulative cost (e.g., comprising CPU allocations and SLO violations). Conversely, contextual bandits also differ from multi-armed bandits by their ability to make decisions based on the observation of the system state, known as the *context* (e.g., RPS). This context can provide valuable information that aids in the learning process (§5.3).

In contrast to the full RL, which optimizes a sequence of future steps, contextual bandits only optimize the current step owing to their assumption that each chosen action only affects the immediate outcome without long-term consequences. Moreover, full RL typically demands extensive offline training before deployment as well as frequent retraining (e.g., upon significant changes in microservices), whereas contextual bandits are more lightweight (with simpler models) and suitable for online learning with considerably fewer samples.

In solving contextual bandit problems, a common approach is to train a cost-prediction model that estimates the cost of taking each action within a context. Due to their inherent partial observability, however, contextual bandits can only observe the costs of actions they select but not the costs of others. To enhance their performance and sample efficiency, a widely adopted improvement is to estimate the costs of unused actions via counterfactual estimates [12, 20, 48]. This approach reduces contextual bandit problems to cost-sensitive classification [35], which can then be addressed using standard supervised learning. We adopt this approach and refer the reader to Bietti et al. [14] for more details.

3.3.2 Realizing contextual bandits in Tower

Next, we describe the contextual bandit algorithm used in Tower. The algorithm operates with a step size of one minute, and it aims to learn to output an action that incurs the lowest cost given the observed context at each step.

Context. Tower selects the average RPS observed in the last step as the context because the optimal CPU throttle target depends on the RPS (§5.3). We refrain from predicting the RPS for the next step due to the inherent difficulty in accurately forecasting RPS; moreover, our Captains have been intentionally designed to tolerate short-term RPS fluctuations (§5.3). Other metrics such as CPU usage are not included in the context as they are merely the byproducts of applying a throttle target to an RPS, with the RPS serving as the primary causal factor. The composition of the workload (i.e., the distribution of different request types) is relevant, but our focus in this work remains on constant workload composition (Appendix A), following the setup in prior work [22, 63].

Action. Given an instantiation of the Captain's resource control algorithm, we search for a ladder of CPU throttle targets as the actions. The search is a one-time process for all applications. By default, our action space consists of 9 throttle targets, ranging from 0 to 0.3 (§4).

Reduction of action space. A microservice-based application can contain 10–1000s of services [11, 25, 31, 40, 64]. In the case of 9 throttle targets, generating a different CPU throttle target for each individual service would result in $9^{\#\text{services}}$ actions, rendering it infeasible for contextual bandits to learn. As a solution, Tower clusters microservices into two classes and outputs an action for each class, effectively reducing the action space to $9^2 = 81$. To implement the clustering, we use the standard k-means algorithm [37] to group microservices based on their average CPU usage. Our empirical results in §5.3 suggest a diminishing return beyond two clusters.

Cost function. We define the cost received per step as follows. When the SLO is met after the step, we only use the total CPU allocation as the cost, since the actual latencies below SLO matter no more. To this end, Tower requests Captains to send their actual CPU allocations as feedback every minute, and then normalizes the total allocation linearly into $[0, 1]$. On the other hand, when the SLO is violated, we set the cost to only contain the tail latency, linearly normalized to $[2, 3]$ considering the higher priority of SLO violations. We arrived at the two normalization ranges above based on their empirical performance compared with other ranges we tested, but we do not claim our cost function is the best.

Noise reduction for costs. Our contextual bandit algorithm learns online and updates its model weights on every (context, action, cost) tuple, i.e., most recent RPS, two throttle targets,

and the incurred cost. In reality, however, we observe highly noisy costs that result in confusion and poor performance of the model, supposedly due to the complex microservice system and the dynamics in Captains. To address this, we buffer and group recent samples using (context, action) as the index after quantizing the RPS. Given a new sample, we use the median cost of the group it falls into—rather than the cost computed for that individual sample—to update the model. Doing so significantly reduces the noise in costs and stabilizes the online learning process.

Exploration. Similar to multi-armed bandits and RL, contextual bandits rely on exploration to acquire knowledge about the costs of different actions, e.g., using ε-greedy [35] to choose a random action with a small probability of ε (the best action is selected otherwise). Despite a reduced action space, randomly exploring all 81 actions within a context remains inefficient as each sample requires one minute to complete; repeated sampling is further required to calibrate noisy cost estimates. To ensure efficient exploration without impeding online learning, we explore only the neighbors of the best action in the action space. Given a sorted ladder of the available CPU throttle target , $r_1 < r_2 < \ldots < r_9$, if the best action consists of (r_i, r_j), $1 \leq i, j \leq 9$, then each of its neighbors (r_i, r_{j-1}), (r_i, r_{j+1}), (r_{i-1}, r_j), (r_{i+1}, r_j) is explored next with an equal probability of $\varepsilon/4$ (subject to boundary conditions). The rationale is that the throttle target ladder is monotonic, allowing Tower to move upward or downward one step at a time without missing the optimal action.

4 Implementation

Our current implementation supports microservice applications deployed as pods on Kubernetes, but it can be easily extended to other container orchestration frameworks (e.g., OpenShift and Docker Swarm). Autothrottle is open-sourced at https://github.com/microsoft/autothrottle.

Captain. Each microservice is associated with a Captain co-located on the same worker node, so we deploy Captains as processes on worker nodes of the Kubernetes cluster. Captain implements the following three functionalities. First, it communicates with the Tower over a TCP socket, exchanging CPU throttle targets and allocations. Second, it collects CPU throttling and usage statistics from Linux cgroup API in every CFS period of 100 ms, as the input to the local resource controller. Third, it runs the resource controller for all microservices on the same worker node, and sets their CPU quotas (`cpu.cfs_quota_us`) accordingly. As Captain only comprises lightweight heuristic-based control loops, it does not require any pre-deployment training.

The pseudocode of Captain is outlined in Algorithms 1 and 2. Our default parameters are $N = 10$, $M = 50$, $\alpha = 3$, $\beta_{\max} = 0.9$, $\beta_{\min} = 0.5$. They can be adjusted accordingly. A larger N or M lowers sensitivity to the noise in CPU usage, hence slower reaction. α sets the supported range of throttle ratios to $(0, 1/\alpha)$. A smaller α increases the upper bound but decreases the tolerance on throttle ratio fluctuations. $\beta_{\max}$ and $\beta_{\min}$ prevent overly small or large allocation changes.

Tower. One instance of Tower runs globally alongside the application (i.e., in the same cluster), initialized with a user-specified SLO. It collects average RPS and tail latencies from the Locust workload generator, but can be extended to hook up to an application gateway. Furthermore, Tower receives the actual CPU allocations from Captains after dispatching CPU throttle targets to them every minute.

Tower leverages the widely used Vowpal Wabbit (VW) library [10] to implement contextual bandits. For each group of microservices, the model outputs one of the 9 throttle targets: 0.00, 0.02, 0.04, 0.06, 0.10, 0.15, 0.20, 0.25, and 0.30. Designed for efficient online learning, VW offers lightweight model options such as linear regression or a shallow neural network with a single hidden layer. We opt for a neural network model with 3 hidden units after performing an ablation study (§5.3), and train it with a learning rate of 0.5. The doubly robust estimator [20] is employed in the bandits for policy evaluation to estimate the costs of untaken actions. Moreover, we disable the native ε-greedy algorithm to implement our customized exploration strategy (§3.3.2). The specific VW usage is detailed in Appendix B.

Online training starts with an exploration stage, which allows VW to randomly explore how different CPU throttle targets would impact application latencies. During this stage, each randomly chosen action will be executed for 2 minutes. Only the second minute is used for cost calculation and training, in order to avoid interference from the previous chosen action. This exploration stage lasts $\sim$6 hours, during which application latencies may exceed the SLO.

After the exploration stage, Tower starts to exploit the best action, while still exploring neighboring actions with a total of 10% probability using ε-greedy. Tower runs every minute to collect last minute's (context, action, cost) sample. All recent samples are grouped using (context, action) as the index with RPS quantized into bins of 20, and each group's cost is defined as the median cost of the group. Since training each unique (context, action) only once is insufficient for contextual bandits, 10,000 training data points are sampled from these groups randomly. A contextual bandit model is then trained on these samples, and predicts the next best action based on RPS. As a reference, this training-and-prediction process takes less than one second in our setup.

5 Evaluation

We evaluate Autothrottle's superior resource saving with three SLO-targeted microservice applications, against state-of-the-art heuristic- and ML-based baselines. Major results include:

(1) Over the best-performing baseline in each application, Autothrottle maintains the given application P99 latency SLO,

while achieving a CPU core saving up to 26.21% for Train-Ticket, up to 25.93% for Social-Network, and up to 7.34% for Hotel-Reservation. Over all baselines, its savings can be up to 93.84%, 55.32%, and 83.99%, respectively.

(2) A 21-day study of Social-Network (with real-world workload trace from a global cloud provider) shows a saving up to 35.2 CPU cores, over the best-performing baseline. Meanwhile, it reduces hourly SLO violations from 71 to 5.

(3) Microbenchmarks evaluate Autothrottle's design and tolerance to workload fluctuations and load-stressing.

5.1 Methodology

Benchmark applications. We deploy three SLO-targeted microservice applications: ***(1)*** Train-Ticket [52], with 68 distinct services, ***(2)*** Hotel-Reservation from DeathStarBench [22], with 17 distinct services, and ***(3)*** Social-Network used in Sinan [63], a variant of the Social-Network application from DeathStarBench, with 28 distinct services including two ML inference serving services: a CNN-based image classifier and an SVM-based text classifier. These applications are representative of real-world microservices, with stateless services (e.g., business logic), data services (e.g., key-value stores), and gateways. Deployments are managed by Docker and Kubernetes. Parent-child service communications are through popular RPC frameworks such as gRPC and Thrift.

Application SLOs are specified on the hourly P99 latency [19]—1,000 ms for Train-Ticket, 200 ms for Social-Network, and 100 ms for Hotel-Reservation.

Comparison baselines. Baselines include ***(1)*** Kubernetes default autoscalers [5] (denoted as "K8s-CPU" and "K8s-CPU-Fast"), and ***(2)*** state-of-the-art ML-driven solution, Sinan [63].

K8s-CPU locally maintains each service's average CPU utilization, with respect to the user-specified CPU utilization threshold (e.g., 50%). Every m=15 seconds, it measures service's CPU usage, and computes the optimal allocation by "CPU usage / CPU utilization threshold." Then, it sets the CPU limit to the largest allocation computed in the last s=300 seconds. We also include a faster version called K8s-CPU-Fast, which has m=1 and s=20. Since Kubernetes relies on users to properly translate the application SLO to CPU utilization threshold, we manually try different thresholds to find the appropriate one for each experiment (Appendix F).

Sinan leverages ML models (e.g., a convolutional neural network and a boosted tree model) to globally assess each service's resource allocation. Starting with the open-sourced Sinan [8], we follow instructions to train application-specific models offline for 20+ hours. Since Sinan relies on users to properly set several hyperparameters, we manually tune for each application. During experiments, we run Sinan every second—given historical resource usage and latencies, Sinan tries to predict the optimal CPU allocation that is unlikely to violate the SLO over both the short and long terms.

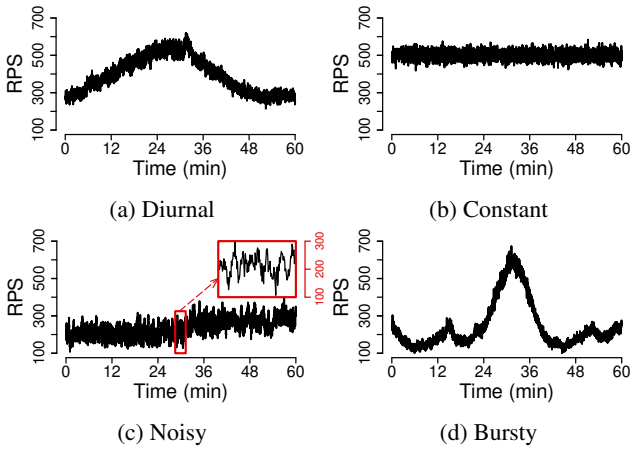


Figure 3: Our workload traces capture common patterns of RPS (requests per second) on an hourly basis. These patterns have been observed in real-world scenarios: Puffer streaming requests [60], Google cluster usage [58], and Twitter tweets [9]. We also recorded a full 21-day workload trace from a global cloud provider for long-term evaluation. We scale these traces accordingly for each benchmark application to saturate the cluster (Appendix E).

Experiment setup. We generate workloads with Locust [7], which is configured to mix application requests (Appendix A) to stress as many services as possible. Locust replays workload traces to reproduce RPS (requests per second). The first set of traces captures hourly RPS patterns, which are commonly observed in production environments: Puffer's streaming requests [60], Google's cluster usage [58], and Twitter tweets [9]. Figure 3 illustrates these patterns: diurnal, constant, noisy, and bursty. We also keep a full 21-day workload trace from a global cloud provider for long-term evaluation. Depending on the complexity of benchmark applications, we scale traces accordingly to saturate the cluster (Appendix E).

Each experiment ends when Locust finishes replaying a trace. For comparisons, we record the following per-hour measurements: ***(1)*** the average number of CPU cores allocated, and ***(2)*** the application end-to-end P99 latency.

Our testbeds consist of a 160-core cluster (over five 32-core Azure VMs with AMD EPYC 7763 processors) and a 512-core cluster (over six 64-core and four 32-core physical servers with Intel Xeon Silver 4216 processors).

5.2 Application SLO and resource saving

We evaluate the amount of CPU resources that Autothrottle saves over baselines, while every algorithm tries to maintain the hourly SLO over time. To ensure that all baselines can achieve their best results, we manually identify and tune their settings prior to experiments (Appendix F).

Table 1 summarizes empirical results on the 160-core cluster, and Autothrottle outperforms baselines in all applications.

Workload	Autothrottle	K8s-CPU	K8s-CPU-Fast	Sinan
Diurnal	30.4	58.0 (↓47.59%)	41.2 (↓26.21%)	278.4 (↓89.08%)
Constant	21.7	24.8 (↓12.50%)	27.3 (↓20.51%)	279.9 (↓92.25%)
Noisy	15.5	23.6 (↓34.32%)	17.7 (↓12.43%)	251.8 (↓93.84%)
Bursty	17.7	27.1 (↓34.69%)	21.9 (↓19.18%)	268.3 (↓93.40%)

(a) Train-Ticket application (SLO: 1,000 ms P99 latency)

Workload	Autothrottle	K8s-CPU	K8s-CPU-Fast	Sinan
Diurnal	77.5	93.9 (↓17.47%)	115.5 (↓32.90%)	162.7 (↓52.37%)
Constant	88.7	115.6 (↓23.27%)	118.8 (↓25.34%)	149.7 (↓40.75%)
Noisy	57.5	66.5 (↓13.53%)	105.1 (↓45.29%)	105.2 (↓45.34%)
Bursty	50.0	67.5 (↓25.93%)	99.7 (↓49.85%)	111.9 (↓55.32%)

(b) Social-Network application (SLO: 200 ms P99 latency)

Workload	Autothrottle	K8s-CPU	K8s-CPU-Fast	Sinan
Diurnal	15.3	15.7 (↓2.55%)	16.5 (↓7.27%)	45.5 (↓66.37%)
Constant	11.2	11.5 (↓2.61%)	11.3 (↓0.88%)	21.2 (↓47.17%)
Noisy	10.8	12.1 (↓10.74%)	11.6 (↓6.90%)	65.9 (↓83.61%)
Bursty	10.1	15.7 (↓35.67%)	10.9 (↓7.34%)	63.1 (↓83.99%)

(c) Hotel-Reservation application (SLO: 100 ms P99 latency)

Table 1: Average number of CPU cores that Autothrottle and baselines allocate to satisfy the SLO (and thus latencies are elided). Percentages in parentheses quantify Autothrottle's CPU savings over each baseline. The overall best-performing baseline for each application is highlighted in gray. For K8s-CPU and K8s-CPU-Fast, we manually search for their optimal utilization thresholds (to minimize the average CPU allocation), per application and workload trace (Appendix F).

We make the following observations, with respect to heuristic-based baselines. First, in Social-Network, Autothrottle saves up to 25.93% of CPU resources (or 17.5 cores) over K8s-CPU, and up to 49.85% of CPU resources (or 49.7 cores) over K8s-CPU-Fast. Delving into empirical results, Figure 4 suggests that tuning the baselines' CPU utilization thresholds does not make them outperform Autothrottle. Taking the diurnal workload as an example, the figure shows that Autothrottle is able to maintain the application SLO with the minimum CPU allocation—Autothrottle achieves a P99 latency of 178 ms with only 77.5 cores, whereas K8s-CPU achieves 177 ms with 115.5 cores and K8s-CPU-Fast achieves 171 ms with 93.9 cores, at best. When allocating a comparable number of CPUs (∼80 cores) to Autothrottle, K8s-CPU and K8s-CPU-Fast would violate the SLO, resulting in latencies of 252 ms and 418 ms respectively. Second, Autothrottle has a relatively low resource reduction on Hotel-Reservation. This is due to the application simplicity where requests traverse an average of only 3 microservices. A similar observation can be made for the constant workload trace, where the relatively static RPS pattern simplifies scaling decisions.

Furthermore, Table 1 shows that Autothrottle outperforms the ML-enabled baseline, Sinan. Its CPU saving is at least

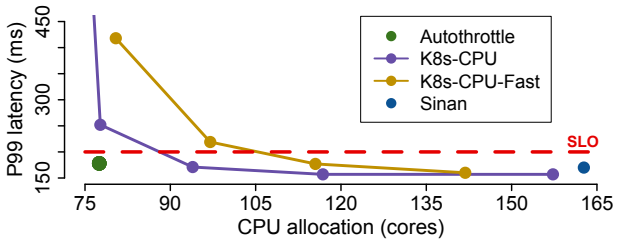


Figure 4: Application latency vs. CPU allocations, as we vary the two baselines' CPU utilization threshold for Social-Network under the diurnal workload trace. Dashed red line illustrates the 200 ms SLO. Autothrottle is able to maintain the SLO with the minimum CPU allocation.

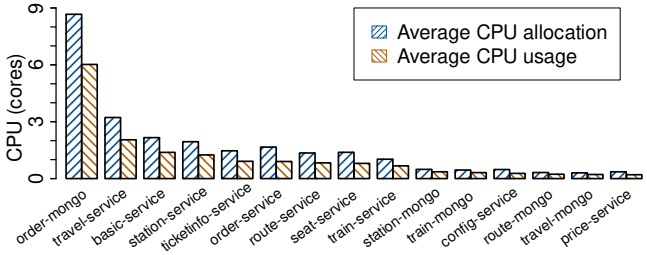


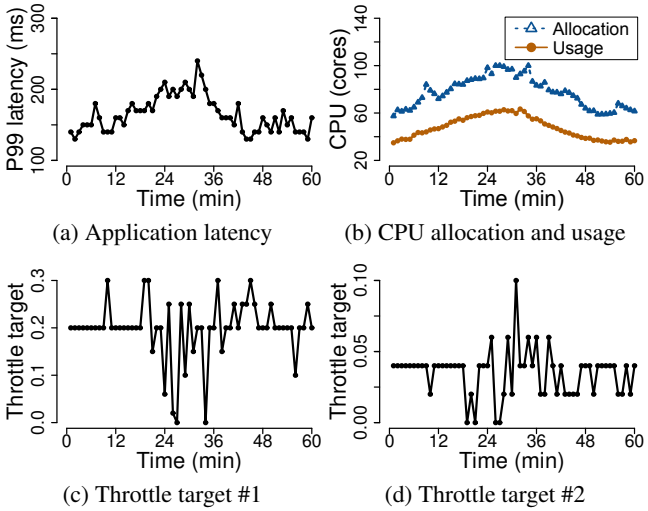
Figure 5: Autothrottle tailors CPU allocations to each microservice's resource usage. Figure shows top 15 microservices with the highest CPU usage in Train-Ticket under the diurnal workload trace.

40.75% (or 61 cores for Social-Network). Deeper investigations suggest two reasons for this gap. First, while we are able to achieve the model accuracy published by authors (e.g., training RMSE of 22.39 and validation RMSE of 22.07, for Social-Network) after 20+ hours of training, this non-negligible error can still mislead scaling decisions, especially for non-constant workloads. Second, in order to reduce training costs, Sinan learns to make relatively coarse-grained CPU allocation adjustments (i.e., ±1 core, ±10% cores, and ±50% cores).

Resource savings from Table 1 are due to Autothrottle's ability to tailor CPU allocations across services and over time. For example, Figure 5 looks at top 15 microservices with the highest CPU usage, under diurnal workload in Train-Ticket. We note that CPU allocation is noticeably lower for services with less CPU usage (e.g., `price-service`). Under the same workload, Figure 6 illustrates how Tower updates performance targets—as the RPS varies over time, Tower selects appropriate throttle targets to adjust CPU allocations and maintain the P99 latency. Note that per-minute P99 latencies are displayed in this figure, different from the hourly P99 latencies shown in the remaining evaluation.

5.3 Microbenchmarks

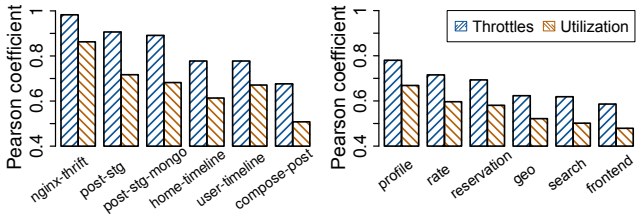
Correlation of proxy metrics to latencies. Compared with the prevalent proxy metric for estimating resource demand—



(a) Application latency (b) CPU allocation and usage

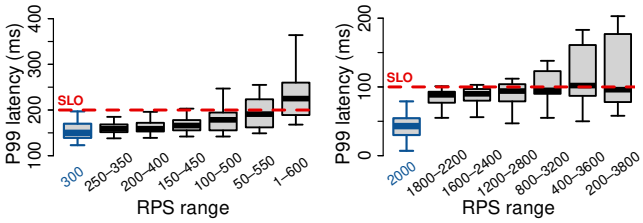
(c) Throttle target #1 (d) Throttle target #2

Figure 6: Measurements of Social-Network under diurnal workload. Figures (a) and (b) show the latency and CPU statistics achieved by Autothrottle. Figures (c) and (d) demonstrate how Tower adjusts throttle targets in response to time-varying workload for the two CPU usage groups (Appendix C).



(a) Social-Network (b) Hotel-Reservation

Figure 7: As a proxy metric, CPU throttles exhibit a higher correlation with application latencies than CPU utilization. The figure shows top microservices with highest CPU usage.



(a) Social-Network (b) Hotel-Reservation

Figure 8: Captain maintains latency SLO under some workload fluctuations. Boxplots show latency variances, from reusing the first blue boxplot's performance target.

CPU utilization, our use of CPU throttles is motivated by the higher correlation with application latencies as demonstrated by Figure 7. For each service in Social-Network, we manually set its CPU quota (i.e., `cpu.cfs_quota_us`) to 40 uniformly distributed values. Then, we measure CPU utilization, CPU throttle counts, and application P99 latency, at 300 RPS. We compute the Pearson correlation coefficient for ***(1)*** latency vs. CPU throttles, and ***(2)*** latency vs. CPU utilization. Figure 7a focuses on Social-Network microservices using the most CPU cores. In all cases, CPU throttles exhibit a higher correlation than CPU utilization, suggesting a stronger linear relationship. Figure 7b shows the same conclusion for Hotel-Reservation.

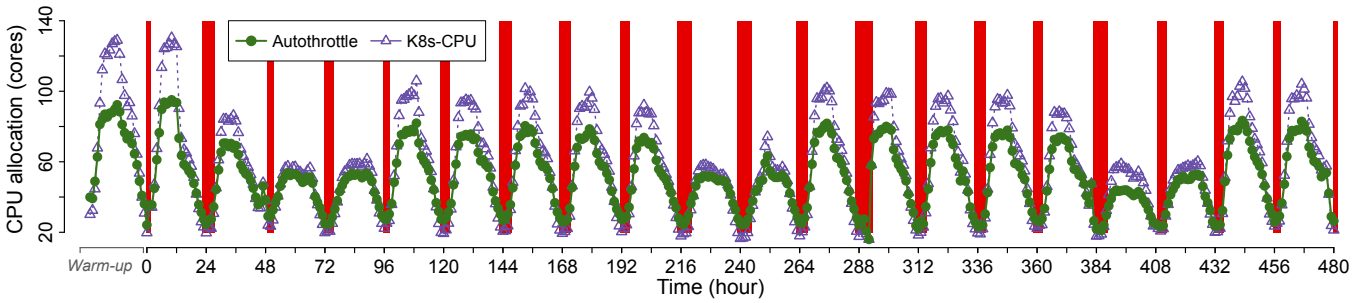
Recall that Captains continuously collect local CPU throttles for resource control (§3.2.2), and Tower distributes CPU-throttle-based performance targets (§3.3.2). A high correlation suggests that CPU throttling is indicative of the latency and suitable to track locally in Captain as a target for maintaining the SLO. The learning process in Tower can also be simplified given a clear relationship between CPU throttles and application latencies.

Tolerance to short-term workload fluctuations. Figure 8 shows that Captains can tolerate short-term local workload fluctuations, even with static throttle targets. The experiment starts by finding a throttle target for Social-Network's 200 ms SLO, at 300 RPS. Then, we reuse this target while instrumenting Locust to fluctuate RPS in a one-minute window for 60 minutes. The fluctuation ranges from 100 (i.e., RPS=250–350) to 600 (i.e., RPS=1–600). In Figure 8a, boxplots summarize the latency variance of 60 windows. Autothrottle can keep the application P99 latency under SLO for a fluctuation range up to 300 (i.e., RPS=150–450), or up to 500 (i.e., RPS=50–550) if we consider the median value instead. Similarly, Figure 8b shows RPS fluctuation tolerance up to 800 (i.e., RPS=1,600–2,400) for Hotel-Reservation.

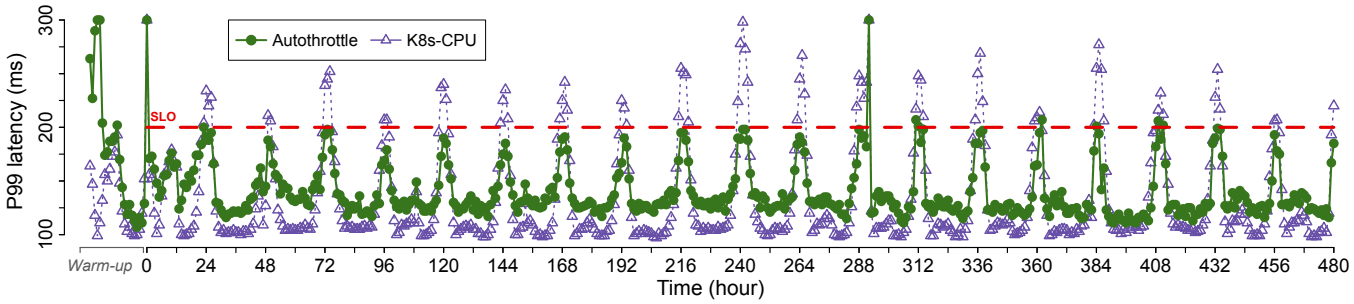
The tolerance to short-term workload fluctuations stems from the use of performance targets (vs. exact resource allocations), which hide service-level resource details from Tower and enable Captains to autonomously adjust resource allocations. This tolerance is vital as it frees Tower from the excessive recomputation of performance targets (§3.3.2).

Number of performance targets. Rather than generating separate performance targets for individual microservices, Tower clusters microservices into two categories based on their average CPU usage, reducing the action space to two targets (§3.3.2). To assess this design, we empirically compare the performance of 1, 2, 3, and 4 targets, under the constant workload trace. In each scenario, we manually search for the best-performing set of throttle targets that satisfy the SLO using the minimum number of CPU cores. For Social-Network, Autothrottle allocates 70.8, 55.9, 55.1, and 54.7 cores with 1 to 4 targets, respectively. Hotel-Reservation consistently uses the largest target (0.3) to meet the SLO on this trace, regardless of the number of targets. For Train-Ticket, the allocation is 18.6, 18.1, and 18.1 cores with 1 to 3 targets (exhaustive search is infeasible for 4 targets). Overall, these results suggest a diminishing return beyond 2 targets.

Load-stressing to the limit. We stress resource managers, by pushing Locust's RPS to the application's upper limit. This is the breaking point (before application crashing) when almost



(a) CPUs allocated by Autothrottle and the K8s-CPU baseline. Red boxes highlight hours of K8s-CPU's SLO violations.



(b) Social-Network's P99 latency, as achieved by Autothrottle and the K8s-CPU baseline. Dashed red line illustrates the 200 ms SLO.

Figure 9: A 21-day study on Social-Network with real-world workload trace from a global cloud provider. Compared with Autothrottle, the K8s-CPU baseline over-allocates an average of 12.1 and up to 35.2 cores, and triggers 71 hourly SLO violations.

all CPU cores are allocated. To this end, we stress Social-Network at constant RPS of 600 and 700, on the 160-core cluster. At 600 RPS, Autothrottle still achieves a CPU core saving of 27.67% and better SLO—it achieves a P99 latency of 202 ms with only 98.3 cores, whereas K8s-CPU achieves 216 ms with 135.9 cores and K8s-CPU-Fast achieves 235 ms with 133.1 cores. Finally, at 700 RPS, Autothrottle achieves a P99 latency of 452 ms with only 106.8 cores, whereas K8s-CPU achieves 600 ms with 153.1 cores, and K8s-CPU-Fast achieves 551 ms with 143.8 cores.

Ablation study for contextual bandits. We investigate two aspects that can impact Tower's contextual bandits. The first is the number of available throttle targets to choose from in the action space (§3.3.2). For the constant workload trace, reducing from 9 to 4 throttle targets results in over-allocating 5.6 CPU cores (or 10.03%) for Social-Network, and 0.7 CPU cores (or 3.49%) for Train-Ticket. The second is the use of neural networks (§4). Under various workload patterns on Social-Network, we test a linear model and neural networks with different numbers of hidden units, but their difference in CPU allocation is small. None of the tested models violates the SLO. We include the results in Appendix B.

5.4 Long-term evaluation

We perform a 21-day study with real-world workload trace from a global cloud provider. Experiments are performed with Social-Network on the 160-core cluster, and an hourly SLO of 200 ms is set on P99 latency. We compare Autothrottle with K8s-CPU, the best-performing baseline from §5.2. We use day 1 for training and tuning Autothrottle and K8s-CPU. For the former, we train the Tower's model. For the latter, we spend 24 man-hours to manually identify its best CPU utilization threshold.

Figure 9 illustrates the results over the entire period. Figure 9a shows the CPU core saving that Autothrottle achieves every hour, over the K8s-CPU baseline. First, Autothrottle can save up to 35.2 cores (or an average saving of 12.1 cores) over K8s-CPU. Second, although there are days when K8s-CPU allocates fewer CPUs (e.g., an hourly average of −2.77 CPU cores on day 4), these are also the days when K8s-CPU has a high chance of triggering SLO violations. In total, K8s-CPU violates the hourly SLO 71 times (highlighted by red boxes in Figure 9a). On the other hand, Autothrottle reduces SLO violations to only 5 times—an investigation reveals that these hours' workloads appear anomalous (i.e., recorded RPS jumps between 0 and ∼400) and unforeseen.

Figure 9b shows Social-Network's P99 latency per hour. One observation is that Autothrottle is able to continuously maintain a P99 latency closer to the 200 ms SLO. Since its P99 latency exhibits a much lower variance over time, this results in a more stable application performance.

5.5 Large-scale evaluation

We now show Autothrottle's scalability on the larger 512-core cluster. It allows us to push RPS beyond the breaking point of the 160-core cluster (§5.3), up to 1,200 on Social-Network (the upper limit for comparison baselines). To fully allocate

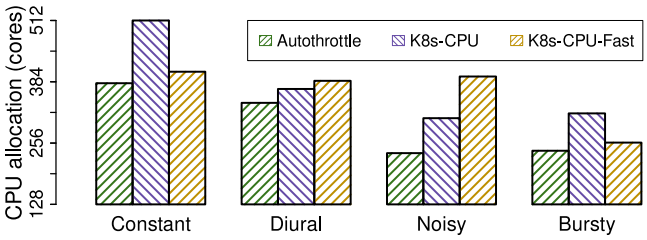


Figure 10: Number of CPU cores that Autothrottle and baselines allocate, to satisfy Social-Network's P99 SLO. Figure shows Autothrottle's scalability on the larger 512-core cluster.

all cores, we replicate Social-Network's CPU-intensive microservices: Nginx (×3) and ML-based image classifier (×6).

Figure 10 shows that Autothrottle is able to allocate fewer CPU cores while meeting Social-Network's 200 ms P99 SLO. Compared to the best-performing baselines, K8s-CPU and K8s-CPU-fast, Autothrottle saves up to 28.24% (or 150 CPU cores) and at least 5.92% (or 24 CPU cores). Finally, we note that K8s-CPU-Fast can have a higher CPU allocation than K8s-CPU, especially for the noisy workload trace. Since K8s-CPU-Fast is more sensitive to CPU utilization changes than K8s-CPU, it can sometimes accidentally scale down and lead to SLO violations. As a result, conservatively setting K8s-CPU-Fast results in the trade-off of higher CPU allocation.

6 Related Work

Cloud resource management. Cloud vendors have long offered services that enable elastic scaling of VMs and their associated resources according to user-defined rules [1, 3, 4]. In addition to rule-based scaling, researchers have proposed predictive scaling, which involves forecasting future demand and adjusting resource allocation in advance of any demand changes [2, 26, 42, 51]. Despite the cost effectiveness of these mechanisms in meeting SLOs, they are primarily designed for VMs (e.g., targeting monolithic applications or relying on VM-specific techniques such as live migration), and cannot be directly applied to microservices. Other cluster management frameworks [18, 27, 49, 55, 56] that schedule jobs to clusters may be used in conjunction with Autothrottle.

Vertical scaling of microservices. Vertical autoscalers adjust the resource limits in a fine-grained manner, e.g., milli-cores. Kubernetes Vertical Pod Autoscaler (VPA) [6] heuristically adjusts resource limits to maintain a user-specified utilization threshold. Autopilot [46] focuses on vertical scaling, selecting resource limits based on moving windows of historical usage and an ML technique akin to multi-armed bandit. Sinan [63] trains ML models to infer the likelihood of SLO violations given a set of proposed CPU limits. FIRM [44] reacts to SLO violations and pinpoints a microservice as the root cause, using reinforcement learning to scale up the service. A recent work [36] (also named "Autothrottle") adjusts the CPU quota of containers using closed-loop control to satisfy their individual network SLOs (e.g., throughput), rather than application latency SLOs. Autothrottle differs from these approaches with its bi-level design and the use of CPU throttle targets.

Horizontal and hybrid scaling of microservices. Horizontal autoscalers operate at a coarse-grained level by adjusting the number of replicas of a microservice. Kubernetes Horizontal Pod Autoscaling (HPA) [5] employs a mechanism similar to VPA at its core, except for choosing the appropriate number of pods to meet an input utilization threshold. GRAF [43] leverages graph neural networks to model service dependencies. COLA [47] uses a multi-armed bandit to collectively determine the number of replicas for each microservice. In addition, there are hybrid autoscalers that combine vertical and horizontal scaling and apply them selectively [24, 33]. Autothrottle focuses on vertical scaling due to its fine-grained and rapid reaction that empowers per-service controllers. As future work, we plan to explore the integration of horizontal scaling with Autothrottle.

Proxy metrics for estimating resource demand. In comparison to CPU throttles, alternative service-level proxy metrics fall short in maintaining end-to-end latency under workload changes. Kubernetes defaults to CPU utilization [5], but high CPU usage does not always indicate an issue if requests can still complete within the SLO [34, 47]. Queue length [61], the number of requests pending at a service, overlooks the complexity of individual requests, while queuing delay [16, 64] depends on the service's threading model [53] and may require manual instrumentation of each service. The scattered nature of queues across the application, OS, and network, further complicates precise measurement of queue length or queuing delay [63]. Regardless, dynamically adapting the thresholds for these metrics may require an application-level controller as proposed by Autothrottle.

7 Conclusion

Autothrottle is a bi-level learning-assisted resource management framework for SLO-targeted microservices. It decouples mechanisms of SLO feedback and resource control, and bridges them through CPU-throttle-based performance targets. Going forward, we are extending to additional resource types such as memory and storage, and exploring integrations with additional scaling strategies such as horizontal scaling.

Acknowledgments

We are grateful to Yang Yue and Jiayi Mao for their contributions in the early stage of this work. We thank Neeraja J. Yadwadkar for shepherding our paper and the anonymous reviewers for their helpful comments and feedback. We also thank Haidong Wang, Chuanjie Liu, Qianxi Zhang, Yawen Wang, Yu Gan, Fan Yang, Mao Yang, Lidong Zhou, and Victor Bahl for their support or insightful discussions.

References

[1] AWS Auto Scaling. https://aws.amazon.com/autoscaling/.

[2] AWS Predictive Scaling. https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-predictive-scaling.html.

[3] Azure Autoscale. https://azure.microsoft.com/en-us/products/virtual-machines/autoscale/.

[4] Google Cloud Autoscaler. https://cloud.google.com/compute/docs/autoscaler/.

[5] Kubernetes Autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[6] Kubernetes Vertical Pod Autoscaler. https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler#vertical-pod-autoscaler.

[7] Locust: An Open Source Load Testing Tool. https://locust.io.

[8] Sinan Open-sourced Repository. https://github.com/zyqCSL/sinan-local.

[9] Twitter Data for Academic Research. https://developer.twitter.com/en/use-cases/do-research/academic-research/resources. Accessed in 2022.

[10] Vowpal Wabbit. https://vowpalwabbit.org.

[11] Adam Gluck. Introducing Domain-Oriented Microservice Architecture, 2020.

[12] Alekh Agarwal, Daniel Hsu, Satyen Kale, John Langford, Lihong Li, and Robert Schapire. Taming the monster: A fast and simple algorithm for contextual bandits. In *International Conference on Machine Learning*, pages 1638–1646. PMLR, 2014.

[13] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *OSDI*. USENIX, 2020.

[14] Alberto Bietti, Alekh Agarwal, and John Langford. A contextual bandit bake-off. *J. Mach. Learn. Res.*, 22:133–1, 2021.

[15] Dave Chiluk. Unthrottled: Fixing CPU Limits in the Cloud (blog post). https://engineering.indeedblog.com/blog/2019/12/unthrottled-fixing-cpu-limits-in-the-cloud/.

[16] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload Control for μs-scale RPCs with Breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 299–314, 2020.

[17] Byungkwon Choi, Jinwoo Park, Chunghan Lee, and Dongsu Han. pHPA: A Proactive Autoscaling Framework for Microservice Chain. In *APNet*. ACM, 2021.

[18] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *SoCC*. ACM, 2015.

[19] Jianru Ding, Ruiqi Cao, Indrajeet Saravanan, Nathaniel Morris, and Christopher Stewart. Characterizing Service Level Objectives for Cloud Services: Realities and Myths. In *ICAC*. IEEE, 2019.

[20] Miroslav Dudík, John Langford, and Lihong Li. Doubly robust policy evaluation and learning. *arXiv preprint arXiv:1103.4601*, 2011.

[21] David Lo et al. Towards Energy Proportionality for Large-scale Latency-critical Workloads. In *ISCA*, 2014.

[22] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *ASPLOS*. ACM, 2019.

[23] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *ASPLOS*. ACM, 2019.

[24] Alim Ul Gias, Giuliano Casale, and Murray Woodside. ATOM: Model-driven Autoscaling for Microservices. In *ICDCS*. IEEE, 2019.

[25] Giulio Santoli. Microservices Architectures: Become a Unicorn like Netflix, Twitter and Hailo, 2016.

[26] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. PRESS: Predictive Elastic Resource Scaling for Cloud Systems. In *CNSM*. IEEE, 2010.

[27] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*. USENIX, 2011.

[28] Xiaofeng Hou, Chao Li, Jiacheng Liu, Lu Zhang, Shaolei Ren, Jingwen Leng, Quan Chen, and Minyi Guo. AlphaR: Learning-Powered Resource Management for Irregular, Dynamic Microservice Graph. In *IPDPS*. IEEE, 2021.

[29] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. PerfIso: Performance isolation for commercial latency-sensitive services. In *ATC*. USENIX, 2018.

[30] Seyyed Ahmad Javadi, Amoghavarsha Suresh, Muhammad Wajahat, and Anshul Gandhi. Scavenger: A Black-Box Batch Workload Resource Manager for Improving Utilization in Cloud Environments. In *SoCC*, 2019.

[31] Jeremy Cloud. Decomposing Twitter: Adventures in Service Oriented Architecture, 2013.

[32] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goirin, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *OSDI*, 2016.

[33] Anthony Kwan, Jonathon Wong, Hans-Arno Jacobsen, and Vinod Muthusamy. HyScale: Hybrid and Network Scaling of Dockerized Microservices in Cloud Data Centres. In *ICDCS*. IEEE, 2019.

[34] Cheuk Lam, Enlin Xu, and David Blinn. Kubernetes CPU Throttling: The Silent Killer of Response Time — and What to Do About It (blog post). https://community.ibm.com/community/user/aiops/blogs/dina-henderson/2022/06/29/kubernetes-cpu-throttling-the-silent-killer-of-res.

[35] John Langford and Tong Zhang. The Epoch-Greedy Algorithm for Multi-Armed Bandits with Side Information. *NIPS*, 2007.

[36] Kyungwoon Lee, Kwanhoon Lee, Hyunchan Park, Jaehyun Hwang, and Chuck Yoo. Autothrottle: Satisfying Network Performance Requirements for Containers. *IEEE Transactions on Cloud Computing*, 2022.

[37] Stuart Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[38] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. In *ICCE*. IEEE, 2018.

[39] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *ISCA*, 2015.

[40] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *SoCC*. ACM, 2021.

[41] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. In *SIGCOMM*. ACM, 2019.

[42] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. AGILE: Elastic distributed resource scaling for infrastructure-as-a-service. In *10th International Conference on Autonomic Computing (ICAC 13)*, pages 69–82, 2013.

[43] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. GRAF: A graph neural network based proactive resource allocation framework for SLO-oriented microservices. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pages 154–167, 2021.

[44] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *OSDI*. ACM, 2020.

[45] Gourav Rattihalli, Madhusudhan Govindaraju, Hui Lu, and Devesh Tiwari. Exploring potential for non-disruptive vertical auto scaling and resource estimation in Kubernetes. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 33–40. IEEE, 2019.

[46] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. Autopilot: workload autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[47] Vighnesh Sachidananda and Anirudh Sivaraman. Collective autoscaling for cloud microservices, 2021. arXiv:2112.14845.

[48] Tobias Schnabel, Adith Swaminathan, Ashudeep Singh, Navin Chandak, and Thorsten Joachims. Recommendations as treatments: Debiasing learning and evaluation. In *International Conference on Machine Learning*, pages 1670–1679. PMLR, 2016.

[49] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364, 2013.

[50] Vishwanath Seshagiri, Darby Huye, Lan Liu, Avani Wildani, and Raja R. Sambasivan. [SoK] identifying mismatches between microservice testbeds and industrial perceptions of microservices. *Journal of Systems Research*, 2(1), 2022.

[51] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14, 2011.

[52] Software Engineering Laboratory of Fudan University. Train Ticket: A Benchmark Microservice System. https://github.com/FudanSELab/train-ticket.

[53] Akshitha Sriraman and Thomas F. Wenisch. μTune: Auto-tuned threading for OLDI microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, 2018.

[54] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction (second edition)*. MIT press, 2020.

[55] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SoCC*, pages 1–16. ACM, 2013.

[56] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[57] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud. In *EuroSys*. ACM, 2021.

[58] John Wilkes. Google cluster data – 2019 traces. https://github.com/google/cluster-data/blob/master/ClusterData2019.md, 2020.

[59] Zhengxu Xia, Yajie Zhou, Francis Y. Yan, and Junchen Jiang. Genet: Automatic curriculum generation for learning adaptation in networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 397–413, 2022.

[60] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning *in situ*: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, Santa Clara, CA, February 2020. USENIX Association.

[61] Hailong Yang, Quan Chen, Moeiz Riaz, Zhongzhi Luan, Lingjia Tang, and Jason Mars. PowerChief: Intelligent Power Allocation for Multi-Stage Applications to Improve Responsiveness on Power Constrained CMP. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 133–146, 2017.

[62] Yanqi Zhang, Inigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and Cheaper Serverless Computing on Harvested Resources. In *SOSP*. ACM, 2021.

[63] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. Sinan: ML-based and QoS-Aware Resource Management for Cloud Microservices. In *ASPLOS*. ACM, 2021.

[64] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload Control for Scaling WeChat Microservices. In *SoCC*. ACM, 2018.

Appendices

A Application workload details

We present the workload composition used in our experiments. Our workload generator, Locust, follows the ratios specified below when generating requests at a given RPS:

Train-Ticket:

- `Mainpage`: 29.41%
- `Travel`: 58.82%
- `Assurance`: 2.94%
- `Food`: 2.94%
- `Contact`: 2.94%
- `Preserve`: 2.94%

Hotel-Reservation:

- `Search`: 60%
- `Recommend`: 39%
- `Reserve`: 0.5%
- `Login`: 0.5%

Social-Network:

- `Read-home-timeline`: 65%
- `Read-user-timeline`: 15%
- `Compose-post`: 20%

B Vowpal Wabbit usage

The following VW parameters are used in our experiments.

- The doubly robust estimator [20] is employed for policy evaluation: `--cb_type dr`
- Number of available actions: `--cb_explore 81`
- The native ε-greedy is disabled to implement our customized exploration strategy (§3.3.2): `--epsilon 0`
- Number of hidden units in the neural network: `--nn 3`
- Learning rate: `-l 0.5`

We also compare different VW models—a linear model and neural networks with 2, 3, 4 and hidden units, on Social-Network under the same workload patterns (Figure 3). Figure 11 shows that Autothrottle perform similarly across different models. We select the neural network model with 3 hidden units, as it performs slightly better on the bursty workload (as indicated by the lower whiskers in boxplots).

C Microservice clustering

Tower clusters microservices into two groups based on their average CPU usage, denoted "High" and "Low", using a standard k-means clustering algorithm (§3.3.2). Table 2 presents a breakdown of the number of services in each group.

D Microservice replicas

Train-Ticket and Hotel-Reservation deploy each service with one replica. For Social-Network, we employ three replicas of `media-filter-service` except in the large-scale evaluation (§5.5). In §5.5, 6 replicas of `media-filter-service` and 3 replicas of `nginx-thrift` are employed.

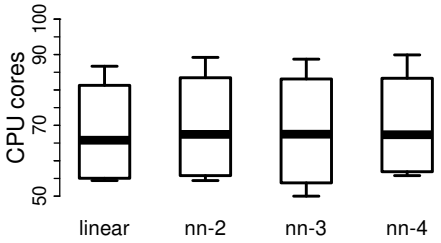


Figure 11: Different VW models—a linear model and neural networks with 2, 3, and 4 hidden units—perform similarly on Social-Network under various workloads (Figure 3).

Application	"High" group	"Low" group
Train-Ticket	8	60
Hotel-Reservation	6	11
Social-Network (160-core cluster)	1	27
Social-Network (512-core cluster)	2	26

Table 2: Number of services in each application assigned to the "High" and "Low" CPU usage groups.

E RPS range of workload traces

We scale the traces presented in Figure 3 to saturate the cluster for each application, as documented in Table 3.

F CPU utilization thresholds in K8s-CPU and K8s-CPU-Fast

In evaluating K8s-CPU and K8s-CPU-Fast, we test and select the best-performing CPU utilization threshold from the set $\{0.1, 0.2, \ldots, 0.9\}$, for each application and each workload trace. The selected thresholds are presented in Table 4.

G Evaluation methodology details

All experiments are performed using one-hour workload traces. Prior to testing, certain applications require additional preparations. We warm up Hotel-Reservation by sending 200 requests per second for 15 seconds and waiting for 60 seconds. For Social-Network, we populate the database with 962 users, 18,812 edges, and 20,000 posts. We then warm up for 3 minutes by incrementally increasing the RPS by 10% every 5 seconds, up to the initial RPS in the one-hour trace. The warm-up phase is excluded from the calculation of P99 latency and resource allocation.

Autothrottle shown in Table 1 is warmed up for 12 hours. The first 6 hours are the (random) exploration stage, followed by 6 hours of normal learning with ε=0.5. We use a separate 1-hour diurnal trace, which is different from the one used for testing but has the same RPS range. Warm up involves running 12 repetitions of this trace. During testing, exploration is turned off completely (with ε set to 0). For Hotel-Reservation,

Workload	Min RPS	Average RPS	Max RPS
Diurnal	145	262	411
Constant	152	200	252
Noisy	75	157	252
Bursty	62	163	442

(a) Train-Ticket

Workload	Min RPS	Average RPS	Max RPS
Diurnal	1721	2627	4003
Constant	1855	2002	2183
Noisy	793	1575	2470
Bursty	768	1633	4037

(b) Hotel-Reservation

Workload	Min RPS	Average RPS	Max RPS
Diurnal	227	394	656
Constant	390	500	588
Noisy	105	236	390
Bursty	104	245	648
Long-term (§5.4)	1	230	592

(c) Social-Network

Workload	Min RPS	Average RPS	Max RPS
Diurnal	479	787	1214
Constant	882	1001	1131
Noisy	232	472	771
Bursty	205	489	1266

(d) Social-Network, large-scale evaluation (§5.5)

Table 3: The RPS range of workload traces after being scaled to saturate the cluster for each application.

RPS is grouped into bins of 200 due to its high RPS, while other applications use the default bin size of 20.

H Captain performance

We demonstrate Captain's ability to achieve Tower's given performance target of CPU throttle ratio (§3.2). This is one factor that determines Autothrottle's effectiveness in maintaining the end-to-end SLO. To this end, Figure 12 dives into Social-Network and illustrates two services from "High" and "Low" CPU usage groups (§C): `media-filter-service` and `post-storage-service`. Two subfigures compare the target throttle ratio and the actual throttle ratio, over a period of 60 min. Captain's heuristics meets the targets relatively well and reacts quickly to target changes, especially when the target is low (Figure 12b). In Figure 12a, we note that the actual throttle ratio is lower than the target. The reason is that the throttle ratio is very sensitive to CPU allocation, especially when the target is high. As a result, Captain tries to err on the safe side, and it can over-allocate to avoid exceeding the targeted throttle ratio due to estimation errors.

Workload	K8s-CPU	K8s-CPU-Fast
Diurnal	0.4	0.6
Constant	0.6	0.6
Noisy	0.5	0.7
Bursty	0.5	0.6

(a) Train-Ticket

Workload	K8s-CPU	K8s-CPU-Fast
Diurnal	0.7	0.7
Constant	0.7	0.8
Noisy	0.6	0.7
Bursty	0.5	0.7

(b) Hotel-Reservation

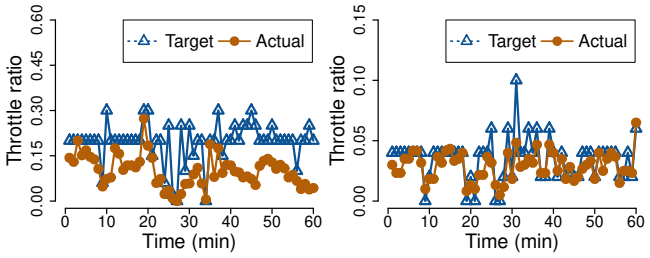
Workload	K8s-CPU	K8s-CPU-Fast
Diurnal	0.5	0.5
Constant	0.5	0.6
Noisy	0.5	0.4
Bursty	0.5	0.4
Long-term (§5.4)	0.5	–

(c) Social-Network

Workload	K8s-CPU	K8s-CPU-Fast
Diurnal	0.6	0.7
Constant	0.5	0.8
Noisy	0.5	0.5
Bursty	0.5	0.7

(d) Social-Network, large-scale evaluation (§5.5)

Table 4: The best-performing CPU utilization thresholds for comparison baselines, per application and workload trace.



(a) `media-filter-service` (b) `post-storage-service`

Figure 12: Captain is able to follow the given performance target over time, by adjusting per-service resources.

Jolteon: Unleashing the Promise of Serverless for Serverless Workflows

Zili Zhang Chao Jin Xin Jin

School of Computer Science, Peking University

Abstract

Serverless computing promises automatic resource provisioning to relieve the burden of developers. Yet, developers still have to manually configure resources on current serverless platforms to satisfy application-level requirements. This is because cloud applications are orchestrated as serverless workflows with multiple stages, exhibiting a complex relationship between resource configuration and application requirements.

We propose Jolteon, an orchestrator to unleash the promise of automatic resource provisioning for serverless workflows. At the core of Jolteon is a stochastic performance model that combines the benefits of whitebox modeling to capture the execution characteristics of serverless computing and blackbox modeling to accommodate the inherent performance variability. We formulate a chance constrained optimization problem based on the performance model, and exploit sampling and convexity to find *optimal* resource configurations that satisfy user-defined cost or latency bounds. We implement a system prototype of Jolteon and evaluate it on AWS Lambda with a variety of serverless workflows. The experimental results show that Jolteon outperforms the state-of-the-art solution, Orion, by up to 2.3× on cost and 2.1× on latency.

1 Introduction

Serverless computing [1–6] aims to simplify cloud programming and relieve developers from infrastructure management. It exposes cloud functions as a key abstraction to developers. Developers use cloud functions to build cloud applications, and the serverless computing platform handles underlying hardware resource management. Serverless computing provides fine-grained resource elasticity and billing at the granularity of functions.

The potential benefits of serverless computing attract many applications, such as data analytics [7–9], video processing [10–12], and machine learning [13–15]. Applications are typically orchestrated as *serverless workflows* on serverless computing platforms [16–19]. Specifically, an application is decomposed into a set of functions, and a serverless workflow corresponds to a directed acyclic graph (DAG) that organizes these functions to implement the application logic.

Cloud platforms are expected to satisfy application-level requirements for cloud applications [20–22]. These requirements typically refer to latency and cost bounds. For example, a developer may expect the end-to-end latency to process a request for an application (i.e., the corresponding serverless workflow) to be no larger than 100ms, or the per-request cost to be no larger than $1. There exists a trade-off between latency and cost. Provisioning more resources results in lower latency but higher cost, and vice versa. Given a bound on one metric (e.g., a latency bound), it is desirable to provision resources to minimize the other metric (e.g., minimize cost).

Serverless computing promises automatic resource provisioning to relieve the burden of developers. Resource provisioning includes two parts, i.e., resource configuration (i.e., the function instance size and the number of function instances for each stage for a workflow instance) and resource scaling (i.e., the number of workflow instances). While current serverless platforms provide auto-scaling based on the realtime load, developers still have to manually configure resources for workflows to satisfy application-level requirements. AWS Lambda provides Power Tuning [23] for developers to profile the latency-cost curve for a particular function and optimize the configuration for one function based on the latency-cost curve. However, Power Tuning does not support tuning the serverless workflow.

Several recent works have explored resource configuration for serverless workflows [7–9, 24, 25]. Orion [24] uses a blackbox model to approximate the latency-cost curve for a workflow, and a heuristic algorithm to search for resource configurations. The nature of the blackbox approach results in inaccurate models and the resource configurations found by the heuristic are sub-optimal. Ditto [9] enables developers to optimize either latency or cost for a workflow. It only provides two extremes (minimum latency or minimum cost), and does not allow developers to explore other trade-offs between latency and cost. A 7% sacrifice on latency may bring a 1.8× reduction on cost, which may be more preferable than the configuration with the minimum latency.

We present Jolteon, a workflow orchestrator that provides automatic resource configuration to satisfy application-level requirements for serverless applications. Developers only need to specify either a latency or cost bound for a workflow. Jolteon automatically configures resources to minimize the execution latency for a cost bound or minimize the cost for a latency bound. By doing so, Jolteon delivers a serverless experience for workflow orchestration, and more importantly, enables developers to navigate the entire latency-cost Pareto front of the configuration space.

There are two challenges in realizing Jolteon. The first challenge is to build an accurate performance model to capture the complex relationship between resource configuration and application requirements. Existing works [7–9, 24, 25] rely

	Jolteon	Orion [24]	Ditto [9]	Caerus [8]	Locus [7]	Aquatope [25]	Stepconf [26]	CostPre [27]	Power-Tuning [23]	Cose [28]
Analytical performance model?	Y	N	Y	Y	Y	N	Y	N	Y	Y
Distribution-aware performance model?	Y	Y	N	N	N	Y	N	Y	N	N
Achieving the Pareto front and guaranteeing the performance bound?	Y	N	N	N	N	N	N	N	Y	Y
Supporting serverless workflow?	Y	Y	Y	Y	Y	Y	Y	Y	N	N

Table 1: The design space of Jolteon against existing works of serverless computing.

on either blackbox modeling or whitebox modeling. Blackbox models capture the performance variability of serverless computing, but lack explainability and ignore the execution characteristics of serverless workflows. In contrast, whitebox models employ deterministic formulas to represent the execution characteristics, which is faster and more accurate on average prediction. However, whitebox models ignore the inherent variability of serverless computing and fail to guarantee the performance bound. Jolteon builds a stochastic performance model. The model uses analytical formulas to capture the execution characteristics of workflow execution, which make the model more accurate and efficient (i.e., the benefit of whitebox modeling). The model uses stochastic functions to capture the performance variability, which can be used to bound the performance (i.e., the benefit of blackbox modeling). This approach leverages the benefits of whitebox and blackbox models, and avoids their drawbacks.

Given the stochastic performance model, the next challenge is to find the optimal resource configuration under a latency or cost bound. As we use random variables to model the execution variability, the problem is mathematically formulated as a chance constrained optimization problem, which is a type of stochastic optimization problem. To solve the problem, we first convert it to a deterministic formulation via Monte Carlo sampling and guarantee the performance bound through a novel inequality. Enumerating all possible configurations under the deterministic formulation is not practical given the large search space and the vast number of constraints. We prove that our formulation is convex. Thus, we leverage the convexity and use an efficient gradient descent algorithm to find the optimal resource configuration under the bound.

We implement a system prototype of Jolteon and evaluate it on AWS Lambda with a variety of serverless workflows. The experimental results show that Jolteon outperforms Orion [24] by up to 2.3× on cost and 2.1× on latency. Compared to Ditto [9] which provides either minimum latency or minimum cost, Jolteon is able to reduce cost by 1.8× or latency by 3.3×, with a ≤11% reduction on the other metric. The evaluation also confirms that Jolteon can satisfy different latency or cost bounds given by users.

In summary, we make the following contributions.

- We present Jolteon, a workflow orchestrator for serverless computing that provides automatic resource configuration to satisfy application-level requirements.
- We propose a stochastic performance model that captures both the execution characteristics and variability, and a bound guaranteed sampler to transform the stochastic optimization problem. We further prove the convexity of the problem and apply a gradient descent algorithm to find the latency-cost Pareto front of the configuration space.
- We implement a Jolteon prototype. The experimental results show that Jolteon outperforms the state-of-the-art solution, Orion, by up to 2.3× on cost and 2.1× on latency.

2 Background and Motivation

In this section, we first introduce the background of serverless computing and application workflows. Then we discuss the limitations of existing work, which motivate the design of Jolteon. Finally, we describe the technical challenges to find the optimal resource configuration that satisfies performance bounds for serverless workflows.

2.1 Serverless Workflows

Serverless computing simplifies cloud programming for cloud application developers. It enables developers to create short-lived, stateless functions triggered by events (e.g., HTTP request). Serverless platforms are responsible for resource scaling and fault tolerance, allowing developers to concentrate on application logic without managing cloud resources [5, 6]. Known for its high elasticity and pay-as-you-go billing, serverless computing meters function run-time at a fine granularity, one millisecond in major platforms [1–3], and bills users only for the resources consumed during function execution.

These benefits lead to a shift of traditional serverful applications to serverless platforms across many fields, including data analytics [7, 8], video processing [10–12], machine learning [13–15], and vector query processing [29]. A serverless application is orchestrated into a workflow, represented as a DAG. Each stage (i.e., node) in the DAG consists of a collection of parallel function instances. The quantity of cloud resources dedicated to a stage is the product of the number of function instances and the function size (e.g., the number of vCPUs). Edges in the DAG denote data dependencies between consecutive stages. The intermediate data is transmitted between stages via external storage services, such as S3 [30],

Developers expect that cloud applications are executed to meet different application-level requirements. For instance, they typically prioritize latency in *online* applications and

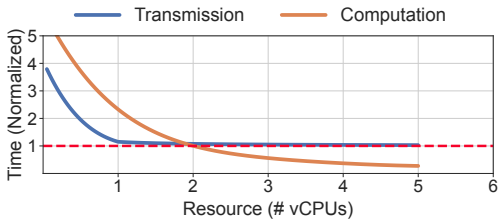


Figure 1: Ditto's performance model.

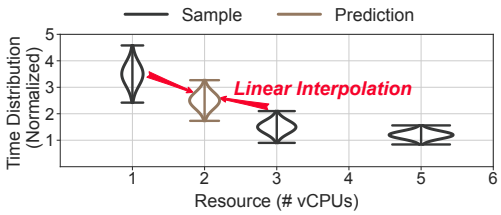


Figure 2: Orion's performance model.

focus on cost efficiency for *offline* applications. A key requirement for workflow resource configuration is to minimize one metric (e.g., minimize cost) while adhering to the limit on another (e.g., a latency bound). Current serverless platforms only offer auto-scaling to scale the workflow instances, and delegate resource configuration (e.g., the function size and the number of function instances in a workflow stage) to developers. To meet the application-level requirements, developers have to understand the resource-to-performance mapping of serverless workflows. This is cumbersome and error-prone, as identifying the optimal trade-off between latency and cost (i.e., Pareto front) is complex and most developers lack expertise in the underlying system characteristics.

2.2 Existing Work and Challenges

In this subsection, we compare Jolteon with existing works and describe the challenges. Table 1 summarizes the key differences of Jolteon against other resource management systems of serverless computing.

Whitebox and analytical performance model. To achieve automatic resource configuration, an essential prerequisite is to capture the resource-to-performance relationship with a comprehensive and accurate performance model. Several works [7–9] leverage whitebox approaches to deterministically predict the function execution time with analytical equations. For instance, Locus [7] observes that the data shuffling time decreases with the increase in resources. Based on Locus, Caerus [8] and Ditto further divide the execution into several fine-grained steps (e.g., transmission and computation). They build individual equations for each step based on its logic.

As depicted in Figure 1, Ditto's performance model divides function execution into transmission and computation steps. The model recognizes that the transmission time remains constant when the number of vCPUs surpasses one, given that a single vCPU can fully saturate the network bandwidth. Hence, it employs a piecewise function, using one vCPU as the boundary, to characterize the relationship between resources and transmission time. These efforts capture the underlying logic of serverless computing and achieve fast and accurate predictions of *average* performance. However, the model neglects the intrinsic performance *variability* in serverless computing, thus failing to guarantee the performance bound.

Blackbox and distribution-aware performance model. Other works [24, 25, 27] note that real-world serverless platforms suffer from performance *variability*. This variability stems from the skew of the input data characteristics, network traffic, workflow structure, and invocation pattern, etc. To address this, these works [24, 25, 27] employ distribution-aware performance models to capture such performance variability. Specifically, they first collect function execution traces as sampling data, and then fit a blackbox model to cover the function execution process.

Among these, Orion [24] utilizes a linear interpolation method. As depicted in Figure 2, the two sampling distributions are illustrated by probability distribution functions. The predicted distribution is fitted through linear interpolation. Aquatope [25] and CostPre [27] rely on Bayesian optimization and mixture density networks, respectively. Although these techniques capture the performance variability, they overlook underlying serverless computing characteristics, resulting in imprecise predictions and time-consuming blackbox fitting. The characteristics of serverless computing refer to the step-by-step breakdown of function execution, i.e., initialization, transmission, and computation. We can exploit the characteristics of each step to build a more accurate and efficient performance model, e.g., the time of transmission step is influenced by the available network bandwidth. We will discuss the details in §4.1.

Pareto front and performance bound. Optimizing the cost or latency for serverless workflows has been explored in prior work. Caerus [8] and Locus [7] utilize proportional resource allocation based on input data size. However, for applications insensitive to data size, these approaches yield sub-optimal performance. To rectify this, Ditto [9] introduces a bottom-up DAG traversal algorithm to minimize either cost or latency, and achieves near-optimal results. Nevertheless, these studies do not support finding the optimal resource configuration under specific performance bounds, and their deterministic models fail to guarantee the performance bound under the variability of serverless computing.

Orion [24], Stepconf [26] and Aquatope [25] aim to reduce cost within a pre-defined latency budget. Orion, considered as the state-of-the-art, leverages a best-first search algorithm that explores resource configurations and selects the first one that satisfies the given latency constraint. Similarly, Stepconf applies heuristic search methods, while Aquatope resorts to Bayesian techniques. Notably, the search space of the three systems is limited, and they cannot enumerate all possible resource configurations. Consequently, they fail to achieve the Pareto front between latency and cost while guaranteeing the performance bound.

Supporting serverless workflows. At the function level, some works such as AWS Lambda Power Tuning [23] and Cose [28] characterize the deterministic latency-cost curve for specific functions. They profile the latency-cost curve for a particular function, and allow users to optimize latency or cost under a given bound of the other metric. However, a serverless workflow (i.e., a DAG) is composed of many functions with varying data dependencies. These works are inadequate for supporting the optimization of serverless workflows.

Challenges. Our goal is to provide automatic resource configuration to meet application-level requirements. Specifically, the system is expected to configure resources automatically to minimize the execution latency for a user-specified cost bound or minimize the cost for a user-specified latency bound. To achieve this goal, two main challenges must be addressed.

The first challenge is to build a performance model to accurately capture the complex relationship between resource configurations and application requirements (e.g., latency and cost). As described above, whitebox models successfully capture the underlying logic of serverless computing and therefore provide fast and accurate predictions of average performance. However, their deterministic nature fails to capture the performance variability. Blackbox models, on the other hand, concentrate on the performance variability but overlook the underlying system characteristics, resulting in imprecise predictions and time-consuming blackbox fitting. It is desirable to develop a performance model that leverages the benefits of whitebox and blackbox while mitigating their shortcomings.

The second challenge is to identify the optimal resource configuration within the given performance bound. With the performance model including distributional factors, the optimization problem becomes stochastic which cannot be solved directly. Even if the problem is deterministic, the huge search space of resource configurations and intricate formulation of the problem make it hard to find the optimal solution. In short, the second challenge lies in the formulation of the stochastic optimization problem with performance guarantees and efficiently solving the problem to find the optimal result.

3 Jolteon Overview

Jolteon is a serverless workflow orchestrator that facilitates automatic resource configuration to meet application-level requirements (i.e., latency or cost). It employs a novel stochastic performance model to accurately profile the resource-to-performance relationship in serverless computing (§4.1). It formulates the chance constrained optimization problem. It then converts it into a deterministic problem with Monte-Carlo sampling and guarantees the performance bound via a novel inequality (§4.2). It employs a gradient descent algorithm that leverages the problem's convexity to achieve the optimal result (§4.3). Figure 3 shows the overview.

User interface. Users define their serverless workflow DAGs and requirements (latency or cost bounds) as input to Jolteon.

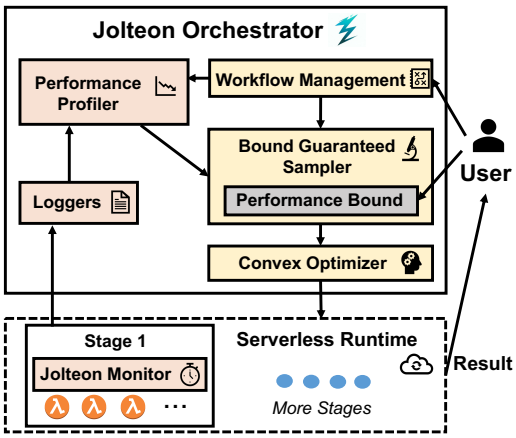


Figure 3: Jolteon overview.

A workflow DAG includes the functions for each stage and the data dependencies between the stages. Jolteon assesses the validity of a DAG (e.g., confirming the absence of cycles) and deploys the functions to the serverless platform. Jolteon then processes the invocation requests for the workflow, and executes the workflow with the optimal resource configuration. The results are returned upon the completion of the requests.

Jolteon orchestrator. The orchestrator receives workflow DAGs and requirements from users, and generates resource configurations. It contains the following components.

Performance profiler. After a workflow is registered, Jolteon's performance profiler periodically polls the data logs of the corresponding workflow. It learns and updates the performance model (§4.1) of each workflow stage, modeled by stochastic functions. The stochastic functions are able to leverage the advantages of both whitebox and blackbox models while mitigating their shortcomings. Specifically, the performance model treats every parameter in the analytical formula as a random variable and fits it as a distribution.

Bound guaranteed sampler. Jolteon formulates the chance constrained optimization problem based on the workflow information and user-specified bound. In the formulation, resource configurations are regarded as independent variables. Since chance constrained optimization cannot be solved directly, this module converts it into a deterministic problem through Monte-Carlo sampling. More samplings lead to a higher confidence level, but more complicated formulation with larger solving time. Jolteon proposes a novel inequality to decide the minimal sample size that guarantees the performance bound with a high confidence level (§4.2).

Convex optimizer. After sampling, the chance constrained optimization problem is converted into a deterministic optimization problem. Solving such problem with existing algorithms [31] is both time-consuming and sub-optimal. Jolteon leverages one key insight: the problem is convex. Consequently, Jolteon employs a gradient descent algorithm to efficiently find the optimal resource configuration (§4.3). Moreover, Jolteon prunes the constraints with the support constraint

Symbol	Description
d_i	The number of function instances of the i_{th} stage
v_i	The number of vCPUs of one function in the i_{th} stage
$\mathbf{d}$	The array of d_i of all stages
$\mathbf{v}$	The array of v_i of all stages
$T_j(d,v)$	The transmission time of the j_{th} function in one stage
$C_j(d,v)$	The computation time of the j_{th} function in one stage
$LS_i(d,v)$	The latency of the i_{th} stage
$CS_i(d,v)$	The cost of the i_{th} stage
$LW(\mathbf{d},\mathbf{v})$	The latency of the entire workflow
$CW(\mathbf{d},\mathbf{v})$	The cost of the entire workflow

Table 2: Key notations in the design.

technique to further reduce formulation complexity. The configuration is then sent to the serverless runtime for execution.

Serverless runtime. The serverless runtime receives the serverless workflow and the optimal resource configuration from the orchestrator. It then executes (i.e., through function invocations) the workflow and records the execution logs. The logs are used by the profiler to build the performance model. The execution results are returned to the users.

4 Jolteon Design

In this section, we first describe the stochastic performance model (§4.1). Then we formulate the chance constrained optimization problem and introduce the bound guaranteed sampler (§4.2). Finally, we introduce the convex optimizer to solve the problem (§4.3). The key notations are listed in Table 2. Lowercase symbols represent deterministic values (e.g., x), uppercase symbols represent random variables (e.g., X), and bold symbols represent vectors (e.g., $\mathbf{x}$).

4.1 Performance Profiler

Jolteon's performance profiler collects the function logs to periodically update the stochastic performance model. In this subsection, we introduce the details of the stochastic performance model which integrates the benefits of both whitebox and blackbox models while mitigating their drawbacks.

The model first divides each workflow stage (i.e., each node of the DAG) into *functions* and *phases*. A stage consists of many parallel function instances. The latency of the *stage* is the *maximum* latency of all function instances. Each function instance is divided into two phases: *initialization* and *execution*. The initialization phase refers to setting up the function environment while the execution phase is to run the user code. The phase is further divided into fine-grained *steps* as follows.

Initialization phase. The initialization phase is to receive the request and set up the function environment. The first step involves a network delay and front door execution. The network delay spans from the moment the function is triggered to when it reaches the API gateway. It is inherently variable, influenced by the network status. Subsequently, front door execution includes request authentication, function routing,

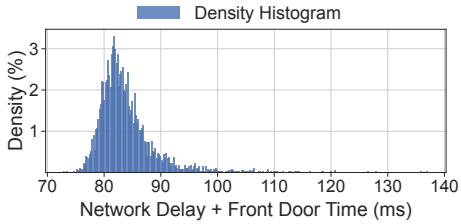


Figure 4: Time distribution of the first step in initialization.

and load balancing. The latency of this process is likewise unpredictable, shaped by the overall system load. This step is inevitable and irrelevant to resource configuration. We have gathered data on the latency from AWS Lambda, where functions are triggered at 30-second intervals via HTTP requests. We omit the failed invocation requests and only record the time of requests with a status code of 200. Evidently, this latency is not stable, as illustrated in Figure 4. We model the latency with a random variable, denoted by D.

The next step is to set up the function environment. The serverless platform determines if a corresponding function instance (e.g., VM or container) is present in the instance pool. In the absence of such an instance (i.e., cold start), the platform will allocate a new one, pull the function image, and initiate the function runtime. Conversely, if an instance does exist (i.e., warm start), the existing function instance is reused. The latency is negligible for warm start. Jolteon profiles cold or warm start according to the specific policy (e.g., pre-warming or keep-alive) and employs the following formula to approximate the time of this step, where C is a random variable of cold start time.

$$G = \begin{cases} 0, & \textit{Warm start} \\ C, & \textit{Cold start}. \end{cases}$$

Execution phase. In the execution phase, the function includes two steps: data transmission and computation. Typically, a workflow stage downloads the data from its preceding stage, processes the data, and uploads the result to its subsequent stage. The allocation of computational resources has a notable influence on the latency of these two steps. Specifically, more vCPUs result in higher network bandwidth and higher computational capacity.

Transmission step. In the data transmission step, data is either downloaded from or uploaded to external storage. We model downloading as an example. Relevant factors for this step include the data size, the available network bandwidth, and the API overhead. The input data may vary due to the diversity of data sources and the stochastic nature of data generation. We model input data size (denoted as S) as a random variable. The stage comprises d parallel function instances. Let the available network bandwidth and the number of vCPUs of a function instance be b and v, respectively. Our goal is to formulate a model for the transmission time of a function, based on the given resource configuration, d and v.

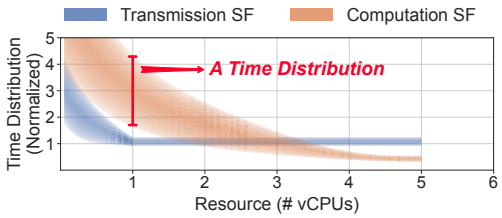


Figure 5: Jolteon's stochastic performance model.

The data transmission time for a serverless function has two components: the propagation time and API invocation overhead. The propagation time is influenced by the data size and the available network bandwidth. The data of the entire stage (with size of S) is partitioned into d parts, one for each function instance. The propagation time of each function instance is $S/(d \times b)$, where b represents the available bandwidth. b scales with the number of allocated vCPUs (i.e., v). However, the bandwidth reaches a saturation point when v is sufficiently large. Thus, b is approximated by $min(v \times W, B)$, where B is the maximum bandwidth and W is the per-vCPU bandwidth. The API invocation overhead is independent of resource allocation and denoted by O_T. Therefore, the transmission time for a function instance is summarized as follows:

$$T(d,v) = \frac{S}{d \times min(v \times W,\ B)} + O_T. \tag{1}$$

Computation step. In the computation step, each function handles a data subset of size S/d with v vCPUs. However, the function logics are diverse. For example, in the frame extraction stage of video analytics, the function ingests S/d videos and cycles through each frame with multi-core processing. The computation time is $A \times \frac{S}{dv}$, where A is the time to process one video. Conversely, in the map stage of data analytics, the function sorts the S/d data units. Since the time complexity of sort is $O(x \ln x)$, the time is $B \times \frac{S}{dv} \times \ln \frac{S}{dv}$, where B is the time to operate one data unit with one vCPU.

To tackle such complexity, we propose a model that mathematically characterizes the distinct computational logics with appropriate equations. Specifically, we focus on two typical function logics: those with polynomial complexity and logarithmic complexity. The execution time of polynomial logic is represented by $\sum_{i=0}^{l} A_i \times x^i$, and the time for logarithmic logic is $\ln x \times (\sum_{i=0}^{m} B_i \times x^i)$. Here, A_i and B_i are positive random variables and x is defined as $\frac{S}{dv}$. For functions without the relevant logic, the coefficients are set to zero. Consequently, the computation time of a function instance is expressed as:

$$C(d,v) = \sum_{i=0}^{l} A_i \times (\frac{S}{dv})^i + \ln \frac{S}{dv} \times \sum_{i=0}^{m} B_i \times (\frac{S}{dv})^i. \tag{2}$$

l and m are decided when fitting the parameters (A_i, B_i) with historical data. Specifically, A_i is fitted one by one in ascending order. The fitting process terminates when a specific number of zero A_i are encountered, and l is the last index i. The profiler monitors if the function supports multi-processing. If latency remains the same when $v > 1$, it is deemed single-processing. Consequently, the function's performance model restricts v's feasible domain to $(0,1]$. Another concern is that Formula 2 is not applicable to function logic with other complexities (e.g., exhaustive search with exponential complexity rather than polynomial complexity). However, since the feasible domain of v and d is discrete and finite (e.g., $v \in (0,6]$), other complexities can be approximated by the above polynomial complexity, i.e., Lagrange interpolating polynomial.

Function model to stage model. The aforementioned performance model pertains to an individual function instance. We must consider the d parallel function instances for a stage. The latency of the entire stage is the maximum latency of the d parallel functions. We define the stage latency as $LS(d,v)$:

$$LS(d,v) = \max\left\{D_j + G_j + T_j(d,v) + C_j(d,v), j = 1 \ldots d\right\}. \tag{3}$$

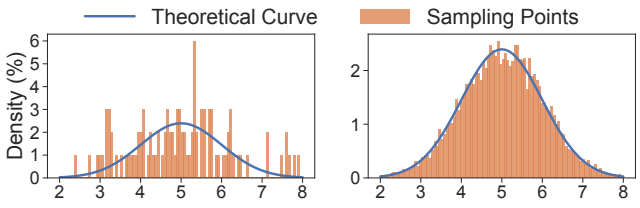
Regarding the cost of a stage, current serverless platforms do not bill users for initialization time. Therefore, the cost is defined as follows, where α is the cost per second of one vCPU and β is the cost per invocation.

$$CS(d,v) = \sum_{j=1}^{d} \left((T_j(d,v) + C_j(d,v)) \times v \times \alpha + \beta\right). \tag{4}$$

Stochastic model. The above formulas serve as analytical (i.e., whitebox) models, and the parameters (e.g., C, W, A_i) are fitted with historical data. The independent variables d and v are the resource configuration. As we discussed in §2.2, treating the parameters (e.g., C, W, A_i) as static values fails to account for the inherent variability. We instead model these parameters as random variables, transforming the deterministic functions to stochastic functions. These random variables are fitted as distributions and marked as uppercase letters. As shown in Figure 5, when the resource configuration is given, the stochastic function (SF) outputs a distribution of the predicted latency rather than a fixed value. Compared to blackbox models, this stochastic model captures the underlying system characteristics through the simple yet potent formulas, which allows higher accuracy and lower fitting overhead. In short, this stochastic model captures not only the variability but also the underlying system characteristics, which leverages the benefits of both whitebox and blackbox models.

4.2 Bound Guaranteed Sampler

Formulation of chance constrained optimization. We first extend the above stage performance model to the entire workflow. In the workflow DAG, the latency of the complete workflow is modeled as the `maximum` latency of paths in the DAG. One path is a sequence of stages that are connected by edges and its latency is the `sum` of all sequential stages' latency. The workflow latency, denoted as $LW(\mathbf{d}, \mathbf{v})$, is calculated through



(a) Sample size = 100. (b) Sample size = 10,000.

Figure 6: Sampling under different sample size.

either addition or maximum on stage latency, $LS_i(d_i, v_i)$. Here, $\mathbf{d}$ and $\mathbf{v}$ are p-dimensional arrays where d_i and v_i specify the resource configuration of the i-th stage (p stages in total). As for the cost of the workflow, denoted as CW, it is the sum of all stage cost, i.e., $CW(\mathbf{d},\mathbf{v}) = \sum_{i=1}^{p} CS_i(d_i, v_i)$.

We then mathematically formulate this stochastic problem as a chance constrained optimization problem. The objective is to minimize either latency or cost, subject to the constraint of guaranteeing the performance bound. The problem is formulated as follows (with cost as the objective function):

$$\begin{aligned} &Minimize \quad CW(\mathbf{d},\mathbf{v}). \\ &s.t. \quad Confidence\big(LW(\mathbf{d},\mathbf{v}) \leq \varepsilon_l\big) \geq \delta. \end{aligned} \tag{5}$$

In this formulation, certain parameters are random variables and the constraint mandates that the latency bound ε_l be guaranteed with a predetermined confidence level δ. The formulation to minimize latency under a cost bound is similar.

Bound guaranteed sampling. Chance constrained optimization, a subfield of stochastic optimization, deals with uncertain parameters in constraint functions, while ensuring the constraints are guaranteed with a high confidence level. Such problem has been extensively studied in various research fields [32–34]. However, adapting this method to our specific problem introduces two challenges.

The first challenge is that the objective and constraint functions contain random variables. Jolteon employs Monte Carlo sampling to transform the chance constraint into a set of deterministic constraints. As for the objective function, the random variables are usually regarded as their expectation values in optimization. Let function $G(\mathbf{d},\mathbf{v},\mathbf{X})$ be the constraint function in Formula 5, where $\mathbf{X}$ is the set of the random variables defined in §4.1, represented as a random vector. For example, $G(\mathbf{d},\mathbf{v},\mathbf{X}) = LW(\mathbf{d},\mathbf{v}) - \varepsilon_l$ if user specifies a latency bound. Monte Carlo sampling is used to sample n vectors from the random variables in the constraint functions, e.g., $\mathbf{x_i}$ is a deterministic sample vector of $\mathbf{X}$. This converts the original stochastic constraint into a series of deterministic constraints:

$$\{G(\mathbf{d},\mathbf{v},\mathbf{x_i}) \leq 0, \ \ for\ i = 1...n\}. \tag{6}$$

All random variables are replaced by their corresponding sample values. The deterministic problem can be solved directly.

The second challenge is to determine the number of samples to ensure the performance bound with a high confidence level. Conceivably, the more samples, the higher confidence the solution has. For example, assume that the random vector, $\mathbf{X}$, only contains one random variable, X. X conforms to normal distribution as shown in Figure 6. The figure contrasts two different sample sizes: Figure 6(a) uses 100 samples, while Figure 6(b) uses 10,000 samples. More samples yield a higher confidence. Specifically, the prediction error for the 95% percentile is 13% in Figure 6(a) and only 0.15% in Figure 6(b). However, a large sample size introduces high sampling overhead and intricate problem formulation with large solving time. The two illustrations in Figure 6 exemplify a single random variable on two specific sample sizes. However, the actual problem arises with many random variables, which further complicates the determination of sample size.

Jolteon leverages Hoeffding's inequality and sample approximation theory to find the lower bound of sample size n. This lower bound allows Jolteon to guarantee the performance bound at confidence level δ while minimizing the sampling overhead and simplifying the problem formulation. The lower bound of sample size is as follows:

$$\frac{1}{2\times(1-percentile)^2} log(\frac{|D|}{1-\delta}). \tag{7}$$

In this content, D denotes the domain of independent variables $\mathbf{v}$ and $\mathbf{d}$. δ refers to the confidence level. Jolteon sets δ to 99.9% which is large enough for most user cases. *percentile* refers to the percentile for the target bound, such as P95 latency bound. The two parameters are both configurable for developers. The theoretical proof of the lower bound of n to guarantee the performance bound is given in Appendix A.1. The proof is based on Hoeffding's inequality [35] and sample approximation theory [36, 37].

The definition domain, D, depends on the structure of workflow DAG and is a large value due to the large resource configuration space. In real workflow executions, n may reach up to thousands, according to Formula 7. However, the subsequent convex optimizer reduces such number with pruning technique and the evaluation (§6.4) shows that solving with thousands of constraints is still efficient and fast.

4.3 Convex Optimizer

The sampler transforms the chance constrained optimization problem (Formula 5) into a deterministic problem with a set of constraints, i.e., random variables become the sampling values. Thus, we replace the uppercase symbols with lowercase symbols. However, solving such problem at runtime remains a challenge due to the complex constraints and huge search space. Jolteon leverages an important insight: the optimization problem is *convex*. A convex problem means the objective function and all constraints are convex functions. This confers a significant advantage—the local extremum also serves as the global extremum, enabling rapid, optimal solutions via established algorithms [31]. We first prove the convexity of the above deterministic problem.

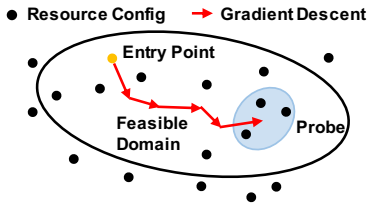


Figure 7: Process diagram of optimization algorithm.

Convexity analysis. According to the definition of convex optimization problems [38], we need to prove that $CW(\mathbf{d},\mathbf{v})$ and $LW(\mathbf{d},\mathbf{v})$ are convex, where $(\mathbf{d},\mathbf{v})$ are independent variables and all other parameters are constants after sampling.

Since the sum and maximum of convex functions remain convex, we only need to prove that $CS_i(d_i,v_i)$ and $LS_i(d_i,v_i)$ are convex. Formula 3 and 4 further simplify the proof by narrowing the focus on proving $T(d,v)$ and $C(d,v)$ are convex. We focus on one stage and omit the index i. All variables and constants are positive due to their real-world meanings.

For the computation step, $C(d,v)$ (i.e., Formula 2), is the sum of two parts: the polynomial part and the logarithmic part. Function $f(x)=x^i$ is convex and monotonically increasing. $\frac{s}{dv}$ is convex because its Hessian matrix is positive-definite. Therefore, the polynomial part is convex due to the convexity of composite functions. Since s is significantly larger than $d\times v$ due to its real-world meaning (i.e., $\frac{s}{dv}>1$), the logarithmic part is also convex according to its positive-definite Hessian matrix. Therefore, $C(d,v)$, the sum of two parts, is a convex function. For the transmission step, $T(d,v)$ (i.e., Formula 1), is also a convex function due to the convexity of piecewise maximum functions, where s,w,o_t are constants.

The detailed analysis of the convexity of the above sum, maximum, composite, logarithmic, polynomial, and piecewise maximum functions is given in Appendix A.2. In summary, the sampling deterministic problem is convex.

Optimization algorithm. Based the convexity, we propose an efficient algorithm to optimally solve the problem, as outlined in Figure 7. Each point in the figure represents one resource configuration (i.e., $\mathbf{d},\mathbf{v}$). The feasible domain circles out the points that satisfy the sampling constraints. The first procedure is a gradient descent algorithm. It starts from a random entry point and takes iterative steps following the gradient until it approaches a local extremum. The second procedure is a probe process. Due to the discrete nature of resource configurations in the real world, the continuous local extremum may not be feasible. To address this, the probe process iteratively examines feasible points surrounding this extremum to find the final configuration. Given that the problem is convex, the local extremum is also globally optimal. Thus, the optimization algorithm is capable of identifying the optimal solution under a given performance bound (i.e., Pareto front).

An excessive number of sample constraints complicate the optimization process and exacerbate the constraints checking overhead. To mitigate this, we employ a support constraint technique to prune redundant constraints. This technique identifies a minimal subset of $\{c_1,c_2,...,c_\alpha\}$ while preserving the feasible domain. If $\mathbf{x_i}<\mathbf{x_j}$ ($\mathbf{x_i}$ is the parameter vector of c_i), then the feasible domain defined by c_j becomes a subset of that defined by c_i. Thus, c_i can be pruned. This is valid under the assumption that all constants and variables are positive and the inequality sign of constraint is $\leq$. For example, if c_1 is $x+y\leq 1$ and c_2 is $2x+2y\leq 1$, the feasible domain of c_2 is encompassed by that of c_1 when $x,y>0$.

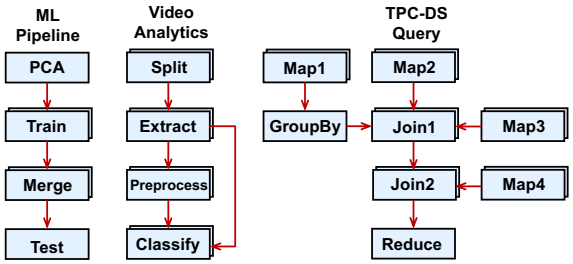


Figure 8: DAGs of the serverless workflows in the evaluation.

The solving time of convex optimizer occupies the majority of Jolteon's orchestration time. In particular, the solving time escalates rapidly with the increasing number of constraints and complexity of the workflow DAG. In §6.4, we evaluate the solving time of the optimizer and show that it is efficient and fast under most use cases. In summary, Jolteon's convex optimizer is capable of identifying the Pareto front, and is efficient with the support constraint pruning technique.

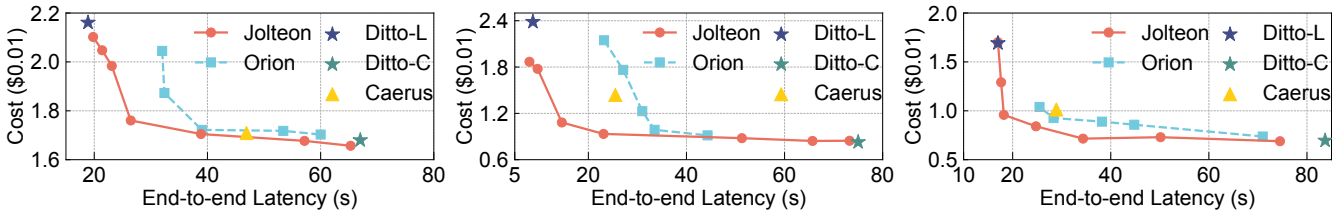
5 Implementation

We implement a system prototype with ~3,800 lines of code in Python. Our prototype supports AWS Lambda [1] as the serverless platform, and uses AWS S3 [30] as the external storage. Jolteon handles execution failures by re-executing the serverless workflow. Existing fault tolerance mechanisms for serverless workflows [39–41] are orthogonal to Jolteon. The code of Jolteon is open-source and is publicly available at https://github.com/pkusys/Jolteon.

Performance profiler. The performance profiler periodically polls the data from the serverless runtime and fits a stochastic performance model through `SciPy` [42]. We employ the nonlinear least squares method to fit the parameters. The profiler updates the expectation value and covariance matrix of each parameter to characterize the distribution. These distributions are then used to generate samples by the sampler.

Bound guaranteed sampler. The sampler module generates a set of sample constraints to transform the original problem into a deterministic problem. The sample size is determined through Formula 7. Since the distributional parameters constitute a multivariate distribution, the sampling process is implemented through joint probability distribution with each parameter's expectation value and covariance matrix.

Convex optimizer. After sampling, we employ the SLSQP (Sequential Least Squares Quadratic Programming) method as the gradient descent algorithm in Figure 7. This method uses a



(a) TPC-DS Query. (b) Video Analytics. (c) ML Pipeline.

Figure 9: Overall performance for different serverless workflows.

series of quadratic approximations to find the optimum under a set of nonlinear constraint functions. Given that it performs optimally when the objective function and constraints are convex, it aligns well with the nature of our problem. The probe procedure is implemented by breadth first search at a fixed depth to iterate the points around the local extremum.

6 Evaluation

In this section, we evaluate Jolteon from the following aspects: (*i*) overall performance against state-of-the-art solutions (§ 6.1); (*ii*) performance guarantees provided by Jolteon (§ 6.2); (*iii*) effectiveness of the performance model (§ 6.3); (*iv*) time analysis for running Jolteon (§ 6.4). We also evaluate the sensitivity of Jolteon to various initial configurations (i.e., entry points) in Appendix A.3.

Setup. We conduct all experiments on AWS. The Jolteon orchestrator is deployed on one c5.12xlarge EC2 instance with 48 vCPUs and 96 GB memory. Jolteon executes the serverless workflow applications on AWS Lambda [1] and uses AWS S3 [30] as the external storage.

Workloads. Our experiments use three representative serverless applications. The workflows of these applications have different characteristics in terms of the number of stages, data dependencies, I/O and compute demands. Figure 8 shows the three workflow DAGs, and the details are as follows.

- **ML Pipeline** is a machine learning workflow adopted from Cirrus [13]. It consists of four stages connected as a chain: dimensionality reduction (PCA), model training, merging, and testing. Using the `LightGBM` library [43], the training stage runs multiple parallel functions to train a set of decision tree models, and the merging stage combines them into a random forest. Each workflow execution trains on 5K images and tests on 2K images of MNIST dataset [44].
- **Video Analytics** is adopted from Pocket [45] and is composed of four stages: video splitting, frame extraction, preprocessing and frame classification. There exists one branch in the workflow, where some frames are preprocessed and then classified by a pre-trained YOLO model [46], while the others are directly dispatched to classification. Each workflow execution processes 32 YouTube videos in "Music" and "News" categories, each with one minute duration.
- **TPC-DS Query** is a data analytics job (Query 95 in TPC-DS benchmark [47]). This workflow is composed of eight stages with complex dependencies. The stages perform a series of `filter`, `groupby`, and `join` operations. Each workflow execution processes 10 GB TPC-DS data.

Baselines. We compare Jolteon with the following baselines.

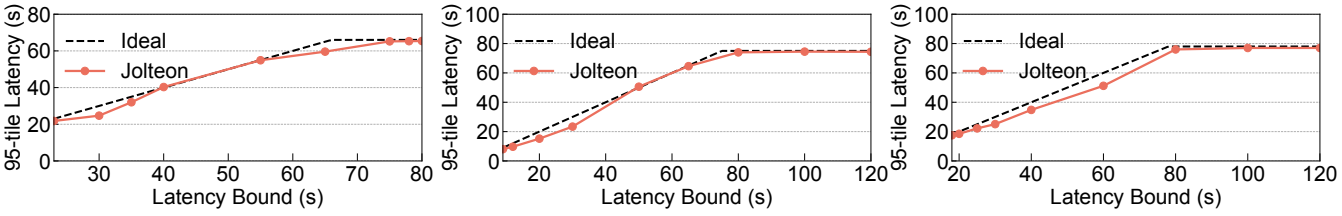
- **Caerus** [8] is a serverless scheduler that uses a heuristic proportional resource allocation strategy based on input data size to optimize latency and cost for a workflow.
- **Ditto** [9] is a state-of-the-art serverless scheduler that utilizes whitebox modeling to provide either minimum latency or minimum cost for a workflow execution, which we refer to as **Ditto-L** and **Ditto-C**, respectively.
- **Orion** [24] is a state-of-the-art serverless scheduler that employs blackbox modeling on the function instance size and uses deterministic numbers of parallel functions. It then uses a heuristic search algorithm with the blackbox model to minimize cost under varying latency requirements.

Metrics. We use end-to-end latency and cost of the workflow execution as the main metrics. The latency refers to the time span from the submission of a workflow request to the receipt of its execution results by the user. The cost is the expense for the serverless function execution, which is extracted from AWS Lambda's "Billed Duration" log entry. We do not use throughput, because throughput is determined by the number of workflow instances, which is controlled by the auto-scaling mechanism of serverless platforms, and the throughput of a single instance, which is the reciprocal of latency.

6.1 Overall Performance

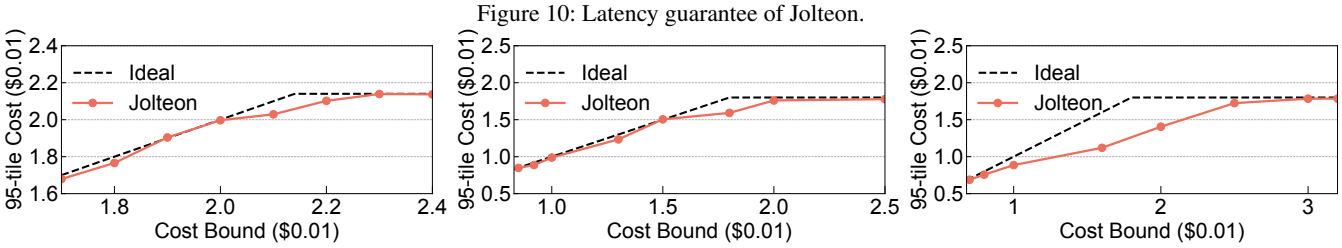
We first compare the overall performance of Jolteon against the baselines. For systems that can meet varying performance requirements (i.e., Jolteon and Orion), we set different latency and cost bounds to obtain the latency-cost curve. For those that do not explore the latency-cost curve (i.e., Ditto and Caerus), we measure the best performance they can achieve. We run the workflows under each resource configuration several times and report the average latency and cost. The results are shown in Figure 9, which we summarize as follows.

- Jolteon outperforms Caerus in *both* latency and cost. Specifically, Jolteon achieves 1.75× lower latency and 1.33× lower cost than Caerus for Video Analytics. This is because the performance model of Caerus only considers the input data size rather than the inherent logic of a serverless function. As a result, it fails to capture the performance accurately and thus the resource configuration is sub-optimal.



(a) TPC-DS Query. (b) Video Analytics. (c) ML Pipeline.

Figure 10: Latency guarantee of Jolteon.



(a) TPC-DS Query. (b) Video Analytics. (c) ML Pipeline.

Figure 11: Cost guarantee of Jolteon.

- Ditto-L and Ditto-C achieve minimum latency and cost, respectively. However, Ditto does not support the trade-off on the Pareto-optimal latency-cost curve. In serverless workflow execution, a small sacrifice on one metric (e.g., latency) may bring a significant reduction on the other metric (e.g., cost). Compared to Ditto, Jolteon is able to trade-off between latency and cost. With only 7% sacrifice on latency, Jolteon reduces the cost by up to 1.77× compared with Ditto-L. And an 11% increase on cost enables Jolteon to reduce the latency by 2.44–3.25× against Ditto-C.
- Orion approximates the latency-cost curve and is able to meet varying performance requirements. But it uses heuristic search that returns as soon as one configuration meets the performance requirement and only adjusts function sizes. This approach significantly narrows the algorithm's search space and is far from the Pareto-optimal. Compared to Orion, Jolteon achieves 1.45–2.07× on latency with the same cost, and 1.04–2.3× on cost with the same latency.

6.2 Performance Guarantees

This set of experiments evaluates the effectiveness of Jolteon to provide performance guarantees. We set the percentile of the target performance requirement to 95%.

Latency guarantee. To evaluate the latency guarantee of Jolteon, we set different latency bounds and measure the latency. Figure 10 shows the actual 95% latency and the ideal latency for the three workflows. The measured actual 95% latency is consistently less than and close to the ideal line, which demonstrates the capability of Jolteon to provide latency guarantee. When we increase the latency bound, the actual 95% latency monotonically increases, indicating that Jolteon can adapt to varying latency bounds. The actual latency does not increase when the bound is loose (e.g., >80 seconds for Video Analytics in Figure 10(b)), since the resource configuration hits the floor to enable the execution.

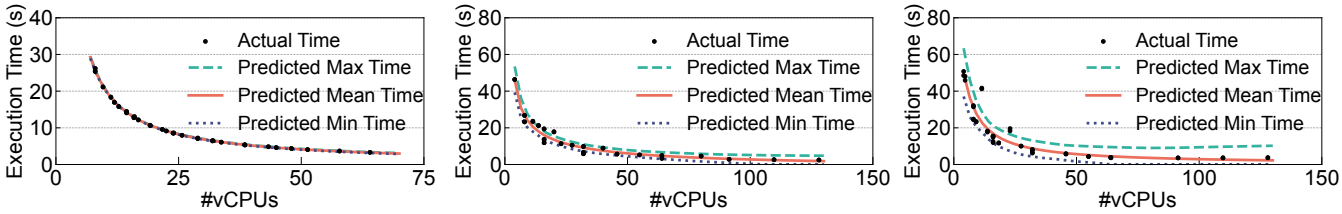
Cost guarantee. We also evaluate the cost guarantee of Jolteon by setting different cost bounds. Figure 11 shows the actual and ideal 95% cost for the three workflows. Similar to the latency guarantee, Jolteon is able to provide bounded cost for all workflows. When the cost bound is relatively loose (e.g., >$0.02 per run in Figure 11(b)), the actual cost is nearly unchanged. This is because the optimization objective is latency. Increasing resources does not further reduce latency. In such cases, Jolteon avoids unnecessary allocation of resources.

6.3 Effectiveness of the Performance Model

To evaluate the performance model, we conduct step-level, stage-level, and workflow-level experiments.

Step-level effectiveness. We first evaluate the performance model for the critical execution steps in the workflow. The critical steps have the longest execution time and the highest cost in the workflow. Figure 12 shows the actual execution time and the predicted time distributions against the total number of vCPUs. The total number of vCPUs is defined as the product of the number of function instances and the number of vCPUs allocated to each function for the step.

The three workflows exhibit different characteristics. TPC-DS Query is an I/O-intensive workflow, where the critical step is to read the data from S3. Since S3 offers a steady bandwidth for reading [7], the execution time of the step under the same vCPU allocation remains stable in Figure 12(a), and can be precisely predicted by the performance mode with less than 1% error. Video Analytics and ML Pipeline are compute-intensive, where the critical steps are computation steps in video splitting and model training stages, respectively. These steps have more variable execution time due to the performance variability of serverless computing, as shown in Figure 12(b) and Figure 12(c). The stochastic performance model predicts the execution time distributions that cover most of the actual execution time.



(a) TPC-DS Query, map1, read. (b) Video Analytics, splitting, compute. (c) ML Pipeline, training, compute.

Figure 12: Effectiveness of the performance model for critical execution steps.

Video Splitting Stage		
Resource Config: (d, v)	**P50**	**P95**
Large: (16, 4)	-6.07%	0.29%
Medium: (8, 2)	6.87%	5.47%
Small: (4, 1)	-5.80%	-4.50%
MAPE	**6.25%**	**3.42%**

Table 3: Errors of Jolteon for stage execution time prediction.

Resource Config	#Funcs (**d**)	#vCPUs/Func (**v**)	#vCPUs in Total
Config 0	(32, 16, 8, 8)	(2.5, 4, 2.5, 2.5)	184
Config 1	(32, 16, 16, 16)	(2, 1.5, 1.5, 2)	144
Config 2	(16, 32, 8, 16)	(2, 1.5, 1.5, 1.5)	116
Config 3	(16, 16, 16, 8)	(1, 1.5, 1.5, 1.5)	76
Config 4	(8, 16, 4, 4)	(1.5, 1, 1, 1)	36
Config 5	(4, 8, 4, 4)	(1.5, 1.5, 1.5, 1.5)	30

Table 4: Six resource configurations to run Video Analytics.

Stage-level effectiveness. Then we use large, medium, and small amounts of resources to examine the effectiveness of the stage-level performance model. Table 3 lists the prediction errors for the video splitting stage (i.e., the critical stage) in Video Analytics. d represents the number of function instances, and v represents the number of vCPUs per function. The error range is [-6.07%, 6.87%]. The mean absolute percentage error (MAPE) is 6.25% for P50 and 3.42% for P95 time prediction, indicating that Jolteon is accurate and is able to capture the performance variability for the stage.

Workflow-level effectiveness. Finally, we evaluate Jolteon's errors in predicting the end-to-end latency distributions for the entire workflow, and compare it with Orion's blackbox and Ditto's whitebox models. We vary the resource configurations with different number of function instances (i.e., **d**) and different number of vCPUs per function (i.e., **v**). The total number of vCPUs is the dot product of vector **d** and **v**. The detailed configurations are shown in Table 4. As shown in Table 5 for Video Analytics, Jolteon has an error range in [-7.59%, 9.29%] for P50 and P95 latency prediction, with a MAPE under 4%. The two baselines perform much worse. Orion experiences a MAPE exceeding 30% for both P50 and P95 latency predictions since its linear interpolation is inaccurate under large configuration space. Ditto has a MAPE of 33.98% for P50 and 28.00% for P95 latency predictions due to its inability to accommodate performance variability in its analytical model.

Video Analytics							
Resource Config	#vCPUs in Total	Jolteon		Orion		Ditto	
		P50	**P95**	**P50**	**P95**	**P50**	**P95**
Config 0	184	0.88%	-7.58%	20.00%	14.62%	51.75%	40.91%
Config 1	144	3.23%	-1.21%	34.38%	36.03%	51.71%	40.89%
Config 2	116	9.29%	5.26%	32.86%	30.20%	75.71%	61.84%
Config 3	76	4.21%	2.56%	52.63%	57.52%	10.88%	8.03%
Config 4	36	3.46%	-1.54%	43.53%	44.28%	9.39%	16.31%
Config 5	30	-2.19%	-1.38%	12.23%	20.12%	-4.46%	-0.02%
MAPE		**3.88%**	**3.26%**	**32.62%**	**33.80%**	**33.98%**	**28.00%**

Table 5: Errors for end-to-end latency prediction of different models.

6.4 Time analysis for Jolteon

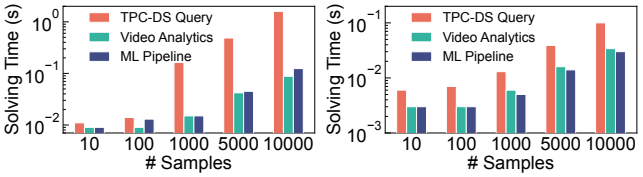
Performance model training time. We evaluate the offline time to train the performance model. For each workflow, Jolteon collects tens of execution profiles with different resource configurations. Then, it uses non-linear least squares to fit the distributions of the random variables (parameters) in the performance model. Table 6 reports the training time for the three workflows. The training time is less than 70 milliseconds, which is negligible compared to the end-to-end latency of tens of seconds.

Solving time. We measure the time of Jolteon's gradient descent algorithm in the convex optimizer. We vary the sample size from 10 to 10,000. Figure 13 shows the results. In the case of latency bound, the algorithm finishes under 0.5 seconds when sample size is less than 5,000. When a cost bound is specified, the algorithm finishes within 0.05 seconds. The higher solving time under a latency bound is due to the specifics of our implementation. With `SLSQP` as the algorithm, constraint functions are smooth and differentiable. The presence of the max operator in the latency constraint violates this condition. To address this, we replace the single latency constraint for the entire workflow with individual latency constraints for each DAG path. It eliminates the use of the max operator but increases the number of constraints.

More complex workflows result in longer solving time. Among the three workflows, TPC-DS Query has the most stages and complicated data dependencies (e.g., eight stages). Our experiment on TPC-DS Query involves 4048 sampling constraints. As Figure 13(a) shows, Jolteon can solve the problem within 0.5 seconds. According to the characterization on Azure Durable Functions [24], 95% of the serverless workflows have fewer than eight stages, which indicates that Jolteon provides sub-second solving time for most use cases.

	Training Time
TPC-DS Query	0.065 s
Video Analytics	0.016 s
ML Pipeline	0.014 s

Table 6: Training time for the performance model.



(a) Latency bound. (b) Cost bound.

Figure 13: Solving time of Jolteon.

7 Discussion

Memory in serverless computing. Besides vCPU, the memory size of a serverless function also impacts its performance. In AWS Lambda, the memory size of a function is proportional to the configured vCPU. Therefore, the Jolteon prototype, which is on top of AWS Lambda, leverages this property and uses the vCPU as the main influence factor. Recent work [48] decouples the memory size from the vCPU in the resource configuration of a serverless function. In such cases, Jolteon chooses the memory size greater than the peak memory usage for the function and adjusts the number of vCPUs to meet different application requirements.

Auto-scaling. Resource provisioning includes two parts: resource configuration and resource scaling. Resource scaling (e.g., auto-scaling) is used to scale the workflow instances to meet the realtime execution load. It is orthogonal to resource configuration and is not the focus of Jolteon's design.

Limitations of Jolteon's performance model. One limitation is that the IO model does not reflect intricate IO patterns. For example, the serverless function may issue a SQL query to an external database. The IO time is determined by the SQL query's logic rather than the returned data size. Another limitation is that the performance model does not consider pipelining in serverless workflows. The upload of the upstream function and the download of the downstream function can be pipelined to reduce latency. Jolteon's performance model can be extended to capture these complex scenarios.

8 Related Work

Characteristics of serverless computing. Different from other cloud computing paradigms, serverless computing exhibits unique performance characteristics. ServerlessWild [49] and Orion [24] analyze the performance of serverless functions in Azure Functions [2], and emphasize the performance variability and the impact of cold starts. FaaSCache [50] categorizes a serverless function's lifespan into initialization and execution phases. Caerus [8] and Ditto [9] view the function execution as fine-grained transmission and computation steps. Jolteon integrates the above insights and introduces a novel stochastic performance model to capture the characteristics of serverless computing.

Resource configuration for serverless computing. Resource configuration is critical to satisfy application-level requirements in cloud computing [20, 22, 51–53] and even more important in serverless computing due to its fine-grained resource allocation [7, 8]. Existing works either use whitebox models [7–9, 23, 26, 28] or blackbox models [24, 25, 27] to predict the performance of serverless functions and perform optimization. Jolteon leverages the advantages of both whitebox and blackbox models while mitigating their drawbacks in the performance model.

Serverless workflow orchestration. Many cloud providers identify the necessity for serverless workflow orchestration to facilitate the development of complex cloud applications. Centralized workflow orchestrators, such as AWS Step Functions [16], Google Workflows [19], and Azure Durable Functions and Logic Apps [17, 18], are adopted by major serverless platforms. Recent works [54–56] propose decentralized workflow orchestrators to improve scalability and reduce network latency. Jolteon can be integrated with them to provide automatic resource configuration.

Serverless workflow execution. Some works [39–41, 57] focus on the fault tolerance of serverless workflows to guarantee the exactly-once semantics. Jolteon can be integrated with them by capturing the overhead of logging in the performance model. Some works [9, 58, 59] exploit shared memory to reduce data transmission time by co-locating two adjacent stags in serverless workflows. Jolteon uses S3 as the external storage and its performance model is also applicable to data transmission through shared memory.

9 Conclusion

We present Jolteon, a serverless workflow orchestrator that facilitates automatic resource configuration to satisfy application-level requirements for serverless applications. Jolteon employs a novel stochastic performance model to capture both the execution characteristics and variability, a bound guaranteed sampler to transform the stochastic problem and an efficient convex optimizer to find Pareto-optimal configurations. We evaluate Jolteon with a variety of serverless workflows. The experimental results show that Jolteon outperforms the state-of-the-art solution by up to 2.3× on cost and 2.1× on latency.

Acknowledgments. We sincerely thank our shepherd Rodrigo Bruno and the anonymous reviewers for their valuable feedback on this paper. This work was supported by the National Key Research and Development Program of China under the grant number 2022YFB4500700. Xin Jin is the corresponding author. Zili Zhang, Chao Jin and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] "AWS Lambda." https://aws.amazon.com/lambda/.

[2] "Azure Functions." https://azure.microsoft.com/products/functions/.

[3] "Google Cloud Functions." https://cloud.google.com/functions/.

[4] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, 2019.

[5] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," *Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley*, 2019.

[6] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," *Communications of the ACM*, 2021.

[7] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *USENIX NSDI*, 2019.

[8] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: Nimble task scheduling for serverless analytics.," in *USENIX NSDI*, 2021.

[9] C. Jin, Z. Zhang, X. Xiang, S. Zou, G. Huang, X. Liu, and X. Jin, "Ditto: Efficient serverless analytics with elastic parallelism," in *ACM SIGCOMM*, 2023.

[10] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads," in *USENIX NSDI*, 2017.

[11] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li, "Lavea: Latency-aware video analytics on edge computing platform," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, 2017.

[12] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *ACM Symposium on Cloud Computing*, 2018.

[13] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *ACM Symposium on Cloud Computing*, 2019.

[14] H. Wang, D. Niu, and B. Li, "Distributed machine learning with a serverless architecture," in *IEEE INFOCOM*, 2019.

[15] C. Zhang, M. Yu, W. Wang, and F. Yan, "Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving," in *USENIX ATC*, 2019.

[16] "AWS Step Functions." https://aws.amazon.com/step-functions/.

[17] "Azure Logic Apps." https://azure.microsoft.com/products/logic-apps/.

[18] "Azure Durable Functions." https://learn.microsoft.com/azure/azure-functions/durable/.

[19] "Google Workflows." https://cloud.google.com/workflows.

[20] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *USENIX NSDI*, 2017.

[21] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, *et al.*, "Autopilot: workload autoscaling at google," in *EuroSys*, 2020.

[22] R. Bhardwaj, K. Kandasamy, A. Biswal, W. Guo, B. Hindman, J. Gonzalez, M. Jordan, and I. Stoica, "Cilantro: Performance-aware resource allocation for general objectives via online feedback," in *USENIX OSDI*, 2023.

[23] "AWS Lambda Power Tuning." https://github.com/alexcasalboni/aws-lambda-power-tuning.

[24] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "Orion and the three rights: Sizing, bundling, and prewarming for serverless dags," in *USENIX OSDI*, 2022.

[25] Z. Zhou, Y. Zhang, and C. Delimitrou, "Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows," in *ACM ASPLOS*, 2023.

[26] Z. Wen, Y. Wang, and F. Liu, "Stepconf: Slo-aware dynamic resource configuration for serverless function workflows," in *IEEE INFOCOM*, 2022.

[27] S. Eismann, J. Grohmann, E. Van Eyk, N. Herbst, and S. Kounev, "Predicting the costs of serverless workflows," in *Proceedings of the ACM/SPEC international conference on performance engineering*, 2020.

[28] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "Cose: Configuring serverless functions using statistical learning," in *IEEE INFOCOM*, 2020.

[29] "AWS Serverless Vector Engine." https://aws.amazon.com/opensearch-service/serverless-vector-engine/.

[30] "Amazon simple storage service (S3)." https://aws.amazon.com/s3/.

[31] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[32] M. A. Lejeune and A. Ruszczyński, "An efficient trajectory method for probabilistic production-inventory-distribution problems," *Operations Research*, 2007.

[33] M. C. Campi and S. Garatti, "A sampling-and-discarding approach to chance-constrained optimization: feasibility and optimality," *Journal of optimization theory and applications*, 2011.

[34] A. K. Takyi and B. J. Lence, "Surface water quality management using a multiple-realization chance constraint method," *Water Resources Research*, 1999.

[35] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *The collected works of Wassily Hoeffding*, 1994.

[36] J. Luedtke and S. Ahmed, "A sample approximation approach for optimization with probabilistic constraints," *SIAM Journal on Optimization*, 2008.

[37] M. C. Campi and S. Garatti, "The exact feasibility of randomized solutions of uncertain convex programs," *SIAM Journal on Optimization*, 2008.

[38] S. P. Boyd and L. Vandenberghe, *Convex optimization*. 2004.

[39] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Fault-tolerant and transactional stateful serverless workflows," in *USENIX OSDI*, 2020.

[40] Z. Jia and E. Witchel, "Boki: Stateful serverless computing with shared logs," in *ACM SOSP*, 2021.

[41] S. Zhuang, S. Wang, E. Liang, Y. Cheng, and I. Stoica, "Exoflow: A universal workflow system for exactly-once dags," in *USENIX OSDI*, 2023.

[42] "Scipy." https://scipy.org/.

[43] "LightGBM Python Library." https://lightgbm.readthedocs.io/en/latest/Python-Intro.html.

[44] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, 2012.

[45] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *USENIX OSDI*, 2018.

[46] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[47] "TPC-DS." https://www.tpc.org/tpcds/.

[48] M. Bilal, M. Canini, R. Fonseca, and R. Rodrigues, "With great freedom comes great opportunity: Rethinking resource allocation for serverless functions," in *EuroSys*, 2023.

[49] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *USENIX ATC*, 2020.

[50] A. Fuerst and P. Sharma, "Faascache: keeping serverless computing alive with greedy-dual caching," in *ACM ASPLOS*, 2021.

[51] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *ACM ASPLOS*, 2019.

[52] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "Firm: An intelligent fine-grained resource management framework for {SLO-Oriented} microservices," in *USENIX OSDI*, 2020.

[53] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: Ml-based and qos-aware resource management for cloud microservices," in *ACM ASPLOS*, 2021.

[54] D. H. Liu, A. Levy, S. Noghabi, and S. Burckhardt, "Doing more with less: Orchestrating serverless applications without an orchestrator," in *USENIX NSDI*, 2023.

[55] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in *USENIX NSDI*, 2023.

[56] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, "Faasflow: Enable efficient workflow execution for function-as-a-service," in *ACM ASPLOS*, 2022.

[57] S. Qi, X. Liu, and X. Jin, "Halfmoon: Log-optimal fault-tolerant stateful serverless computing," in *ACM SOSP*, 2023.

[58] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *USENIX ATC*, 2020.

[59] S. Qi, L. Monis, Z. Zeng, I.-c. Wang, and K. Ramakrishnan, "Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing," in *ACM SIGCOMM*, 2022.

A Appendix

A.1 Lower bound of sample size

Theorem A.1 *One of the lower bounds of the sample size, n, to guarantee the performance bound under confidence level* δ, $(0<\delta<1)$ *is:*

$$\frac{1}{2\times(1-percentile)^2}log(\frac{|D|}{1-\delta})$$

Proof. Let Pr be the probability function, i.e., the percentile function of a random variable, D be the definition domain and FD be the feasible domain of $(\mathbf{d},\mathbf{v})$. FD (i.e., the ideal feasible domain) is constrained by $Confidence\big(Pr\{G(\mathbf{d},\mathbf{v},\mathbf{x_i}) \leq 0\} \geq percentile\big) \geq \delta$ with percentile as Pr function, while FD' (i.e., the sampling feasible domain) is constrained by n samples $\{G(\mathbf{d},\mathbf{v},\mathbf{x_i}) \leq 0,\ for\ i=1...n\}$.

For simplicity, *percentile* is replaced by p. With the percentile of performance metric (i.e., p), we have: $Pr\{G(\mathbf{d},\mathbf{v},\mathbf{x_i}) \leq 0\} \geq p$ if $(\mathbf{d},\mathbf{v}) \in FD$ and $Pr\{G(\mathbf{d},\mathbf{v},\mathbf{x_i}) \leq 0\} < p$ if $(\mathbf{d},\mathbf{v}) \notin FD$. We define the random variable Z_i by $Z_i = 1$ if $G(\mathbf{d},\mathbf{v},\mathbf{x_i}) \leq 0$ and $Z_i = 0$ otherwise. Now, we consider the probability of $\exists(\mathbf{d},\mathbf{v}) \in FD'$ but $(\mathbf{d},\mathbf{v}) \notin FD$ (denoted by *pro*).

$$pro = Pr\{\sum_{i=1}^{n} Z_i \geq n \text{ and } E(Z_i) < p\}$$
$$\leq Pr\{\big(\sum_{i=1}^{n} Z_i - E(\sum_{i=1}^{n} Z_i)\big) \geq n - np\}$$

According to Hoeffding's inequality ($0 \leq Z_i \leq 1$), we have:

$$pro \leq e^{-2n(1-p)^2}$$

Since there may be $|D|$ points in this domain, one of the upper bounds of $1-\delta$ (i.e., the probability of violating the confidence level) is:

$$1-\delta < 1-(1-pro)^{|D|} < |D| \times e^{-2n(1-p)^2}$$

Therefore, one of the lower bounds of sample size n is $\frac{1}{2\times(1-p)^2}log(\frac{|D|}{1-\delta})$.

A.2 Convexity Analysis

Lemma A.1 *Given two convex functions $f(x)$ and $g(x)$, $h(x) = f(x)+g(x)$ is also a convex function, $x \in \mathcal{D}$.*

Proof. Functions $f(x)$ and $g(x)$ are convex, which satisfy the following inequality:

$$f(\lambda x_1+(1-\lambda)x_2) \leq \lambda f(x_1)+(1-\lambda)f(x_2)$$
$$g(\lambda x_1+(1-\lambda)x_2) \leq \lambda g(x_1)+(1-\lambda)g(x_2)$$

Here, λ is an arbitrary value in $[0,1]$ and $x_1, x_2 \in \mathcal{D}$. We have:

$$h(\lambda x_1+(1-\lambda)x_2) = f(\lambda x_1+(1-\lambda)x_2)+g(\lambda x_1+(1-\lambda)x_2)$$
$$\leq \lambda f(x_1)+(1-\lambda)f(x_2)+\lambda g(x_1)+(1-\lambda)g(x_2)$$
$$= \lambda h(x_1)+(1-\lambda)h(x_2)$$

Therefore, $h(x)$ is a convex function.

Lemma A.2 *Given two convex functions $f(x)$ and $g(x)$, $h(x) = \max(f(x),g(x))$ is also a convex function, $x \in \mathcal{D}$.*

Proof. First, we prove a basic inequality for max operation:

$$\begin{aligned}&\max(\alpha_1,\alpha_2)+\max(\beta_1,\beta_2)\\&=\max(\alpha_1+\max(\beta_1,\beta_2),\alpha_2+\max(\beta_1,\beta_2))\\&\geq \max(\alpha_1+\beta_1,\alpha_2+\beta_2)\end{aligned} \tag{8}$$

We can set $\alpha_1 = \lambda f(x_1)$, $\alpha_2 = \lambda g(x_1)$, $\beta_1 = (1-\lambda)f(x_2)$ and $\beta_2 = (1-\lambda)g(x_2)$, where $x_1, x_2 \in \mathcal{D}$. Based on Formula 8 and Lemma A.1, we have:

$$h(\lambda x_1+(1-\lambda)x_2) = \max(f(\lambda x_1+(1-\lambda)x_2), g(\lambda x_1+(1-\lambda)x_2))$$
$$\leq \max(\lambda f(x_1)+(1-\lambda)f(x_2), \lambda g(x_1)+(1-\lambda)g(x_2))$$
$$\leq \max(\lambda f(x_1),\lambda g(x_1))+\max((1-\lambda)f(x_2),(1-\lambda)g(x_2))$$
$$= \lambda h(x_1)+(1-\lambda)h(x_2)$$

Therefore, $h(x)$ is a convex function.

Lemma A.3 *Given two convex functions $f(x)$ and $g(x)$ ($f(x)$ is a monotonically increasing function), $h(x) = f(g(x))$ is also a convex function, $x, g(x) \in \mathcal{D}$.*

Proof. $\forall x_1, x_2 \in \mathcal{D}$ and $\lambda \in [0,1]$, we have $f(g(\lambda x_1+(1-\lambda)x_2)) \leq f(\lambda g(x_1)+(1-\lambda)g(x_2))$ since $f(x)$ is monotonically increasing and $g(x)$ is convex. Based on this, we derive:

$$h(\lambda x_1+(1-\lambda)x_2) = f(g(\lambda x_1+(1-\lambda)x_2))$$
$$\leq f(\lambda g(x_1)+(1-\lambda)g(x_2))$$
$$\leq \lambda f(g(x_1))+(1-\lambda)f(g(x_2)) = \lambda h(x_1)+(1-\lambda)h(x_2)$$

Therefore, $h(x)$ is a convex function.

Lemma A.4 $(\frac{s}{xy})^\alpha \times \ln\frac{s}{xy}$ *is a bivariate convex function, where s is a positive constant, $\frac{s}{xy} \in (1,\infty)$, $x,y \in (0,\infty)$ and α is a nonnegative integer.*

Proof. We divide the proof into two cases.

Case one: $\alpha = 0$**.** Since $\ln\frac{s}{xy}$ has second derivatives when $\frac{s}{xy} \in (1,\infty)$. We calculate the Hessian matrix of the bivariate function:

$$Hessian(\ln\frac{s}{xy}) = \begin{bmatrix} \frac{1}{x^2} & 0 \\ 0 & \frac{1}{y^2} \end{bmatrix}$$

Evidently, such matrix is positive semi-definite and $\ln\frac{s}{xy}$ is a convex function.

Case two: $\alpha \geq 1$**.** We define $f(z) = z^\alpha \times \ln z$, and $g(x,y) = \frac{s}{xy}$. We first calculate the derivative of $f(z)$ $(z>1)$: $f'(z) = (1+\alpha\ln z)\times z^{\alpha-1} > 0$. Therefore, $f(z)$ is a monotonically increasing function. Then, we prove that $f(z)$ is a convex

Initial Config	#Funcs (**d**)	#vCPUs/Func (**v**)	#vCPUs in Total
Large	(16, 16, 16, 16)	(4, 4, 4, 4)	256
Medium	(8, 8, 8, 8)	(2, 2, 2, 2)	64
Small	(4, 4, 4, 4)	(1, 1, 1, 1)	16
Mix1	(4, 8, 16, 4)	(1, 2, 4, 1)	88
Mix2	(16, 4, 8, 16)	(4, 1, 2, 4)	148

Table 7: Detailed initial configurations.

function when $z > 1$. When $\alpha = 1$, we calculate the second derivative $f(z) = z^{\alpha} \times \ln z$ as follows:

$$f''(z) = (\ln z + 1)' = \frac{1}{z} > 0$$

When $\alpha \geq 2$, we calculate the second derivative as follows:

$$\begin{aligned} f''(z) &= ((1 + \alpha \ln z) \times z^{\alpha-1})' \\ &= \alpha(\alpha-1)z^{\alpha-2} \ln z + (2\alpha-1)z^{\alpha-2} > 0 \quad (z > 1, \alpha \geq 2) \end{aligned}$$

Therefore, $f(z)$ is a convex function when $z > 1$. Last, we need to prove that $g(x,y)$ is a convex function. We calculate the Hessian matrix of $g(x,y)$ as follows (s,x,y are positive and s is constant):

$$Hessian(g(x,y) = \frac{s}{xy}) = \begin{bmatrix} \frac{2s}{x^3y} & \frac{s}{x^2y^2} \\ \frac{s}{x^2y^2} & \frac{2s}{xy^3} \end{bmatrix}$$

Since s,x,y are all positive, this matrix is positive semi-definite. Therefore, $g(x,y)$ is a convex function. In summary, $f(z)$ is a monotonically increasing convex function, and $g(x,y)$ is a convex function. Based on Lemma A.3, we derive that $h(x,y) = f(g(x,y))$ is a convex function.

Lemma A.5 *$f(x,y) = \frac{s}{x \times min(y,w)}$ is a bivariate convex function, where s,w are positive constants and x,y are positive independent variables.*

Proof. We are able to convert to $f(x,y)$ to $max(\frac{s}{x \times w}, \frac{s}{x \times y})$. The Hessian matrix of $\frac{s}{x \times w}$ is:

$$\begin{bmatrix} \frac{2s}{wx^3} & 0 \\ 0 & 0 \end{bmatrix}$$

It is a positive semi-definite matrix, and $\frac{s}{x \times w}$ is a convex function. As for $\frac{s}{x \times y}$, its convexity is proved in Lemma A.4. According to Lemma A.2, $f(x,y) = \frac{s}{x \times min(y,w)} = max(\frac{s}{x \times w}, \frac{s}{x \times y})$ is a convex function.

A.3 Sensitivity of Jolteon

To evaluate the sensitivity of Jolteon's convex optimizer algorithm (§4.3) to various initial configurations (i.e., entry points), we use five different initial configurations for Video Analytics: large, medium, small, mix1, and mix2, with total number of vCPUs ranging from 16 to 256. Table 7 shows them in detail, where **d** and **v** represent the vectors (four stages) of the number of function instances and the number of vCPUs per function instance, respectively. We run Jolteon with three different latency bounds: 12 seconds for tight bound, 36 seconds for moderate bound, and 80 seconds for loose bound.

Initial Config	Output Config		
	Tight Bound (12 s)	Moderate Bound (36 s)	Loose Bound (80 s)
Large	Config A	Config B	Config C
Medium	Config A	Config B	Config C
Small	Config A	Config B	Config D
Mix1	Config A	Config B	Config C
Mix2	Config A	Config B	Config C

Table 8: Output configurations of Jolteon for Video Analytics under different initial configurations and bounds.

Output Config	#Funcs (**d**)	#vCPUs/Func (**v**)	#vCPUs in Total
Config A	(32, 16, 16, 16)	(5, 1.5, 1.5, 5)	288
Config B	(8, 8, 8, 8)	(1.5, 1.5, 1.5, 1.5)	48
Config C	(4, 4, 4, 4)	(1, 1, 1.5, 1.5)	20
Config D	(4, 4, 4, 4)	(1, 1.5, 1.5, 1)	20

Table 9: Detailed output configurations of Jolteon.

As Table 8 indicates, Jolteon produces identical configurations across all initial configurations under the tight and moderate bounds. Under the loose bound, Jolteon generates a slightly different configuration for the small initial configuration. Table 9 further illustrates these output configurations. Notably, Config C and Config D under the loose bound share the same number of function instances and total vCPUs. The slight discrepancy between them lies in the specific allocation of vCPUs, i.e., (1, 1, 1.5, 1.5) vs. (1, 1.5, 1.5, 1). We run Video Analytics under these two configurations and obtain similar performance. Config C and Config D yield end-to-end latency of 75.9 and 74.5 seconds and cost of \$0.00916 and \$0.00874, respectively. The difference between the two configurations in latency and cost are negligible, which are 1.9% and 4.8%, respectively. In summary, Jolteon's convex optimizer algorithm is insensitive to initial configurations.

Can't Be Late: Optimizing Spot Instance Savings under Deadlines

Zhanghao Wu, Wei-Lin Chiang, Ziming Mao,
Zongheng Yang, Eric Friedman†, Scott Shenker†, Ion Stoica
University of California, Berkeley †*UC Berkeley and ICSI*

Abstract

Cloud providers offer spot instances alongside on-demand instances to optimize resource utilization. While economically appealing, spot instances' preemptible nature causes them ill-suited for deadline-sensitive jobs. To allow jobs to meet deadlines while leveraging spot instances, we propose a simple idea: use on-demand instances judiciously as a backup resource. However, due to the unpredictable spot instance availability, determining when to switch between spot and on-demand to minimize cost requires careful policy design. In this paper, we first provide an in-depth characterization of spot instances (e.g., availability, pricing, duration), and develop a basic theoretical model to examine the worst and average-case behaviors of baseline policies (e.g., greedy). The model serves as a foundation to motivate our design of a simple and effective policy, Uniform Progress, which is parameter-free and requires no assumptions on spot availability. Our empirical study, based on three-month-long real spot availability traces on AWS, demonstrates that it can (1) outperform the greedy policy by closing the gap to the optimal policy by 2× in both average and bad cases, and (2) further reduce the gap when limited future knowledge is given. These results hold in a variety of conditions ranging from loose to tight deadlines, low to high spot availability, and on single or multiple instances. By implementing this policy on top of SkyPilot, an intercloud broker system, we achieve 27%-84% cost savings across a variety of representative real-world workloads and deadlines. The spot availability traces are open-sourced for future research.[1]

1 Introduction

As organizations continue to migrate their workloads to clouds, the need to minimize operational costs has become a critical concern [41]. One of the top contributors to cloud spending is the cost of compute instances [8], which are typically offered in two pricing models: *on-demand* and *spot*.[2]

On-demand instances are available but come at a premium cost. In contrast, *spot* instances are typically 3–10× cheaper (Table 1), but are less available and they can be preempted unexpectedly. As a result, more applications such as analytics [13], AI [28, 37, 40], HPC [29], and CI/CD workloads [1], are leveraging spot instances to reduce costs. To handle preemptions, these jobs either checkpoint periodically and recover from the last checkpoint on restart [25, 46], or re-execute the entire job.

	V100 GPU	64-core CPU
AWS	3×	2–6×
Azure	3–6×	3–10×
GCP	3×	4–11×

Table 1: Cost savings of spot vs. on-demand instances.

However, while many applications can tolerate uncertainties introduced by spot instance preemptions, others cannot. One such category is delay-sensitive applications where a job needs to finish by a certain deadline [24]. Examples include processing new user data to keep an AI model up-to-date in a recommendation system, or analyzing the latest information to make timely decisions in a trading application. Therefore, most of deadline-sensitive applications eschew spot instances in favor of on-demand instances, thus trading off cost for predictability.

In this paper, we resolve this tradeoff by enabling an application to leverage spot instances while still meeting its deadline. For simplicity, we focus on recoverable jobs running on a single instance, and assume the running time of the job is known, as well as its deadline. A job can be in one of three states: (1) running on a spot instance, (2) running on an on-demand instance, or (3) idle, i.e., waiting for a spot instance to become available. We design scheduling policies that periodically decide whether a job should remain in the same state or switch to another state. When a job switches to a non-idle state we assume there is a delay, e.g., the overhead of provisioning/setting up a new instance, and re-starting from a previous checkpoint. Due to the high unpredictability in spot instance availability (§2.2), the key challenge lies in striking a balance between cost optimization and deadline adherence to effectively leverage the low cost of spot instances without missing the deadline.

A simple solution to this problem would be for a job to

[1] See spot traces: https://github.com/skypilot-org/spot-traces

[2] In this paper, we do not consider "reserved" instances, whose economics involves volume contracts and is more complex.

use a spot instance up to the point at which the remaining computation time equals the remaining time to deadline, and then switch to an on-demand instance until it finishes. While this "greedy" policy (§3.4) guarantees that the job will meet its deadline, we show that it is sub-optimal. We do so by developing a theoretical framework to study the worst-case behavior (e.g., competitive ratio) of the policy (§4).

To address the limitations of "greedy" policy, we propose a simple and effective policy, called *Uniform Progress*, which aims to make uniform progress towards deadline, by distributing the job computation uniformly across the time. Uniform Progress requires no assumption about spot instances' availability and is parameter-free. Using simulations on real-world traces we show that Uniform Progress outperforms greedy policy and approaches an optimal policy with limited knowledge of the future (knowing how long the next spot instance is going to last) in a variety of scenarios—from loose to tight deadlines, and from low to high spot availability. We build a prototype of Uniform Progress and evaluate it in a cloud setting on three real-world workloads: ML training, scientific batch jobs, and data analytics. Results show that Uniform Progress achieves 27–84% cost savings while meeting deadlines.

This paper is organized as follows. First, we provide an in-depth characterization of spot instances across various cloud regions, examining their availability patterns, pricing, and lifetime to inform our policy design (§2). Next, we develop a theoretical model that captures the essential dynamics of spot instances, which enables us to examine the worst-case behavior of a given policy (§3, §4). We then present our policies for jobs with both single and multiple instances (§5) and conduct a comprehensive empirical study on months-long real-world traces of spot instances (§6). We build a prototype implementation that supports the proposed policies in a cloud setting, and evaluate these policies on three real-world workloads (§7). Finally, we review related work in §8.

In summary, this paper makes the following contributions:

1. We introduce a problem of using spot instances to minimize the cost of running a job with deadline adherence.
2. We develop a theoretical framework to study the worst and average-case behavior of baseline policies, providing insights on the tradeoff between cost and deadline.
3. We propose Uniform Progress, a simple but effective policy which is parameter-free and requires no assumptions on spot availability. Empirically, we show the significant improvement of the policy in a wide variety of scenarios.
4. We implement a prototype system with Uniform Progress, and evaluate it on real-world workloads.

Finally, we open source our three-month traces of spot instance availability to encourage future research in this area.

2 Characterization of Spot Instances

In this section, we characterize spot instance availability and pricing over time and across availability zones. We observe high volatility in availability but a smooth pricing pattern. We use these insights to drive the design of our scheduling policy.

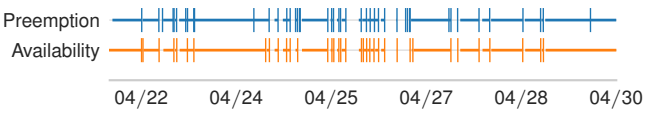


Figure 1: Real spot preemptions and availability are highly correlated. Trace is in AWS us-west-2b. **Upper**: preemptions. Horizontal lines represent a running spot instance. Vertical bars are preemptions. **Lower**: availability. Horizontal lines are spot instance available periods. Vertical bars are changes from availability to unavailability. Grey gaps are unavailability periods. Note that although some vertical bars look immediately followed by a horizontal line, there are still gaps in between.

2.1 Methodology of Spot Trace Collection

We collect *spot availability traces* from public clouds. A trace is a time series showing whether a particular spot instance type is available at a given time in a zone. We collect these traces over a three-month period and in nine AWS availability zones (three in *us-west-2*, four in *us-east-1*, two in *us-east-2*).

A key challenge of trace collection is that it can be prohibitively expensive. For example, a spot V100 instance costs about $1/hour. If we collect a real *preemption* trace where an instance is kept live as much as possible modulo preemptions, collecting three-month long traces in all nine zones could cost over $10,000. Instead, we propose an approximation: we collect *availability* traces, where we try to launch a spot instance every 10 minutes to probe if it is available and then immediately terminate it. To validate this approach, Figure 1 shows a high correlation between the real preemption and availability signals over a week-long period. This approach reduces the cost of trace collection by about 100×. For completeness, we also include real preemption traces in our evaluation of policy performance on multi-instance jobs (§6.6) and real-world workloads (§7.2).

In this work, we focus on a few *scarce* instance types, i.e., Nvidia V100 and K80 GPU instances, which are now in high demand [4] due to the rise of Generative AI and large language models (LLMs). Focusing on these scarce instance types is thus both useful and interesting, as they are frequently preempted, providing a good testing ground for scheduling policies.

2.2 High Variance in Spot Availability

Our analysis reveals that spot availability varies significantly *across zones* and *over time*. Figure 2 (left) shows the availability traces of 9 AWS zones over 2 weeks (4 example zones are in Figure 2 and the rest 5 zones are in §A.1). We observe a large difference across zones (e.g., us-west-2a vs us-east-1a). The periods of unavailability can last for hours or even days.

To understand spot availability distributions, we overlay 6-hour windows on a 2-week period (thus, $14 \times 24/6 = 56$ windows per zone) and count the fraction of availability probes that succeeded in each window. Figure 2 (right) plots the distributions of *spot availability fractions* in the 56 windows per zone, which approximate the fraction of time spot instances are available in each zone. We observe that each

Figure 2: Spot Availability is highly unpredictable and volatile. Traces are across four of nine AWS zones collected. **Left**: Availability. Horizontal lines are available periods. Vertical bars are changes from available to unavailable, followed by grey gaps indicating unavailable period. **Right**: Boxplots of spot availability fraction, *i.e.,* percentage of the time an instance is available in 6-hour windows.

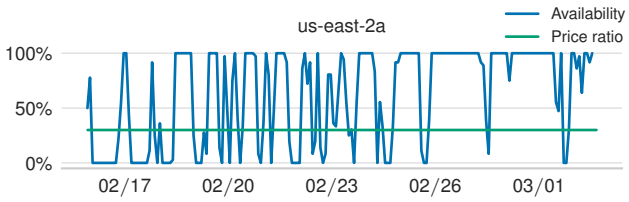


Figure 3: High volatility of spot availability fraction. Availability can jump from 100% to 0% within hours. Price ratio: spot price divided by on-demand price.

zone can go from being highly available to mostly unavailable across time (e.g., in us-east-2b, the difference between p25 and p75 is about 70%) and there is little correlation across zones. In addition, Figure 3 shows changes in spot availability fractions over time. We observe a highly volatile pattern: availability can change from 100% to 0% within hours.

The results above suggest that scheduling policies should be robust to highly unpredictable availability patterns. For generality, in this paper, we make no assumptions on spot availability patterns. We discuss existing prediction-based approaches in §8 and leave this direction to future work.

2.3 Relative Stability in Spot Pricing

In contrast, we observe that spot pricing is much more stable than availability. Figure 3 shows the price ratio of spot to on-demand for AWS stays almost constant despite significant changes in availability. In the three-month-long trace, we observe only a 5% price variation on average over any one-week period, validating the recently introduced smooth pricing model on AWS [5]. GCP's spot instance prices are even more stable as it is guaranteed to only change once every 30 days [3].

2.4 Correlation of Multi-Instance Availability

To understand the behavior of multiple spot instances, we analyze 2-week *preemption* traces and 2-week *availability* traces for clusters of 4 and 16 instances, respectively (see §6.1 for details). Notably, over 94% of the time, either all or none of the instances are available in each cluster. This suggests availability tends to change *simultaneously* for multiple instances (bulk preemption is also observed in [16]), up to a count of 16.

3 Using Spot for Deadline-Sensitive Jobs

In this section, we present a simple model to formulate the problem, discuss when a policy matters, and then give three rules for policy design followed by a basic greedy policy.

3.1 Problem Setup

We consider two types of instances with the same hardware: an on-demand instance, which is always available,[3] and a spot instance, whose availability is unpredictable. We assume that spot availability is non-adversarial, meaning that it is independent of the job's choices and observable factors, except for §4.1, where we adopt competitive analysis for the worst case study.

We focus on long-running (hours to days) jobs where preemptions are likely. We firstly assume each job uses one instance. We will extend it to multiple instances in §5.5 and evaluate it in §6.6.

For a deadline-sensitive job, we denote *remaining computation time* at time t as $C(t)$ and *remaining time-to-deadline* as $R(t)$. This implies that the job's total computation time is $C(0)$, and deadline is $R(0)$. Based on the definition, we can derive that $R(t) = R(0) - t$ and when a job is progressing, $\partial C(t)/\partial t = -1$.

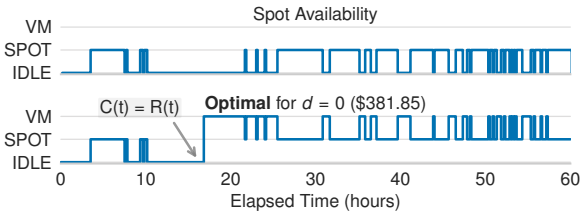
We assume that both $C(0)$ and $R(0)$ are given and the job is fault-tolerant to interruptions. For example, ML training typically has a consistent per-epoch time, indicating a predictable total runtime, and the model weights can be checkpointed and resumed for fault tolerance. Additionally, computation times for many recurring jobs (e.g., data analytics, scientific HPC, CI/CD) can be derived from past executions.

To account for overheads of starting the job on a new instance, we introduce *changeover delay*, d, which includes the time required to launch an instance, set up dependencies, and recover any potential progress loss caused by gaps between checkpoints or restarting the most recent unsaved execution. Whenever a job switches to a new spot or on-demand instance, a changeover delay occurs, meaning that $C(t)$ does not decrease for a duration of d while $R(t)$ continues to decrement. A delay d is charged at the new instance type's price. Switching from an instance to idle (i.e., termination) does not incur a delay. We will extend the model to consider variety with $C(0)$ and d in §5.6, and evaluate it in §6.7.

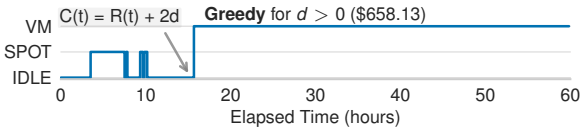
The goal is to minimize the cost for completing job's computation time $C(0)$ before deadline $R(0)$, *i.e.,* $C(R(0)) \leq 0$, using spot and on-demand instances. For simplicity, we define the price for an on-demand instance to be $k > 1$, and a spot instance to be 1. We assume that cloud providers charge every second when an instance is alive.[4] Based on the observation

[3] This is a simplifying assumption. In practice, some on-demand instance types can hit unavailability.

[4] Cloud providers have different billing practices, *e.g.,* AWS does not



(a) Without changeover delay.



(b) With changeover delay.

Figure 4: Example decision traces of policies on real spot availability on AWS.

in §2.3, we assume both the on-demand and spot price are fixed throughout the time before deadline $R(0)$.

3.2 Scheduling Policy

At any time t, a job can be in one of the following three states: idle, running on a spot instance, or running on an on-demand instance. While we assume that on-demand instances are always available, spot instances can be in one of two states: available or unavailable. The job's state space is the combination of any of the instance state and the spot state, except for an impossible case where the instance state is spot with spot state unavailable (Table 2). A scheduling policy is invoked to decide how a job moves across instance states.

Spot State \ Instance State	Idle	Spot	On-Demand
Spot Available	①	③	④
Spot Unavailable	②	-	⑤

Table 2: State space for a job.

In the ideal case where changeover delay $d = 0$, the problem is simple. An optimal policy is to use a spot instance whenever it is available, *i.e.,* transition between state ② and ③, until $C(t) = R(t)$. After that, the job cannot stay idle, as it needs to utilize all the remaining time before deadline to make progress. Since there is no changeover delay, the policy can use spot whenever it is available and switch to on-demand when it is not, *i.e.,* transition between ③ and ⑤. This policy is optimal because it utilizes all available spot instance lifetimes before the deadline, without additional cost. Figure 4a shows an example decision trace of how this policy performs for a job with $C(0) = 48$ hours and $R(0) = 60$ hours on a real spot availability trace, where the policy utilizes every spot lifetime, and runs the remaining computation with on-demand instances.

However, when changeover delay $d > 0$, which is the practical case, the problem becomes non-trivial. The policy now has to decide whether it is worth switching to a different instance at the expense of losing time d without making progress, which increases the risk of missing the deadline. For example, applying the optimal policy above for $d > 0$ would result in missing the deadline, since every switch costs an additional time d.

In the remainder of this paper, we focus on designing policies for the more practical $d > 0$ scenario.

3.3 Rules for Policy Design

Based on the problem setting, we propose three basic rules that all policies without future knowledge should follow to avoid unnecessary cost or missing the deadline.

Thrifty Rule. The job should remain idle after $C(t) = 0$.

Safety Net Rule. When a job is idle and $R(t) < C(t) + 2d$, switch to on-demand and stay on it until the end.

The policy is required to guarantee the job finished by the deadline. After $R(t) < C(t) + 2d$ becomes true, it is no longer safe to move from idle to spot. Otherwise, when the changeover delay of the spot finishes, the remaining time will become $R(t) < C(t) + d$, which means any preemption to the spot instance will result in missing deadline. Note that one could wait until $R(t) = C(t) + d$ then move to on-demand, but there is no gain for waiting an additional d if the job is idle.

Exploitation Rule. Once start using a spot instance, stay on it until it is preempted.

If the job is on a spot instance, any progress made will always cost the minimum price any policy could get, *i.e.,* the spot price. Voluntarily switching from spot to idle or on-demand will have no benefit, but less progress or more cost.

This rule will not violate the deadline because the Safety Net Rule guarantees that $R(t) \geq C(t) + 2d$ holds at the time t when the job is moved to the current spot instance. After the changeover delay is incurred and the job starts progressing, $R(t) - C(t)$ will not change, *i.e.,* $R(t) \geq C(t) + d$ holds, meaning the remaining time is enough for at least one changeover even if the current spot is preempted. The job will be able to switch to on-demand when Safety Net Rule kicks in.

3.4 Greedy Policy

Based on the three rules, we propose a straightforward greedy policy. The greedy policy behaves as follows:

1. Stay on any available spot instance until it is preempted (Exploitation Rule), and keep waiting if no spot instance is available, *i.e.,* transition between ② and ③ in Table 2.
2. (Safety Net Rule) When $R(t) < C(t) + 2d$ holds and the job is idle, move to on-demand and stay there until the end.

In Figure 4b, we show the decision trace of the greedy policy on the same spot availability trace as before (Figure 4a). The greedy policy acts much more conservatively than the previous optimal policy without changeover delay. That is because greedy can no longer afford frequent switches between on-demand and spot instances as before without missing the deadline. Thus, we now turn our attention to: can we do better than greedy while not assuming future knowledge?

4 Theoretical Analysis

In this section, we delve into theoretical aspects of the problem and prove the existence of a policy that is better than greedy in both worst and average cases.

charge for spot instances preempted within the first hour, while GCP does.

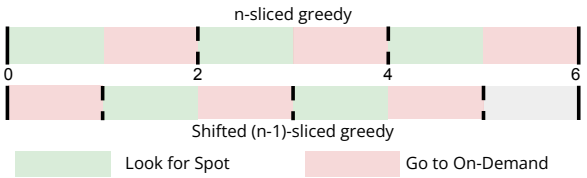


Figure 5: Example slicing for randomized shifted greedy (RSF) policy, where deadline $R(0)=6$, computation time $C(0)=3$, and slices $n=3$. Dashed lines indicate boundaries of slices.

4.1 Worst Case with Competitive Analysis

We first look into the worst case by investigating the competitive ratio c of a policy without knowledge of future spot availability, which is the ratio of the cost of the policy to the best omniscient policy with full knowledge of future spot availability. By "worst" case, we assume that spot instances are chosen by an oblivious adversary, who can base their decisions on complete knowledge of the job's policy but not on random coin flips used by the policy. Our goal is to prove that there is a policy with lower competitive ratio c than greedy, *i.e.,* performs better in the worst case.

To simplify the presentation, we assume changeover delay d is small and ignore the term $O(d)$. Also, we use $R(t)=C(t)+d$ as Safety Net Rule's condition, instead of $R(t)=C(t)+2d$, which will not affect the conclusion, due to negligible $O(d)$.

A natural bound for c is $1\le c\le k$, where k can be reached when the oblivious adversary choose a case that a given policy have to use all on-demand, and the omniscient policy could use all spot instances. We can prove that for any $R(0), C(0)$, a deterministic policy cannot perform better than greedy (see §A.2.1).

Theorem 1. *For any deterministic policy P, $c\ge k-O(d)$.*

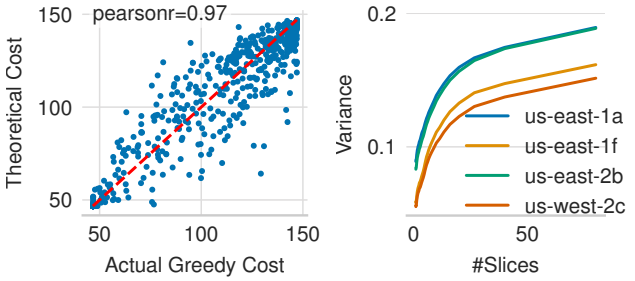
With Theorem 1, we can conclude that a policy has to be randomized to beat greedy, whose competitive ratio $c=k$, as an adversary can simply make spot available from t', where $R(t')=C(t')+d$. We now construct a better policy on top of greedy. We first extend greedy to an *n-sliced greedy* policy, in which we divide the time into n even slices with length $\frac{R(0)}{n}$ and apply greedy in each of these slices with $\frac{C(0)}{n}$ progress to make. The upper figure in Figure 5 is an example of n-sliced greedy, with a deadline $R(0)=6$ and 3 slices. In each slice, the policy enforces the job to make $\ge\frac{C(0)}{n}=1$ units of progress within $\frac{R(0)}{n}=2$.

We then shift the n-sliced greedy policy by $\frac{C(0)}{n}$ to get *shifted* $(n-1)$*-sliced greedy* policy, which uses on-demand for time $\frac{C(0)}{n}$ from start (1 in the example Figure 5) and then applies $(n-1)$-sliced greedy from $t=\frac{C(0)}{n}$ until $t=R(0)-\frac{R(0)-C(0)}{n}$.

Although both policies have $c=k$, we can define a *randomized shifted greedy (RSF)* policy by using either the n-sliced or the shifted $(n-1)$-sliced greedy with equal probability at any time t. We can prove that the competitive ratio for RSF is bounded and lower than greedy (see §A.2.1).

Theorem 2. *If $R(0)\ge 2C(0)$, then for RSF policy has $c\le\frac{k+1}{2}+\frac{k-1}{2n}+O(d)<k$.*

When deadline $R(0)$ is more than $2\times$ longer than computation time $C(0)$, the worst case (the largest gap to omniscient policy) for RSF policy is bounded, *i.e.,* provably better than greedy.



(a) Theoretical vs actual greedy cost with delay $d=0.01h$.

(b) Variance vs number of slices with an 80-hour deadline.

Figure 6: Numerical results for validating the theoretical greedy cost and the assumption for increasing variance in the stochastic model. Both analysis are conducted on sampling sub-traces from 2-month AWS spot availability traces.

For $R(0)\le 2C(0)$, we can simply use on-demand until $R(t)=2C(t)$ then start using RSF policy. We denote this *modified RSF (MRSF)* policy.

Corollary 1. *Let $a=\frac{R(0)}{C(0)}-1$ for $0<a\le 1$. MRSF policy has:*

$$c\le k-ak+a(\frac{k+1}{2}+\frac{k-1}{2n})+O(d)=k-\frac{a(k-1)(n-1)}{2n}+O(d)<k$$

With MRSF policy, we shown that there exists a policy that performs better than greedy for any $R(0), C(0)$ in worst cases by randomization and distributing job progress.

4.2 Average Case with Stochastic Model

Since spot availability is a complex stochastic process, we propose a simpler model that is analytically tractable for the development of practical policies. With that model, we will show that n-sliced greedy is better than greedy in the average case.

In order to model the spot process, we consider a smoothed version where we assume that a fractional spot is always available, with a ratio $r<1$, *i.e.,* a job running on the fractional spot makes r amount of progress per unit of time. For example, if spots have 4-hour average lifetimes and 1-hour average wait times after preemption. Then, the fractional spot has a ratio, $r=4/(4+1)=0.8$, and a job using it makes 0.8 amount of progress per unit of time.

Similar as §4.1, for simplicity, we assume that d is relatively small and ignore terms of $O(d)$. We first consider greedy policy. It will use the fractional spot until $R(t')=C(t')+O(d)$ and then switch to on-demand. At time t', the job progress on the fractional spot would be $C(0)-C(t')=rt'-O(d)$, *i.e.,* $C(t')=C(0)-rt'+O(d)$, and the remaining time would be $R(t')=R(0)-t'$. We can derive t' and expected payment (total cost) p:

$$R(t')=C(t')\Longrightarrow R(0)-t'=C(0)-rt'+O(d) \quad (1)$$

$$t'=\frac{R(0)-C(0)+O(d)}{1-r} \quad (2)$$

$$p=rt'+(R(0)-t')k+O(d)=(r-k)t'+kR(0)+O(d) \quad (3)$$

We can observe that the payment depends on the fractional spot ratio r. For simplicity, we will drop $O(d)$ in following formulas. Since $r-k<0$, payment p reduces when the time

t' spent on the fractional spot increases.

In Figure 6a, we calculate both actual and theoretical costs, p, for greedy policy on real availability traces for a 48-hour job with various deadlines (52 to 92 hours) and small changeover delays. It illustrates that theoretical costs with the significant simplified stochastic modeling matches well with actual costs.

We now consider the n-sliced greedy policy from §4.1. For a fixed r, the n-sliced greedy has the same expected payment as original greedy. However, when we started considering the expected payment across difference traces, variance for fractional spot involves. We show that n-sliced greedy works better than original greedy in average.

Consider spot fraction $\mathcal{R}$ as a random variable with mean r and variance v. We can prove that the expected time on the fractional spot $E[t']$ increases with the variance v (see A.2.2). With the formula in §A.2.2, we calculate the difference of n-sliced (with variance $\hat{v}$) to original greedy (with variance v):

$$\Delta = \frac{R(0)-C(0)}{(1-r)^3}(\hat{v}-v)$$

where $\hat{v}$ is the variance over slices with length $\frac{R(0)}{n}$ and v is the variance for traces with length $R(0)$. Since $\mathcal{R}$ is averaged over time, we expect $\hat{v} > v$ (shown in Figure 6b), *i.e.*, $\Delta > 0$. We can conclude that n-sliced greedy has larger $E[t']$, leading to a lower expected cost p than original greedy in average case. Also, as v increases with n, n-sliced policy can achieve better performance with more slices, when d is relatively small.

5 Methodology

Building on our theoretical analysis, we now propose policies for real-world cloud settings. In this section, we will examine the performance of a Time Sliced policy derived from the theoretical analysis, and extend it to a parameter-free Uniform Progress policy. Additionally, we present a upper bound of cost savings through the Omniscient policy, which has the knowledge of future spot availability, and a Partial Lookahead Omniscient policy that only has a shorter lookahead of the future (*e.g.*, 6 hours). Then, we will discuss an interesting scenario when the next spot lifetime is given, and propose an extension that combines Uniform Progress with a Next Spot Lifetime Oracle. Lastly, we extend the policies to multiple instances, and relaxed job computation times and changeover delays for generality.

5.1 Time Sliced

Based on the n-sliced greedy policy in §4.1, we propose the Time Sliced policy. We divide the time before deadline, $R(0)$, into slices, and assign each slice a proportionate computation time $C(0)/n$ and deadline $R(0)/n$, denoted as C_i and R_i for slice i. In each time slice, we apply greedy policy – switching to on-demand instances when $R_i(t) < C_i(t) + 2d$. We make two changes compared to the n-sliced greedy policy: (1) jobs can continue on spot instances whenever available after $C_i(t) \leq 0$, and (2) if a slice makes more progress than required, we reduce the required computation in the succeeding slice, C_{i+1}. We do not apply randomness as in the competitive analysis for simplicity based on the assumption that clouds are non-adversarial.

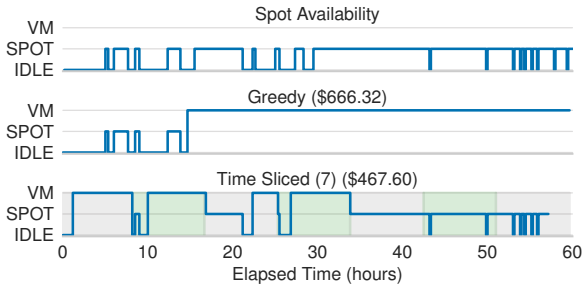


Figure 7: Example decision traces comparing Time Sliced and greedy policy. Time Sliced policy cuts costs by better utilization of available spot near deadline.

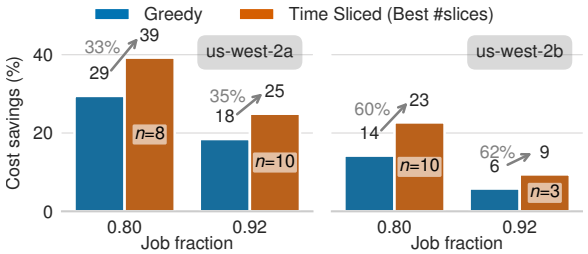


Figure 8: Cost savings (*higher is better*) vs. on-demand with Greedy and Time Sliced policies. Job fraction is $\frac{C(0)}{R(0)}$, and n is the best number of slices chosen for the Time Sliced policy.

Figure 7 presents example decision traces for both greedy and Time Sliced. The spot availability trace shows when spot is available on cloud. The greedy policy utilizes all available spot until $R(t) < C(t) + 2d$. At this point, the job cannot tolerate another changeover delay and must stay on on-demand until the end, rendering available spots close to deadline unusable. In contrast, Time Sliced policy's decision is divided into seven slices (with alternating colors), with greedy applied in each slice. Due to the progress made in earlier slices, Time Sliced allows more slacks to switch between spot and on-demand instances when the deadline is close. This enables better utilization of spot instances, reducing total cost. In this specific example, Time Sliced reduces 30% cost compared to greedy.

In Figure 8, we evaluate Time Sliced by comparing it to greedy in terms of average cost savings across 600 random p3.2xlarge availability traces on AWS. Picking the optimal number of slices enables Time Sliced to achieve 33-62% additional cost savings for relatively tight deadlines. These results suggest that *ensuring uniform progress throughout a job's lifetime* leads to better utilization of spot availability in expectation. We apply this idea in the design of Uniform Progress below.

5.2 Uniform Progress

Although Time Sliced policy with the best slice number n outperforms greedy, selecting the optimal n for different cases is not practical. We take the uniform progress idea from Time Sliced policy and design a parameter-free policy, denoted as Uniform Progress.

5.2.1 Pushing the Slices to the Extreme

Time Sliced policy guarantees uniform progress by enforcing it in discrete slices. While progress can be left behind within a slice, it is ensured by the end of each slice.

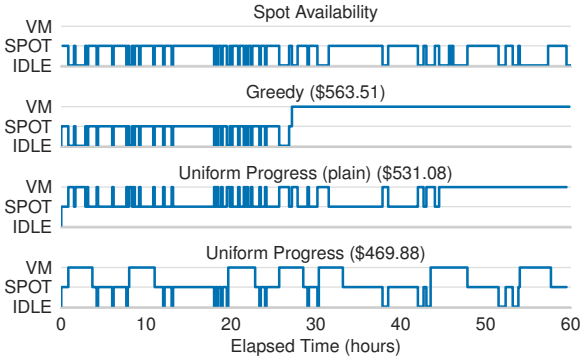


Figure 9: An example decision trace for Uniform Progress.

At the end of a slice i, $t_i = i\frac{R(0)}{n}$, *i.e.,* $i = t_i\frac{n}{R(0)}$. The *current progress*, $cp(t_i) = C(0) - C(t_i)$, is guaranteed to meet the *expected progress*, $ep(t_i)$:

$$cp(t_i) \geq ep(t_i) = i\frac{C(0)}{n} = t_i\frac{C(0)}{R(0)} \tag{4}$$

Note that when the slice number $n = 1$, there is only one t_i, *i.e.,* $t_1 = R(0)$, and Time Sliced becomes greedy policy and only enforces progress $C(0)$ at deadline $R(0)$. When more slices involve, with larger n, (4) applies to more time steps $t \in \{\frac{R(0)}{n}, \frac{2R(0)}{n}, ..., \frac{nR(0)}{n}\}$. According to the stochastic model in §4.2, n-sliced greedy will perform better when n increases, given small changeover delays. Intuitively, this is due to a more aggressive enforcement of progress. For instance, increasing n from 2 to 10 within a 50-hour deadline ensures expected progress made every 5 hours instead of every 25 hours.

We adapt this idea into Time Sliced by pushing $n \to \infty$, making each slice infinitesimal. That enforces (4) at any $t \leq R(0)$, *i.e.,* fully distributing progress within the deadline:

$$cp(t) \geq ep(t) = t\frac{C(0)}{R(0)}, \forall t \leq R(0) \tag{5}$$

5.2.2 Uniform Progress Policy

We propose a parameter-free policy, called *Uniform Progress (plain)*, that switches among three instance states: idle, spot, and on-demand. The policy, based on (5) and the rules in §3, has the following rules:

1. Uniform Progress: When the job is idle and $cp(t) < ep(t)$, switch to on-demand and stay on it to catch up progress.
2. Taking Risks: Switch to spot whenever it is available (even when $cp(t) < ep(t)$). Stay on the spot until it is preempted (Exploitation Rule).

To avoid missing deadline, we also apply Safety Net Rule on top. The first rule asks the policy to maintain steady progress, while *Taking Risks* rule allows the policy to utilize any available spot instances by taking the risk of changeover delays.

In Figure 9, we show an example decision trace. Similar to Time Sliced, Uniform Progress (plain) can achieve better cost savings compared to the greedy policy by evenly distributing progress within the deadline. However, during periods when spot life/wait time are relatively short, the policy suffers from frequent switches between spot and on-demand instances. When the job is on on-demand, and a spot becomes available,

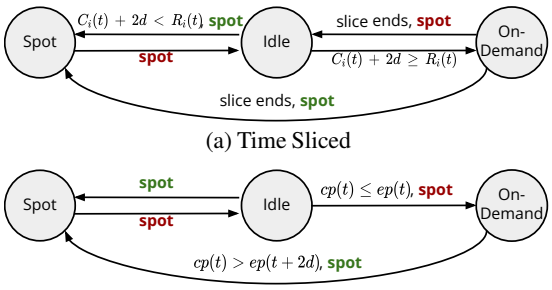


Figure 10: State machine diagram for Time Sliced and Uniform Progress. **spot** means spot unavailable and **spot** means spot available. The Safety Net Rule is left out for simplicity.

our policy will immediately switch to spot. If the spot is preempted by the cloud shortly, the job may make little progress. When that happens, $cp(t) < ep(t)$ can still hold and the job will be scheduled to on-demand again, wasting two changeover delays, $2d$ (one for spot and one for on-demand).

To address that, we propose adding hysteresis to the policy. Although the policy does not know or control the lifetime of a spot instance, it can ensure that the progress made on on-demand instances is sufficient to compensate for potential losses in the worst-case scenario. We thus add another rule:

3. Hysteresis: When the job is on on-demand, stay on it until $cp(t) \geq ep(t+2d)$.

We call the resulting policy *Uniform Progress*. Figure 9 shows that the hysteresis mitigates frequent switching by enforcing more progress on on-demand, and improves cost savings.

Figure 10 compares the state transitions of Uniform Progress and Time Sliced. Both policies share the uniform progress idea, but Time Sliced is discretized, relying on Safety Net Rule within each slice and slice boundaries to jump off an on-demand instance. In comparison, Uniform Progress replaces slice parameters with a global uniform progress checker, $cp(t) \geq ep(t)$, and a hysteresis, $cp(t) \geq ep(t+2d)$.

We will evaluate the policies above in §6. In order to properly assess a policy's performance relative to the best cost savings, we next discuss several policies, which have access to future knowledge, and use them as cost saving upper bounds.

5.3 Omniscient

First, we propose the Omniscient policy, which assumes full future knowledge and generates the theoretically optimal plan.

5.3.1 Omniscient Policy

The Omniscient policy minimizes cost for a given availability trace and deadline $R(0)$. We define some binary variables:

- $a(t)$ whether a spot instance is available at time t.
- $s(t)$, $v(t)$ indicate the policy choose to use a spot/on-demand instance at time t.
- $x(t)$, $y(t)$ represent changeover delays happen to a spot/on-demand instance at time t.

By discretizing time, we can represent the policy as a cost

minimization problem:

$$\min_{s(t),v(t)} \sum_{t=0}^{R(0)} [s(t)+v(t)k] \tag{6}$$

$$\forall t, s(t)+v(t) \le 1, s(t) \le a(t) \tag{7}$$

$$\sum_{t=0}^{R(0)} [s(t)+v(t)] \ge d \sum_{t=1}^{R(0)} (x(t)+y(t))+C(0) \tag{8}$$

$$\forall t, x(t) \le s(t), x(t) \le 1-s(t-1), x(t) \ge s(t)-s(t-1) \tag{9}$$

$$\forall t, y(t) \le v(t), y(t) \le 1-v(t-1), y(t) \ge v(t)-v(t-1) \tag{10}$$

(7) ensures the policy to choose only one instance at a time and only use spot when it is available; (8) requires the total time on spot and on-demand instances to be larger than sum of the time spent on changeover delays and the job runtime; (9) and (10) set variables $x(t)$ and $y(t)$ to 1 when a changeover occurs for spot and on-demand instances, respectively. The resulting formula is an integer linear programming (ILP) problem and can be solved using ILP solvers [14, 31].

5.3.2 Partial Lookahead Omniscient Policy

Omniscient, with complete knowledge of future spot availability, produces an unachievable bound. To better understand the impact of partial knowledge, we propose Partial Lookahead Omniscient, which has limited foresight into future spot availability. By partitioning the deadline into n slices, it can only see complete availability within each slice. To incorporate that knowledge, we modify Omniscient formula to minimize the average cost of progress made in a slice i while ensuring the job progress at the end to be at least $iC(0)/n$. Further details can be found in §A.3.

5.4 Next Spot Lifetime Oracle

Both Omniscient and Partial Lookahead Omniscient policies assume complete knowledge of future availability with different lookahead windows. We propose a more realistic scenario where cloud providers offer an oracle $o(t)$ that returns the lifetime of the next spot instance a job can acquire at the current time t. This assumption is reasonable as providers can determine when to reclaim a spot instance.

Uniform Progress can be extended to leverage this oracle. We introduce two new conditions to replace the hysteresis:

1. If the job is idle, we only switch to spot when the average cost per unit of progress is lower than on-demand cost $\frac{o(t)}{o(t)-d} < k$, *i.e.,* $o(t) > \frac{kd}{k-1}$.
2. If the job is on on-demand instance, we switch to spot only when the average cost per unit of progress, considering switching to spot and back to on-demand, is less than staying on the current on-demand: $\frac{o(t)+kd}{o(t)-d} < k$, *i.e.,* $o(t) > \frac{2kd}{k-1}$.

5.5 Extending to Multiple Instances

All the discussions above are based on single-instance scenario. We now extend the policies to multiple instances. We assume gang-scheduling is required, *i.e.,* all instances must be running for a job to progress. This is typical in distributed ML training [22, 23, 36] and HPC workloads [11]. A cluster may consist solely of spot instances, on-demand instances, or a mix of both. We call clusters with an identical resource type *homogeneous* and those with a mix *heterogeneous*. Changeover delays are incurred when a cluster is reconfigured, *i.e.,* the number of spot/on-demand instances in it changes, unless it has no instance after reconfiguration.

We introduce a new rule for all multi-instance policies:

Polarization Rule. For a job requiring $N > 1$ instances, a policy should either use no instance or N instances at any time.

Since gang-scheduling is required, a cluster with fewer than N instances incurs unnecessary costs without job progress. Thus, once any instance is preempted, a policy should immediately reconfigure the cluster to either 0 or N instances.

We now extend previous policies to multiple instances.

Extending Greedy and Uniform Progress. First, observing that spot availability tends to change simultaneously for multiple instances (§2.4), we propose each policy should produce homogeneous clusters. We will show that this assumption does not harm performance on reasonably large clusters (§6.6). Combining this with *Polarization Rule*, the action space for a policy is simplified to either: N spot, N on-demand, or no instances at any time t.

The problem for multiple instances is now equivalent to the single instance, with the one-to-one mapping of states (§3.2):

- *Cluster state*: N spot, N on-demand, or no instances map to spot, on-demand or idle states for single-instance jobs.
- *Spot state*: If available spot instances $a(t) < N$, it is equivalent to a spot being unavailable in the single-instance scenario, and $a(t) = N$ maps to a spot being available.

Thus, for multi-instance jobs, we directly execute greedy and Uniform Progress using the mappings above.

Extending Omniscient. For Omniscient, we can also restrict it to produce homogeneous clusters and get Omniscient (Homogeneous). The detailed formulation is in §A.7.2. To obtain a better theoretical upper bound for cost savings, however, we further adapt Omniscient to support heterogeneous clusters, denoted as Omniscient (Heterogeneous), by modifying the ILP (6) to factor in a mixed cluster configuration (§A.7.3).

5.6 Relaxing Computation Time and Changeover Delay

In real-world scenarios, exact computation times and changeover delays may be uncertain. We generalize our model to accommodate such variability.

Computation time. To account for the inaccuracies of a user-provided job computation time $\bar{C}(0)$, we denote the difference to the actual job computation time as $\delta = C(0) - \bar{C}(0)$. Given that no policy can predict $C(0)$ precisely beforehand, we adjust the deadline guarantee of the policies to be best effort, ensuring a finish time within the original deadline plus the difference, $R(0)+\delta$. This is guaranteed by having all policies stay on the current instance and switch to on-demand,[5] after the job does not finish but has already made $\bar{C}(0)$ progress, *i.e.,* $\bar{C}(t) \le 0$. When a user overestimates a job's computation time $\bar{C}(0) > C(0)$, it should finish before the original deadline.

[5] If the job was on a spot instance, it should switch to on-demand after the spot instance is preempted (Exploitation Rule).

Otherwise, if the job computation time is underestimated $\bar{C}(0) < C(0)$, the job should finish within the original deadline plus the difference. Note that there is no additional d, as Safety Net Rule guarantees that when $\bar{C}(t) \to 0$, the job should either be on on-demand already or $R(t) \geq \bar{C}(t) + d$, *i.e.,* there is a spare d for the job to switch to on-demand.

Changeover delay. We now adjust the model to factor in system stragglers and variations in changeover delay. We assume that no policies can foresee the exact changeover delay until its occurrence, though the average changeover delay is given. If the maximum possible changeover delay, $\hat{d}$, is also given (*e.g.,* the most significant possible progress loss is triggered), we can prove that policies should be able to ensure a deadline of $R(0) + 2(\hat{d} - d)$. The proof can be found in §A.4. If a user would like to ensure the original deadline with a given maximum changeover delay, they can specify a new deadline $R(0) - 2(\hat{d} - d)$.

With the new model, policies can account for the variety, by guaranteeing a bounded relaxed deadline $R(0) + \delta + 2(\hat{d} - d)$.

6 Evaluation

In this section, we conduct experiments to assess the performance of the proposed policies using real spot instance traces collected from the cloud.

6.1 Datasets and Setup

We collected spot availability traces on AWS (§2.1). These traces include a 2-week availability trace started on 10/26/2022, with four instance types: p3.2xlarge/p3.16xlarge (1/8 V100), p2.2xlarge/p2.16xlarge (1/8 K80), and two availability zones: us-west-2a and us-west-2b. Moreover, we collect a 2-month long availability trace started on 02/15/2023 for p3.2xlarge instances across nine zones from regions, us-east-1, us-east-2, and us-west-2. For multiple instances, we collect 2-week *preemption* traces for 4 p3.2xlarge in 3 AWS zones (*us-east-1f*, *us-east-2a*, *us-west-2c*), and 2-week *availability* traces for 16 p3.2xlarge in 3 zones (*us-east-2a*, *us-west-2b*, *us-west-2c*). All the availability traces were collected with a 10-minute probe interval. As demonstrated in §2.1, availability and preemption traces are highly correlated, indicating that the performance of the policies on availability traces should reflect their real-world performance. We will use preemption traces in §6.6 for multiple instances benchmark and §7.2 for real system evaluation.

We evaluate the policies on both 2-week traces, and 2-month traces. For all experiments, we randomly sample 300 starting points for each trace, considering each pair of instance type and zone. We consider cases where the *job fraction* $\frac{C(0)}{R(0)} > 0.6$, *i.e.,* the deadline is relatively tight, as the problem becomes less interesting when deadlines are loose and available spot instances within deadline are sufficient to complete the job. For loose deadlines, jobs can utilize spot instances whenever they are available until the remaining time-to-deadline $R(t)$ is relatively tight compared to the remaining computation time $C(t)$, and then start applying policies (details in §A.8). The computation time is set to 48 hours for consistent comparison across different settings (experiments for different computation times can be found in §A.9). Unless noted, changeover delays d are set to 0.2 hours and costs are normalized by on-demand costs in all experiments.

Policy	**On-Demand** (hours)	**Spot** (hours)	**Spot Util.**
On-Demand	48.0 ± 0.0	0.0 ± 0.0	0%
Greedy	30.8 ± 17.7	17.2 ± 17.7	63%
Uniform Progress	25.1 ± 15.3	22.9 ± 15.4	84%
Omniscient	20.7 ± 15.5	27.4 ± 15.5	100%

Table 3: Compute time spent on on-demand and spot instances, averaged across 8 scenarios for a job fraction of 0.8. "Spot Util." indicates the fraction of compute time on spot leveraged by a policy vs. the Omniscient policy.

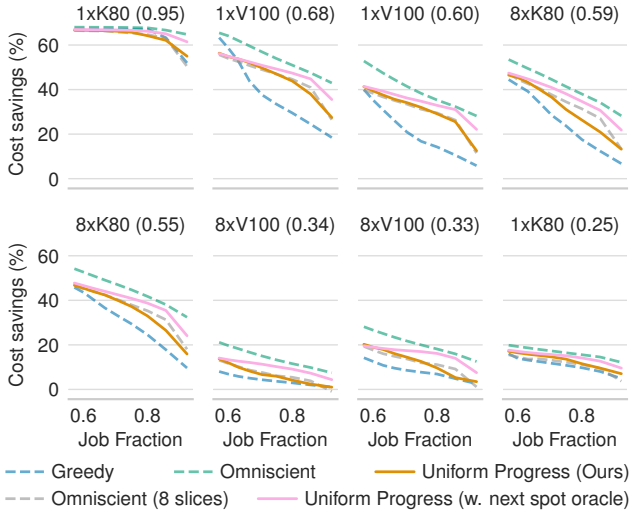


Figure 11: Cost savings (*higher is better*) against on-demand instances on real spot availability traces. Omniscient (8 slices) is Partial Lookahead Omniscient. Larger job fraction means tighter deadline. Each sub-plot is on a (instance type, zone) trace. Values in '(x)' are average spot fractions (percentages of time a spot instance is available) across all samples in the trace.

Baselines. To our knowledge, existing methods in literature (§8) do not consider switching between spot and on-demand in a cost optimization and deadline adherence setting for batch jobs. Thus, we compare our results against policies with future knowledge, which serve as strong upper bounds.

6.2 Time Spent on On-demand and Spot Instances

We first show different policies' overall compute times on on-demand vs. spot instances, which exclude changeover delays. Such breakdowns examine how well spot instances are utilized. Table 3 shows the results with a fixed job fraction $\frac{C(0)}{R(0)} = 0.8$ on the 2-week traces, averaging across eight (instance type, availability zone) pairs, each with 300 randomly sampled traces. We observe that our Uniform Progress runs on spot instances 21% longer than greedy policy on average, reducing the gap to Omniscient's spot usage by 57%.

6.3 Various Deadlines

Figure 11 evaluates the cost savings achieved by the policies across various deadlines (represented as job fractions)

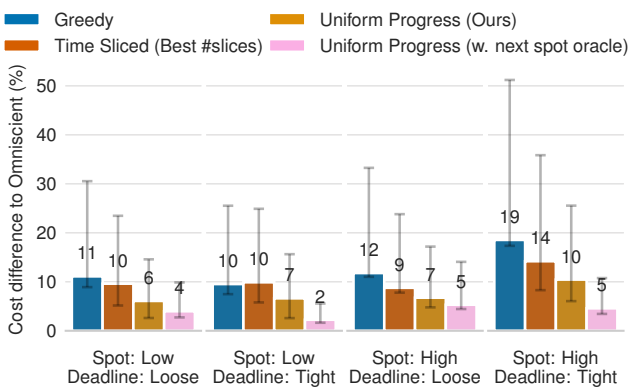


Figure 12: Cost difference compared to Omniscient policy (normalized by on-demand cost, *lower is better*), measuring a policy's proximity to Omniscient. Error bars range from p25 to p75. "Spot" represents spot fraction. It is on 2-week availability traces, aggregated on 300×8=2400 sampled traces.

on the 2-week availability traces. Our Uniform Progress consistently surpasses the greedy in cost savings in all cases, while approaching savings of Omniscient policy.

While Uniform Progress excels, there is still a gap to Omniscient. We compare Uniform Progress with Partial Lookahead Omniscient policy with 8 slices, which assumes strong knowledge of the future (around 6 hours of lookahead): Uniform Progress achieves similar performance in most cases, despite lacking future knowledge. This suggests any other policy without future knowledge may not yield much higher savings.

We also investigate the potential improvement of Uniform Progress policy by assuming cloud providers offering an oracle for the lifetime of the next spot instance (§5.4). With such knowledge, cost savings improve significantly, nearing the theoretical optimum when deadlines are tighter.

The conclusions also hold on the 2-month traces (§A.5).

6.4 Impact of Spot Fraction and Deadline

To better understand the influence of *spot fractions* (the percentage of time a spot instance is available) and deadlines on policy performance, we categorize them into two dimensions: low or high spot fraction, and loose or tight deadline. Tight deadline represents job fraction $\frac{C(0)}{R(0)} > 75\%$, while high spot fractions are defined as those exceeding 50%. Our 2-week traces have an even distribution between high and low spot fractions, while the 2-month traces show a dominance of high spot fractions, which forms 72% of all cases. This aligns with our earlier observation of the volatile nature of spot instance availability (§2.2). Figure 12 presents the performance of the policies compared to Omniscient policy (theoretical upper bounds for cost savings) in the four categories. Results based on the 2-month traces are covered in §A.5.

For tight deadlines, the number of feasible instance switches is limited to at most $\frac{R(0)-C(0)}{d}$, demanding strategic planning of each changeover. When spot availability is high and deadline tight (the rightmost group of bars), all policies lacking future knowledge exhibit a relatively large gap to the optimal.

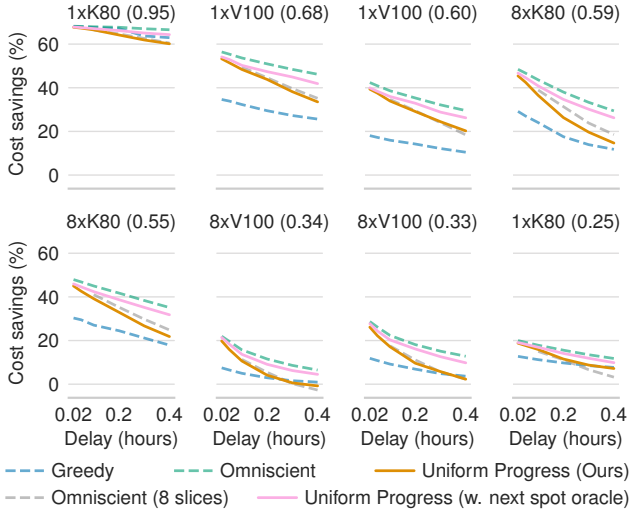


Figure 13: Impact of changeover delays (d). Values in '(x)' are average spot fractions over all samples in the trace.

Nevertheless, Uniform Progress still reduces the gap by ∼2× compared to the greedy policy. This efficiency arises from its uniform progress guarantee and hysteresis, which optimize spot utilization within the deadline while avoiding frequent changeovers. The small gap between Uniform Progress with the next spot lifetime oracle and Omniscient policy confirms that the ability to skip short spot lifetimes and strategically switching from on-demand to spot with the opportunity cost in mind is crucial to achieve close to optimal performance.

As deadlines loosen and spot availability increases, all policies perform closer to Omniscient policy, as jobs have greater flexibility to wait for spot instances and switch between resource types, *i.e.,* judicious planning becomes less important.

Additionally, we show the performance of Time Sliced policy with the best number of slices (within 50 slices). Time Sliced policy outperforms Greedy because of uniform progress it guarantees, but worse than Uniform Progress, potentially due to a higher overhead between slice switches.

Regardless of the different categories, our Uniform Progress policy reduces the gap to optimal by nearly 2× compared to greedy policy for both average and tail (p75) cases.

6.5 Different Changeover Delays

In Figure 13, we evaluate the performance of policies across various changeover delays. Our Uniform Progress performs consistently similar to the Partial Lookahead Omniscient policy. As changeover delays increase, cost savings compared to on-demand instances are reduced. This is because for each spot instance being used, a larger changeover delay means we pay the same price for less actual progress, so that switching to spot instances becomes less economical. Both Uniform Progress and Partial Lookahead Omniscient approach the greedy policy as d increases. However, Uniform Progress combined with the next spot lifetime oracle consistently remains close to the upper bound, due to its ability to skip short spot lifetimes and judiciously calculate the opportunity

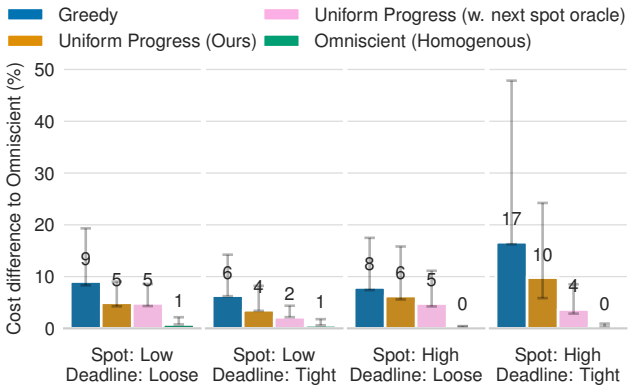


Figure 14: Cost savings for jobs on 4 instances, compared to Omniscient with heterogeneous clusters.

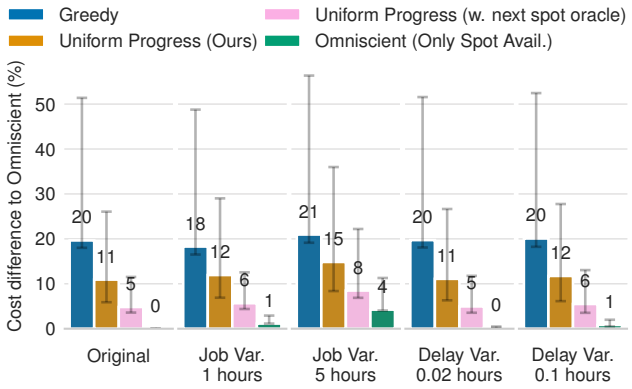


Figure 15: Cost savings with relaxed job computation time or changeover delays. All policies are compared against Omniscient knowing exact spot availability, computation time, and changeover delays in advance. Omniscient (Only Spot Avail.) only has the information of spot availability.

cost of switching from an on-demand to a spot instance.

6.6 Multiple Instances

We now evaluate the policies on multi-instance jobs. Figure 14 shows the cost savings on 4-instance clusters for various policies compared to the theoretical upper bound set by our Omniscient (Heterogeneous) policy (§5.5). The difference between Omniscient (Homogeneous) and Omniscient (Heterogeneous) is negligible (at most 1%), which validates the use of homogeneous clusters in our policy formulation. Our Uniform Progress consistently outperforms the greedy policy, especially in high-spot-availability, tight-deadline conditions, which agrees with the conclusion on single-instance jobs (§6.4). We observe a similar win for clusters with 16 instances (§A.7.4). Due to monetary budget limits, we leave the extension to larger clusters ($N > 16$) to future work.

6.7 Relaxed Computation Time and Changeover Delay

We show that the variations for computation time and changeover delays introduced in §5.6, marginally influence the cost savings. In Figure 15, we apply a uniformly distributed variance to the computation time and changeover delays, and compare all policies with Omniscient policy, which possesses exact knowledge of the job and delays. The experiments are conducted in the same settings as §6.4, with a single instance, high spot fraction, and tight deadline. All policies can guarantee deadlines in the new model. The performance of Omniscient with only spot availability information degrades when the variance of computation time increases. When a user-provided job computation time is larger than the actual one $\bar{C}(0) > C(0)$ (overestimate), it cannot fully utilize spot instances close to the deadline, while, for $\bar{C}(0) < C(0)$ (underestimate), it has to use on-demand after exceeding the original deadline $R(0)$. Similarly, it performs worse when the variance of changeover delay increases, due to sub-optimal decisions made with partial information. However, in all cases, we observe Uniform Progress outperforms greedy with a relatively stable gap, indicating its robustness.

7 Practical Usage

In this section, we discuss our implementation of the prototype and evaluate it with three real-workload: machine learning training, bioinformatics (HPC), and data analytics.

7.1 Implementation

We implemented the policies on top of a real multi-cloud system, SkyPilot [44], that supports launching instances on the public cloud providers. Given an availability zone and an instance type to use, our policies drive a job's resource provisioning and switching decisions.

In the system, a controller is in charge of monitoring spot availability and managing the job with heartbeats. All policies are invoked by the controller behind a simple interface as follows. Periodically, the policy observes current_instance_state (in {idle, spot, on-demand}) and a boolean is_spot_available through the controller, and then uses them to compute a decision (in the same state set). If the decided state differs from the current instance state, the decision is executed by the system's provisioner module (*e.g.,* switch from on-demand to spot). To obtain the boolean is_spot_available, the controller invokes cloud-specific capacity reservation APIs (*e.g.,* AWS EC2 offers a create_capacity_reservation API) which return whether a zone has capacity for a spot instance type.

7.2 Real Workloads

We validate our policy across AWS and GCP platforms using real-world preemption traces with spot availabilities ranging from 70% to 90%. Metrics like changeover delays and other system lags are measured directly from the implementation and included in the evaluation. We summarize the settings of the three workloads, Machine Learning (ML) Training, Bioinformatics, Data Analytics, in Table 4 and explain details in §A.6.

We consider two different deadlines (job fractions 90% and 75%) for each workload. We first present detailed cost breakdowns for the ML workload with loose deadlines in Figure 16. Uniform Progress achieves 48% cost savings compared to only using on-demand. It outperforms Greedy (15%) and approaches the optimal (55%). Similar patterns are observed in the other two workloads. We show the cost savings in Table 5. For Bioinfo's c3-highcpu-88, the spot price is 91%

Workload	Location	Instance Type	Spot Price (Discount)	Computation	Deadlines	Changeover Delay
ML Training	AWS (us-west-2b)	p3.2xlarge	$0.92/hr (-67%)	72 hrs	84/100 hrs	4+5+9 mins ≈0.3 hrs
Bioinformatics	GCP (us-east1-b)	c3-highcpu-88	$0.34/hr (-91%)	22.5 hrs	24/28 hrs	2+1+8 mins ≈0.2 hrs
Data Analytics	AWS (us-east-1c)	r5.16xlarge	$1.85/hr (-55%)	27 hrs	30/36 hrs	4+1+7 mins ≈0.2 hrs

Table 4: Detailed characteristics of real workloads. Deadlines are derived from job fractions 90% and 75%, and changeover delays are the sum of VM provisioning, environment setup, and job recovery progress loss time.

		Uniform Progress	
Workload	On-demand	Tight DDL (0.9)	Loose DDL (0.75)
ML	$233.5	$138.2 (**-41%**)	$122.0 (**-48%**)
Bioinfo	$140.5	$51.9 (**-63%**)	$22.8 (**-84%**)
Analytics	$109.6	$80.0 (**-27%**)	$74.1 (**-32%**)

Table 5: Cost savings for real workloads. Results of two deadlines are shown (job fractions 0.9 and 0.75).

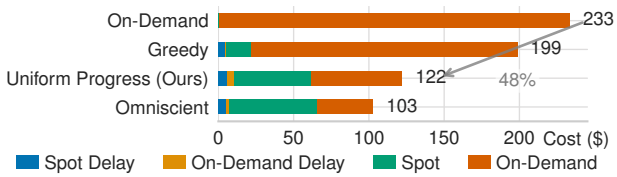


Figure 16: Cost breakdown of each policy for ML workload.

cheaper than on-demand. This allows Uniform Progress to achieve 63% cost savings even when the deadline is tight and 84% savings when the deadline is loose. For the analytics workload, the spot price discount is much smaller (55%). In this case, Uniform Progress achieved 27% and 32% savings for tight and loose deadlines, respectively. Note, however, these savings still approach those achieved by the Omniscient (32% and 46%, for tight and loose deadlines).

8 Related Work

Spot pricing and availability modeling. AWS pioneered spot instances in 2009, using a bidding mechanism to monetize unused cloud capacity [5]. The pricing model has evolved to offer more stability, diminishing bidding, with other cloud providers adopting similar strategies, such as GCP's constant 30-day spot price [2], Oracle Cloud's fixed 50% discount for preemptible instances [6], and Azure's stable regional pricing [21]. While spot pricing is relatively stable, modeling spot availability remains challenging due to its black box nature. While prior work attempted to model preemption patterns [20] and employed ML prediction methods [16, 42, 43], we design our policy to be robust against potential changes in spot eviction strategies of the cloud providers.

Applications using spot instances. The cost-effective nature of spot instances has driven their adoption for savings. Frameworks like Bamboo [38], Spotnik [40], and Srifty [28], was developed for machine learning on spot instances. Narayanan *et al.* [33] showed significant reductions in machine learning training costs using spot instances across multiple clouds. CompuCache [47] leverages spot instances for in-memory data caching. However, preemptions can negatively impact application performance [10, 43], and deadline-constrained applications may struggle to effectively utilize spot instances.

Job scheduling with preemptions. Running jobs on preemptive devices is investigated on intermittent systems, where jobs can be interrupted due to sporadic harvestable energy. Many studies [12, 15, 18] focus on scheduling multiple real-time IoT tasks, due to the limited computation resources on these devices. Spot instances introduce preemption to resource-demanding batch jobs on clouds. From the cloud providers' perspective, existing work [9, 17, 19] investigates how to maximize revenue, or enhance runtime guarantees. For end-users, earlier studies explored bidding-based policies for bag of tasks with deadlines [30, 35, 39, 45], but these approaches are less applicable to current spot markets due to changes in pricing model. Recently, Snape [43] investigates using a mix of spot and on-demand instances for long-running services. It optimizes for SLO which require the number of instances available to be close to the target one at any time. It is different from deadline-sensitive batch jobs studied in this paper, where the job can stay idle for long periods, as long as it can meet the deadline.

9 Conclusion

Spot instances are economically appealing, but unreliable due to the preemptions. In this paper, we resolve a critical challenge of minimizing the cost for delay-sensitive jobs by utilizing a mix of spot and on-demand instances. Our work features a comprehensive analysis of spot instances and presents a theoretical framework to assess policies in both worst and average cases. This inspires the development of our proposed policy, Uniform Progress, which is simple, parameter-free, and effective without relying on assumptions of spot availability. Our empirical study using 3-month real-world traces demonstrates a significant improvement in cost savings compared to the greedy policy, closing gaps with the optimal policy by approximately 2× on both single or multiple instances. We also find that if cloud providers were willing to offer an oracle for the next spot instance's lifetime, it could further improve applications' cost efficiency, by enabling our Uniform Progress to approach the upper bound of cost savings. We implemented a prototype on top of SkyPilot, and showcased the effectiveness of Uniform Progress on three real workloads, reducing the cost by 27%-84%. We open source the spot traces for future research.

Acknowledgements. We thank the NSDI reviewers for their valuable feedback. This work is in part supported by gifts from Accenture, AMD, Anyscale, Google, IBM, Intel, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, Uber, and VMware.

References

[1] Amazon EC2 Spot customers. https://aws.amazon.com/ec2/spot/customers/.

[2] GCP Spot VMs Pricing. https://cloud.google.com/compute/docs/instances/spot#pricing.

[3] Google Cloud Spot VM Pricing. https://cloud.google.com/compute/docs/instances/spot#pricing.

[4] Navigating the High Cost of AI Compute. https://a16z.com/2023/04/27/navigating-the-high-cost-of-ai-compute/.

[5] New Amazon EC2 Spot pricing model: Simplified purchasing without bidding and fewer interruptions. https://aws.amazon.com/blogs/compute/new-amazon-ec2-spot-pricing/.

[6] Oracle Computing Pricing. https://www.oracle.com/cloud/compute/pricing/.

[7] Pretraining RoBERTa using your own data. https://github.com/facebookresearch/fairseq/blob/main/examples/roberta/README.pretraining.md.

[8] Vantage Cloud Cost Breakdown Report. https://www.vantage.sh/cloud-cost-report/2023-q1.

[9] F. Alzhouri, A. Agarwal, and Y. Liu. Maximizing cloud revenue using dynamic pricing of multiple class virtual machines. *IEEE Transactions on Cloud Computing*, 9(2):682–695, 2018.

[10] P. Ambati, I. Goiri, F. Frujeri, A. Gun, K. Wang, B. Dolan, B. Corell, S. Pasupuleti, T. Moscibroda, S. Elnikety, M. Fontoura, and R. Bianchini. Providing SLOs for Resource-Harvesting VMs in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751. USENIX Association, Nov. 2020.

[11] I. Buch, M. J. Harvey, T. Giorgino, D. P. Anderson, and G. De Fabritiis. High-throughput all-atom molecular dynamics simulations using distributed computing. *Journal of Chemical Information and Modeling*, 50(3):397–403, 2010.

[12] M. Chetto. Optimal scheduling for real-time jobs in energy harvesting computing systems. *IEEE Transactions on Emerging Topics in Computing*, 2(2):122–133, 2014.

[13] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. N. Tantawi, and C. Krintz. See spot run: using spot instances for mapreduce workflows. *HotCloud*, 10:7–7, 2010.

[14] J. Forrest, T. Ralphs, H. G. Santos, S. Vigerske, J. Forrest, L. Hafer, B. Kristjansson, jpfasano, EdwinStraver, M. Lubin, Jan-Willem, rlougee, jpgoncal1, S. Brito, h-i gassmann, Cristina, M. Saltzman, tosttost, B. Pitrus, F. MATSUSHIMA, and to st. coin-or/cbc: Release releases/2.10.10, Apr. 2023.

[15] H. E. Ghor, M. Chetto, and R. H. Chehade. A real-time scheduling framework for embedded systems with environmental energy harvesting. *Computers & Electrical Engineering*, 37(4):498–510, 2011.

[16] A. Harlap, A. Chung, A. Tumanov, G. R. Ganger, and P. B. Gibbons. Tributary: spot-dancing for elastic services with latency SLOs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 1–14, Boston, MA, July 2018. USENIX Association.

[17] S. M. Iqbal, H. Li, S. Bergsma, I. Beschastnikh, and A. J. Hu. Cospot: A cooperative vm allocation framework for increased revenue from spot instances. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 540–556, New York, NY, USA, 2022. Association for Computing Machinery.

[18] B. Islam and S. Nirjon. Scheduling computational and energy harvesting tasks in deadline-aware intermittent systems. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 95–109. IEEE, 2020.

[19] N. Jain, I. Menache, J. Naor, and J. Yaniv. Near-optimal scheduling mechanisms for deadline-sensitive jobs in large computing clusters. *ACM Transactions on Parallel Computing (TOPC)*, 2(1):1–29, 2015.

[20] J. Kadupitige, V. Jadhao, and P. Sharma. Modeling the temporally constrained preemptions of transient cloud vms. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, page 41–52, New York, NY, USA, 2020. Association for Computing Machinery.

[21] S. Lee, J. Hwang, and K. Lee. Spotlake: Diverse spot instance dataset archive service. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 242–255, 2022.

[22] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, Oct. 2014. USENIX Association.

[23] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, et al.

Pytorch distributed: Experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment*, 13(12), 2019.

[24] R. Liaw, R. Bhardwaj, L. Dunlap, Y. Zou, J. E. Gonzalez, I. Stoica, and A. Tumanov. Hypersched: Dynamic resource reallocation for model development on a deadline. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 61–73, 2019.

[25] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[26] H. Liu, Q. Zeng, J. Zhou, A. Bartlett, B.-A. Wang, P. Berube, W. Tian, M. Kenworthy, J. Altshul, J. R. Nery, H. Chen, R. G. Castanon, S. Zu, Y. E. Li, J. Lucero, J. K. Osteen, A. Pinto-Duarte, J. Lee, J. Rink, S. Cho, N. Emerson, M. Nunn, C. O'Connor, Z. Yao, K. A. Smith, B. Tasic, H. Zeng, C. Luo, J. R. Dixon, B. Ren, M. M. Behrens, and J. R. Ecker. Single-cell dna methylome and 3d multi-omic atlas of the adult mouse brain. *bioRxiv*, 2023.

[27] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[28] L. Luo, P. West, P. Patel, A. Krishnamurthy, and L. Ceze. Srifty: Swift and thrifty distributed neural network training on the cloud. *Proceedings of Machine Learning and Systems*, 4:833–847, 2022.

[29] A. Marathe, R. Harris, D. K. Lowenthal, B. R. de Supinski, B. Rountree, and M. Schulz. Exploiting redundancy and application scalability for cost-effective, time-constrained execution of hpc applications on amazon ec2. *IEEE Transactions on Parallel and Distributed Systems*, 27(9):2574–2588, 2015.

[30] I. Menache, O. Shamir, and N. Jain. On-demand, spot, or both: Dynamic resource allocation for executing batch jobs in the cloud. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 177–187, Philadelphia, PA, June 2014. USENIX Association.

[31] S. Mitchell, M. O'Sullivan, and I. Dunning. PuLP: A Linear Programming Toolkit for Python. 2011.

[32] R. O. Nambiar and M. Poess. The Making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, page 1049–1058. VLDB Endowment, 2006.

[33] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia. Analysis and exploitation of dynamic pricing in the public cloud for ml training. In *Workshop on Distributed Infrastructure, Systems, Programming, and AI*, August 2020.

[34] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 48–53, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[35] D. Poola, K. Ramamohanarao, and R. Buyya. Fault-tolerant workflow scheduling using spot instances on clouds. *Procedia Computer Science*, 29:523–533, 2014. 2014 International Conference on Computational Science.

[36] A. Sergeev and M. Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[37] J. Thorpe, P. Zhao, J. Eyolfson, Y. Qiao, Z. Jia, M. Zhang, R. Netravali, and G. H. Xu. Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, Boston, MA, Apr. 2023. USENIX Association.

[38] J. Thorpe, P. Zhao, J. Eyolfson, Y. Qiao, Z. Jia, M. Zhang, R. Netravali, and G. H. Xu. Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, Boston, MA, Apr. 2023. USENIX Association.

[39] P. Varshney and Y. Simmhan. Autobot: Resilient and cost-effective scheduling of a bag of tasks on spot vms. *IEEE Transactions on Parallel and Distributed Systems*, 30(7):1512–1527, 2019.

[40] M. Wagenländer, L. Mai, G. Li, and P. Pietzuch. Spotnik: Designing distributed machine learning for transient cloud resources. In *Proceedings of the 12th USENIX Conference on Hot Topics in Cloud Computing*, pages 4–4, 2020.

[41] S. Wang and M. Casado. The Cost of Cloud, a Trillion Dollar Paradox. https://a16z.com/2021/05/27/cost-of-cloud-paradox-market-cap-cloud-lifecycle-scale-growth-repatriation-optimization.

[42] F. Yang, B. Pang, J. Zhang, B. Qiao, L. Wang, C. Couturier, C. Bansal, S. Ram, S. Qin, Z. Ma, I. n. Goiri, E. Cortez, S. Baladhandayutham, V. Rühle, S. Rajmohan, Q. Lin, and D. Zhang. Spot virtual machine eviction prediction in microsoft cloud. In *Companion Proceedings*

of the Web Conference 2022, WWW '22, page 152–156, New York, NY, USA, 2022. Association for Computing Machinery.

[43] F. Yang, L. Wang, Z. Xu, J. Zhang, L. Li, B. Qiao, C. Couturier, C. Bansal, S. Ram, S. Qin, Z. Ma, I. n. Goiri, E. Cortez, T. Yang, V. Rühle, S. Rajmohan, Q. Lin, and D. Zhang. Snape: Reliable and low-cost computing with mixture of spot and on-demand vms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 631–643, New York, NY, USA, 2023. Association for Computing Machinery.

[44] Z. Yang, Z. Wu, M. Luo, W.-L. Chiang, R. Bhardwaj, W. Kwon, S. Zhuang, F. S. Luan, G. Mittal, S. Shenker, and I. Stoica. SkyPilot: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 437–455, Boston, MA, Apr. 2023. USENIX Association.

[45] M. Zafer, Y. Song, and K.-W. Lee. Optimal bids for spot vms in a cloud for deadline constrained jobs. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 75–82, 2012.

[46] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, oct 2016.

[47] Q. Zhang, P. Bernstein, D. S. Berger, B. Chandramouli, B. T. Loo, and V. Liu. Compucache: Remote computable caching using spot vms. In *Conference on Innovative Data Systems Research (CIDR 2022)*, January 2022.

A Appendix

A.1 Spot Availability and Preemption Traces

In Section 2.2, we highlighted the variability in spot availability across four out of nine AWS availability zones. For a comprehensive view, Figure 17 presents data for all nine zones. It demonstrates the fluctuations in spot availability both across zones and over time.

A.2 Proofs for Theoretical Analysis

In this section, we show the detailed proofs for the theorems and statements in our theoretical analysis (§4).

A.2.1 Worst Case with Competitive Analysis

We first prove Theorem 1 (see §4.1), which states that a deterministic policy cannot perform better than the greedy policy in competitive analysis.

Theorem 1. *For any deterministic policy P, $c \geq k - O(d)$.*

Proof. Since the policy P is deterministic, the adversary can choose spot availability as follows. It makes the spot available only when P starts using on-demand or $R(t) = C(t) + d$. If P switches to the spot, the adversary waits for d units of time, then preempts the spot, so P makes no progress on any spot instances, *i.e.,* must use at least $C(0)$ units of on-demand.

With that adversary, we examine P have to use all on-demand while omniscient policy can finish the job with all spots. Consider the first time t', where $R(t') = C(t') + d$. Over $t' \leq t \leq R(0)$, P cannot switch to spot, but the omniscient policy could as it knows the spot will remain available, *i.e., P* makes $C(t')$ progress on on-demand, while omniscient is on spot. Next, P must have accumulated the $C(0) - C(t')$ before t'. Due to the adversary, any work accumulated before t' should be on on-demand when a spot is available. Thus, the omniscient policy can make $C(0) - C(t')$ of progress on those spots before t'. □

We now show the proof for Theorem 2, *i.e.,* the competitive ratio for randomized shifted greedy (RSF) policy is bounded and lower than greedy.

Theorem 2. *If $R(0) \geq 2C(0)$, then for RSF policy has $c < k$.*

Proof. By ignoring the terms of $O(d)$, at any time t before the last split, at least one of the policies is looking to use a spot (as shown in Figure 5), so any available spot is used for at least half of the time. Thus, except for the last $C(0)/n$ progress, at least half of the remaining progress is done on spot instances:

$$c \leq (\frac{1}{n} + \frac{1-1/n}{2})k + \frac{1-1/n}{2} + O(d) = \frac{k+1}{2} + \frac{k-1}{2n} + O(d) < k \quad \square$$

A.2.2 Average Case with Stochastic Model

In §4.2, we inferred that the expected payment (total cost), p, for greedy policy in the stochastic model decreases (lower the better) when the time t' spent on the fractional spot increases. Consider spot fraction $\mathcal{R}$ as a random variable with mean r and variance v. We now prove that the expected time spent on the fraction spot $E[t']$ increases when v increases, *i.e.,* larger v indicates lower expected cost.

Proof. Let $\mathcal{R} = r + \delta$. Based on (2), we have $E[t']$:

$$E[t'] = E[\frac{R(0) - C(0)}{1 - (r+\delta)}] = \frac{R(0) - C(0)}{1-r} E[1 + \frac{\delta}{1-r} + \frac{\delta^2}{(1-r)^2} + \cdots]$$

where the second equation is derived from Taylor expansion for $\delta \to 0$. By construction, $E[\delta] = 0$ and $E[\delta^2] = v > 0$. When we take the first three terms, we get an approximation:

$$E[t'] = \frac{R(0) - C(0)}{1-r}(1 + \frac{v}{(1-r)^2})$$

We calculate the difference of the expected time on the fractional spot $E[t']$ for policies with variance v_1 and v_2

$$\Delta = \frac{R(0) - C(0)}{(1-r)^3}(v_1 - v_2)$$

Since $\Delta > 0$ when $v_1 > v_2$, we can conclude that $E[t']$ increases with the variance v. □

A.3 Partial Lookahead Omniscient Formulation

Instead of minimizing the total cost for the progress, in Partial Lookahead Omniscient policy, a job can make more progress than it is assigned in each slice and reduce the computation time required in the next slice by the additional

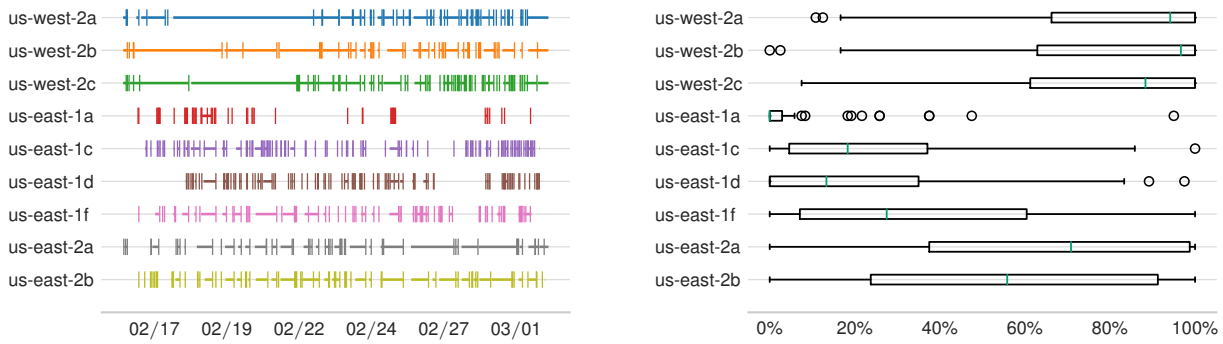


Figure 17: Spot availability is highly unpredictable and volatile. Traces are across nine AWS zones collected.

progress made. Therefore, we modify the ILP formula for the Omniscient policy with the following formula for a slice i to minimize the average cost of the progress made in that slice:

$$\min_{s(t>t_{i-1}),v(t>t_{i-1})} \sum_{t=t_{i-1}}^{t_i} [s(t)+v(t)k]/P_i \tag{11}$$

$$\forall t, s(t)+v(t)\le 1, s(t)\le a(t) \tag{12}$$

$$P_i = \sum_{t=t_{i-1}}^{t_i} [s(t)+v(t)] - d \sum_{t=t_{i-1}+1}^{t_i} (x(t)+y(t)) \tag{13}$$

$$P_i \ge \frac{iC(0)}{n} - \sum_{j=1}^{i-1} P_j \tag{14}$$

$$\sum_{j=1}^{i} P_i \le C(0) \tag{15}$$

$$\forall t, x(t)\le s(t), x(t)\le 1-s(t-1), x(t)\ge s(t)-s(t-1) \tag{16}$$

$$\forall t, y(t)\le v(t), y(t)\le 1-v(t-1), y(t)\ge v(t)-v(t-1) \tag{17}$$

where the (13) ensures the total progress at the end of the slice is at least $cp(t_i)\ge iC(0)/n$, and the (15) avoids making more total progress than the job computation time.

A.4 Deadline for Changeover Delay Extension

In §5.6, we relaxed the model to account for the variations of the changeover delay. We now prove that with the extension, the deadline guaranteed by all policies should be $R(0)+2(\hat{d}-d)$.

Proof. With the assumptions, all variance of changeover delay will be directly reflected in $C(t)$.

We consider the last moment a job is idle before it finishes (at time t), there are three cases:

1. $R(t) > C(t) + 2d$: Safety Net Rule should never be triggered, causing the maximum time the job finishes to be $R(0)-R(t)+C(t)+\hat{d}<R(0)+\hat{d}-2d$.
2. $R(t)=C(t)+2d$: Safety Net Rule kicks in at $t+\varepsilon$, and the guaranteed deadline should be $R(0)-R(t)+C(t)+\hat{d}$, *i.e.*, $R(0)+\hat{d}-2d$.
3. $R(t) < C(t) + 2d$: It means the job was on a spot instance and got preempted. In this case, at the time t' the job jumped onto the spot instance, we have $R(t') \ge C(t') + 2d$. Thus, the worst case for the guaranteed deadline would be the job experience two maximum changeover delays, once for jumping onto a spot, and once for jumping onto an on-demand instance. That said, the bound for the finish time would be $R(0)-R(t')+C(t')+2\hat{d}\le R(0)+2(\hat{d}-d)$.

Combining the three cases, the bound for the deadline guaranteed should be $R(0)+2(\hat{d}-d)$ □

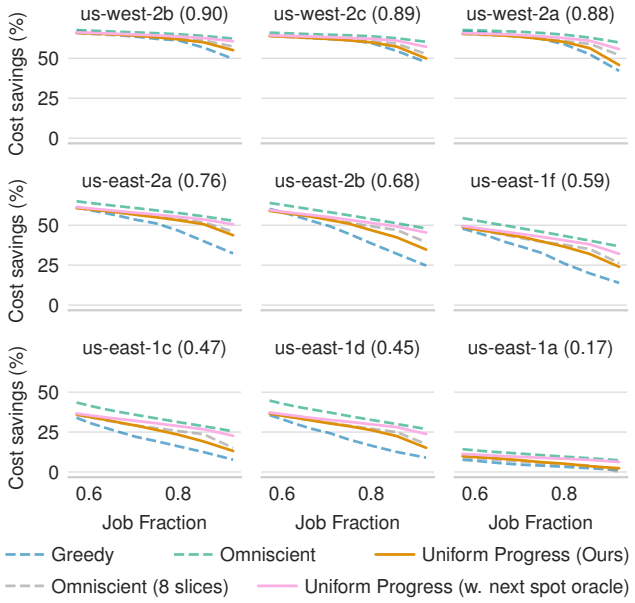


Figure 18: Cost Savings against on-demand instance for different policies on 2 months of spot availability traces.

A.5 Performance on 2-month Availability Traces

Various Deadlines. Figure 18 shows the cost savings of different policies we consider on the 2-month spot availability trace. Similar conclusions as §6.3 can be drawn from the figure that our Uniform Progress has a much lower gap to the Omniscient policy, with similar performance as the Partial Lookahead Omniscient policy in various cases, including loose and tight deadline as well as low and high spot fraction. The Next Spot Lifetime Oracle improves our Uniform Progress even further making the performance approach to the theoretical optimum.

Impact of Spot Fraction and Deadline. Figure 20 compares the cost saving difference of the policies against Omniscient policy on 2-month spot availability traces. Similar conclusions as in §6.4 can be drawn that Uniform Progress consistently outperforms greedy policy in all the scenarios, and applying

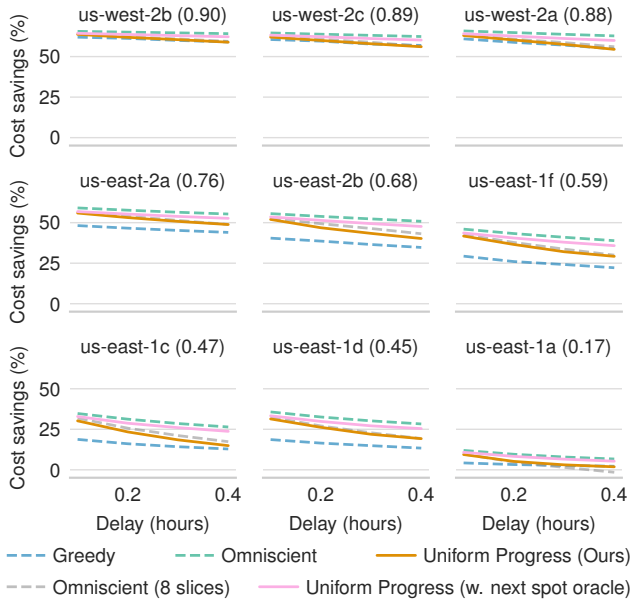


Figure 19: Cost savings against on-demand instances for different changeover delays.

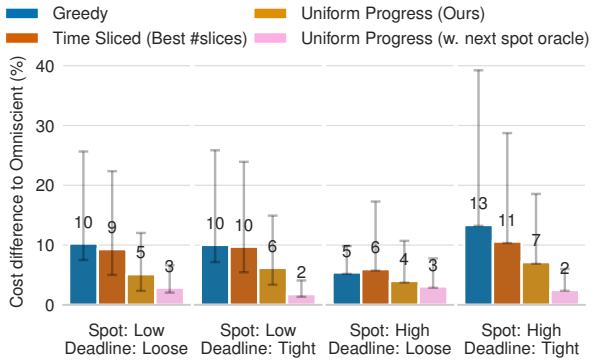


Figure 20: Cost difference compared to the Omniscient policy (normalized by on-demand cost, *lower is better*). It is on 2-month availability traces starting from 2/15/2023, aggregated on 300×9=2700 sampled traces.

Next Spot Lifetime Oracle further increases the cost savings.

Different Changeover Delays. We also evaluate the performance of the policies for different changeover delays on the 2-month availability trace. Figure 19 illustrates that our policy performs consistently better than the greedy policy for different changeover delays. Similarly as §6.5, the gap of Uniform Progress to the Omniscient policy increases while the Next Spot Lifetime Oracle helps it regain the advantage.

A.6 Setup of Real Workloads

We benchmark all policies on real workloads in §7.2. We describe detailed setups of the workloads in this section.

Machine Learning Training. We consider pre-training a RoBERTa [27] model on a subset of Wikipedia, WikiText-103, with a V100 GPU instance (p3.2xlarge) on AWS. We follow the configuration of FairSeq's reproduction [7, 34] to train the model for around 110 epochs (each takes about 40 minutes). To be fault-tolerant, we checkpoint the model weights twice per epoch to a cloud object store (AWS S3). The average progress loss and the time to reload the model into GPU are included in the changeover delay.

Bioinformatics. We run a bioinformatics workload of mapping DNA cells of sequencing data [26] on GCP. The workload has 90 independent tasks, each with a relatively short duration (15 minutes). Each task requires a powerful multi-core CPUs for parallelization. We use GCP's latest C3 generation of compute instance, c3-highcpu-88. In this workload, interrupted tasks need to be recomputed entirely after recovery. We use the average task duration as the changeover delay (see Table 4).

Data Analytics. We run Apache Spark [46] (v3.2.0) on a widely-used benchmark, TPC-DS [32]. We use scale factor 1000 to generate 300 GB of data on a 64-core CPU instance (r5.16xlarge). The data is stored on a persistent disk, which is attachable for future instances. We run all queries 10 times. Similar to the bioinformatics workload, each query needs to start over if interrupted. We add a weighted average of query runtimes (7 mins) as progress loss into the changeover delay.

A.7 Extending to Multiple Instances

We extend Omniscient policy to gang-scheduling jobs with multiple instances as mentioned in §5.5.

A.7.1 Omniscient policy

We extend Omniscient's ILP (6) to multiple instances.

A.7.2 Omniscient with homogeneous clusters.

We first extend the formula for the homogeneous cluster case, where all instances in a cluster with N instances should be the same type (all spot, all on-demand, or none). We revise the semantics of the original variables:

- $a(t)$: the number of spot instances available at time t.
- $s(t), v(t)$ indicate the policy chooses to use all spot or all on-demand for the cluster at time t.
- $x(t), y(t)$ represent the changeover delay that happens to the spot/on-demand cluster at time t.

The Omniscient policy with the same instance type can be represented as:

$$\min_{s(t),v(t)} \sum_{t=0}^{R(0)} N[s(t)+v(t)k] \tag{18}$$

$$\forall t, s(t)+v(t) \le 1, s(t) \le a(t)/N \tag{19}$$

$$\sum_{t=0}^{R(0)} [s(t)+v(t)] \ge d \sum_{t=1}^{R(0)} (x(t)+y(t)) + C(0) \tag{20}$$

$$\forall t, x(t) \le s(t), x(t) \le 1-s(t-1), x(t) \ge s(t)-s(t-1) \tag{21}$$

$$\forall t, y(t) \le v(t), y(t) \le 1-v(t-1), y(t) \ge v(t)-v(t-1) \tag{22}$$

A.7.3 Omniscient with heterogeneous clusters.

We further generalize the Omniscient policy to support heterogeneous clusters, allowing a mix of spot and on-demand instances in a cluster. This is the theoretical upper bound of the cost saving we can achieve under the problem setting in §5.5. We update the definition of variables as follows:

- $s(t), v(t)$: the number of spot and on-demand instances in the cluster at time t.

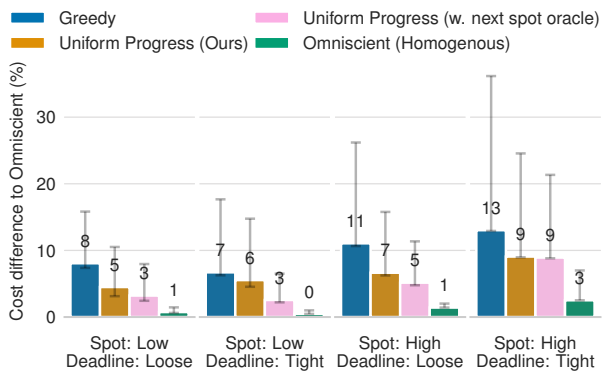


Figure 21: Cost savings for gang-scheduling jobs on 16-instance clusters, compared against theoretical upper bound (Omniscient policy allowing heterogeneous clusters). Uniform Progress consistently outperforms greedy policy.

- $p(t)$: whether the cluster is UP at time t.
- $z(t)$: whether changeover delay is triggered at time t.
- $m(t), n(t), j(t), k(t)$: intermediate binary variables.

The following is the Omniscient policy for multi-nodes with gang scheduling.

$$\min_{s(t),v(t)} \sum_{t=0}^{R(0)} [s(t)+v(t)k] \tag{23}$$

$$\forall t, s(t)+v(t)-N\cdot p(t)=0, s(t)\le a(t) \tag{24}$$

$$\sum_{t=0}^{R(0)} [s(t)+v(t)] \ge d \sum_{t=1}^{R(0)} z(t)+C(0) \tag{25}$$

$$\forall t, s(t)-s(t-1)\le N\cdot z(t) \tag{26}$$

$$\forall t, v(t)-v(t-1)\le N\cdot z(t) \tag{27}$$

$$\forall t, m(t)\le s(t)-s(t-1)+(N+1)\cdot j(t) \tag{28}$$

$$\forall t, m(t)\le s(t-1)-s(t)+(N+1)\cdot(1-j(t)) \tag{29}$$

$$\forall t, n(t)\le v(t)-v(t-1)+(N+1)\cdot k(t) \tag{30}$$

$$\forall t, n(t)\le v(t-1)-v(t)+(N+1)\cdot(1-k(t)) \tag{31}$$

$$\forall t, z(t)\le m(t)+n(t) \tag{32}$$

(26) and (27) set $z(t)=1$, when either the number of spot or on-demand increases in the cluster; (28) to (32) enforces $z(t)=0$ when $s(t)=s(t-1)\wedge v(t)=v(t-1)$, *i.e.,* the number of spot or on-demand used by the job does not change. That said, (26) to (32) make sure $z(t)=1$ iff changeover delay happens at time t.

A.7.4 Cost Savings on 16 Instances

We scale up the experiments to 16 instances using real spot preemption traces. In Figure 21, we can observe that the homogeneous cluster constraint only has negligible influence on the Omniscient policy on 16-instance clusters, which is also because the spot market is efficient enough, similar availability of all the 16 instances in a cluster. Our Uniform Progress still outperforms greedy policy on all 4 different scenarios with a smaller gap to the best cost savings a system can achieve.

A.8 Loose Deadline

In this paper, we mainly discuss policy design for jobs with relatively tight deadlines, as very loose deadlines will likely lead to jobs able to finish on spot instances only. As mentioned in §6.1, when a loose deadline is given, a job can utilize spot instances whenever available until timestamp t_0, when the remaining time to deadline $R(t_0)$ becomes tight compared to the job progress $C(t_0)$, and apply the policy. We conduct experiments for loose deadlines, by setting the switch point at $\frac{C(t_0)}{R(t_0)}=0.7$.

In Figure 22, we can see that greedy policy gets close to the upper bound of the cost savings for very loose deadlines, as jobs are likely to be able to finish on spot instances only. It is worth noticing that job fraction 0.25 represents the deadline

Figure 22: Cost savings for policies with very loose deadlines.

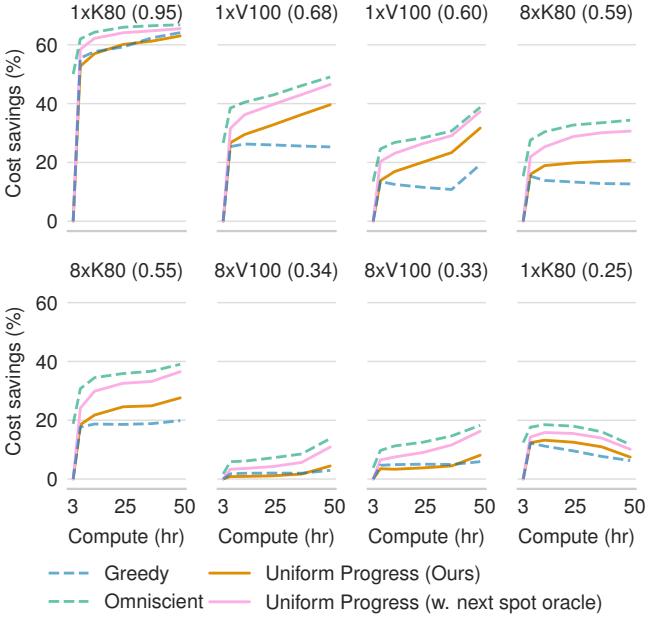


Figure 23: Cost savings with various job computation time. Job fraction (computation time/deadline) is set to 85%.

is 4x longer than the job duration. By allowing jobs to utilize as many spot instances as possible when the remaining time is abundant, all policies perform similarly in cases where loose deadlines are given.

A.9 Various Job Computation Time

We compare the cost savings for the policies across different job computation times with the same job fraction 85% in Figure 23. When job computation time is very small (comparable to the changeover delay), all policies' cost savings drop quickly, as switching between spot and on-demand instances is not worth the cost caused by the changeover delay. However, when the job computation time increases, there is more optimization opportunity for the policies, as more changeover delay can be tolerated, leading to a larger gap between Uniform Progress and greedy policy.

Towards Intelligent Automobile Cockpit via A New Container Architecture*

Lin Jiang†
Xi'an Yunzhiji Technology, China
pppaass@163.com

Feiyu Zhang
Xi'an Yunzhiji Technology, China
1123022283@qq.com

Jiang Ming†
Tulane University, USA
jming@tulane.edu

Abstract

An intelligent cockpit is now crucial in automobiles, not just to provide digital instrumentation and in-vehicle controls but also to offer a wide range of entertainment functionalities. To cater to the demands of these intelligent vehicles, the automotive industry starts employing virtualization technology to offer a unified hardware and software architecture that can simplify system management and enhance resource utilization. Particularly in the domain of intelligent cockpits, virtualization can tightly integrate systems with different criticality levels (e.g., safety and real-time) on a single hardware platform, improving inter-system communication quality and the timely response to user-initiated requests. Currently, microhypervisor virtualization has been used in production to achieve intelligent automobile cockpit. However, in addition to the performance concern and high production costs, this solution is suffering from the global shortage of chips capable of running microhypervisor systems.

Our key insight is that, most functions within intelligent cockpit systems are non-safety-critical and non-real-time multimedia tasks. Based on this characteristic, in this paper we present *AutoVP*, a new cockpit virtualization architecture. The hardware foundation of AutoVP consists of two low-cost chips: 1) a consumer-grade System-on-Chip (SoC) multi-core processor as the main chip; 2) a typical automotive-grade Microcontroller Unit (MCU) as the auxiliary chip. The MCU auxiliary chip is responsible for hosting real-time and safety-critical tasks, while the SoC main chip primarily handles multimedia tasks, such as entertainment systems and digital instrumentation. Further more, we construct an Android container virtual environment on the SoC main chip. This environment integrates multiple media functions onto a single chip, resulting in efficient utilization of chip computational resources and high system scalability. Our comparative performance evaluation demonstrates that AutoVP is a cost-effective and efficient solution to build intelligent cockpits.

*Operational Systems Track
†Corresponding authors.

1 Introduction

The automotive electrical and electronic systems comprise hundreds of sensors, actuators, and Electronic Control Units (ECUs). These units are responsible for running various subsystems, such as instrumentation, entertainment, and advanced driver assistance systems. They collaborate while remaining relatively independent, ensuring the highest level of performance, safety, and functionality. Today's automobiles are approaching the limits of their complexity. In the future, the automotive industry will provide always-connected vehicles, operating advanced autonomous driving functions and an increasing array of cutting-edge applications. In this environment, both the hardware and software components within vehicles are experiencing exponential growth in line with the increasing number of applications, leading to an explosion in the complexity of vehicle architectures [1]. Furthermore, the proliferation of connectivity and applications also results in a larger attack surface [2]. This distributed computing architecture is making automotive electrical and electronic systems increasingly bulky, with challenges emerging in wiring, thermal management, and power distribution [3].

A promising approach to address the aforementioned problem involves the adoption of a virtualization solution. Virtualization primarily serves the purpose of integrating multiple business subsystems that originally operated on different ECUs [4, 5], thus reducing software and hardware costs, and shortening product time-to-market. At the beginning of the 21st century, this technology was first adopted as a centralized software technique for avionics systems [6]. Nowadays, several commercial virtualization solutions in the market have achieved significant success in safety-critical applications, including aerospace, national defense, and healthcare. There is currently a drive to promote virtualization as the preferred integrated solution within the automotive electrical and electronic architecture [1].

In the automotive landscape, vehicles can be categorized into various functional domains, including the Powertrain domain, Chassis domain, Body/Comfort domain, Cockpit/In-

fotainment domain, and Autonomous Driving domain [7]. The Cockpit/Infotainment domain encompasses both safety-critical tasks such as instrumentation and driver assistance, as well as non-critical multimedia entertainment functions like central control and heads-up displays. These subsystems operate relatively independently while collaborating, interacting directly with users to provide intelligent experiences [8, 9]. By employing a centralized virtualization solution to integrate multiple subsystems within the Cockpit/Infotainment domain, it can reduce costs, enhance collaborative processing efficiency, and optimize user experience.

Currently, commercial intelligent cockpit virtualization products are dominated by the hypervisor-based solution [10, 11], such as QNX Hypervisor [12], INTEGRITY Multivisor [13], and PikeOS Hypervisor [14]. They enable the operation of multiple virtual systems on a single set of hardware (e.g., ECU), with each virtual system running in a relatively independent virtual environment, hosting different functions.

These vehicle hypervisor solutions typically adopt a microkernel architecture [15, 16]. However, compared with user-attractive Android applications, microkernel architecture has a notable drawback, namely the absence of a rich software ecosystem [17–21]. For instance, if the cockpit needs to run advanced driver assistance systems (ADAS) functions (e.g., automated parking), they require support for deep learning, computer vision, video encoding/decoding, and 3D graphics, necessitating substantial resource investment for development. Furthermore, existing hypervisor solutions must handle both safety-critical tasks and complex multimedia operations, leading to specific hardware requirements. Consumer-grade SoC chips or even standard automotive-grade MCU chips fail to meet these demands. Automotive-grade SoC chips are capable of accommodating such intricate software systems [22], but their development entails significant challenges and high production costs. Even worse, the auto industry is facing a global shortage of automotive-grade SoC chips [23–25].

We have three observations motivating a new cockpit virtualization design: 1) only a minority of functions within an intelligent cockpit are safety-critical (e.g., instrumentation and driver assistance), while the majority of modules classified as non-critical operations (e.g., all multimedia entertainment tasks); 2) a typical automotive-grade MCU chip can host these real-time and safety-critical functions, while a cheap consumer-grade SoC chip (e.g., smartphone chips) can handle the remaining non-critical functions; 3) these two kinds of chips are affordable, and they have not been significantly impacted by recent supply chain shortages. As a result, this paper presents a cost-effective virtualization architecture for constructing an intelligent cockpit, called *AutoVP*.

As shown in Figure 1, AutoVP employs Android container technology [26, 27] to integrate non-safety-critical subsystems, such as instrument display, central control system, and passenger entertainment system, into a single entity, which is deployed on a cheap consumer-grade SoC chip. Container

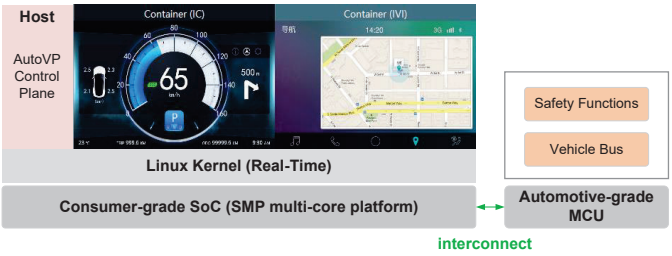


Figure 1: AutoVP's mixed-criticality decoupled design.

is a lightweight virtualization technology [28–30] that can efficiently leverages the computational power of the SoC chip while ensuring the independent operation of each business subsystem, thereby enhancing service quality. Safety-critical tasks such as safety monitoring and vehicle bus, are handled by an automotive-grade MCU chip. Interconnection between the SoC chip and MCU chip is facilitated through inter-chip communication technologies, forming a complete intelligent cockpit system. Please note that AutoVP's mixed-criticality decoupled design also complies with automotive functional safety requirements, as outlined in ISO 26262 [31].

We developed a new Android container framework on top of Cells [26, 27]. This container imposes no specific hardware requirements and incurs very small virtualization performance overhead, making it suitable for widespread deployment across various low-power SoC chips. Besides, in the automotive industry, the deployment of safety-critical tasks using automotive-grade MCU chips is a well-established and mature method [32]. In addition to container-based virtualization, AutoVP's differences in design also involve monitoring mechanisms, implementing corrective measures for abnormal behaviors, and ultimately isolating the safety-critical tasks from the business functions. Hence, deploying the AutoVP-powered cockpit system within intelligent vehicles is relatively straightforward.

We compared AutoVP with a commercial hypervisor product on the same automotive-grade platform. Our extensive performance experiments demonstrate that AutoVP incurs significantly lower performance overhead. When examining various aspects such as CPU overhead, memory usage, power consumption, startup time, peripheral performance, and frame rate, AutoVP's container outperforms the hypervisor solution.

In a nutshell, we make the following key contributions:

- We propose a mixed-criticality decoupled architecture to address the need for centralized deployment of various business subsystems within the intelligent cockpit domain, all while offering ease of construction, low production costs, and small performance overhead.
- We present a new Android container framework tailed for intelligent cockpits. Our work represents the latest progress in mobile container-based virtualization, and it is an ideal solution to host the non-safety-critical subsystems that require display screens for user interaction.

- Our evaluation and real-world deployment demonstrate that AutoVP is a viable in-vehicle virtualization alternative to mitigate the ongoing automotive chip crisis.

Real-world Deployment AutoVP has been deployed in two flagship electric vehicle models under a leading automotive manufacturer.* The installation volume in the past year has exceeded one million units.

Open Source We have released a prototype of AutoVP's container to facilitate reproduction and reuse, as all found at https://github.com/jianglin-code/AutoVP.

2 Background and Related Work

In this section, we first provide background information on the evolution of in-car virtualization. We also review existing approaches for intelligent cockpit virtualization and identify their limitations, which have prompted our work. Next, we underline the need of using the Linux container technology for in-car virtualization. Finally, we introduce the specific Android container framework that we leverage to implement AutoVP's container.

2.1 The Development of In-Car Virtualization

With the emergence of multi-core embedded System-on-Chip (SoC) devices, the integration of multiple applications with varying levels of criticality on a single platform has become increasingly popular. This platform is referred to as a mixed-criticality system, which needs to meet various requirements, including real-time constraints, operating system (OS) scheduling, and memory/OS isolation [33].

The cockpit system of intelligent vehicles represents a typical mixed-criticality system. It comprises non-critical subsystems such as entertainment, networking, and voice systems, as well as safety-critical subsystems like the vehicle bus and advanced driver assistance systems (ADAS). The non-critical subsystems often involve multimedia subsystems that rely on hardware modules for tasks such as display, encoding/decoding, networking, AI, and voice processing. Therefore, these subsystems are typically deployed on SoC chips to provide users with a seamless intelligent experience. On the other hand, safety-critical subsystems must adhere to functional safety requirements to ensure error-free operation [31]. They are usually deployed on automotive-grade MCU chips, offering users a stable safety and automated experience.

A key challenge in the design of mixed-criticality systems is the isolation of software applications with varying degrees of criticality on a common hardware platform. In the automotive domain, a common practice for isolating safety-critical applications is through the proliferation of multiple hardware Electronic Control Units (ECUs). These control units are dedicated to various tasks, including basic operations like intelligent infotainment, as well as critical functions such as automated parking and adaptive cruise control. However, this approach is highly inefficient, as many of the resources within these ECUs often remain underutilized. However, with the evolution of multi-core architectures and the introduction of new hardware extensions such as virtualization, securely executing multiple applications on the same platform while reducing costs and vehicle weight to enhance resource utilization has become feasible [33].

*Until the publication of this paper, we have not been granted authorization to disclose the name of the automotive manufacturer.

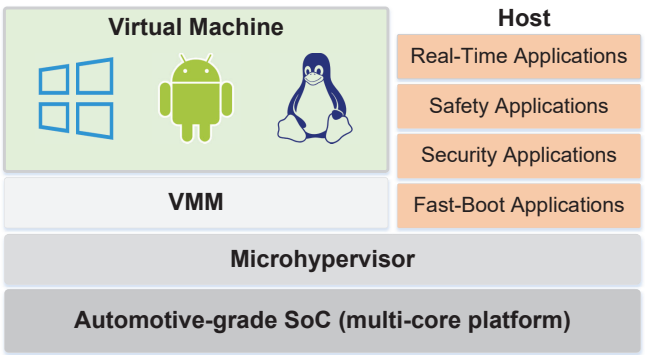


Figure 2: In-vehicle microhypervisor architecture.

Virtualization is considered a solution for isolating operating systems within virtual machines, and its advantages include cost reduction through the abstraction of the host platform [10]. Additionally, features provided by a hypervisor, such as the abstraction of memory, CPU, and interrupts, aid in the isolation of operating systems. This approach, as exemplified by commercial products like QNX Hypervisor [12], INTEGRITY Multivisor [13], and PikeOS Hypervisor [14], can be applied to create mixed-criticality systems. A common industry practice is to employ a hypervisor system with a microkernel architecture to ensure real-time capabilities and functional safety [15, 19]. We will further discuss the pros and cons of this approach in §2.2 and §2.3.

Android Automotive [34] is a new vehicle OS based on Android, designed to run in-vehicle infotainment systems and pre-installed Android applications. However, Android Automotive is not a virtualization solution and thus lacks a mechanism for system isolation. If several different types of services are running on the same Android Automotive system at the same time, sharing system resources, such as network, storage, multimedia, etc., these services will interfere with each other, affecting the user experience and in-vehicle safety.

2.2 Microkernel + Hypervisor

Embedded systems in vehicles often adopt microkernel-based operating systems due to their inherent real-time and security features. Such microkernel OSs are responsible for hosting safety-critical tasks; besides, they employ hypervisor techniques to run virtual OSs that have a richer software ecosystem, addressing the limitation of their own software ecosys-

Table 1: Comparison of three in-vehicle chips. The unit prices displayed represent their median ranges.

Chips	Production Cost	Unit Price ($)	Supply Chain Shortages?
Automotive-grade SoC	High	300∼500	Yes
Consumer-grade SoC	Low	70∼150	No
Automotive-grade MCU	Minimum	5∼10	No

tem [12–14]. Running this type of software-configured kernel is referred to as a "microhypervisor," which combines both microkernel and hypervisor functionalities [35].

The microhypervisor is responsible for providing resource containers, execution contexts, scheduling, inter-process communication, and synchronization mechanisms for its user mode. As shown in Figure 2, the user mode of the microhypervisor can directly run applications, making it suitable for executing real-time tasks, fast boot tasks, and functional safety tasks, among others. Furthermore, the user mode of the microhypervisor can also run virtual machines through a virtual machine manager, making it suitable for running OSs with a rich software ecosystem, such as virtual Linux systems.

From a functional safety perspective, because virtual OSs have a vast software ecosystem that can support complex hardware for business tasks, while the microhypervisor system can directly host safety-critical tasks, this approach is a feasible design for in-vehicle virtualization. The hypervisor host domain serves as the guardian of the virtual machine domain, similar to the role of an independent MCU.

2.3 Limitations of In-Vehicle Microhypervisor

However, in-vehicle microhypervisor solutions present two significant challenges. One is the requirement for a specific high-end hardware base, and the other is the performance of inter-process communication (IPC), which can become a bottleneck for the entire software-hardware system.

The microkernel can provide verified real-time capabilities and a minimal trusted computing base for safety-sensitive applications, thus requiring a hardware platform with safety and reliability. Furthermore, the microkernel can run complex multimedia and intelligent AI tasks by executing virtual machines. In this scenario, it demands a hardware foundation with substantial computational power, abundant peripheral modules such as display units, GPU modules, AI components, and network modules. Consumer-grade SoC chips lack safety and reliability assurances, while automotive-grade MCU chips often cannot meet the computational and peripheral requirements of complex multimedia tasks. Typically, automotive-grade SoC chips are needed to simultaneously run safety-critical and non-critical tasks. However, as shown in Table 1, the production costs of automotive-grade SoC chips are exceptionally high, and they are currently being adversely affected by global supply chain shortages [23–25].

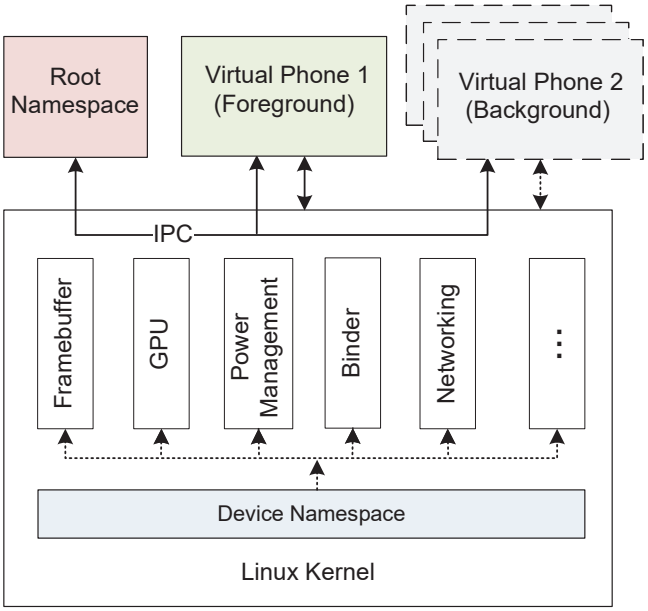


Figure 3: The overview of Cells [26, 27]. Only the virtual phone running in the foreground is displayed at any time.

Furthermore, within microkernels, any communication between different user processes is based on IPC, and this is an operation-intensive process. For instance, if a client process writes data to an external block device, it first communicates with the file system, which then notifies the disk device driver to write data to the block device. All communication is carried out through IPC. Therefore, IPC performance is a crucial technical metric for microkernels [36].

2.4 Linux Container for In-Car Virtualization

One way to address the challenges discussed in §2.3 is to combine a consumer-grade SoC chip with an automotive-grade MCU chip. High-computational tasks like deep learning, computer vision, video encoding/decoding, and 3D graphics run on the ordinary SoC chip, while critical tasks run on the MCU chip. This approach not only alleviates the shortages of automotive-grade SoC chips but also reduces costs. Furthermore, this combined chip structure does not require the use of a complicated microhypervisor. Instead, the Linux container technology, together with an independent MCU solution, can better meet the "one-core, multiple-screen" requirements of intelligent cockpits [37]. The entire solution involves isolating non-critical subsystems using the container technology to ensure software service quality. Simultaneously, it employs a dedicated MCU chip to host safety-critical tasks.

2.5 Android Container Framework

The Linux container technique used by AutoVP is derived from the Cells project [26, 27], a lightweight Android virtualization framework. It enables multiple containerized Android

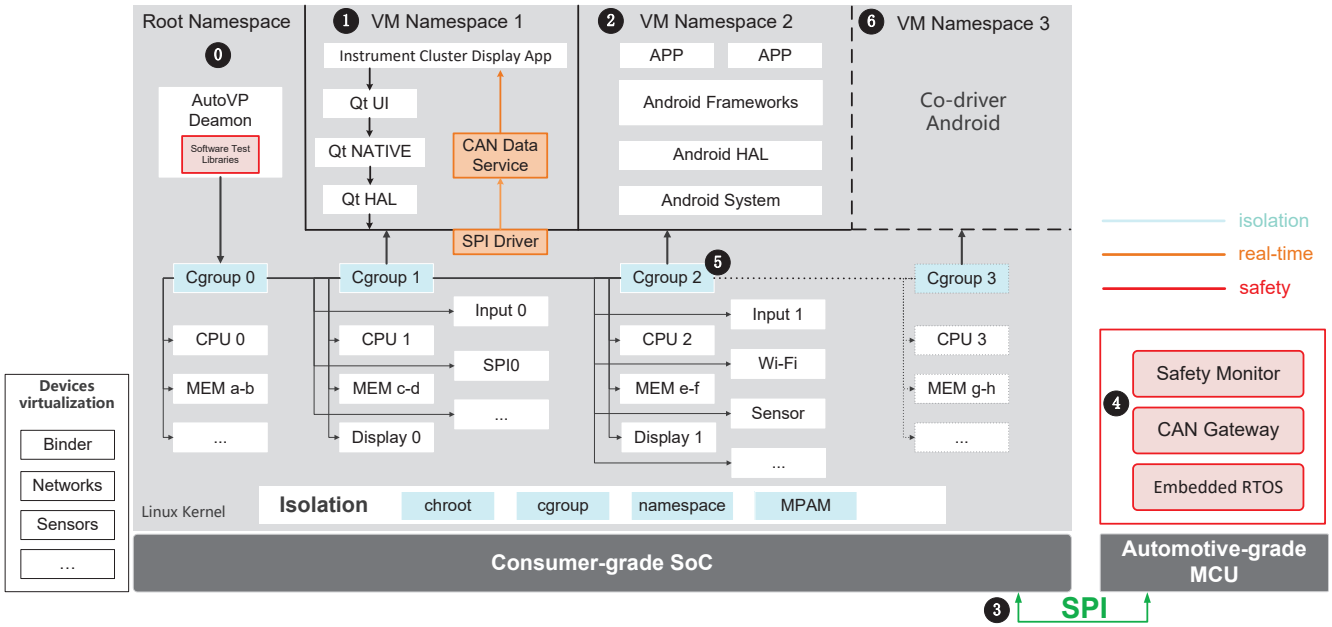


Figure 4: Overview of AutoVP's architecture. AutoVP supports multiple Android containers. For example, both driver and co-driver can have their own in-vehicle infotainment systems running on isolated Android containers.

instances to run simultaneously on the same mobile device in an isolated manner. As shown in Figure 3, Cells introduces the concept of foreground and background container systems, where only one container system is displayed in the foreground while others operate in the background. Cells implements a new device namespace mechanism and user-level proxy method, which, in conjunction with Linux namespaces, allows for the multiplexing of hardware resources across multiple containerized Android instances while providing nearly lossless performance.

However, the foreground-background design used by Cells is not applicable to the automotive application scenario, in which, each virtual system needs to directly provide services to users simultaneously; in other words, all virtual systems must be running in the foreground. Additionally, Cells primarily focuses on how to share hardware resources among multiple virtual Android systems through software virtualization methods. In contrast, in the automotive application scenario, the emphasis is on isolating and decoupling multiple virtual systems and reducing interference between them.

Therefore, the virtualization methods of Cells, such as user-level proxy and mutual-exclusion use of hardware modules, may not be applicable in automotive cockpits. Furthermore, in Cells, all virtual phones share hardware resources such as CPU and memory. When multiple virtual phones run simultaneously, they contend for these resources, leading to mutual interference. In automotive systems, hardware resources also need to be forcibly isolated.

3 AutoVP Overview

Figure 4 illustrates the architecture of AutoVP, including the mixed-criticality decoupled design using two low-cost chips. The consumer-grade SoC chip runs a root namespace and two container spaces. The root namespace runs AutoVP's control plane (❶ in Figure 4), which has a small software stack primarily responsible for managing the startup, shutdown, resource allocation, and isolation functions of container systems. The other two container spaces are dedicated to instrument cluster (IC) display and in-vehicle infotainment (IVI) functions, respectively. These container systems have undergone software-based resource isolation for CPU, memory, and peripheral resources. Additionally, the separate MCU chip hosts safety-critical operations, such as safety monitoring and Controller Area Network (CAN bus). Interconnection between the SoC chip and MCU chip is facilitated through inter-chip communication technologies such as Serial Peripheral Interface (SPI). AutoVP employs software virtualization techniques for various devices such as binder, WiFi network, and sensors, allowing multiple virtual systems to simultaneously utilize these devices.

The container hosting the instrument cluster display (❶ in Figure 4) consists of the Qt framework abstraction layer [38], Qt native API, Qt UI, CAN message parsing service, and instrument display app. The vehicle data required for the digital instrumentation comes from the vehicle standard communication service running on the MCU chip. The instrument display app on the SoC chip exchanges data with the MCU chip through the SPI interface (❸ in Figure 4). This con-

tainer needs to occupy GPU and display hardware resources to visualize instrument graphical interfaces.

The second container (❷ in Figure 4) runs a complete Android system. It is responsible for hosting complex in-vehicle infotainment functions such as high-definition navigation map, voice broadcasting, Bluetooth headset, and WiFi network services. It also provides smooth interactive functions to respond promptly to user command operations. This container occupies most of the SoC chip's hardware resources, such as GPU, display, WiFi, camera, Bluetooth, and USB.

Both the two containers are configured with independent touchscreens for user operations. The display and input subsystems of the two containers are completely independent. Please note that AutoVP supports multiple Android containers. For example, two isolated Android containers (❷ vs. ❻ in Figure 4) can host separate in-vehicle infotainment systems for the driver and co-driver, respectively.

For safety-critical functions, AutoVP utilizes a hardware-based isolation by employing an external MCU chip to host them (❹ in Figure 4). This MCU chip is responsible for running safety-critical functions such as automotive regulation communication on an embedded real-time OS, serving as a guardian system to the main SoC chip.

4 In-Vehicle Container Implementation

The development of an in-vehicle container presents a plethora of challenges, ranging from intricate hardware resource multiplexing to the need for a fine-grained isolation mechanism, and to interactions with safety-critical tasks. In this regard, AutoVP has made significant advancements over the Cells solution [26, 27][†] to fulfill the requirements of an intelligent cockpit. Unlike Cells, AutoVP does not rely on the foreground-background container system design. Instead, all container systems operate in the foreground. Therefore, our approach's crux is to ensure that these virtual systems operate independently without interfering with each other.

Compared to Cells, AutoVP's container mainly differs in three perspectives: device virtualization methods, isolation, and monitoring mechanisms. AutoVP's monitoring mechanism allows for the early detection of potential failures in complex systems running on the SoC chip. We will introduce them in the follow-up subsections.

4.1 Device Virtualization Methods

AutoVP's device virtualization involves a significant workload, encompassing various board-level and peripheral devices. In particular, the device virtualization methods can be summarized as follows.

[†]Cells's virtualization methods to many hardware devices (e.g., filesystem, network, display, and power) have been obsolete since Android 6.0.

Table 2: The list of virtualized devices and services. Virtualization method ID: (1) multiple identical devices; (2) kernel-level device virtualization; (3) user-level device virtualization.

Virtualization Method ID	Virtualized Devices and Services
(1)	Display, Input, Audio, Bluetooth
(2)	Binder, Power Management, Network Sensors, GPS, SELinux
(3)	WiFi, Adb

1. Multiple Identical Devices: In this approach, the hardware base supports multiple instances of the same type of device, each serving different virtual systems. For instance, in an automotive cockpit, there may be multiple touchscreen displays. These displays can be allocated to different virtual systems by modifying the system configuration.

2. Kernel-level Device Virtualization: In cases where the hardware base is equipped with a single device, such as power management, we made modifications to the kernel power driver to provide data isolation and multiplexing capabilities for that hardware module, allowing the hardware module to respond to power control requests from multiple virtual systems simultaneously.

3. User-level Device Virtualization: In scenarios where virtualization at the kernel level for complex devices is challenging, like the WiFi module, modifications are made to the user-level WiFi service process. Then, we implement a system-level IPC mechanism to enable multiple virtual systems to share the WiFi functionality provided by the customized WiFi service process.

AutoVP employs the above methods to conduct specific device virtualization work, resulting in the list as shown in Table 2. Simultaneously, in order to meet the requirements of smart cockpit applications, adjustments have been made to the virtualization methods of certain modules.

One-Core, Multiple-Screen To fulfill the demand for a "one-core, multiple-screen" smart cockpit, the SoC chip supports multiple display interfaces, such as Camera Serial Interface and DisplayPort, allowing the system to connect to multiple independent touchscreens. We configure separate touchscreens for both the IC display container (❶ in Figure 4) and the IVI container (❷ in Figure 4) to accommodate user interactions. The display and input subsystems of these two containers are entirely independent.

Inter-system Communication Within the automotive cockpit, multiple touchscreens are positioned in close proximity. This configuration necessitates frequent collaboration among these screens, such as sharing map services from the driver's in-vehicle infotainment (IVI) system to the instrument display screen or sharing a video between the driver's IVI and the

co-driver's IVI systems. Therefore, the inter-system communication mechanism becomes pivotal.

AutoVP achieves this by virtualizing the *binder* IPC mechanism [39], ensuring that each virtual machine (VM) has its own virtual binder node. The binder driver is a pseudo device in the kernel and does not correspond to any actual hardware. We modify the binder driver so that each VM has its own independent set of data structures within the binder driver; these binder data structures for different VMs do not interfere with each other. Furthermore, we introduce a routing and forwarding mechanism, adding "bridges" between virtual binder nodes, making it easier for virtual machines to access each other's functional services. The virtual binder routing and forwarding mechanism effectively meets the requirements for inter-VM screen casting and data transmission. For instance, AutoVP provides compositor image stacking services in the IC display container, allowing the IVI container to access the compositor service to project navigation maps onto the instrument cluster display screen, thereby enhancing the user experience.

4.2 Isolation Mechanisms

As an intelligent cockpit system, ensuring an isolation among different functions is imperative to prevent interference between them. AutoVP achieves the isolation of various business subsystems through several mechanisms within the host system. First, it employs the *chroot* mechanism to isolate the file system environment, thereby separating the software stacks of each business subsystem. Second, it utilizes the *cgroups* mechanism to allocate system hardware resources such as CPU, memory, and peripherals, ensuring that high-priority tasks have independent resources to execute critical functions. Third, AutoVP employs the *namespace* mechanism to segregate the kernel resources of different systems, effectively hiding them from each other, thereby reducing inter-system coupling and ensuring minimal interference among the various business subsystems. Additionally, it leverages the new MPAM (Memory System Resource Partitioning and Monitoring) feature [40] to dynamically isolate resources like cache and memory bandwidth, thus mitigating performance interference between different workloads at the hardware level and ensuring stable performance for high-priority tasks.

Cgroups AutoVP employs the *cgroup* mechanism [41] to isolate the hardware resources utilized by each business subsystem, thus reducing coupling and interference between these subsystems (❺ in Figure 4). For instance, the *cpuset* subsystem is used to allocate independent CPU cores and memory nodes to task groups; the *memory* subsystem is employed to allocate memory usage to task groups, and the *devices* subsystem is utilized to allocate peripheral resources to task groups. AutoVP, taking into account the characteristics of each business subsystem, engages in precise management of the hardware resources consumed by these subsystems. This

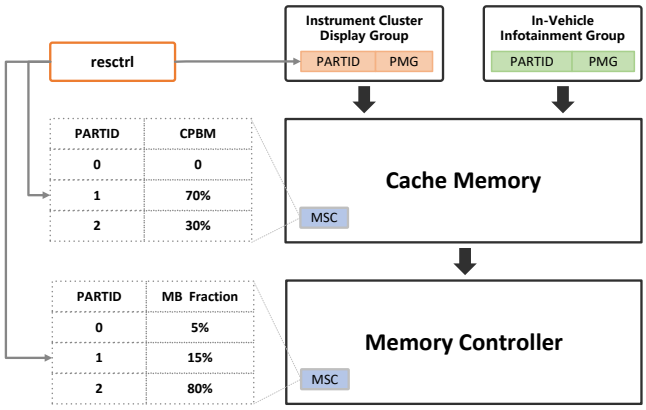


Figure 5: MPAM utilizes PARTID (Propagation of a Partition ID) and PMG (Performance Monitoring Group) to label business groups, and it employs MSC (Memory-System Component) configuration list for various hardware modules.

Table 3: Cache and memory bandwidth allocation strategies for various business subsystems.

Business Subsystem	Cache	Memory Bandwidth
Instrument Cluster Display (❶ in Figure 4)	70%	15%
In-Vehicle Infotainment (❷ in Figure 4)	30%	80%
Root Namespace (❶ in Figure 4)	0	5%

prevents resource contention among the business subsystems during their operation, thereby avoiding mutual interference. For example, in certain scenarios where the IVI container (❷ in Figure 4) plays multiple videos, consuming a significant amount of memory resources, this might lead to memory starvation in the IC display functions running in ❶ of Figure 4. By implementing resource isolation through the cgroup mechanism, AutoVP effectively ensures the long-term stability and robust operation of each business subsystem.

Namespace AutoVP utilizes the *namespace* mechanism [42] to isolate the kernel resources of each system, such as processes, networking, file systems, and driver data. This separation allows multiple business subsystems to hide from each other, reducing coupling between subsystems and striving to ensure the independent operation of each business subsystem without mutual interference. For instance, it becomes possible to halt a specific business subsystem without affecting the normal operation of other business subsystems.

MPAM MPAM (Memory System Resource Partitioning and Monitoring) is a new feature introduced in ARM v8 [43], enabling the allocation and monitoring of resources such as cache, memory bandwidth, and SMMU (System Memory Management Unit). This feature, operating at the hardware

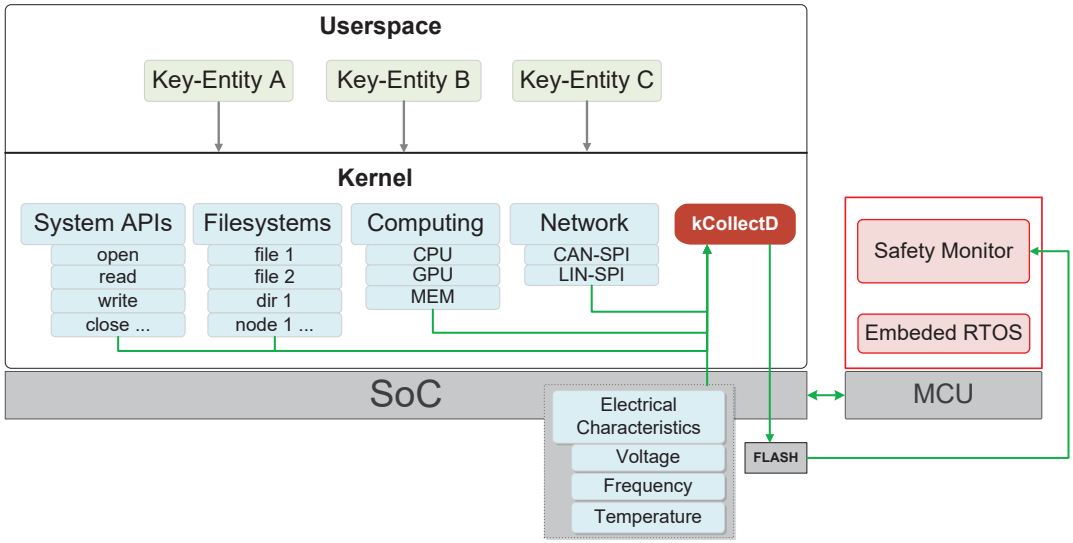


Figure 6: The kernel kCollectD module periodically collects data related to critical entities' interactions with kernel resources and hardware characteristics. This data is then stored in an external flash memory. The Safety Monitor module running on the MCU chip analyzes the data stored in the flash memory and provides a determination or assessment.

level, reduces the interference between different workloads, ensuring the stability of high-priority task performance.

AutoVP leverages the hardware feature of MPAM to dynamically isolate resources at runtime for various business subsystems. For instance, cache and memory bandwidth for the IC display subsystem (❶ in Figure 4) and the IVI subsystem (❷ in Figure 4) are isolated using MPAM. As shown in Figure 5, different configurations of cache and memory bandwidth access policies are tagged with PARTID (Propagation of a Partition ID). These PARTIDs are then bound to the IC display subsystem and the IVI subsystem, respectively. Consequently, every time cache and memory resources are accessed, the resource queries the resource usage policy of the subsystem associated with the bound PARTID. This control of resource utilization boundaries reduces resource conflicts and competition when the IC display subsystem and the IVI subsystem access cache and memory resources.

We empirically configure cache and memory bandwidth allocation strategies for business subsystems based on their specific characteristics, as outlined in Table 3. The IC display subsystem, although much smaller compared to the IVI subsystem, demands higher real-time performance. Therefore, it is allocated a greater share of cache space and a smaller portion of memory bandwidth. This resource allocation strategy allows the instrument cluster subsystem to dominate more than half of the cache resources, significantly enhancing cache hit rates. Besides, it is also assigned 15% of the memory bandwidth, reducing interference from the complex central control Android container. By employing this configuration strategy to elevate the priority of dynamic resource utilization for critical instrument cluster tasks, we are able to reduce IC display function latency and improve stability.

4.3 Monitoring Mechanisms

In Figure 4, the subsystems built on top of the main SoC chip primarily host non-safety-critical tasks. However, these subsystems include certain critical functions that have to interact with safety-critical data. For example, the IC display subsystem is responsible for displaying real-time vehicle information such as speed, fuel consumption, warning indicators, and other vehicle status data while the vehicle is in operation. These pieces of information are particularly crucial for the driver during the driving process. As a result, AutoVP needs to implement mechanisms for real-time monitoring of such functions. These mechanisms are designed to determine whether these critical functions are operating according to the prescribed procedures. In the event of any irregularities or malfunctions detected in these critical functions, the system must promptly alert the driver and employ redundancy mechanisms to rectify the abnormal functions.

kCollectD Kernel Module AutoVP establishes a monitoring module within the system kernel. As shown in Figure 6, this module, called *kCollectD*, periodically collects behavioral data from critical entities within the system, analyzing and assessing whether these entities are operating in accordance with predefined procedures. The critical entities under scrutiny include processes and kernel modules. The behavioral data encompass various aspects such as which system interfaces were invoked, which files were manipulated, the utilization of heap and stack memory, the CPU time slices consumed, and the number of frames rendered, among others. Given that collecting behavioral data from these entities can incur performance overhead, it is imperative to target the acquisition of key data specific to the relevant business processes and analyze essential behavioral characteristics. For example, in

the case of the instrument cluster display subsystem where the displayed content is crucial for the driver, it is essential to monitor the number of frames of graphics rendered by the IC display subsystem per second. If, during the periodic analysis phase, it is determined that a critical entity has deviated from the prescribed actions, an audit alarm is triggered immediately. This leads to the termination of the relevant entity's operations, followed by the initiation of redundancy mechanisms to rectify the error. For instance, if it is observed over a period of time that the number of frames rendered per second by the IC display subsystem is lower than expected, indicating a freeze in the display program, the system will sound an alarm to alert the driver and activate an emergency instrument cluster program to take over the display screen.

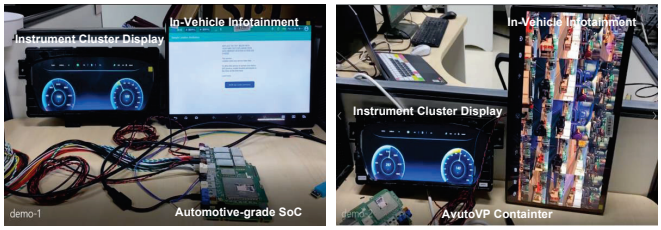
Monitoring Hardware Data Simultaneously, AutoVP's monitoring module, kCollectD, also collects hardware information from the SoC chip in real-time, such as power, clock, reset, and temperature. Both collected kernel resources (e.g., APIs, files, and networks.) and hardware characteristics (e.g., voltage, frequency and temperature) are then stored in an external flash memory. The Safety Monitor module, which runs on the MCU chip, analyzes the data stored in the flash memory and assesses whether the critical functions running on the SoC are operating normally. For example, if the Safety Monitor module detects an excessively high CPU voltage, changes in clock frequency, or excessive temperature on the SoC chip, it indicates that the software running on the SoC may not be functioning correctly or could potentially result in abnormal behavior. In such cases, the Safety Monitor triggers an alarm sound, notifying the driver of the system's abnormal operation and the need for manual intervention.

5 Evaluation

Our experiments focus on measuring performance metrics from seven aspects: startup time, memory usage, battery consumption, CPU, GPU, network performance, and real-world workloads. We compare AutoVP with a microhypervisor-based in-vehicle virtualization product. Although they are designed to run on different hardware bases, we deploy them on the same automotive-grade SoC chip to preserve the same test conditions. The commercial product used in our experiments has terms of use that disallow publication of tool performance and tool output. Therefore, we anonymize the product name in this paper. Furthermore, as MPAM is a new feature introduced in ARM v8 [43], we also want to evaluate the effect of MPAM on dynamically isolating cache and memory bandwidth.

5.1 Experiment Setup

We conduct performance experiments on top of an automotive-grade SoC chip (configuration: 8-core ARM Cortex-A710 at 2.15 GHz, ARM Mali G78 GPU, 8GB DDR, 128GB UFS) to run the microhypervisor-based in-vehicle virtualization system (commercially licensed) and AutoVP. Please note that here we also run AutoVP on an automotive-grade SoC instead of a consumer-grade SoC. The primary reason is that Microhypervisor solutions can only run on top of automotive-grade SoCs; otherwise, we cannot perform the comparative evaluation. Due to safety considerations, we are unable to conduct road testing on an actual vehicle; instead, we use the simulated data provided by the automotive manufacture as vehicle-related data. Figure 7 shows the effect of running both in-vehicle virtualization solutions with simulated vehicle-related data. The specific subsystems and their boot sequence for both solutions are listed in Table 4. Next, we present detailed software & hardware configurations for each solution. *Please note that with regards to the static allocation of specific hardware resources for each subsystem, we adhere to the reference configuration information provided by the automotive manufacturer*.



(a) Microkernel + Hypervisor (b) AutoVP Container

Figure 7: Two in-vehicle virtualization solutions are running.

Table 4: The subsystems of two in-vehicle virtualization solutions. The numbers in Row 2∼4 represent the boot sequence of each subsystem. "IC" is short for instrument cluster.

Boot Sequence	Microhypervisor	AutoVP
(1)	Application Domain (Instrument Cluster)	Root Namespace
(2)	System Domain	VM Namespace 1 (IC Display)
(3)	Android VM (In-Vehicle Infotainment)	VM Namespace 2 (In-Vehicle Infotainment)

Microhypervisor Configurations The instrument cluster (IC) subsystem, deployed within the microkernel's application domain, consists of a set of graphical display programs. Hardware resources, including CPU physical core 1 and 2, 500MB of memory, Display 1, and 50% of GPU partition resources are allocated to the IC subsystem. This group of programs also includes driver modules that can directly utilize the corresponding hardware devices. The microkernel's system domain comprises system components, hardware modules, and some device drivers, such as network device driver. The system domain is allocated with CPU physical core 3 and 4, 2GB of memory, storage, and network devices. The in-vehicle infotainment (IVI) subsystem is deployed in the form

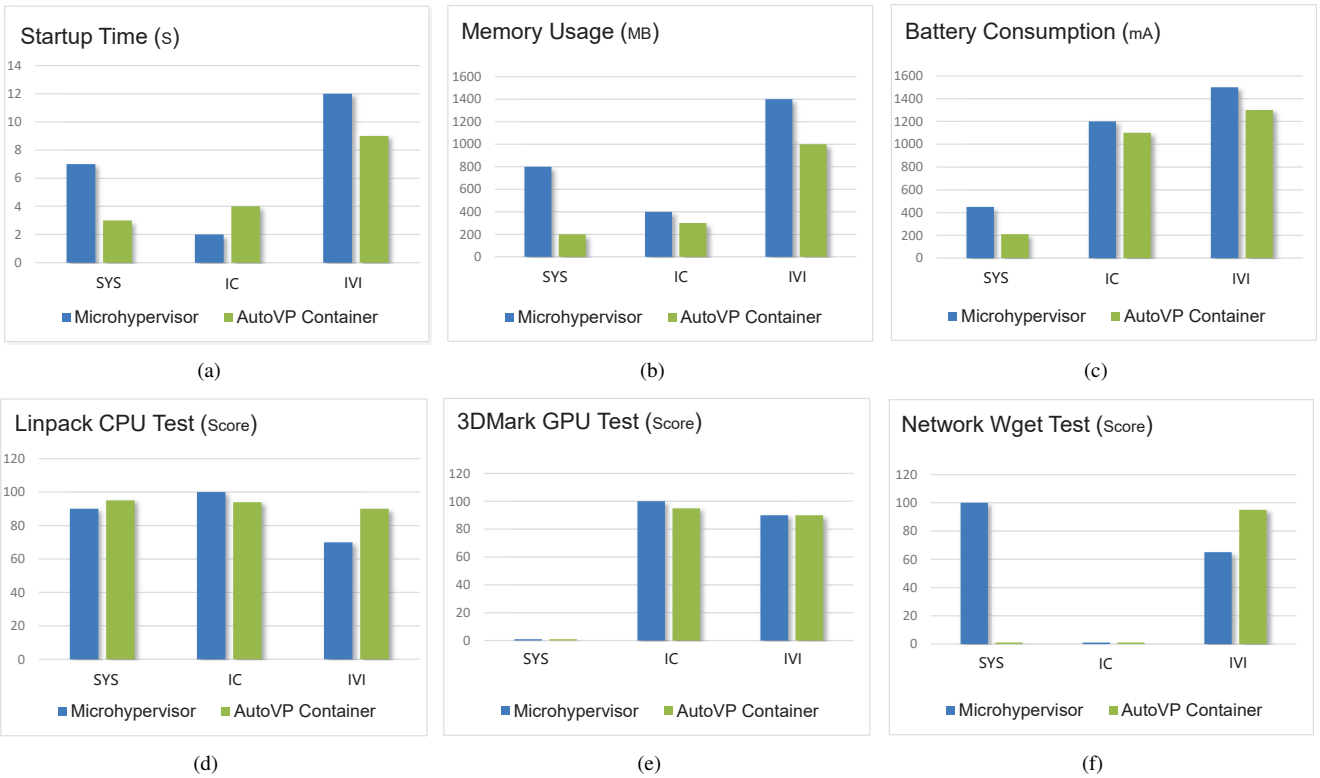


Figure 8: Standard benchmark evaluation results. "SYS" represents the system domain for the microhypervisor product and the root namespace for AutoVP, respectively. "IC" is short for instrument cluster, and "IVI" means in-vehicle infotainment.

of an Android virtual machine, which runs the Android version of 12.0. We allocate all remaining hardware resources to the virtual machine, such as CPU physical cores 5∼8, 5.5GB of memory, 50% of GPU partition resources, and Display 2.

AutoVP Configurations The Linux kernel version used by AutoVP is 5.10.221. The root namespace require low computing resources to run management panel software, and thus AutoVP employs cgroup technology to allocate CPU physical core 1 and 300MB of memory to the root namespace. The instrument cluster display subsystem runs the Qt graphical display framework (version 5.12.); CPU physical core 2 and 3, 500MB of memory, Display 1, and 50% of GPU partition resources are allocated to this subsystem. The container running Android 12.0 hosts the in-vehicle infotainment subsystem, which occupies all remaining hardware resources of the chip, including CPU physical cores 4∼8, 7.2GB of memory, 50% of GPU partition resources, and Display 2.

5.2 Methodology

According to the boot sequence of each subsystem (see Table 4), the method for collecting startup time data involved sequentially starting the application domain, system domain, and Android virtual machine for the microhypervisor solution and recording the startup time. This process was repeated 20 times, and the average startup time was calculated in seconds. Measuring AutoVP's startup time followed a similar procedure, but with a boot sequence of the root namespace, IC display, and Android container. We collect real-time memory usage data for each subsystem after it has been running stably for a period of two hours. Similarly, battery consumption data are recorded within the same two-hour time window.

To collect CPU performance data, we installed the Linpack benchmark in the application domain, system domain, and Android virtual machine for the microhypervisor solution, respectively. We ran the Linpack benchmark for 1 hour and recorded normalized resource utilization data—we repeated this process 10 times and calculated the average value. The CPU measurement of AutoVP followed a similar procedure, but with performance data collected in the root namespace, IC display, and Android container. A higher Linpack score indicates more efficient CPU utilization, resulting in lower performance overhead. It should be noted that the Linpack benchmark evaluates the single-core CPU performance.

The method for collecting GPU performance data was similar to that for CPU performance testing, but we used the 3DMark benchmark. A higher 3DMark score indicates superior GPU utilization, leading to reduced performance overhead. In a similar vein, the method for collecting network performance data leverages the wget application. A high wget score is indicative of an efficient utilization of network bandwidth, thus resulting in lower bandwidth consumption.

5.3 Performance Measurements

Startup Time Figure 8(a) shows the results of startup time. The microhypervisor solution, due to its microkernel structure, can minimize the startup time of the instrument cluster (IC) subsystem. The IC operates as a microkernel application running on top of the microkernel OS, enabling a swift initiation of the IC display. In contrast, AutoVP employs a monolithic kernel approach, with many drivers deployed in the kernel mode. During the system startup phase, it is necessary to initialize all driver modules as a priority, even if certain driver modules will not be immediately used. This results in a longer initialization time for the Linux kernel itself in AutoVP. Additionally, AutoVP requires the container management program to be started before the IC display can be initiated. Experimental data indicates that the startup time for the IC display in AutoVP is slower compared to the microhypervisor solution.

However, the microhypervisor needs to first run the system domain and then initiate complex software virtualization frameworks, such as the virtual machine manager, before launching the virtualized Linux kernel and subsequently the Android VM that hosts in-vehicle infotainment functions. AutoVP, by contrast, requires initializing the Linux kernel only once and then running the container management program to start the Android container. As a result, AutoVP starts the IVI subsystem faster than the microhypervisor solution.

Memory Usage Figure 8(b) shows the comparison of memory usage. Both of these two solutions are allocated 500MB of memory for the IC subsystem. However, in the microhypervisor, the IC subsystem also includes display driver and GPU driver, whereas in AutoVP, the required display and GPU drivers for the IC run within the Linux kernel. Therefore, the memory usage of the IC subsystem in the microhypervisor is slightly higher. Furthermore, the microhypervisor requires the prior execution of complex software virtualization framework, such as the virtual machine manager, before running the virtualized Linux kernel and subsequently the Android VM. In contrast, AutoVP can directly run the Android container on the Linux kernel. Experimental data demonstrates that the memory usage of the container in AutoVP is lower than the that of Android VM in the microhypervisor solution.

Battery Consumption Figure 8(c) shows the measurement of battery consumption. The IC subsystem in the microhypervisor requires running display driver and GPU driver, as well as directly managing hardware resources for IC display peripherals and GPU components. In contrast, the required display and GPU drivers in AutoVP run within the Linux kernel. This can explain why the IC subsystem in the microhypervisor consumes more power. The microhypervisor solution necessitates the prior execution of complex software virtualization framework. Additionally, some hardware devices are directly allocated to and managed by the virtualized Android system. Conversely, AutoVP can directly run the Android container on the kernel, with the kernel managing all peripheral hardware. Therefore, experimental data demonstrates that the power consumption of the Android container in AutoVP is lower than that of the virtualized Android system in the microhypervisor solution.

CPU Performance Figure 8(d) shows the scores of Linpack benchmark, which is the most popular benchmark for ranking of high performance systems. Since both the microhypervisor and AutoVP allocate two CPU physical cores and 500MB of memory to the IC subsystem, the CPU performance data for the IC subsystem are comparable between the two solutions. However, the virtualization framework and virtualized Linux kernel running in the microhypervisor solution consume a significant amount of CPU and memory resources. As a result, experimental results indicate that the CPU performance data for AutoVP's container is superior to that of the virtualized Android system in the microhypervisor.

GPU Performance Figure 8(e) shows the scores of 3DMark benchmark that tests the system's GPU performance. Both the microhypervisor and AutoVP employ GPU partitioning technology, a hardware resource slicing technique that effectively addresses the isolation of GPU resources between the IC subsystem and the Android VM. Neither the system domain of the microhypervisor nor AutoVP's root namespace utilizes GPU resources. Therefore, the GPU performance data obtained from tests conducted on the IC subsystems and the Android VMs are comparable between these two solutions.

Network Performance The network performance data measured by wget is depicted in Figure 8(f). While the microhypervisor's WiFi module is deployed in the system domain, enabling it to benefit from superior network performance, AutoVP's root namespace does not utilize the network in its normal functioning. Furthermore, neither of the IC subsystems in these two solutions use the network. However, the Android VM in the microhypervisor necessitates software virtualization techniques to time-share the WiFi functionality in the system domain. On the other hand, the Android container in AutoVP requires only Linux kernel features to access external networks through the WiFi module. Experimental results show that the network performance of AutoVP's container is significantly better than that of the virtual Android system in the microhypervisor solution.

Real-world Workloads We run common in-vehicle applications such as the dashboard, navigation, music, movie, climate control, vehicle settings, infotainment, Bluetooth connectivity, voice navigation, and reverse camera in both solutions. We use Android *gfxinfo* tool to measure the frame rate (frames drawn per second) of each application. The frame rate metric reflects the performance of the application interface and can assess whether user interactions with the application are smooth. The evaluation results are depicted in Figure 9, wherein a higher frame rate correlates with a better quality of the video or animation. If an application is relatively smooth with a high frame rate, it indicates good performance in the current

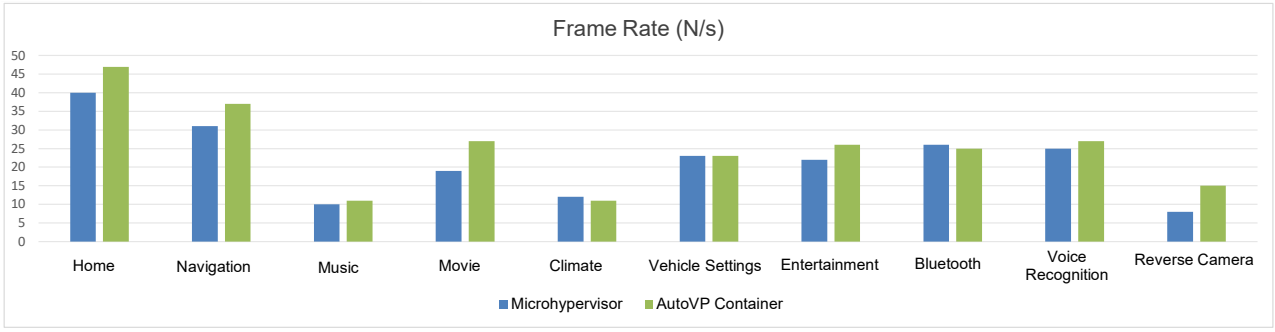


Figure 9: Comparative performance evaluation results with common in-vehicle applications. The higher the frame rate, the better the quality of the video or animation will be.

system environment, including CPU, GPU, and file I/O. Experimental results indicate that the frame rates per second for common in-vehicle applications are higher when running in AutoVP compared to the Hypervisor solution.

Summary The microhypervisor solution, owing to its microkernel structure, enables a quick launch of its IC subsystem. In contrast, AutoVP runs the IC display on the Linux monolithic kernel, which fails to ensure a rapid startup of the IC display. The distinct advantage of AutoVP lies in its ability to ensure efficient utilization of system resources by the Android container. In this configuration, the Android container does not need to run complex virtualization frameworks, and thus can make better use of CPU, network, memory, and battery, resulting in a smoother user experience when running common in-vehicle applications. Due to the same GPU resource allocation, the microhypervisor's VM achieves GPU resource utilization efficiency that is on par with AutoVP's container.

5.4 MPAM Measurements

Due to the noticeable isolation effect of MPAM on mixed-criticality systems [40, 43], we conduced a separate experiment for MPAM resource isolation scenarios in AutoVP. The experiment involved disabling/enabling the MPAM mechanism for IC display and IVI subsystems. For the initial 14 minutes, IC display and IVI shared CPU cache and memory bandwidth resources. At the 15th minute, MPAM was activated. We follow the specific cache and memory bandwidth allocation strategies for various business subsystems, as outlined in Table 3.

As show in Appendix Figure A1(a), it is evident that without enabling MPAM, there is intense competition between IC and IVI services for cache resources, leading to significant mutual interference. However, upon enabling MPAM, the usage of cache resources by IC and IVI services stabilized. Notably, IVI's usage of cache resources was significantly suppressed, with IC's cache usage percentage increasing to about 70%. This drastic improvement notably enhanced cache hit rates and consequently improved IC's performance. However, Appendix Figure A1(b) reveals that the impact of MPAM mechanism on memory bandwidth control was minimal. One contributing factor is that MPAM is a new mechanism introduced by ARM v8, and the current testbed does not offer comprehensive support for the new MPAM mechanism.

6 Discussion & Conclusion

The development of smart automobile cockpits for civilian vehicles needs to balance multiple factors such as safety, reliability, and production costs. The advantages of AutoVP are evident: cost-effectiveness and an almost entirely open-source system. Thanks to the lightweight virtualization features of containers, it allows non-safety-critical tasks to efficiently utilize hardware resources. AutoVP is poised to have enduring significance, persisting beyond the resolution of the automotive chip crisis [23–25]. At present, our approach involves a static resource allocation strategy, as outlined in Table 3. Future endeavors will delve into enabling dynamic resource allocation. Furthermore, we will also explore how to provide real-time capabilities for safety-critical tasks on a Linux system without the need for an external MCU chip.

Conclusion In this paper, we present a new intelligent cockpit virtualization architecture. We segregate safety-critical functions from other non-critical functions into two low-cost chips, respectively. We also run an Android container on the main SoC chip to host non-safety-critical tasks. Our Android container solution features a rich software ecosystem, excellent performance, and inherent cost-effectiveness. By incorporating automotive-grade MCUs to handle safety-critical tasks, the entire system complies with automotive standards. Our comparative evaluation results with a commercial in-vehicle microhypervisor product are exciting.

Acknowledgments

We sincerely thank NSDI anonymous reviewers and our shepherd Amit Levy for their insightful comments. Jiang Ming was supported by the National Science Foundation under grant CNS-2312185 and Carol Lavin Bernick Faculty Grant.

References

[1] Victor Bandur, Gehan Selim, Vera Pantelic, and Mark Lawford. Making the Case for Centralized Automotive E/E Architectures. *IEEE Transactions on Vehicular Technology*, 70(2), 2021.

[2] Sekar Kulandaivel, Shalabh Jain, Jorge Guajardo, and Vyas Sekar. Cannon: Reliable and Stealthy Remote Shutdown Attacks via Unaltered Automotive Microcontrollers. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P '21)*, 2021.

[3] S. Kanajan, C. Pinello, Haibo Zeng, and A. Sangiovanni-Vincentelli. Exploring Trade-off's Between Centralized versus Decentralized Automotive Architectures Using a Virtual Integration Environment. In *Proceedings of the 2006 Design Automation & Test in Europe Conference (DATE '06)*, 2006.

[4] Jan Pelzl, Marko Wolf, and Thomas Wollinger. Virtualization Technologies for Cars: Solutions to Increase Safety and Security of Vehicular ECUs. In *Proceedings of the 2009 Embedded World Conference*, 2009.

[5] Marius Strobl, Markus Kucera, Andrei Foeldi, Thomas Waas, Norbert Balbierer, and Carolin Hilbert. Towards Automotive Virtualization. In *Proceedings of the 2013 International Conference on Applied Electronics*, 2013.

[6] Thomas Gaska, Brian Werner, and David Flagg. Applying Virtualization to Avionics Systems — The Integration Challenges. In *Proceedings of the 29th Digital Avionics Systems Conference*, 2010.

[7] Victor Bandur, Vera Pantelic, Matthew Dawson, Alexander Schaap, Bryon Wasacz, and Mark Lawford. A Domain-Centralized Automotive Powertrain E/E Architecture. Technical report, SAE Technical Paper 2021-01-0786, 2021.

[8] Ashraf Gaffar and Shokoufe Monjezi. Using Artificial Intelligence to Automatically Customize Modern Car Infotainment Systems . In *Proceedings of the International Conference on Artificial Intelligence (ICAI '16)*, 2016.

[9] Arm. The Evolution of Car Cockpit and IVI Systems. https://www.arm.com/markets/automotive/digital-cockpit, 2023.

[10] Michele Paolino, Walt Miner, Daniel Bernal, Artem Mygaiev, Tiejun Chen, and Rich Persaud et al. The Automotive Grade Linux Software Defined Connected Car Architecture. Technical report, The Linux Foundation, June 2018.

[11] Srdjan Usorac, Dejan Bogdanovic, Dario Peric, and Zeljko Lukac. Adding Android Capabilities in Automotive Linux Infotainment: Available Virtualization Technologies. In *Proceedings of the 29th Telecommunications Forum*, 2021.

[12] BlackBerry QNX. QNX Hypervisor Virtualization Software. https://blackberry.qnx.com/en/products/foundation-software/qnx-hypervisor, 2023.

[13] Green Hills Software. INTEGRITY Multivisor. https://www.ghs.com/products/rtos/integrity_virtualization.html, 2023.

[14] SYSGO GMBH. PikeOS: Certifiable RTOS & Hypervisor. https://www.sysgo.com/pikeos, 2023.

[15] Asif Iqbal, Nayeema Sadeque, and Rafika Ida Mutia. An overview of microkernel, hypervisor and microvisor virtualization approaches for embedded systems. *Report, Department of Electrical and Information Technology, Lund University, Sweden*, 2009.

[16] Everton de Matos and Markku Ahvenjärvi. seL4 Microkernel for Virtualization Use-Cases: Potential Directions towards a Standard VMM. *Electronics*, 11(24), 2022.

[17] Hermann Härtig, Michael Roitzsch, Adam Lackorzynski, Björn Döbel, and Alexander Böttcher. L4 – Virtualization and Beyond. *Korean Information Science Society Review*, 2, 2008.

[18] Dong-Guen Kim, Sang-Min Lee, and Dong-Ryeol Shin. Design of the Operating System Virtualization on L4 Microkernel. In *Proceedings of the 4th International Conference on Networked Computing and Advanced Information Management*, 2008.

[19] Jörn Schneider and Tillmann Nett. Safety Issues of Integrating IVI and ADAS functionality via Running Linux and AUTOSAR in Parallel on a Dual-Core-System. *Automotive - Safety & Security*, 2015.

[20] Jason Belt, John Hatcliff, Robby, John Shackleton, Jim Carciofini, Todd Carpenter, Eric Mercer, Isaac Amundson, Junaid Babar, Darren Cofer, David Hardin, Karl Hoech, Konrad Slind, Ihor Kuz, and Kent Mcleod. Model-Driven Development for the SeL4 Microkernel Using the HAMR Framework. *Journal of Systems Architecture*, 134, 2023.

[21] Aswin Sampath Kumar and Tuğrul Daim. Study on Consumer Requirements for Automotive Infotainment Systems. *R&D Management in the Knowledge Era: Challenges of Emerging Technologies*, 2019.

[22] Alessandra Nardi and Uyen Tran. Key Requirements for Automotive SoC Design. Technical report, Synopsys, April 2023.

[23] Keith Naughton. Automotive Chip-Shortage Cost Estimate Surges to $110 Billion. Bloomberg Technology, May 2021.

[24] Xiling Wu, Caihua Zhang, and Wei Du. An analysis on the crisis of "chips shortage" in automobile industry-Based on the double influence of COVID-19 and trade friction. In *Journal of Physics: Conference Series*, volume 1971. IOP Publishing, 2021.

[25] Kristin Dziczek et al. Why the Automotive Chip Crisis Isn't Over (Yet). *Chicago Fed Letter*, 2022.

[26] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, 2011.

[27] Christoffer Dall, Jeremy Andrus, Alexander Van't Hof, Oren Laadan, and Jason Nieh. The Design, Implementation, and Evaluation of Cells: A Virtual Smartphone Architecture. *ACM Transactions on Computer Systems*, 30(3), August 2012.

[28] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '07)*, 2007.

[29] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. In *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software*, 2015.

[30] Michael Eder. Hypervisor- vs. Container-based Virtualization. In *Proceedings of the Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications*, 2016.

[31] International Organization for Standardization. ISO 26262: Road Vehicles — Functional Safety. https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-2:v1:en, 2018.

[32] J Langheim, Bruno Guegan, Laurent Maillet-Contoz, Kamel Maaziz, Gilles Zeppa, S Boutin, H Aboutaleb, F Philippot, and Pierre David. System architecture, tools and modelling for safety critical automotive applications–the R&D project SASHA. In *Proceedings of the 2010 Embedded Real Time Software and Systems Conference*, 2010.

[33] Pierre Lucas, Kevin Chappuis, Michele Paolino, Nicolas Dagieu, and Daniel Raho. VOSYSmonitor, a Low Latency Monitor Layer for Mixed-Criticality Systems on ARMv8-A. In *Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS '17)*, 2017.

[34] Google. Automotive OS. https://developers.google.com/cars/design/automotive-os, 2023.

[35] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*, 2010.

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

[37] Business Wire. Automotive Cockpit Multi & Dual Display Trends Report. https://www.researchandmarkets.com/r/61epwn, 2020.

[38] Qt Group. Tools for Each Stage of Software Development Lifecycle. https://www.qt.io/, 2023.

[39] Android Open Source Project. Using Binder IPC. https://source.android.com/devices/architecture/hidl/binder-ipc, [2022].

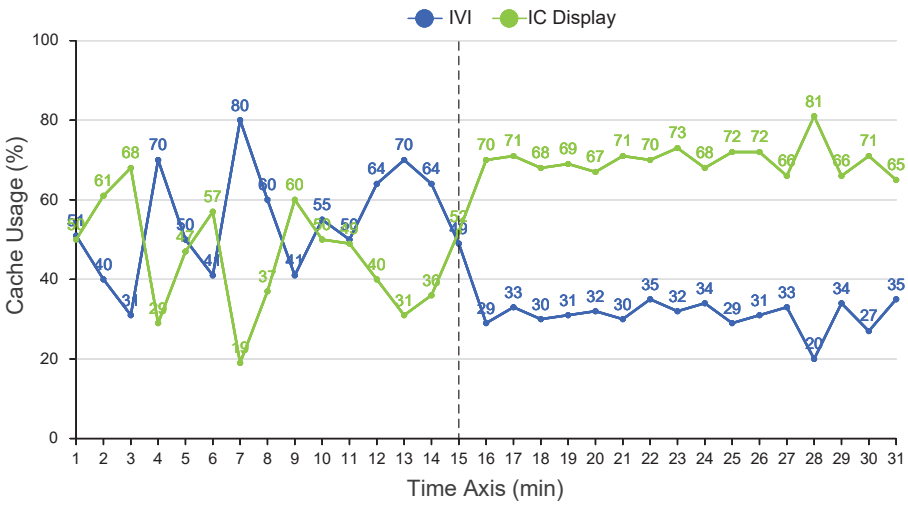
[40] Arm. Memory System Resource Partitioning and Monitoring (MPAM), for A-profile architecture. Arm Architecture Reference Manual Supplement, 2022.

[41] Chris Down. 5 Years of Cgroup v2: The Future of Linux Resource Control. In *Proceedings of the 34th Large Installation System Administration Conference (LISA '21)*, 2021.

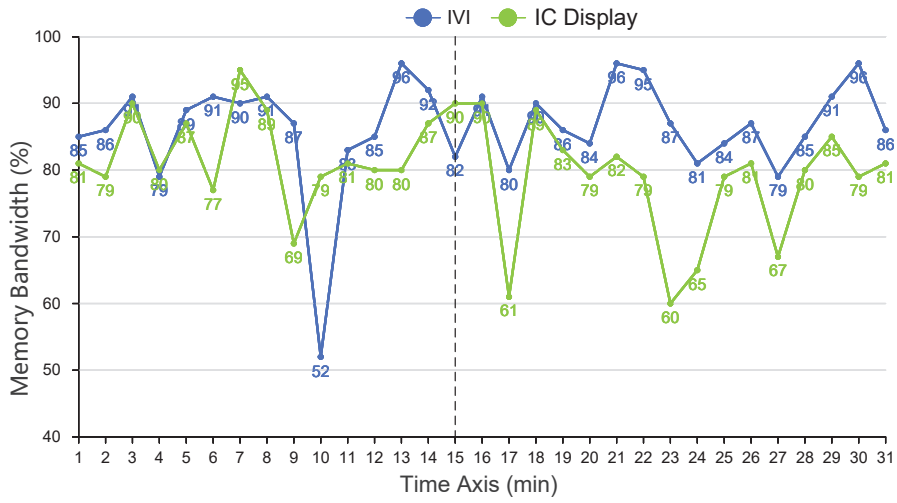
[42] Rami Rosen. Resource management: Linux kernel namespaces and cgroups. *Haifux, May*, 186:70, 2013.

[43] Matteo Zini, Daniel Casini, and Alessandro Biondi. Analyzing Arm's MPAM From the Perspective of Time Predictability. *IEEE Transactions on Computers*, 72(1), 2023.

Appendix



(a) Real-time Cache Usage



(b) Real-time Memory Bandwidth

Figure A1: The measurement of MPAM effects on dynamically isolating cache and memory bandwidth.

MuCache: A General Framework for Caching in Microservice Graphs

Haoran Zhang*, Konstantinos Kallas*, Spyros Pavlatos, Rajeev Alur, Sebastian Angel, Vincent Liu
University of Pennsylvania

Abstract

This paper introduces *MuCache*, a framework for extending arbitrary microservice applications with inter-service caches. MuCache significantly improves the performance of microservice graphs (commonly found in large applications like Uber or Twitter) by eliminating the need for one microservice to call another when the relevant state has not changed. MuCache is enabled by a novel non-blocking cache coherence and invalidation protocol for graph topologies that minimizes critical-path overhead. For this protocol, we prove a strong correctness result: any execution observed by the cache-enabled microservice application could have been observed by the original application without caches. Our evaluation on well-known microservice benchmarks shows that MuCache reduces the median request latency by up to $2.5\times$, and increases throughput by up to 60%.

1 Introduction

Many of today's largest web services, such as Uber, Lyft, Twitter (now X), and Meta structure their applications as microservices to enjoy the benefits of developer agility, resource provisioning, maintainability, fault tolerance, and other important metrics. Fortunately, these benefits can be reaped by smaller companies and individual developers too; recent runtimes and service mesh frameworks like Dapr [3], Envoy [4], and Istio [6] have been created to help design, deploy, and manage microservices. In a microservice architecture, applications are decomposed into a call graph of services that interact with each other and with end-users through API calls. The 'root' of the microservice call graph is typically a client-facing frontend service, while the 'leaves' are databases that store service state. This call graph is dynamic in the sense that it may be different for each request.

The call graphs of today's services are complex. Services like Airbnb and Uber consist of thousands of unique microservices that support their functionality [5, 40]. Each user request will flow through a substantial subset of these microservices. A typical Twitter request, for instance, can traverse a call graph that is 6 levels deep with significant fan-out at each level [36], and requests to popular Facebook pages often involve hundreds of servers [30]. Given the depth and breadth of modern microservice call graphs and that each edge corresponds to a network request, it is important to avoid making such calls whenever possible. One way to do so is by having each microservice reuse the responses of the services it contacts if it knows that a request will produce the same answer. Caching the responses in this fashion improves both the latency of the target request (if the branch would have been on the request's critical path) and the system's throughput as a whole (by freeing resources for other requests).

Response caching is, of course, a common technique in system design that many services are already employing to great effect. For example, a recent Alibaba analysis of its storefront microservice architecture found that around 40% of its call graphs have a depth of 3 because they hit a cache; while requests that do not hit caches reach call graph sizes of more than 40 [28]. Similarly, half of Twitter's cache clusters are used to cache intermediate computation results [38], and a study of the cache clusters at Facebook [37] reported that the cache-hit ratio for a specific cluster was more than 80%.

Unfortunately, adding caching to a microservice graph is difficult, and is something that only well-resourced companies can afford to do correctly. Indeed, no existing service mesh provides support for inter-service caching, leaving mid-size and small companies to deal with caching on their own.

The complexity of caching in microservices stems from the fact that—unlike traditional cache coherence protocols where one can reason about individual read and write operations to a target object—microservice responses are a function of the input request, the service's state, *and* an unbounded number of downstream services that are recursively called during the processing of a request. The resulting web of dependencies means that developers must carefully study the interactions between all services in the system.

Consider, for example, a user's home timeline in a social network (e.g., [14, 36]). A cached timeline response can be invalidated by changes to the user's followees, the content of contained posts, the security policies of users included in the timeline, and tweaks to the ranking algorithm; these changes can stem from requests that never touch the home timeline mixer, modifications to objects from services several hops downstream, and revisions to the control flow of dependent services. Note that, depending on the nature of the change, subtrees of the call graph may be cacheable even if the final response is not, and effective caching approaches should consider that distinction.

Given the benefits of caching and the absence of automated tools that developers can use to help them with this task, what

*Equal contribution.

can developers do today? Broadly speaking, developers today add caching by either (a) creating manual or application-specific coherence protocols, which are error-prone and fail to generalize; (b) focusing on the backend-storage layer [24, 30], which ignores the significant advantages of terminating request call graphs early; or (c) giving up on consistency and implementing simple TTL-based eviction mechanisms [1, 27, 33] that can produce stale responses.

In this paper, we propose MuCache, a framework that extends existing service meshes like Dapr to automatically provide microservice applications with inter-service caches that improve performance. Users declare the read-only methods of each service's API, and then MuCache caches the results of calls to these methods to avoid re-executing them if the data has not changed. To keep caches coherent, MuCache implements a novel lazy-invalidation cache coherence protocol for dynamic graphs of services. This new protocol has two important features. First, all cache accesses are non-blocking, so requests never need to wait for other requests to finish. This greatly reduces latency during cache hits and ensures that in the event of a cache miss, the overhead of having had to access the cache is both constant and small. Second, the protocol is provably correct and provides a very strong consistency guarantee: *all executions of a cache-enabled application are equivalent to an execution of the original application without caches*. Note that optimal tuning of these caches (e.g., selecting the optimal cache size or eviction policy) is out of the scope of this paper; instead, MuCache allows the developer to use other tools to tune each cache separately without having to worry about coherence.

Our experimental evaluation shows that MuCache achieves a median request latency reduction of up to 2.5× and a 95th-percentile tail latency reduction of up to 1.8× for well-known microservice benchmarks and applications. Additionally, MuCache increases throughput by up to 60% and allows applications with MuCache to scale as well as the original implementation. We also perform worst-case tests, artificially reducing the cache hit rate to 0%, and the results indicate that MuCache's overheads are minimal.

MuCache is open-source at `https://github.com/eniac/mucache`.

2 Background

In this section, we briefly review microservice architectures and the *graphs* that are formed by their interconnections.

Microservices. Applications today often comprise various services, each of which handles an incoming request, performs some task, and returns a response. This *microservice* paradigm has many benefits over prior (monolithic) architectures in which all functionality exists within a single component. For example, microservices are modular, so they can be implemented in any language and with any features so long as they expose an appropriate API (e.g., REST). This also allows teams to design, develop, deploy, scale, and optimize each microservice independently. Furthermore, each microservice can manage its own datastore, ensuring *data sovereignty*, which is important for fault tolerance and when regulations place strict access control requirements on data.

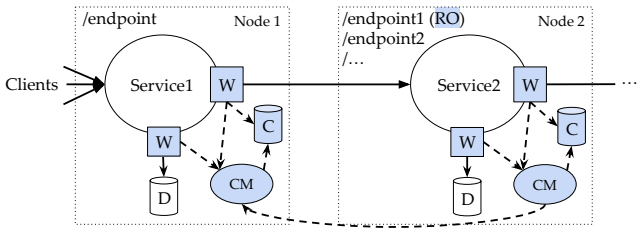


FIGURE 1—**MuCache's Architecture.** (C) denote caches, (CM) cache managers, (W) wrappers, and (D) the datastores. Wrappers are interceptor functions in the sidecar of each service. Solid arrows denote baseline communication while dashed arrows and blue components denote additions by our system. RO means read-only.

Microservice graphs. Today's microservice applications collate information from multiple backend sources and distill it into a single user interface with the help of intermediate processing functions. Such a design naturally leads to a directed graph of microservices that collectively implement the application's behavior. In this graph, the vertices are microservices, and the edges represent calls between them.

A drawback of this approach is that it incurs higher communication costs and response latency compared to monolithic systems. Whereas in a monolithic system, all invocations would typically be local function calls that leverage a machine's fast memory and native data structures, in a microservice graph, the caller needs to create the request (e.g., an HTTP request with a serialized JSON payload) and send it over the network to the callee (which may include compression and encryption), who must then deserialize it and execute it—potentially having to call further microservices. The response will incur similar overheads. For long chains of calls, the additional latency can be in the order of 100ms.

3 Goals and Overview

Given the prevalence of microservice architectures and the high costs associated with inter-service communication, we wish to design a general caching layer that avoids having one microservice call another whenever such a call is unnecessary, e.g. when the request is similar to a prior request and the state of the callee has not changed. While there is a vast design space that one could consider for achieving this high-level goal, we are grounded by a set of pragmatic requirements:

- *Correctness*: The cache should not introduce behaviors that are not part of the original application.
- *Non-blocking and low overhead*: The cache should not require blocking on the critical path, and any overhead should be minimal.
- *Dynamic graphs*: It should support microservices where

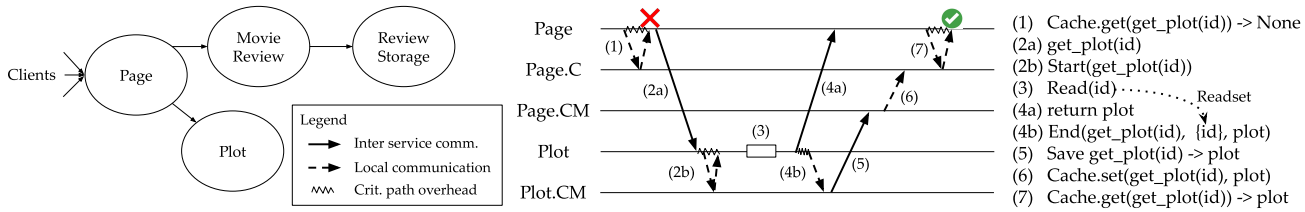


FIGURE 2—(Left) Moview Review application fragment. (Right) An example execution of this application. Each line corresponds to a different component, and arrows denote communication. (C) components are caches and (CM) cache managers.

the call-graph varies per-request and is not known a priori.

- *Sharding*: It should support microservices that are deployed with multiple shards to enable scaleout.
- *Application and datastore agnostic*: It should not require any modifications to application logic or backend datastores for easier adoption.
- *Incremental deployment*: It should provide benefits even when only deployed on a subset of the microservice graph, e.g., if subgraphs are managed by different organizations.

3.1 Overview of MuCache

To meet the above requirements, we design and implement MuCache, a new caching framework for microservice graphs; we depict its architecture in Figure 1. In this figure, MuCache extends an application that consists of two microservices, *Service1* and *Service2*, each of which has its own datastore and exposes a set of available methods that clients or other services can call. We refer to these methods as *endpoints*, borrowing from REST terminology. Throughout the paper, when a Service1 calls a Service2, we refer to Service1 as the upstream and Service2 as the downstream.

MuCache extends each service with *wrappers* (W), a *cache manager* (CM), and a *cache* (C). The wrappers are a shim layer that intercepts all communication among services and their datastores. MuCache's wrappers are implemented on top of Dapr [3], a distributed application runtime that orchestrates service invocations and datastore accesses through its API. Dapr supports many datastores through the same API, so our wrappers inherit this compatibility without additional effort. The cache manager saves and deletes entries from the cache to maintain coherence by tracking all inter-service communication and datastore accesses through the wrappers. It is deployed as a separate executable on the same node with the service. Microservices are often sharded across multiple instances to support larger workloads; in such cases, each shard has its own cache manager and cache. MuCache does not impose any configuration restrictions on the cache which can be configured to have any eviction policy, cache size, etc. MuCache acquires knowledge of the graph topology in a decentralized way: each cache manager only knows about its immediate predecessor cache managers.

Workflow. To deploy MuCache, developers must first declare the read-only (RO) endpoints that do not mutate the datastore. If developers use REST APIs, MuCache can automatically

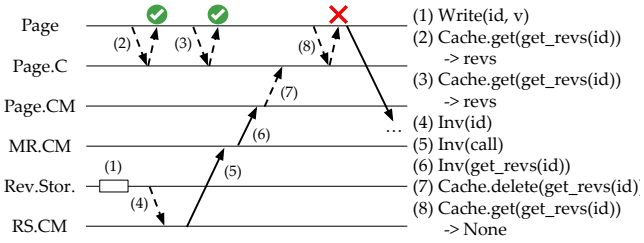


FIGURE 3—An example execution of the movie review application that includes an invalidation.

infer this by treating 'GET' endpoints as RO. The cache will then store the responses of successful requests to the RO endpoints of other services. For example, the cache of Service1 would store the return values of requests to `/endpoint1` of Service2. The cache manager of Service2 would then track all of the keys that were read during each RO call, and whenever one of these keys is modified by a write, it sends messages to all of its caller's cache managers (in this case, the CM of Service1) to invalidate the relevant cache entries.

As a concrete example, consider Figure 2 (left), which shows a fragment of a movie review application (Cf. IMDB) [24]; clients request the page of a specific movie from the `Page` service, which in turn calls the `MovieReview` and `Plot` service to compute its results. The right side of Figure 2 shows one example execution. The first time `Page` tries to get the plot for movie `id`, it does not find it in the cache (1) and then invokes `Plot`. After the call returns, the cache manager of the `Plot` service informs the cache manager of `Page` to save the return value of this call (5). For a subsequent call with the same arguments (7), the return value is found in the cache, and the `Plot` service is never contacted. Figure 3 shows an execution where some other user adds a new review to the movie `id` (1). While the write happens, different users successfully access the page of that movie from the cache (2), (3). The invalidation is propagated in the background between cache managers, until it invalidates all affected saved cache entries, including the one in the cache of `Page`.

We note that most of the processing to update and invalidate caches in MuCache is done off the critical path. With reference to Figure 2, the only operations in the critical path are steps (1), (2b), and (4b), all of which are local accesses. In particular, (1) is an access to a local cache, and (2b) and (4b) involve communication with a co-located cache manager. Furthermore, MuCache supports sharding without any com-

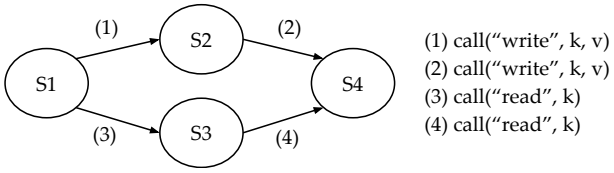


FIGURE 4—An application exhibiting a "diamond" pattern.

munication between shards on the request processing critical path—cache managers of different shards only communicate invalidation in the background. At the same time, the invalidation delays in MuCache are very small (ms)—orders of magnitude smaller than standard values of TTL used in practice to evict cache items (seconds to hours) (§7).

Correctness. The correctness condition for MuCache is based on classical refinement modulo reordering, i.e. that all behaviors exposed by a cache-enabled application are equivalent to a behavior of a cache-free version after potentially reordering independent observable events. The execution in Figure 3 is correct because it could have been observed from the original application if the write (1) had happened right after (2) and (3) since they are independent requests from different clients.

Guaranteeing correctness is challenging for call graphs with more than one path between the same two services, i.e., when a request accesses the same backend service twice in its lifetime. Figure 4 shows such an example of a 'diamond' pattern. In this example, a service S1 first calls S2, which in turn calls S4 that writes to its store. Then S1 calls S3, which calls S4 trying to read from the same value that was written by S2. It is possible that S1 could find the result of a previous call to S3 in its cache, reading a stale value, leading to an execution that would not be observable without caches. Since microservice call graphs are dynamic, this pattern cannot be identified and prevented statically (before execution). MuCache addresses this at runtime by keeping track of visited services during request processing, not checking a cached entry if it depends on a service that has already been visited.

Summary. We conclude this section by describing how MuCache satisfies the previously stated requirements:

- *Correctness*: We prove that MuCache does not introduce behaviors that are not part of the original application (§5).
- *Non-blocking and low overhead*: Cache managers do all processing in the background and the wrappers only send messages to them, never blocking for a response.
- *Dynamic graphs*: MuCache tracks dependencies to guarantee correctness in the presence of dynamic call graphs.
- *Sharding*: MuCache supports sharding without any additional communication on the critical path.
- *Application and datastore agnostic*: MuCache does not require any modification to the application or datastore code because wrappers intercept all communication.
- *Incremental deployment*: Developers can gradually declare read-only endpoints to get incremental benefits.

4 MuCache Protocol

Figure 5 shows the complete MuCache protocol for the wrappers of a single service shard and its cache manager in Python-like pseudocode. The wrapper of each service communicates with its associated cache manager through an ordered message queue (using `SendToCM`). Downstream cache managers also issue `Save/Inv` events to upstream ones through the same queue. Cache managers in different shards of the same service also communicate with each other when broadcasting invalidations using `SendToShardCM`.

The code on the left depicts wrapper logic run before a request starts processing (`preReqStart`), when a request has finished processing (`preReturn`), when a request reads from a key (`preRead`), before a request performs a call to another service (`preCall`), and after a request writes to a key (`postWrite`). The code on the right depicts cache manager logic, which processes events in the message queue sent by the wrappers and cache managers of other services.

Wrapper. The wrappers keep two types of state. The first is a global (per service shard) `readsets` map from request identifiers to sets of keys and call arguments, which keeps the dependencies of each pending read-only (RO) request. The second is the per-request context `ctx`, which is carried around while a request is processed. `ctx` contains (1) the id of the request (`ctx.call_id`); (2) the hash value of the request's arguments (`ctx.ca`); (3) the caller of the request (`ctx.caller`); (4) the visited services of the request and its subrequests (`ctx.visited`); and (5) whether the current request is read-only and, therefore, cacheable by its caller (`ctx.isRO`). Wrappers send a `Start(ca)` message to their associated cache manager before a request starts processing and then maintain the request readset when a read or a subrequest is performed. Once the request is complete, the entire readset, along with the call arguments, the caller, the return value, and the visited services are sent to the cache manager as an `End(ca, rs, caller, ret, vs)` message. Wrappers also send `Inv(k)` messages to cache managers after a datastore key `k` is modified. `preCall` checks the cache before invocation and returns directly upon cache hits.

Cache manager. The cache manager controls the contents of the cache. The cache manager contains two global (per service shard) state components: `saved` and `history`. The `saved` map acts as an inverted index of wrappers' `readsets` by mapping keys (or call arguments) to the corresponding service that has read (or called) them. When a key or a set of calls is invalidated, the cache manager looks up `saved` to locate all the affected upstream services and asks them to invalidate the set of relevant calls that they have cached by sending them `Inv` messages. The second state component, `history`, is a sequence of calls and invalidations used to determine whether a call can be safely cached upstream. When a request with readset `rs` is complete, the cache manager scans the history in reverse chronological order for invalidations that intersect

```
global readsets : map(Key, set(Key | CallArgs))

def preReqStart(ctx):
    if ctx.isRO:
        cid, ca = ctx.call_id, ctx.ca
        readsets[cid] = set()
        SendToCM(Start(ca))

def preReturn(ctx, ret):
    if ctx.isRO:
        cid, ca = ctx.call_id, ctx.ca
        rs = readsets.pop(cid)
        caller = ctx.caller
        vs = ctx.visited
        SendToCM(End(ca, rs, caller, ret, vs))

def postWrite(ctx, k, _v):
    SendToCM(Inv(k))

def preRead(ctx, k):
    if ctx.isRO:
        cid = ctx.call_id
        readsets[cid].insert(k)

def preCall(ctx, ca):
    if ctx.isRO:
        cid = ctx.call_id
        readsets[cid].insert(ca)
    # Check if ca refers to a read-only endpoint and if
    # the visited services are disjoint with the cache
    # item subtree
    if ca.isRO and visited_disjoint(ctx, ca):
        return cache.get(ca)
    return None
```

```
# Tracks which keys and calls will invalidate
#  which cache entries upstream
global saved : map(Key | CallArgs, map(Service, CallArgs))
# Sequence of calls and invalidations
global history : list(Call(CallArgs) | Inv(Key | CallArgs))

def startHandler(ca):
    history.append(Call(ca))

def endHandler(ca, rs, caller, ret, vs):
    # Checks if there are any invalidations
    # to the readset since the call start
    if empty([for Inv(k) in history.invs_after(Call(ca))
              if k in rs]):
        SendToCM(caller, Save(ca, ret, vs))
        saved.store(rs, ca, caller)

def invHandler(k):
    match type(k):
    case Key:
        history.append(Inv(k))
    case CallArgs:
        history.extend([Inv(ca) for ca in k])
    # Inform CMs of same-service shards
    SendToShardCMs(Inv(k)) # (see Sec. 4.1)
    # Ask all affected callers to invalidate
    affected = saved.pop(k)
    for caller, cas in affected:
        SendToCM(caller, Inv(cas))

def saveHandler(ca, ret, vs):
    save_visited(ca, vs)
    cache.set(ca, ret)
```

FIGURE 5—(Left) The wrapper code of the protocol that intercepts the start of request processing, returns, writes, reads, and calls. (Right) The cache manager code that processes work queue items sent by the wrappers and other cache managers.

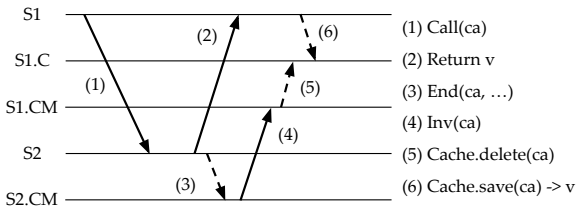


FIGURE 6—A bug that would occur if invalidate messages were allowed to overtake saves.

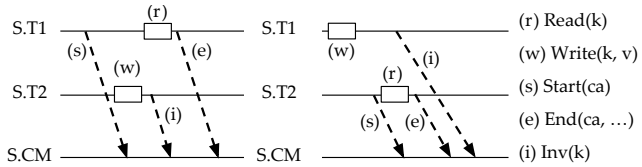


FIGURE 7—Possible imprecisions in invalidation. The three lines represent two service threads processing requests and the cache manager.

with `rs` since the call started. If there is no such invalidation, it asks the upstream cache manager to save the result. The cost of this scan is proportional to the product of request rate and average request duration, which is typically a small number. For example, a service handling 10,000 requests per second, each lasting 100 milliseconds, requires scanning several thousand items.

Saving a new cache entry. A naive method of saving a new entry involves the caller immediately saving it to the cache upon the result's arrival, rather than awaiting an explicit `Save` message from the callee's cache manager. This is not correct, as it allows the bug shown in Figure 6 where the invalidation message by the S2 cache manager "overtakes" the save done by S1, leading to the cache entry never being invalidated. Thus, it is necessary for Invs and $Save$s to not be reordered. MuCache achieves that by issuing them sequentially through the cache manager.

Invalidating an entry. Invalidations are triggered when a key used in a cached result is modified. Naively, the cache manager could track the exact order of all reads and writes to precisely track invalidations. Since requests are being processed concurrently, this would require coordination across different service threads, which would significantly slow down request processing along the critical path. MuCache relaxes the tracking of reads and writes in two ways that do not jeopardize correctness, but reduce the synchronization overhead. First, all reads of a request are gathered by the wrappers (`preRead`) and are only sent to the cache manager at the end of the request (the `rs` argument in the `End` message). To ensure correctness, the cache manager then assumes that all reads happened at the start of the call, considering the call invalid if a write happened in its duration even if it happened before the reads (Fig. 7, Left). Second, writes are intercepted in a non-atomic fashion after they have been completed (`postWrite`). This could allow for a call to start and complete in between the actual write and `postWrite`, leading to its cached response being unnecessarily invalidated (Fig. 7, Right).

Evicting an entry. There are two types of evictions in MuCache. First, a cache could fill up with entries and needs to

evict an entry to make space for new ones; in this case, the eviction is safe without any additional work since the protocol is robust to re-invalidations (i.e., it is safe to invalidate a cache entry that was previously evicted). Second, the cache manager might need to reclaim space if it is keeping track of the dependencies of many calls. It reclaims space by evicting a key or call from its `saved` dependencies and consequently sends invalidation messages to all affected calls upstream as if the key were invalidated (see `inv(k)` in Figure 5).

Garbage collection. The cache manager has two state components that grow during execution: (1) the history and (2) the dependencies. It keeps the history bounded by removing completed calls when processing an `End` request, adding minimal overhead. The protocol preserves correctness in the presence of multiple pending calls with the same arguments by removing the latest occurrence of a call start (potentially overapproximating the duration of the other calls). When the cache manager reaches a memory limit, it deletes some of its saved dependencies as long as it informs the upstream caches to evict relevant entries (similarly to a normal invalidation). The current implementation evicts dependencies following an LRU policy, though other choices could be used.

Sharding. MuCache supports sharded service deployments by attaching a cache manager to each shard; the only requirement being that read-only calls with the same arguments are always processed by the same shard (e.g., by load balancing these calls based on a hash of the call arguments). This guarantees that a single cache manager is the sole authority for the invalidations of each read-only call, ensuring that they will be the only ones to send cache-save and cache-invalidate messages for that call. The only change in the protocol is that a cache manager processing an invalidate due to a write needs to broadcast it to all cache managers of the other shards of the same service, so that they can invalidate their relevant calls (see L.25 in Figure 5). It is important to note that broadcasts only happen upon users' writes; transitive invalidations propagated upstream do not trigger broadcasts. Broadcasting out of the critical path is safe because, similarly to the single-shard protocol, overapproximating the write duration might lead to additional invalidations but not fewer. MuCache, therefore, does not add any latency overhead on the request processing critical path to support sharded services.

Handling Dynamic Call-graphs. Microservice applications can exhibit a diamond pattern (Figure 4) where a request performs multiple subrequests to the same service through its lifetime. In such applications naive caching could lead to executions that cannot be observed without caches. MuCache addresses this by keeping track of the visited services in two locations. First, each request keeps visited services in its context (`ctx.visited`); whenever a subrequest `ca` returns, the parent request adds all the visited services of the subrequest (`ca.visited`) to its own visited services (`ctx.visited`). Second, when saving a cache entry for call `ca`, the cache manager also stores the services, S', that were visited during the processing of `ca`. Before checking the cache, the wrapper checks if the downstream service has ever visited a service in S' that has also been visited by the current request `visited_disjoint(ctx, ca)`; if so, it does not retrieve the return value from the cache to preserve correctness. MuCache tracks visited services using a binary encoding that keeps its size small—less than 1 KB for 1000 services.

5 Protocol Correctness

To demonstrate the correctness of MuCache, we show that clients cannot differentiate a MuCache-enabled application from the original without caches. We give semantics to microservice applications (with and without caches) using observable execution events and traces. Events are indivisible actions (steps) that can be performed by a microservice application; examples of events include reading from a key in the datastore and receiving a response from a completed subrequest. An application can be uniquely described by the set of traces (event sequences) that can be observed in it. Two traces are said to be *equivalent modulo reordering* when all events in one trace exist in the other trace but potentially in a different order. Reorderings are necessary for our correctness theorem to allow reads and writes to proceed concurrently (as in Figure 3). In this section, we informally describe three assumptions that are central to our formal development, the first two hold for all microservice applications, and the last one is a requirement of MuCache. We then state our main theorem and give the high-level intuition for the proof. The complete formal development and proof can be found in Appendix A, which is available in the supplementary material.

(A1) Always enabled requests. Requests in a microservice application only block when waiting for subrequests that they have invoked to finish executing and there is no blocking communication across independent requests. In other words, if a trace can be observed in an application, then we can pick and execute *any* pending request, or any of its subrequests, until it produces an execution event, and the new trace will also be part of the application's set of traces.

(A2) Reordering independent events. Two events are dependent when the first event affects the execution of the second: some examples include two events that are part of the same request, or a write and a read event to the same key in a service datastore. The complete definition of dependent events is given in Appendix A. We assume that due to multithreading, independent events commute; that is, reordering any two consecutive independent events in an application trace results in a trace that can also be observed by the application.

(A3) Linearizable datastores. We assume that the datastore of each service is *linearizable* [26]: operations on an object take place atomically, in an order consistent with the operations' real-time order. For instance, if a write completes before a read begins, then the read must observe the effects

of the write and complete after it. This is necessary due to the requirement that MuCache does not modify the underlying datastore and can only observe writes to the datastore before or after they are completed. If we were to use a non-linearizable datastore a write could take effect after it returns, making it impossible to track which calls it invalidates.

Theorem 1 (Protocol Correctness)**.** *For all traces in a cache-enabled application, there exists a trace in the original application without caches, such that all the client events in the two traces are equivalent modulo reordering.*

Proof intuition. To show the theorem, we prove a stronger lemma, namely that for all cache-enabled traces, we can construct an original trace where (1) all request subtraces are the same (modulo the missing requests due to cache hits), and (2) that the application state is the same at the end of both traces. The proof proceeds by induction on the length of traces and has three phases: (1) given a trace in the cache-enabled application that ends with a cache-hit, it uses assumption (A2) to move writes that happened before the cache-hit but would later invalidate its entry to the end of the trace (together with their dependencies); (2) it then uses the inductive hypothesis to construct a trace in the original application for the prefix up to the cache-hit; and (3) it uses assumption (A1) to fill in all subrequest events that are missing due to the cache-hit, and then it fills in the writes and all their dependencies (A3), ending up with a trace that satisfies the requirement.

6 Implementation

The MuCache implementation comprises roughly 2k LoC of Go [12], including the wrappers that intercept invocations and state accesses, and the cache manager that makes invalidation and saving decisions. Communication between wrappers and the cache manager happens with ZeroMQ [16] and between cache managers with HTTP. Our current implementation uses Redis [9] as the cache, but any in-memory store could be used in its place. We use 32-bit FNV-1a [11] algorithm to compute the hash values of call arguments.

Batching. Cache managers instruct their upstream counterparts to save or invalidate cache entries by sending HTTP requests that might become a bottleneck when the load is high. To increase throughput at high loads without affecting correctness, MuCache allows batching requests that are sent upstream. At low loads, batching increases the time it takes for an invalidation to propagate through the system based on the batching timeout, which is currently set to 1ms. Batching also enables the simplification of upstream requests by canceling out operations at the sender, i.e., invalidates and saves override previous invalidates and saves on the same key. This reduces the size of requests and the number of operations upstream cache managers have to process, while incurring minimal cost since it requires a single pass over the batch.

General support. MuCache is designed to not be limited to a single communication protocol, cache, or datastore. Our wrappers are built on top of Dapr [3], a service mesh extended to also support state accesses through its API. Dapr supports custom middlewares that can be used to intercept invocations and state accesses. It also provides a common abstraction for many service communication protocols and different storage backends, allowing us to implement our wrappers once and inherit support for all the alternatives.

Dependencies between client requests. MuCache's caching protocol treats client requests as independent and allows them to be reordered, processing reads and writes from different clients without synchronization. However, this might not always be desirable, e.g., when a client request expects to see the effects of a previous request. To support this, we extend MuCache's dependencies (Sec. 4) to client requests. Specifically, when a client request is complete, visited services are included in the result, and passed to the subsequent request of the same client (if one is performed), allowing MuCache to avoid violating dependencies across client requests.

Supporting third-party services. Microservice applications often perform requests to third-party services that might not be extensible with MuCache, e.g., if they are owned by a different organization. To support such applications, MuCache allows declaring requests to third-party services as read-only using a TTL, saving their values to the cache on return, but invalidating them when the TTL has passed instead of waiting for a downstream cache manager. This setup provides caching benefits with at least as strong guarantees as if all the caches in the application were configured with a TTL, however for the complete subtrees of the microservice graph that are MuCache-enabled the guarantees are stronger.

7 Evaluation

Our evaluation aims to answer these high-level questions:

- **(Q1) Throughput and latency benefits:** Does MuCache provide throughput and latency benefits compared to other caching alternatives? Does it scale with sharding? How do cache sizes affect its performance? How are its benefits affected by the application call-graph? (§7.3)
- **(Q2) Costs:** What are the costs of deploying MuCache? What is its CPU and memory usage, total network costs, and its latency overhead on the critical path? Does the cache manager throughput become a bottleneck? (§7.4)
- **(Q3) Invalidation:** How fast can MuCache invalidate cache entries? (§7.5)

Before we answer these questions, we describe the experimental setup (§7.1) and our methodology and baselines (§7.2).

7.1 Experimental Setup

We deploy a Kubernetes [7] cluster on CloudLab [2] m510 machines that have 8-core 2.0 GHz CPUs, 64GB RAM, 256GB NVMe SSDs, and 10GB NICs. Machines run Ubuntu 20.04. The average round-trip time between servers is 0.15ms. Ex-

cept for sharding experiments, we utilize a single Kubernetes cluster where the number of worker nodes is equal to the number of services, plus one node acting as a control plane. Each service is deployed via Dapr [3] and is affinitized to a single node. We use Redis [9] configured with an LRU eviction policy as the cache. Unless otherwise noted, MuCache is configured with a sending batch size of 20 and a 1 ms timeout. Cache manager dependencies are stored in a LRU cache, with the maximum number of entries being proportional to the user cache size. In our experiments, the cache manager stores 100 dependencies per 1 MB of user cache (e.g., a user cache of 20 MB allows the storage of 2,000 dependencies).

7.2 Applications, Method, and Baselines

Throughout our evaluation we perform experiments on four open-source microservice applications, as detailed in Figure 8, along with four synthetic ones. Workloads are adapted from the original testbeds, including the dataset and request distribution. Cache sizes are set relative to the application working data set; small enough that they do not fit the entire working data set but big enough so that there is a non-negligible amount of cache-hits.

SocialMedia. A social network application (Cf. Twitter or Facebook) that provides three main endpoints, viewing a user's homepage timeline (RO), viewing a user's personal timeline (RO), and composing a post. The workload ratio is 60% homepage, 30% user timeline, and 10% compose post. The cache size for each service is set to 20 MB. When there are no new posts and each timeline contains 10 posts, the total cacheable posts are around 20 MB.

MovieReview. A movie review application (Cf. IMDB or Rotten Tomatoes) that offers two main endpoints: viewing the page of a movie (RO) and creating a review. The workload ratio is 90% viewing a page and 10% creating reviews. The cache size for each service is set to 70 MB.

HotelRes. A hotel reservation application (Cf. Booking or Airbnb) that offers two main endpoints: searching for hotels in a specific area (RO) and making a reservation. The workload ratio is 80% searching for hotels and 20% making a reservation. The cache size for each service is set to 20 MB.

OnlineBoutique. An online store application (Cf. Amazon or Walmart) that offers multiple endpoints, retrieving the store homepage (RO), updating the currency rate, viewing a product (RO), adding a product to the cart, and checking out. The workload ratio is 75% read-only (homepage, viewing products, and carts) and 25% non-read-only (updating the currency, updating the cart, checking out). The cache size for each service is set to 80 MB.

Synthetic Benchmarks. Figure 9 shows four synthetic applications: ProxyApp, a two-service app where a stateless frontend forwards requests to the backend, which in turn reads/writes to a key-value store; and three applications that extend ProxyApp with archetype call-graph patterns—chain, fan-out, and fan-in. ChainApp has four stateless services and a stateful backend. FanoutApp has a single frontend forwarding requests to four backends. FaninApp has four separate frontends, each forwarding requests to one backend.

	Benchmark	Services	LoC	RO/NonRO	Sources
1	SocialMedia	6	532	90/10	[10, 24, 32]
2	MovieReview	12	913	90/10	[13, 24]
3	HotelRes	6	608	80/20	[24]
4	OnlineBoutique	9	1,088	75/25	[8]

FIGURE 8— Real-world applications used in our evaluation.

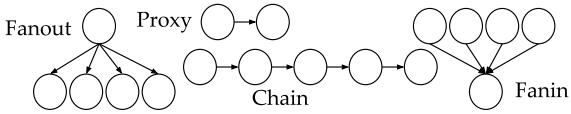


FIGURE 9—Shapes of synthetic benchmark call-graphs.

Method. We measure throughput and latency (median and 95th percentile) using the wrk2 [15] HTTP benchmarking tool. Experiments include a 30-second cache pre-warming period, followed by a 60-second testing period. Each experiment is run three times, and the average is reported. We run MuCache and the baselines with the same CPU resources; that is, MuCache's cache managers are not given extra cores but share resources with the application.

Baselines. We compare MuCache to the following baselines.

BC (Backend Cache): A baseline that lacks inter-service caching and only caches data from the backend datastore.

TTL: A baseline that reflects the current best practices for automated inter-service caching [1, 27, 33]. Caching occurs at both the backend and intermediate services. Upon invocation, the caller saves the result in the cache asynchronously without communicating with any cache manager. The caches can then evict an entry when they become full or, in the case of an inter-service cache, when a configured time-to-live (TTL) timer has expired. Cached data can be *inconsistent* and *arbitrarily stale* (depending on the TTL and access pattern).

TTL-∞: A special case of TTL that serves as an upper bound on the performance achievable by TTL implementations; cache entries never expire and are only evicted when the cache reaches maximum capacity.

7.3 (Q1) Throughput and Latency Benefits

We first measure the throughput and latency of a set of real-world applications with and without MuCache (§7.3.1). We then compare it against different TTL baselines (§7.3.2), we evaluate whether it limits throughput scalability in the presence of sharding (§7.3.3), and we evaluate whether configuring caches with different sizes and whether different application call-graphs affect MuCache's benefits (§7.3.4–7.3.5).

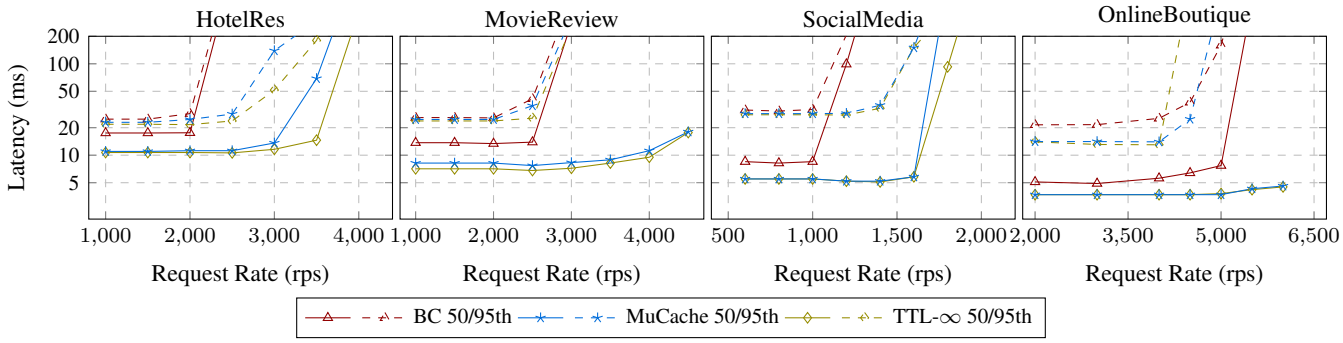


FIGURE 10—Latency and throughput of real-world applications (described in Figure 8).

7.3.1 Real-world applications

We evaluate MuCache's benefits on throughput and latency on the four open-source microservice applications. We compare MuCache against (1) BC to evaluate performance benefits over not having inter-service caches, and (2) TTL-∞ to evaluate how close MuCache is to an implementation that caches results but provides no consistency guarantees.

Results. Figure 10 shows the results, where the X-axis is request rate, and the Y-axis shows latency in ms. MuCache reduces median latency by up to 1.8× in HotelRes, 2.5× in MovieReview, 1.5× in SocialMedia, and 2.1× in OnlineBoutique. The tail latency between MuCache and BC is similar, except for OnlineBoutique, where MuCache reduces tail latency by up to 1.8× by avoiding many invocations from the Checkout service, such as retrieving product information, getting shipping quotes, etc. Furthermore, MuCache improves throughput by 1.6× in HotelRes, 1.5× in MovieReview, and 1.4× in SocialMedia, while achieving similar throughput in OnlineBoutique. Compared to TTL-∞, MuCache's median latencies are up to 1.2× higher before saturation, and MuCache's throughput is around 0.95×.

Take away. MuCache outperforms BC in terms of median and tail latency, and throughput across all workloads. MuCache also performs close to the upper bound TTL-∞. Improvements in median latency can be attributed to cache hits, while improvements in throughput are due to lower utilization of backend services.

7.3.2 Comparison with TTL baselines

Tuning TTL values for caches in real systems is complex and depends on the application requirements; suggested values could range from seconds to hours [18, 23]. To simulate that in a shorter experiment, we vary TTL from 100 ms to 10 s—values under 100 ms lead to negligible cache hits, and a TTL of 10 s is already a large fraction of the total experiment (60 s).

Results. Figure 11 shows the results. As the TTL increases from 0.1 to 10 s, median latency drops from 18.2 ms to 10.9 ms, tail latency drops from 29.3 ms to 10.9 ms, and throughput increases from 2,489 to 3,470 rps. MuCache outperforms TTL-1s (1.3× lower median latency), but is outperformed by TTL-10s (which performs similarly to TTL-∞).

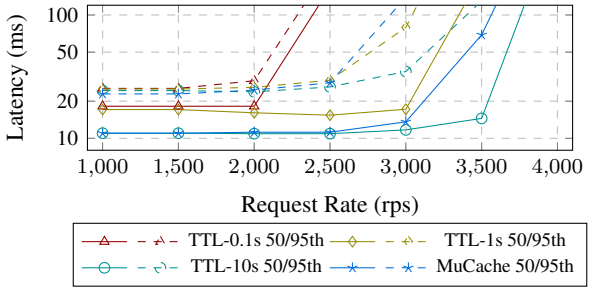


FIGURE 11—HotelRes: Latency and throughput of MuCache compared with various TTL.

Take away. Getting comparable performance to MuCache with a TTL-based caching approach requires setting the TTL to a high value (>1 s)—orders of magnitudes higher than the MuCache invalidation times (on the order of ms per call-graph depth as shown in Section 7.4.3). Furthermore, finding an appropriate TTL value is challenging for developers, as this value has implications for the correctness of the application. In contrast, MuCache requires no tuning of expiration times, and invalidations happen automatically and correctly.

7.3.3 Sharding Scalability

We evaluate the scalability of MuCache by deploying SocialMedia to multiple shards. We provision a fixed pool of machines and restrict each shard to a fixed CPU usage of 2 cores (1 running the service, 1 running the Dapr sidecar) to have multiple shards on a single machine. Each shard is deployed with its own cache manager. We compare against BC to determine whether MuCache limits scalability.

Results. Figure 12 shows the maximum throughput of the SocialMedia when deployed using 1, 2, and 4 shards, with and without MuCache. MuCache scales as well as BC (achieving 1.44×, 1.38×, and 1.37× the throughput of BC).

Take away. MuCache does not limit scalability for sharded applications as the only cost occurs in the background; when the cache manager of a shard broadcasts received writes to all cache managers that belong to the same service shards.

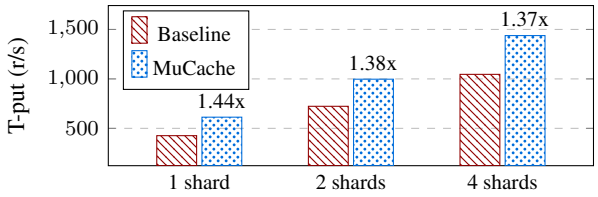


FIGURE 12—Throughput of MuCache and BC when sharding the services in SocialMedia.

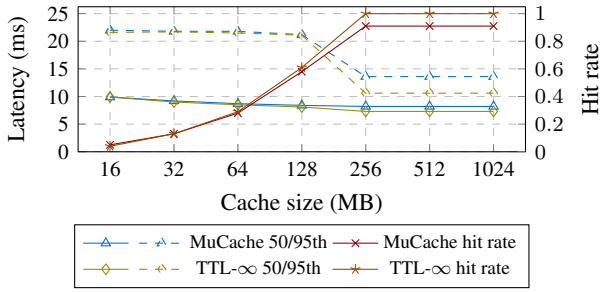


FIGURE 13—HotelRes: Impact of different cache sizes on latency (left Y-axis) and combined cache hit rate (right Y-axis).

7.3.4 Cache size effect

To evaluate how MuCache responds to the cache size of each service, we measure latency and cache hits on HotelRes with a fixed load of 1K req/s while varying the cache size from 16 MB to 1024 MB. TTL-∞ acts as an upper-bound baseline.

Results. Figure 13 shows the results. Increasing the cache size lowers the median latency of MuCache from 9.9 ms to 8.2 ms and tail latency from 22 ms to 13.6 ms; it also increases the cache hit rate from 5% to 91%. Similarly, in TTL-∞, the median latency decreases from 9.9 ms to 7.3 ms, tail latency from 21.6 ms to 10.6 ms, and cache hit rate from 5% to 100%.

Take away. Caching with MuCache reduces mean and tail latency. Furthermore, the reductions achieved by MuCache are close to those achieved by TTL-∞ across all cache sizes.

7.3.5 Application call-graph effect on performance

To evaluate how the application call-graph pattern affects the benefits of MuCache, we use the three synthetic applications in Figure 9. We use a synthetic workload with 50% cache hit rates and compare against BC.

Results. Figure 14 shows the results. For ChainApp, MuCache's median latency is 2.6–3.1× lower than that of BC, while its tail is comparable before reaching saturation. Its maximum throughput is 1.5× higher. For FanoutApp, the median latency and maximum throughput of MuCache are similar to that of the BC, but its tail latency is up to 1.6× lower. In FaninApp, MuCache improves median latency by 1.1–1.3× and 95th percentile latency by up to 1.9×; maximum throughput is 1.75× higher than BC.

Take away. MuCache provides different benefits depending on the call-graph shape. For long call-chains MuCache reduces latency by avoiding network hops; for fan-out it slightly

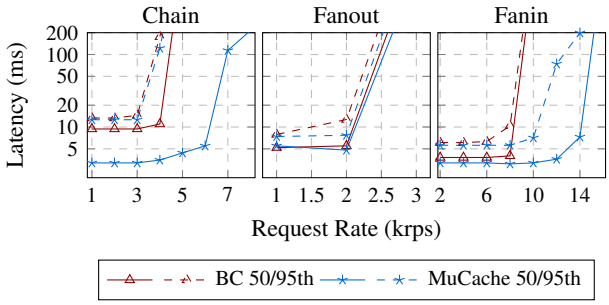


FIGURE 14—Latency and throughput of the graph shape microbenchmarks (Fig. 9).

	Benchmark	Average (MB)	Max (MB)	Cache Size (MB)
1	HotelRes	0.08	0.27	20
2	MovieReview	0.07	0.31	70
3	SocialMedia	0.02	0.09	20
4	OnlineBoutique	0.1	0.45	80

FIGURE 15—Cache manager state and cache size for each service.

improves tail-latency but not median latency since the frontend has to wait for the slowest path to respond; and when the backend is the bottleneck it improves throughput by reducing the number of requests that reach the backend.

7.4 (Q2) MuCache costs and overheads

In order to evaluate the costs of MuCache, we measure its CPU, memory, and network usage (§7.4.1), its latency overhead on the critical path (§7.4.2), and the cache manager's throughput and whether it can be a bottleneck (§7.4.3).

7.4.1 Memory / CPU / Network costs

We evaluate MuCache's memory cost on all four applications and its CPU and network usage on HotelRes. We evaluate MuCache's network usage by measuring data transfer between nodes using `iftop`. We measure the memory cost of each cache manager instance as the average size of its state (history and dependencies) and CPU cost as the average CPU usage of each service during the experiment. We use standard cache sizes and load (2K req/s for HotelRes, 2.5K req/s for MovieReview, 1K req/s for SocialMedia, and 3.5K req/s for OnlineBoutique) for 300 seconds.

Results. Figure 15 shows the cache manager state size and the cache size across services. The average size of the CM state across services ranges from 0.1–0.4% of the cache size per service. Garbage collection plays an important role in keeping the memory usage low: without GC, the CM state in HotelRes goes up to 5 MB in 1 minute. Figure 16 shows the average CPU usage of each service during the experiment. Usage is broken down between the service logic, the Dapr sidecar, and the cache manager. The average CPU usage across services with and without MuCache is 4.2 and 5.1 cores respectively. The average CM CPU usage across services is 0.5 cores. The

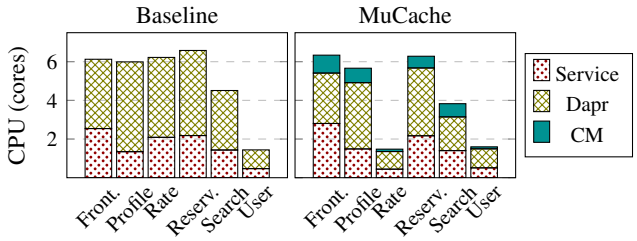


FIGURE 16—CPU usage per service for HotelRes.

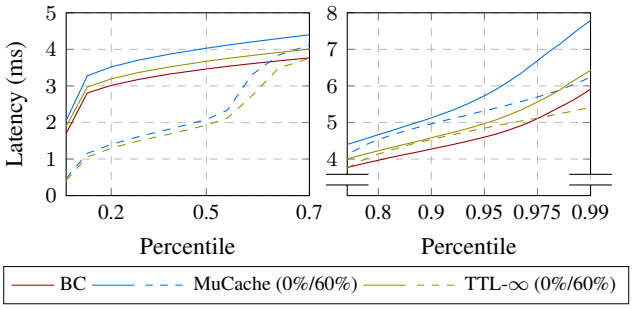


FIGURE 17—Latency distribution w.r.t. hit-rate for ProxyApp. Solid and dashed lines show the latencies when the hit rate is 0% and 60% respectively. Split in 70th percentile for clarity.

average network usage per service without MuCache is 9.0 MB/s, while the average with MuCache is 6.6 MB/s, of which cache managers use 2.9 MB/s.

Take away. Memory costs are low compared to the cache size (<0.4% on average). The CPU usage of MuCache is 13% of the total service CPU on average while at the same time reducing the total CPU usage of the whole application due to some backend services being less utilized because of cache hits in the frontend. Though cache managers use some bandwidth to save/invalidate caches, MuCache reduces the total network usage by 27% due to local cache hits.

7.4.2 MuCache latency overhead

We evaluate MuCache's latency overhead by focusing on ProxyApp, which performs minimal work, to measure the worst-case overhead. We create a synthetic workload with 0% and 60% cache hit rates and compare against (1) BC to evaluate the overhead over no caches when there are no hits, and (2) TTL-∞ to evaluate the wrapper overhead.

Results. Figure 17 shows the complete request latency distribution. We report overheads as absolute values because they are constant and independent of the work that the services do. For a hit rate of 0%, MuCache's median latency (4 ms) is 0.5 ms higher than BC and 0.3 ms higher than TTL-∞, while the 95th-percentile (5.7 ms) is 0.9 ms and 0.5 ms higher respectively. When the hit rate is 60%, MuCache's median and 95-th percentile latencies are 0.15 ms and 0.5 ms higher than TTL-∞. When the hit rate is 60%, MuCache median latency is 1.4 ms better than BC (3.5 ms to 2.1 ms).

Take away. Even in a worst-case scenario (an application that performs minimal work), MuCache imposes a low ($\sim$10%) latency penalty on cache misses.

Batch Size	1	2	5	10	20	50
Throughput (krps)	19.2	26.5	44.5	74.7	75.2	74.6

FIGURE 18—Batching effects on cache manager throughput.

Chain Size	2	3	4	5
Mean invalidation time (ms)	3.94	6.13	8.41	10.66

FIGURE 19—Invalidation time for different chain sizes.

7.4.3 MuCache's throughput

To determine whether MuCache's cache manager can be a bottleneck, we measure the its maximum throughput on the ProxyApp and load the backend's cache manager directly because the backend service becomes the bottleneck otherwise. The load is 80% read-only requests and we vary the batch size of the HTTP sending buffer between cache managers.

Results. Figure 18 shows the throughput in terms of the number of events the cache manager processes per second. Without batching, the cache manager has a throughput of $\sim$19K events per second, while gradually increasing the batch size up to 20 improves it to $\sim$75K events per second.

Take away. The cache manager has a reasonably high throughput and is not the bottleneck even for an application with minimal computation. To further increase throughput, developers may deploy multiple shards for each service.

7.5 (Q3) Invalidation time

We evaluate the time needed for invalidations to reach the root of the call-graph, namely the frontend service, by measuring the observed inconsistency window [19], the elapsed time between the write happening in the backend and the invalidation becoming visible in the frontend. The invalidation time in our experiment is determined solely by the depth of the call graph. To measure the increase in invalidation time per hop, we conducted experiments on a microservice chain consisting of 2 to 5 services, which represents the typical depths of call-graphs in the applications that we studied.

Results. Figure 19 shows the results. For a two-service application, the invalidation time is $\sim$4 ms; for a five-service application it is $\sim$10 ms. Each additional service in the chain increases invalidation time by $\sim$2.2 ms.

Take away. MuCache's invalidation time is $\sim$2.2 ms per call-graph hop—orders of magnitude smaller than the typical invalidation times observed in TTL-based approaches (which range from seconds to hours [18, 23]).

8 Related Work

Caching in microservice applications. Several works study cache usage in real-world microservices, including work from Alibaba [28], Twitter [38], and Facebook [37]. These papers

confirm that caches are heavily used in microservice applications and provide significant performance benefits, but only mention manual, ad-hoc, or inconsistent coherence schemes and do not propose an automatic way to manage these caches.

Caching frameworks for web services. There is a lot of work on caching frameworks for web services for both static and dynamic data. These frameworks focus on three key aspects: (1) content admission, (2) cache size management, and (3) invalidation and data freshness (for a more detailed classification see a recent survey [29]). The first two aspects are orthogonal to our work since we do not focus on optimizing the performance of a cache given a specific workload, but rather propose a general system for keeping caches coherent in a microservice setting. To the best of our knowledge, all frameworks that focus on invalidation (e.g., [21, 22, 31]) are designed as a single cache layer on top of a database without taking into account the inter-service caching.

Cache coherence protocols. There is extensive literature on cache coherence protocols (see survey [34]), none of which considers inter-service caching. Lazy caching [17] exploits the fact that writes do not always require exclusivity (M or E in MOESI [35]), allowing cores to perform concurrent buffered writes, albeit blocking reads to ensure that dependencies are not violated. Our work extends this insight by avoiding all blocking communication on the request's critical path—allowing writes downstream without immediately informing the upstream caches and without blocking on reads.

Incremental computation. Caches are also used to enable incremental and reactive computation: some examples include Reactive Caching [20], Noria [25], and Diamond [39]. Reactive Caching proposes caches for graphs of single-threaded services to support reactive computation, i.e., writes downstream are propagated upstream to refresh the results. Noria is an incremental stream processing engine that uses caches for fast propagation of updates in a dataflow. Both differ from our work in two ways: (1) they only provide eventual consistency that violates dependencies when there are multiple paths between two services (see Fig. 4); and (2) they do not support true multi-threading, as Noria limits writes to a single thread and Reactive Caching only supports single-threaded services. Diamond is a system that automates data management for distributed reactive applications by providing reactive transactions to clients. Similarly to MuCache, Diamond reactively informs clients about data invalidations in the backend store, but in contrast to MuCache it does not support service graphs.

9 Discussion and Limitations

Supporting transactions and non-KV stores. Our implementation does not currently support transactions or non-KV stores. Supporting single-service transactions would require that the wrappers perform the `postWrite` after the transaction has completed to overapproximate the time that the write operation completed. Supporting multi-service transactions would be more challenging since caches should not violate transactional guarantees, which would require additional synchronization in the protocol. Supporting non-KV stores, such as relational databases, would require monitoring the dependencies of read-only calls and determining when to invalidate cache entries, which could be done by leveraging the expressive semantics of SQL (as in the case of Noria [25]).

Supporting weaker consistency datastores. The correctness of MuCache depends on the datastores being linearizable; MuCache needs to be sure that after a write has completed, it has taken effect in the database. Being able to determine the order of reads and writes by intercepting the datastore accesses is necessary so that MuCache is database-agnostic (see requirements in Section 3). Supporting weaker consistency datastores would likely require a more intrusive design with modifications to a datastore—tightly integrating wrappers in the store to provide additional metadata to the cache managers about the precise order of reads and writes—forfeiting the generality of being database-agnostic.

Application debuggability. Extending an application with MuCache provides performance benefits and does not affect the application behavior but adds complexity to the end-to-end deployment and therefore increases the effort required to maintain and debug it. This is an inherent software engineering challenge—the bigger a codebase is, the harder it is to maintain it. A direction for future work that could help address this is to integrate MuCache with existing distributed tracing and debugging tools for microservices, so that engineers have visibility on MuCache's state and actions.

Write-intensive workloads. Even though a service might offer a read-only endpoint, its workload might be write-intensive, leading to overheads without the accompanied benefits if extended with MuCache. Developers can currently manually detect such cases and avoid declaring those endpoints as read-only, but it would be interesting to explore whether MuCache can be extended with an adaptive monitoring mechanism that only enables caching if the read-write ratio of a service is above some threshold.

Sharding. MuCache requires hard affinity sharding of read requests to ensure correctness, i.e., all read-only calls with the same arguments need to be processed by the same shard. Write requests have no such limitation and can be dispatched to any shard. An interesting avenue for future research would be to lift the requirement for hard affinity, allowing for more flexible load balancing and autoscaling.

Acknowledgments

We thank the NSDI 24 and SOSP 23 reviewers, our shepherd, Gábor Rétvári, as well as Achilles Benetopoulos, Akis Giannoukos, Jiali Xing, Nathaniel Hoaglund, and Nikos Vasilakis, for discussions and feedback on the paper. This work was partially supported by NSF awards CCF 2124184, CNS 2107147, and CNS 2321726.

References

[1] Caching Guidance - Azure Architecture Center. https://learn.microsoft.com/en-us/azure/architecture/best-practices/caching.

[2] CloudLab - A testbed for cloud computing research. https://www.cloudlab.us/.

[3] Dapr - Distributed Application Runtime. https://dapr.io/.

[4] Envoy Proxy. https://www.envoyproxy.io/.

[5] From Monolith to Microservices: How to Scale Your Architecture. https://www.youtube.com/watch?v=N1BWMW9NEQc.

[6] Istio Service Mesh. https://istio.io/latest/about/service-mesh/.

[7] Kubernetes - An open-source container orchestration system. https://kubernetes.io/.

[8] Online Boutique – Microservices Demo. https://github.com/GoogleCloudPlatform/microservices-demo.

[9] Redis - An open-source in-memory data store. https://redis.io/.

[10] Rutgers Social Network Graph. https://networkrepository.com/socfb-Rutgers89.php.

[11] The FNV Non-Cryptographic Hash Algorithm. https://datatracker.ietf.org/doc/html/draft-eastlake-fnv-17.html.

[12] The Go programming language. https://go.dev/.

[13] The Movie Database. https://www.themoviedb.org/.

[14] Twitter's recommendation algorithm. https://github.com/twitter/the-algorithm.

[15] wrk2: A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2.

[16] ZeroMQ - An open-source universal messaging library. https://zeromq.org/.

[17] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1993.

[18] AWS. Caching Best Practices. https://aws.amazon.com/caching/best-practices/, 2023.

[19] David Bermbach and Stefan Tai. Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. In *Workshop on Middleware for Service Oriented Computing (MW4SOC)*, 2011.

[20] Sebastian Burckhardt and Tim Coppieters. Reactive caching for composed services: polling at the speed of push. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2018.

[21] K Selçuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling dynamic content caching for database-driven web sites. In *Proceedings of the ACM SIGMOD Conference (SIGMOD)*, 2001.

[22] Jim Challenger, Arun Iyengar, and Paul Dantzig. A scalable system for consistently caching dynamic web data. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 1999.

[23] Cloudflare. Edge and Browser Cache TTL. https://developers.cloudflare.com/cache/how-to/edge-browser-cache-ttl/, 2023.

[24] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[25] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M Frans Kaashoek, and Robert Tappan Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[26] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), July 1990.

[27] Joydip Kanjilal. Scaling microservices architecture using caching. https://www.developer.com/design/scaling-microservices-using-cache/, 2021.

[28] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2021.

[29] Jhonny Mertz and Ingrid Nunes. Understanding application-level caching in web applications: a comprehensive introduction and survey of state-of-the-art approaches. In *ACM Computing Surveys (CSUR)*, 2017.

[30] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[31] Dan RK Ports, Austin T Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache.

In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[32] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the AAAI conference on artificial intelligence (AAAI)*, 2015.

[33] Irfan Saleem, Pallavi Nargund, and Peter Buonora. Data caching across microservices in a serverless architecture. https://aws.amazon.com/blogs/architecture/data-caching-across-microservices-in-a-serverless-architecture/, 2008.

[34] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. In *IEEE Computer*, 1990.

[35] Paul Sweazey and Alan Jay Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. In *ACM SIGARCH Computer Architecture News (SIGARCH)*, 1986.

[36] Alex Xu. Twitter architecture 2022 vs. 2012. what's changed over the past 10 years?, Nov 2022.

[37] Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Characterizing Facebook's Memched Workload. In *IEEE Internet Computing*, 2013.

[38] Juncheng Yang, Yao Yue, and KV Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[39] Irene Zhang, Niel Lebeck, Pedro Fonseca, Brandon Holt, Raymond Cheng, Ariadna Norberg, Arvind Krishnamurthy, and Henry M Levy. Diamond: Automating data management and storage for wide-area, reactive applications. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[40] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. CRISP: Critical path analysis of Large-Scale microservice architectures. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2022.

A Detailed Protocol Correctness

Preliminaries. We start with some basic notation:

- n_S denotes a service name
- ca denotes the arguments of a service call, including the name of the service (which can be extracted using name(ca)) and the endpoint.
- $i \in R$ denotes request identifiers (each request has a unique i). The service name and the arguments to the request can be extracted using name(i) and ca(i). We will define a binary relation $sr \subseteq R \times R$ that determines when a request is spawned by another request. We can also define sr^* as the reflexive transitive closure of sr. There is also a client(i) predicate, which returns true for requests that are initiated by a client.
- v denotes a return value
- k denotes a key that indexes values in the state of a service
- rs(i, t) is a function that returns all of the keys that a particular request (and all of its subrequests) have read in trace t. We will often ignore t when it is obvious which trace we refer to.

Events and traces. We will describe microservice applications and their executions using traces, i.e., sequences of events that describe application actions. We are only interested in events that describe interactions between services and other services and actions on their states. We call the set of all events Σ, and we now define all events in it.

- Req$_i(ca)$ denotes the start of processing of a single request with id i and arguments ca.
- Ret$_i(v)$ denotes that a request with id i has finished processing and is returning value v.
- Read$_i(k, v)$ denotes that request with id i performed a read of key k from its state and returned v.
- Write$_i(k, v)$ denotes that request with id i performed a write with value v to key k of its state.
- Call$_i(ca, i')$ denotes that request with id i performed a call to another service with arguments ca and the request id of that internal request is i'.
- Resp$_i(v, i')$ denotes that request with id i received a response with value v from a finished call with id i'.

We represent the set of all events for a request with identifier i as Σ_i and the set of all read (or write) events as Σ_R (or Σ_W). We also define a set of output events $\Sigma_O = \{\text{Ret}_i(v) : \forall i, v\} \cup \{\text{Read}_i(k, v) : \forall i, k, v\} \cup \{\text{Write}_i(k, v) : \forall i, k, v\} \cup \{\text{Call}_i(ca, i') : \forall i, ca, i'\}$ that are events that are determined by the program when processing a single request, and input events $\Sigma_I = \{\text{Req}_i(ca) : \forall i, ca\} \cup \{\text{Resp}_i(v, i') : \forall i, v, i'\}$ that are events that are given as inputs to the processing of a single request. Finally, we can define the set of client events $\Sigma_C = \{\text{Req}_i(v) : \forall i, \text{client(i)}\} \cup \{\text{Ret}_i(v) : \forall i, \text{client(i)}\}$ We can now describe complete executions of microservice applications using traces t, i.e., sequences of the above events. We can project all events of a trace t from a particular set Σ using $t[\Sigma]$, e.g., $t[\Sigma_W]$ are all the write events in a trace. Note that this projection creates an ordered sequence of events by maintaining the trace order.

Applications and Assumptions. We can now define the behavior of a microservice application $P \in \mathbb{P}$ using its execution traces, $[\![P]\!] \subseteq \Sigma^*$, and state some assumptions on these traces. First of all, an application determines the processing of each request using the step : $\mathbb{P} \times R \times \Sigma^* \times (\Sigma_O \cup \{\bot\})$ relation, that determines the next step of the processing of a request, or $\bot$ if the request is waiting for a response or hasn't started yet. Now we define what it means for a trace to be well formed.

Property 2 (Well-formed traces)**.** *All traces $t \in [\![P]\!]$ are well-formed, i.e., for each trace t the following properties hold: (1) Req$_i(ca)$ are the first events for any request i and Ret$_i(v)$ are the last; (2) for each $i \in t$ there exists a unique Req$_i(ca)$ and at most one Ret$_i(v)$; (3) a Req$_i(ca)$ always comes after a Call$_i(ca, i')$ except in the case of client requests; (4) a Resp$_i(v, i')$ always comes after a Call$_i(ca, i')$ and Ret$_{i'}(v)$; (5) for all Call$_i(ca, i')$, $sr(i, i')$; and for all prefixes $t' = t_0.e$ with $e \in \Sigma_I$, either step(P, i, t_0, e) or step$(P, i, t_0, \bot)$; (6) for all $e \in \Sigma_C$ for any $i \in t$, s.t. client(i) holds, then $\nexists$Call$_{i'}(ca', i) \in \Sigma_C$.*

The last requirement relates the step relation with the traces, i.e., each event in the trace is the result of stepping a request or a request start or response. We also know that the events in a trace are equivalent up to an injective renaming of request identifiers.

Property 3. *For any microservice application P, for all traces $t \in [\![P]\!]$ and for all $i \in t$, then for any $i' \notin t$ we can construct a new trace $t' = t\,[i \mapsto i']$, s.t. $t' \in [\![P]\!]$.*

In addition to the above, we also know that requests are always enabled in microservice applications, i.e., a pending request can always take a step.

Definition 1 (Pending Requests)**.** *We say that a request Req$_i(ca)$ is pending in a trace t iff Ret$_i(v)$ does not exist in t.*

Property 4 (Request Step Always Enabled)**.** *For any microservice application P, for all traces $t \in [\![P]\!]$, and for all pending requests Req$_i(ca)$ for some i, there exists a trace $t' \in [\![P]\!]$ such that $t' = t.e_i$, where $e_i \in \Sigma_{i'}$ and $sr^*(i, i')$.*

Property 4 means that requests are always enabled to take a step, sometimes through their subrequests. This is a valid assumption for microservice applications since they are multithreaded, and therefore a single request can not block other requests from proceeding, and a request can only block while waiting for a response from its subrequests. Note that this assumption requires that the network does not drop requests, i.e., calls eventually lead to request starts and that returns eventually lead to response events.

We also know that the values of read events depend on the latest write to the same key or the original value.

Property 5 (Read return value)**.** *For all applications P, request identifiers i and traces t s.t. step$(P, i, t,$ Read$_i(k, v))$ holds, then either $\exists i'$, Write$_{i'}(k, v) =$ last$(t[\Sigma_{W(k)}])$ or $v = \bot$.*

Intuitively, this means that writes are immediately visible to reads, thus that the underlying stores are linearizable, which is a valid assumption for most key value stores.

We can now define read-only calls, that is calls that never perform writes (even in their subrequests).

Definition 2 (Read-only requests)**.** *Given an application P a request with request identifier i and call arguments ca, i.e. $ca(i) = ca$, is read-only for this application iff for all traces $t \in [\![P]\!]$, and for all i' such that $sr^*(i, i')$, it holds that $t[\Sigma_W \cap \Sigma i'] = \emptyset$. We define a predicate $RO(i)$ that holds for read-only requests.*

State. We represent the state of an application as $\sigma \in \mathbb{D}$. Concretely, a state σ is a tuple of maps from keys to values, one for each service. We define the function $S : \Sigma^* \to \mathbb{D}$ that returns the state of an application after the trace t. Due to Property 5 the state at each point in the execution depends on the prefix of write events and the starting state. We assume that all executions start from the same starting state σ_0.

Caching. Up to this point we have established all important properties of microservice applications without mentioning caches. A cache-enabled application $\widetilde{P}$ can be similarly defined by its execution traces, $[\![\widetilde{P}]\!] \subseteq \widetilde{\Sigma}^*$, where $\widetilde{\Sigma}^*$ is a superset of the set of events of applications without caches, i.e. $\Sigma \subseteq \widetilde{\Sigma}$. The additional cache related events are defined as followed:

- CacheHit$_i(ca, v)$ denotes a cache-hit that replaces a Resp$_{i'}(v, i)$ for some i' (also conforming to its well-formedness conditions Property 2).
- Save(n_S, i, v) denotes that the cache of service n_S has saved the value v for request i with call arguments ca(i).
- Inv(n_S, i, i') denotes an invalidation of the cache of service n_S with ca(i') from a write with identifier i.

Essentially, a cache-enabled application $\widetilde{P}$ is a transformation of a regular microservice application P. We know that our protocol does not affect the stepping of requests other than allowing some calls to return immediately with call hits. We can also lift the step relation to account for cache-enabled applications. The lifted step relation describes the logic of our cache coherence protocol.

Property 6 (Cache Stepping)**.** *For any application P the transformed $\widetilde{P}$ can step, i.e. step$(\widetilde{P}, i, t, e)$ holds, if*

- *step(P, i, t, e) when $e \in \Sigma$ or*
- *$e =$ Save(n_S, i', v) and $\exists$Ret$_{i'}(v) \in t$ or*
- *$e =$ Inv(n_S, i', i'') and $\exists$Write$_{i'}(k, v) \in t$ with $k \in$ rs(i'').*
- *$e =$ Inv(n_S, i', i'') and $\exists$Inv$(n_S, i', i''') \in t$ with ca$(i''') \in$ rs(i'').*
- *$e =$ CacheHit$_i(ca, v)$ and $\exists$Save$($name$(i), i', v) \in t$ and $\nexists$Inv$($name$(i), i'', i''') \in t$ and $ca = ca(i') = ca(i''')$ between the save and the cache-hit.*

Intuitively, Property 6 means that the cache-enabled application does not affect the next steps of any specific request other than sometimes finding a result in the cache.

Definition 3 (Dependency)**.** *We say that event $e' \in \Sigma_{i'}$ is a dependency of $e \in \Sigma_i$ in a trace t if e' is after e and if either:*

- *$i = i'$, i.e. the two events are part of the same request*
- *$e =$ Call$_i(ca, i')$ and $e' =$ Req$_{i'}(ca)$ $i \neq i'$ and $sr^*(i, i')$, i.e., the second event is a part of a subrequest of the first event*
- *$e =$ Ret$_i(v)$ and $e' =$ Resp$_{i'}(v, i)$, i.e., the events are a pair of return and handle response.*
- *$e =$ Write$_i(k, v)$ and $e' =$ Read$_{i'}(k, v')$ or $e =$ Write$_i(k, v)$ and $e' =$ Write$_{i'}(k, v')$ or $e =$ Read$_i(k, v)$ and $e' =$ Write$_{i'}(k, v')$, i.e., read and write events to a key k are dependencies of a prior write to the k and write events are dependencies of a prior read.*
- *$e =$ Ret$_i(v)$ and $e' =$ Save(n_S, i, v) for some n_S*
- *$e =$ Write$_i(k, v)$ and $e' =$ Inv(n_S, i, i') for some n_S*
- *$e =$ Inv(n_S, i', i''') and $e' =$ Inv(n_S, i', i'') with ca$(i''') \in$ rs(i'').*
- *$e =$ Save$($name$(i), i', v)$ and $e' =$ CacheHit$_i(ca, v)$ where ca$(i') = ca$*
- *$e =$ Save(n_S, i', v) and $e' =$ Inv(n_S, i, i'') for some i and ca$(i') =$ ca(i'')*

We will use deps(e) to refer to all the transitive dependencies of an event e. We now state a final assumption on application traces, namely that two independent events can be commuted.

Property 7 (Commute independent events)**.** *For any trace $t \in [\![P]\!]$ with $t = t_0.e.e'.t_1$ and $e' \notin deps(e)$, then $t' = t_0.e'.e.t_1$ can also be observed by the application, i.e. $t' \in [\![P]\!]$.*

This holds because in microservice applications independent requests do not affect each other except through reads and writes to the same key in the same service datastore.

We are now ready to state the main theorem that describes the correctness of our protocol.

Theorem 8 (Protocol Correctness (corresponds to Theorem 1))**.** *For all traces t in a cache-enabled application $[\![\widetilde{P}]\!]$, there exists a trace t' in the original application without caches $[\![P]\!]$, such that their respective client events are equivalent (but potentially reordered), i.e., $\forall i\ t[\Sigma_{C(i)}] = t'[\Sigma_{C(i)}]$.*

This makes sense because correctness is only relevant from the perspective of the clients and not all of the internal events that an application performs. Actually, the cache implementation does not contain the same traces because some calls return immediately on cache-hits without triggering all the internal events. In order to prove this theorem, we show that something stronger holds, a lemma that is stated below. Before stating it, we need to define what it means for an event in the cache-enabled event set to be equivalent to the original one.

Definition 4 (Equivalent events)**.** *Equivalence between a cache-enabled event e_c and an original event e (denoted with $e_c \simeq e$) is defined as followed:*

- *if $e \in \Sigma$ and $e_c \in \Sigma$ are the same event or*
- *$e_c = CacheHit_i(ca(i'), v)$ and $e = Resp_i(v, i')$*

We can lift the equivalence relation of events to account for sequence of events in a straightforward way.

Lemma 1. *Given an arbitrary trace $t \in [\![\widetilde{P}]\!]$ we can construct a trace $t' \in [\![P]\!]$ such that (i) the states at the end of the traces are the same for both traces, i.e. $S(t) = S(t')$, and (ii) for all i, $t[\Sigma_i] \simeq t'[\Sigma_i]$ modulo the missing events due to the cache-hits.*

At a high-level the proof proceeds by constructing a t' from t in the missing events and also by moving some write events later in the trace. Theorem 8 follows directly from Lemma 1 since client events will be the same in both traces.

Proof sketch. We will proceed by induction on the size of traces and for the inductive case we will focus on the only interesting scenario where the trace t ends with a cache-hit event $\text{CacheHit}_i(ca, v)$, because these are the only events for which the effects of our cache-subsystem are observed by the rest of the application. For illustrative purposes we extend traces with the state of all services σ_n between each event.

$$t = t_0|_{\sigma_n}.\text{CacheHit}_i(ca, v)$$

For this cache-hit to have happened, the step relations implies that there must exist some $\text{Save}(n_S, i', v)$ before it, such that $\text{name}(i') = n_S$. Similarly, for the cache save to have happened, there must have been a completed request with call arguments $ca = \text{ca}(i')$.

$$t = \cdots .\text{Req}_{i'}(ca)|_{\sigma_1}.\cdots .\text{Ret}_{i'}(v)|_{\sigma_2}.\cdots .$$
$$\cdots .\text{Save}(n_S, i', v)|_{\sigma_3}.\cdots|_{\sigma_n}.\text{CacheHit}_i(ca, v)$$

Given Property 2 (extended in a straightforward way to support cache events), we know that $t[\Sigma_{i'}]$ can be produced by the step relation. The inductive hypothesis and the fact that t is finite ensure the equivalence of the traces even in the presence of cache-hits for subrequests of the original request. We will now do a case analysis on the existence of a $\text{Write}(k, v_1)$ where $k \in \text{rs}(i')$ between $\text{Req}_{i'}(ca)$ and $\text{CacheHit}_i(ca, v)$.

No such write exists. If no such write exists, then $\sigma_1|_{\text{rs}(i')} = \sigma_2|_{\text{rs}(i')} = \ldots = \sigma_n|_{\text{rs}(i')}$. Then, we can construct a trace $t_1 \in [\![P]\!]$ using the inductive hypothesis and by replacing $\text{CacheHit}_i(ca, v)$ with $\text{Call}_i(ca, i'')$ for some fresh i'' (due to Property 6).

$$t_1 = \ldots|\sigma_n.\text{Call}_i(ca, i'')$$

Then, given that $\sigma_1|_{\text{rs}(i')} = \sigma_n|_{\text{rs}(i')}$ and that Properties 5 and 3 hold, we can construct the same request steps t_c as in the original trace ($t[\Sigma_{i'}]\,[i' \mapsto i'']$) using the step relation, ending up with a trace $t_2 \in P$ such that:

$$t_2 = t_1.t_c.\text{Resp}_i(v, i'')$$

Since $\text{CacheHit}_i(ca, v) \simeq \text{Resp}_i(v, i'')$ and read-only requests do not modify the state, we are done with this case.

Write exists. We now need to focus on the case where a write $\text{Write}(k, v_1)$ with $k \in \text{rs}(i')$ exists between $\text{Req}_{i'}(ca)$ and $\text{CacheHit}_i(ca, v)$. We can first show that the write is between $\text{Save}(n_S, i', v)$ and $\text{CacheHit}_i(ca, v)$, because if it was earlier, it would have been processed by the cache manager, prohibiting $\text{Save}(n_S, i', v)$ to have happened. However, there could be an invalidate between the cache save and the cache-hit that has originated from a previous write in another service between $\text{Req}_{i'}(ca)$ and $\text{Save}(n_S, i', v)$. We can now use Property 7 to move all writes together with their dependencies to the end of the trace to get a trace t_w.

$$t_w = \cdots .\text{Save}(n_S, i', v)|_{\sigma_3}\cdots|\sigma_n.\text{CacheHit}_i(ca, v).\cdots .t_{wd}$$

where t_{wd} contains all the writes and their dependencies. This is possible because $\text{CacheHit}_i(ca, v)$ is not a dependency of the writes between the save and the cache-hit; if it was, there must have been a subcall to the service where the write happened, which would have been caught by our dependency tracking (see Section 4). Second, all i' events are not dependencies of the writes between $\text{Req}_{i'}(ca)$ and $\text{Save}(n_S, i', v)$ because (1) i' is read-only (so it cannot have performed those writes or subcalls that performed those writes), and (2) the

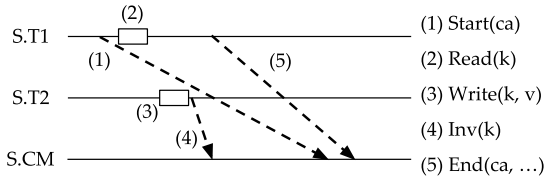


FIGURE 20—A bug that would occur if `preReqStart` does not wait until the `Start` event is added in the CM workqueue.

write happened after the call to start i'.

We can now follow the same reasoning as in the no-write-exists case. We first use the inductive hypothesis on the prefix until the cache-hit and Property 6 to get the following trace:

$$t_1 = \cdots | \sigma_n.\text{Call}_i(ca, i')$$

We then construct the original trace that caused the save using the step relation (like in the no-write-exists case) to get

$$t_2 = \cdots | \sigma_n.\text{Call}_i(ca, i'). \cdots .\text{Resp}_i(v, i')$$

Finally, given that both prefixes and states are the same for t_w and t_2, we can use Property 4 to step all the writes and their dependencies to acquire the same exactly events as the suffix of t_w, proving that the states are the same and the traces for each request in the end are equivalent.

B MuCache protocol design details

B.1 Waiting for events to be added in the queue

It is crucial that the caller waits until the event is added to the queue when sending a `Start` message, otherwise the bug shown in Figure 20 could occur. In this scenario, thread T1 of a service S starts processing a RO request `ca` before waiting for the `Start(ca)` event to be added to the workqueue. In the meantime, another thread T2 of S performs a write which invalidates the results of the call `ca`. However, since the `Start(ca)` event was added in the cache manager workqueue after the `Inv(k)`, the cache manager does not detect the invalidation.

A large-scale deployment of DCTCP

Operational Systems Track

Abhishek Dhamija[1], Balasubramanian Madhavan[1], Hechao Li[2], Jie Meng[1], Shrikrishna Khare[1], Madhavi Rao[1], Lawrence Brakmo, Neil Spring[1], Prashanth Kannan[1], Srikanth Sundaresan[1], and Soudeh Ghorbani[1,3]

[1]*Meta*, [2]*Netflix*, [3]*Johns Hopkins University*

Abstract

This paper describes the process and operational experiences of deploying the Data Center TCP (DCTCP) protocol in a large-scale data center network. In contrast to legacy congestion control protocols that rely on loss as the primary signal of congestion, DCTCP signals in-network congestion (based on queue occupancy) to senders and adjusts the sending rate proportional to the level of congestion. At the time of our deployment, this protocol was well-studied and fairly established with proven efficiency gains in other networks. As expected, we also observed improved performance, and notably decreased packet losses, compared to legacy protocols in our data centers. Perhaps unexpectedly, however, we faced numerous hurdles in rolling out DCTCP; we chronicle these unexpected challenges, ranging from its unfairness (to other classes of traffic) to implementation bugs. We close by discussing some of the open research questions and challenges.

1 Introduction

Congestion control algorithms (CCAs) modulate traffic entry into the network, seeking high utilization, low latency, and relative fairness by making frequent decisions about how much data to send and when. These decisions are based on congestion signals, and the base signal in many CCAs is the *dropped packet.* CCAs tend to increase the amount of data in flight until the point to induce packet loss. Subsequently, they create queue buildup and increase delay. This impacts latency requirements for our datacenter applications. These workloads generate a large number of small request and response flows across the datacenter that, combined, complete a user-requested computation. For a fast response time, each of these short flows should be completed fast.

A class of CCAs [4, 5, 12, 14, 17, 20, 24] tailored specifically to the requirements of datacenters workloads, leverage a diverse set of congestion signals (notifications from the network, measured delay at endpoints, etc.) to detect and react to imminent congestion faster. As one of the earliest and most mature protocols in this class, Data Center Congestion Control (DCTCP) [4] uses Explicit Congestion Notifications (ECNs) from the switches to adjust the sending rate proportional to the level of congestion. ECN remedies large queue buildups and drops by providing a congestion signal before queues overflow, and DCTCP interprets the fraction of ECN-marked packets to scale its response, avoiding persistent queueing and overflows that lead to loss.

A few features of our network made DCTCP's potential worth considering. First, our top of rack switches had "shallow buffers," providing limited space for queueing. Second, buffer sharing and contention over buffers led to variable and unpredictable queue capacities [13]. Finally, our approach to distributing jobs across datacenters created a set of racks with a mixture of large throughput-heavy flows and small latency-sensitive flows that had to compete for the limited and variable buffer space in the network. DCTCP could moderate large flows' use of switch buffers, providing more isolation between jobs sharing shallow-buffered switches.

Using DCTCP required resolving a challenge: designed specifically for short-RTT datacenter traffic, DCTCP had to be applied exclusively to in-region[1] traffic and not cross-backbone traffic. This translates to a few problems: (a) we had to identify in-region traffic, separate it from cross-region traffic, and negotiate DCTCP only for the former. Doing so in a network with the scale and complexity of ours, without risking a broken network, required a large engineering effort, and (b) the in-region DCTCP and cross-region Cubic traffic had to co-exist and share the network. This required careful parameter tuning and network configuration to strike a balance between various classes of traffic. Although challenging, these were surmountable issues that our engineers could overcome to apply DCTCP only to the in-region traffic alongside the Cubic cross-region traffic.

Lawrence Brakmo and Hechao Li contributed to this work during their time at Meta.

[1]A *region* is a collection of datacenters in roughly the same location.

In addition to fitting our workloads' needs, one specific feature of DCTCP made it an operationally appealing choice: it was a relatively simple, mature, and well-established protocol. When we initiated an effort to change our datacenter congestion control algorithm from Cubic in 2018, DCTCP's design had been published eight years prior (2010), its design was well-understood, it had been added to the Linux kernel for four years (since 2014), and had widespread hardware support for ECN marking. We anticipated a smooth transition to DCTCP. Contrary to our anticipation, however, nearly every point in the end-to-end stack presented a riddle to solve to rollout DCTCP: the kernel had bugs; optimizations such as receive offloading could not always inter-operate with it as DCTCP's smaller congestion windows were not always enough to trigger prompt delivery; some switches dropped ECN-capable packets despite having space to buffer them, resulting in poor application performance; not all switches could consistently and reliably support ECN; we could not change the congestion control of long-running connections in the middle of their data transfer, etc.

In this paper, we share our experience of deploying DCTCP in Meta Datacenters. This project started in earnest in 2018, balancing successful tests with head-scratching problems. We share the stumbling blocks we discovered in the hope of helping researchers consider the *deployability* of new data center congestion control algorithms. In particular, switch and NIC implementations are diverse, vendor specific, not always known to datacenter operators, and evolve frequently as the scale and demand of our networks change. Protocols that are dependent on parameter tuning for optimal performance are hard to deploy and maintain. In a large-scale and diverse network, each congestion control algorithm will coexist with a large and diverse set of protocols. Finally, bugs are inevitable in any large-scale, complex network. Ideally, the congestion control protocols should be equipped with mechanisms to detect and gracefully handle bugs and corner cases. We found every problem in the text that follows to be rich and deep, with quite a few surprises. In the moment, we questioned why is this so hard: switches can mark ECN and the kernel implements DCTCP. What more does there have to be? But reflecting on the experience, by expecting unforeseen trouble, careful deployment and monitoring paid off.

Ours is not the first report on production DCTCP deployment; Judd [15] shared experiences that influenced our design. They showed that DCTCP could be unfair to an established Cubic flow, motivating our study of mark and drop thresholds to keep them fair. He also noted that SYN and SYN/ACK packets should be ECN-capable, despite standards; we were surprised to find the same even after our switch thresholds prevented DCTCP from starving out Cubic. His deployment supported ECN marking only on top-of-rack switches; this became the starting point we focus on in this paper.

We organize the paper by characterizing problems by where they occurred. In the next section, we briefly overview DCTCP. Feel free to skip this section or read Alizadeh et al. [4] instead. Section 3 describes how we chose to enable DCTCP for in-region TCP connections but not cross region ones; focusing on deployment safety. Section 4 describes how to configure switches to mark DCTCP traffic and encourage balance with competing Cubic flows; for some devices in our network, this was unexpectedly elaborate. Section 5 describes what we built to monitor the deployment, looking to confirm that congestion-experienced bits were set, DCTCP was negotiated and not falling back to Reno, switch buffers were less occupied, etc. Section 6 describes the often subtle kernel and driver bugs that ECN marking packets and smaller congestion windows surfaced. Section 7 describes a few extensions we applied to the initial DCTCP deployment, extending where congestion can be instrumented with ECN. We conclude in section 8, discussing lingering problems in congestion and reflecting on how the design of nearly every component in the network influenced this deployment.

2 Background

Data Center TCP (DCTCP). DCTCP uses Explicit Congestion Notification (ECNs) signals from switches. It uses 2 bits in the IP header for ECN information. If neither of the two bits are set, the flow does not support ECN and switches will not mark the packets. When only one bit is set, the flow supports ECN signals and no congestion has been encountered. Finally, when both bits are set, the flow supports ECN and the packet has encountered congestion. Congestion, in this context, is usually defined as the queue sizes of the switches on the packet's path passing a pre-defined threshold. That is, when a switch receives an ECN-enabled packet, if the queue used to enqueue the packet is larger than some threshold, the switch marks the packet as having experienced congestion. This signal then arrives at the receiver. The receiver notifies the sender by echoing back the congestion signals on the TCP header of the ACK packet.

Pre-DCTCP, senders treat ECN-marked ACKs as packet loss. TCP Reno, for instance, reduces its congestion window (CWND) by 50%. This aggressive throttling can lead to link under-utilization. Another issue is that legacy protocols do not differentiate between short bursts and standing congestion. For example, sub-RTT queue buildups due to microbursts still result in reducing CWND, a suboptimal outcome.

In contrast, DCTCP reduces CWND proportional to the level of congestion by tracking the percentage of bytes per RTT. For example, if 100% of bytes encounter congestion, DCTCP reduces its CWND by 50% but if only 50% of bytes do so, it reduces CWND by 25%. DCTCP also leverages a moving average to avoid overreacting to transient bursts. For example, if 100% of the bytes in an RTT encounter congestion, but there was no congestion in previous RTTs, then the CWND would only be reduced by 1/32 instead of 1/2 [9].

Given the characteristics of our workloads, notably their

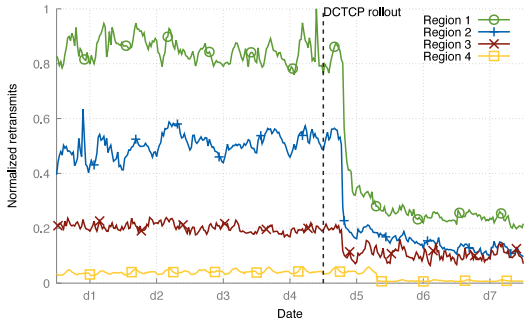


Figure 1: Retransmission rates drop following DCTCP rollout.

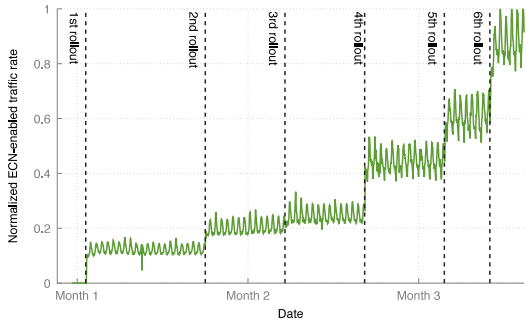


Figure 2: DCTCP's rollout accelerated over time.

burstiness [13, 27], we expected DCTCP's faster reaction to bursts to improve the performance of our applications.

DCTCP helped our network. Encouraged by positive test results, we gradually rolled out DCTCP to each region in the fleet. Overall, we observed improvements in network metrics such as reduced top-of-rack switch congestion discards and queue lengths, resulting in the reduction in the number of retransmitted packets that hosts received. We compared the overall volume of retransmissions for each region after deploying DCTCP and compared it against one week before the transition and observed a reduction of around 75%. Figure 1 shows the change in normalized retransmission rates[2] for four regions a few days before and after the rollout of DCTCP in each region. Note that the reduction in retransmissions after deploying DCTCP is not immediate. In §3.2, we discuss a potential reason (the delay in changing the congestion control algorithm for long-running connections).

Along with retransmissions, we tracked base host metrics like throughput, and the congestion window size, as well as general system state metrics such as CPU and memory utilization. We did not observe any regressions in these metrics. For four regions, we measured the changes in retransmission rates, throughput, the average CWND, and the average smooth RTT (`srtt`) after transitioning to DCTCP. Table 1 reports the results. Note the variance across different metrics and regions, e.g., while `srtt` did not change in Region 4, it did improve in Region 5 albeit not as dramatically as the retransmission rate (7% *vs.* 50%). Moreover, for one of our data-intensive services, we measured the changes in read latency in a region before and after transitioning to DCTCP and observed 38% reduction in the 90th and 99th percentiles of latency (from 65*ms* to 40*ms* and from 130*ms* to 80*ms*, respectively).

Rollout timeline. It should be noted that we did not upgrade all selected regions to DCTCP at the same time; we proceeded gradually over a four-month period while carefully monitoring the impact of the change on our networks (§5). Figure 2 shows the timeline of the per-region rollout, overlapped with the aggregate rate of ECN-enabled packets during this period. The rate of ECN-enabled packets in our networks is a proxy of DCTCP's adoption, as DCTCP traffic is by default ECN-enabled. During this interval, the DCTCP was the only class of ECN-enabled traffic in our networks. To compute the normalized rates in Figure 2, we divided the number of ECN-enabled packets received by all datacenter hosts by the maximum number of ECN-enabled packets received by all hosts once the rollout process was completed. The figure depicts progressively smaller and smaller time gaps between successive rollouts; our deployment process gradually accelerated as our initial deployments helped us identify and resolve the issues that we will discuss in the rest of the paper.

To gauge the impact and effectiveness of DCTCP after the rollout, we temporarily disabled it for a few hours in one region. This resulted in around a 10% drop in throughput and 4.5× increase in retransmissions in that region.

Metric	Region 3	Region 4	Region 5	Region 6
Retrans.	33% ↓	79.5% ↓	50% ↓	73.4% ↓
Throughput	10.3% ↑	4.8% ↑	2% ↑	5.8% ↓
Avg. `srtt`	No change	No change	7% ↓	7% ↓
Avg. `cwnd`	30% ↑	22.3% ↑	20% ↑	18.1% ↑

Table 1: Changes in performance metrics.

3 Enablement

We wanted to enable DCTCP only for in-region connections, which have small enough RTTs that DCTCP is effective. This presented several immediate challenges for us as our available options could not achieve this, e.g., the selective enablement requirement meant that `sysctl` is not viable, while complexity ruled out options such as `setsockopt` and `routes`.

As we explored and experimented with enablement options, we also encountered several other challenges related to kernel features, and obscure DCTCP fallback behavior. We discuss these challenges below.

3.1 Enabling DCTCP for only in-region flows

We discuss the drawbacks of available knobs in greater detail before presenting our approach for resolving this challenge.

[2] *Normalized retransmission rates* are retransmission rate of each region divided by the maximum retransmission rate across all regions.

3.1.1 Potential knobs for changing congestion control

Our goal was to identify knobs to target short-RTT connections and change their congestion control algorithm (CCA). For DCTCP, this should be done before the 3-way handshake since ECN is negotiated during connection establishment. We targeted sub millisecond latency which is typical for traffic within a region. In our data center network, we can use IP addresses as a proxy for RTT, and classify connections created within the region by looking at the source and destination IP addresses. CCAs can be changed in many ways, notably via `sysctl`, `setsockopt`, and `routes` as we discuss next.

`sysctl` `sysctl` is the most common and easiest approach to change CCA. This takes effect for *all new sockets*. This was not ideal for two reasons: (a) we could not easily distinguish between inter- and intra-region traffic and limit the adaptation of DCTCP only to the latter, and (b) the transition process would be slow. Since changes through `sysctl` take effect only on new sockets, listening sockets had to be reset for the passive side of the connection to pick-up this change. At a region-scale, resetting listening sockets requires restarting services which takes days or weeks.

`setsockopt` The most granular approach is to call `setsockopt()` per socket. This also provides the most flexibility since it enables running arbitrary logic for each individual connection. On a datacenter scale, `setsockopt` involves building a library that is used by every service and fitting into the continuous integration and delivery schedule of thousands of services which run across the fleet. Although this was conceptually feasible, this tight coupling was operationally not ideal as it would complicate debugging and fault isolation. Plus, it offered little benefit since DCTCP did not require any service-specific information that would justify inter-dependency with services.

`routes` Linux supports customizing TCP parameters per IP destination route. By changing the granularity of `routes`, we could target different scopes, ranging from an entire region to a single host. For enablement, we needed to enumerate all the intra-region route prefixes and create multiple route table rules. This approach was attractive since it had the lowest dependencies across all options. However, it was not *extensible*: it did not support matching based on other criteria such as the port number, the kernel version, and the NIC model that we envisioned using in future designs. Another risk of this approach was modifying the host routing table and depending on aggregatable IP prefixes. Given how routes control reachability in our fleet, these created substantial risks.

In summary, none of the approaches discussed above was adequate for our use case. We had to develop a new approach based on eBPF which we discuss next.

3.1.2 TCP socket hook eBPF

We developed a method based on eBPF to address the challenges discussed above. By providing hookpoints across the kernel stack, eBPF allows us to customize the network stack. Since its introduction in Linux 4.11, `sockops` has provided a way to run eBPF programs during TCP socket events. A major advantage of `sockops` is its flexibility and programmability. It enables running an eBPF program at the start of every connection which can use user-space configuration to select a CCA for every connection. We use the same IP-based classifier to identify connections to enable DCTCP.

By design, `sockops` is restricted to a specific version of `cgroup` (`cgroup-v2`).[3] Alas, in our fleet, we deployed heterogeneous kernel versions with two versions of `cgroup`. Kernel and `cgroup` limitation meant this cannot be used standalone.

To work around the dependence of `sockops` on `cgroup` version, we used *two* BPF programs to enable DCTCP for new in-region TCP connections: (1) On `cgroup-v2` hosts, we attached a per-connection `sockops` BPF program to set DCTCP as the CCA for in-region connections while leaving inter-region connections unchanged, and (2) on `cgroup-v1` hosts, we used `sysctl` to set DCTCP as the default CCA and attached a per-packet Traffic Control (TC) program to clear ECN-related bits on inter-region SYN packets, forcing those connections to fallback to Cubic. This program not only disables DCTCP for inter-region traffic, but also enables DCTCP for connections on pre-existing listening sockets.

3.1.3 Enablement plan

Our intended scope for deploying DCTCP was an instantaneous region-wide enablement, so that we could minimize disruption in the network caused by in-region DCTCP and Cubic flows interacting in the same switch buffer. This was a substantial change that had never been done before in our network and had to be performed safely without causing disruption to services. This was a challenge given the heterogeneity of our fleet (e.g., in terms of *cgroup* and kernel versions). Ensuring safety despite the diversity of the fleet required having visibility and monitoring into enablement knobs to detect problems, and the ability to revert fast if there was a problem. Safety, flexibility, and fail-open characteristics made eBPF a good choice for this problem. Given we had 90% of the fleet already on `cgroup-v2` when we started with this effort, we took on more complexity in the short-term to handle `cgroup-v1`, but over time, we retired the `cgroup-v1` solution and evolved eBPF to be able to customize both the algorithm and deployment capabilities that we had originally envisioned. We explain the two separate eBPF knobs for the two `cgroup` versions next.

3.1.4 `cgroup-v2` and 4.11+ kernels

We attach a per-connection sockops BPF program [8] to the root-cgroup. A sockops program is invoked for different state

[3] Control groups or *cgroups* is a Linux kernel feature that limits, monitors, and isolates the resource usage of a collection of processes.

```
int sockops_program(skops) {
  switch (skops->op) {
    // Before sending SYN on client or SYN-ACK on server
    // if peer in-region request ECN
    case BPF_SOCK_OPS_NEEDS_ECN:
      if(in_region(skops)) {
          skops->reply = 1;
      }
      break;
    // Once connection is established
    // If we see ECN then enable DCTCP, else Cubic
    case BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB:
    case BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB:
      if(ecn_enabled(skops)) {
        bpf_setsockopt(skops, TCP_CONGESTION, "dctcp");
      } else {
        bpf_setsockopt(skops, TCP_CONGESTION, "cubic");
      }
  }
}
```

Figure 3: Summary of ECN enablement in cgroup-v2 sockops

transitions in the lifetime of a TCP connection. During specific TCP events, `sockops` eBPF programs can change some TCP parameters using `setsockopt()` calls or influence kernel actions. For example, when client or server have to decide ECN negotiation, eBPF programs registered on this callback can opt-in for ECN negotiation after analyzing the TCP and IP headers in the packet or any user-space map state. Our `cgroup-v2` program uses the source and destination IP address and calculates the scope (rack/cluster/dc/region) of the connection. It uses an IP prefix eBPF map populated with all intra-region IP prefixes. It takes the following actions also summarized in Figure 3.

#1: NEED_ECN: When kernel asks whether a socket needs ECN or not, choose "YES" for in-region traffic and "NO" for inter-region traffic.

#2: CONN_ESTABLISHED: When a connection is established, if it is in-region and ECN enablement succeeded, then call `bpf_setsockopt` to set the socket's CCA algorithm to DCTCP. Inter-region CCA remains unchanged and uses the default specified via `sysctl`.

3.1.5 `cgroup-v1` and older kernels

On hosts with `cgroup-v1` or older kernels where sockops eBPF hookpoint was not available (less than 10% hosts), we still needed a solution to selectively change CCA. We chose a combination of `sysctl` and TC eBPF to solve this. We used `sysctl` to set DCTCP as the default CCA and used a Traffic Control (TC) eBPF program to terminate ECN negotiation for inter-region connections, thereby forcing them to fall back to Cubic as we explain below. Table 2 shows the sequence of events that happen during a 3-way handshake (the blue parts are specific to ECN/DCTCP).

To selectively disable DCTCP for inter-region scope, we use a per-packet TC eBPF program on the server in the ingress direction. TC eBPF programs can access socket buffer (skb) and through this modify packet contents. This program can thus selectively clear all ECN state set by the client (step 1 from Table 2). This is possible because in the ingress direction TC programs are executed before the TCP stack can negotiate ECN. This program uses a similar scope resolver as explained in the sockops section above to detect inter-region clients and match TCP SYN packets with ECN negotiation bits (ECE/CWR bits [6]). If there is a pattern match, this program clears the ECN-related bits in the TCP and IP header of the SYN packets thus causing the ECN negotiation to fail. Without ECN, DCTCP is configured to fallback [7] and this eBPF program exploits this configuration.

Step	Direction	IP flags	TCP Flags
1	Client→Server	ECT(0)	SYN + ECE + CWR
2	Server→Client	ECT(0)	SYN + ACK + ECE
3	Client→Server	ECT(0)	ACK

Table 2: ECN + DCTCP Handshake

Although the Linux DCTCP implementation was configured to fall back to TCP Reno as the CCA when ECN negotiation failed, we did not want this extra algorithm to be added to the set of algorithms in our networks. Recall that the co-existence of Cubic and DCTCP traffic was a major challenge in deploying DCTCP. Adding a new protocol to the set of CCAs would exacerbate the situation and require re-running the entire process of parameter tuning and switch configuration with a new combination of protocols. To avoid this problem, our kernel team altered the DCTCP code [7] to fallback to Cubic instead, a change we did not publish upstream.[4] This was our only custom kernel change for the effort, and we removed this patch by replacing the logic with BPF code that could itself handle the fallback.

3.2 Long-lived connections

ECN is negotiated at the onset of the connection: this created a challenge for changing the CCA of ongoing connections from Cubic to DCTCP as we could not enable ECN for a flow in the middle of the connection. Unfortunately, many connections in our networks are long, running for days or longer. This slowed down the process of migrating to DCTCP. The naive solutions such as terminating the existing connections and collaborating with service owners to force all services to be restarted were suboptimal due to their complexity and perceived impact on service performance.

We realized that one of our internal disaster recovery tools, Maelstrom, can be repurposed to aid with the CCA upgrade process. Maelstrom [26] is a large-scale disaster recovery system. It provides a traffic management framework with modular primitives that can be composed to safely and efficiently drain the traffic of interdependent services from one or more failing datacenters to the healthy ones. Maelstrom encoded inter-service dependencies and had a safe way to temporarily

[4] There are some drawbacks to including this fallback policy in the mainline kernel, e.g., Cubic is an optional module in Linux, while Reno is not.

drain traffic from a region. We leveraged Maelstrom to run multiple drain exercises to gradually upgrade the connections to DCTCP. Each drain exercise resulted in approximately 50% of the connections to flip to DCTCP resulting in significant coverage for the CCA change.[5]

To enable future upgrades and maintenance, eventually, we enhanced our eBPF framework with a connection iterator which can iterate over all existing connections, upgrade congestion control, and make other transport changes. The ability to trigger this program on-demand and share similar code as sockops simplified CCA's evolvability and maintenance.

4 Switches and buffers

DCTCP relies on network support—switches mark packets with ECN when buffer occupancy exceeds a certain threshold. Most modern switches support ECN, but it quickly became clear that our specific network characteristics made widespread deployment of ECN unfeasible. We list the specific challenges we faced in deploying ECN in our network.

4.1 Switch queues are a scarce resource

DCTCP, as the name suggests, is designed for data center flows with low RTT ($\sim 1ms$). It is not expected to work well with long-distance flows (10s to 100s of *ms*). This is because the ECN signal, based on queue occupancy, is ephemeral; current queue occupancy is meaningless several milliseconds in the future. So, for our cross-region flows, we continue using Cubic, which relies on packet loss as its signal.

It is common that DCTCP and Cubic flows terminate at the same host, a service can talk to other hosts both in-region and cross-region. In such cases, ideally, we would isolate the feedback signal of the two CCAs in the network. For Cubic and DCTCP, isolation is even more important because they rely on two very different signals. ECN-based CCAs aim to keep bottleneck buffer utilization low, while loss-based CCAs drives the buffer to capacity and loss. In switch terms, conceptually, isolation is easy enough to achieve – we just need to put each CCA in its own queue where it gets access to its own allocation of the dynamically shared buffer.

However, switch queues are primarily used for traffic classification in our network; we support several classes of traffic [3], and each class is allocated to a queue. To isolate CCAs from each other, we would need two queues per traffic class. While later generations of switches supported this, initially, our switches did not have enough queues to support this configuration. This meant that we had to put both DCTCP and Cubic in the same queue, and depending on the switch vendor, the switch supported ECN and Drop Tail (or WRED) in the same queue, or only either ECN or Drop Tail (or WRED). Luckily for us, the bottleneck layer in our data center network had the switches that supported both.

[5] Another benefit of Maelstrom was that it could keep the region in the drained state if we observe any CCA-related outage. This enabled us to manually connect to hosts and remediate the issue.

4.2 ToR switches were sufficient for ECN

We analyzed our production network, and found that bottlenecks largely occurred in the ToR switch downlinks [13]–to hosts–or in our backbone WAN network. The latter is out of scope for this work, as DCTCP does not traverse regions, so we focused only on the ToR downlinks.

A natural question is why we saw congestion largely in the ToR downlink. The answer is a combination of factors. First, due to the nature of our hardware, there was very high disparity in link speeds in the ToR downlink compared to the rest of the fabric, making this layer more susceptible to bursty incasts. Secondly, provisioning and topology ensured high cross-section bandwidth between racks. Finally, rack-agnostic job scheduling ensured a high level of heterogeneity in traffic, largely preventing potential hotspots caused by concentrated placement of network-heavy services.

As we mentioned previously, not all layers in our fabric had switches that supported dual mode thresholding to support DCTCP and Cubic. However, since congestion was rarely an issue elsewhere in the network (and therefore buffer utilization not a big enough issue to cause discards), we could effectively ignore them, and focus our attention of getting the thresholds right for the ToR downlink.

4.3 How to set a mark and drop threshold

Our switches had the ability to mark ECN or do Drop-Tail/WRED on a flow depending on ECN-Capable Transport (ECT) marking, so both DCTCP and Cubic would receive the correct signal, but the buffer itself was shared across both classes of flows, resulting in no isolation.

ECN signaling tries to keep DCTCP queues low, but offers no guarantees – it is entirely possible that a burst of DCTCP traffic can occupy a buffer well beyond the ECN threshold. Cubic flows, on the other hand will try to maintain high buffer utilization. This is a challenging scenario: high buffer occupancy with bursty Cubic flows will result in very high ECN marks for a competing DCTCP flow, while high buffer occupancy with bursty DCTCP flows will result in very low available buffer for competing Cubic flows. Complicating the issue further, not all DCTCP flows are bursty (unless within an incast), while even a few long-RTT Cubic flows could be bursty due to the high BDP.

Our hypothesis was that we need to prevent either CCA from entirely taking over the buffer – but we were more concerned about Cubic's ability to do so, so we narrowed down to a solution where we set relatively high ECN thresholds for DCTCP and low droptail thresholds for Cubic. This was a sub-optimal solution for both CCAs, because it diluted ECN for DCTCP and made loss likelier for Cubic, but it solved for

what we thought was the dominant problem in our network – Cubic bursts undermining DCTCP.

Post-deployment experience proved our hypothesis to be not entirely correct: while Cubic indeed could capture a shallow buffer, incast scenarios were far more serious in our network than we originally anticipated, and had to implement further features to control DCTCP burstiness and buffer utilization, particularly for services that had both sets of flows.

4.4 The multi-host NIC

Multi-host NICs use a single connection to a ToR switch to provide connections to the PCI busses of (typically two or four) different hosts. The multi-host (MH) NIC design is efficient in space, power, and hardware, but is a network component not typically modeled in congestion control work.

The MH NIC connects multiple machines to the network and has a buffer (on the order of 0.5 to 1 MB in size), but it does not act as a conventional switch. The MH NIC does not mark ECN mark by default; earlier versions do not even have this feature. The buffer is not partitioned explicitly across hosts or queues, so it does not provide isolation, nor is able to preferentially deliver high priority traffic. It does not have predictable downlink rates; PCI bus rates are split across the hosts, resulting in per-host rates being significantly less than the MH NIC rate. Furthermore, when a host kernel is unable to keep up with interrupts, or is unable to supply free buffers, the delivery speed to an individual host can drop further.

As the MH NIC buffer fills, the NIC can send ethernet pause frames back to the top-of-rack switch, leading to queuing at the switch. This can happen when a single host is the target of incast: the complete bitrate of the NIC can be applied to deliver the burst of traffic from switch to NIC, reaching a bottleneck at delivering the data to the memory of the destination host. This delivery can also be slowed enough to send pause frames when a host is processing difficult-to-accept data such as small frames from many different connections not amenable to offloaded reassembly.

Unmodified, the ECN marking threshold at the ToR means that the effective queue backlog to the host is the ToR ECN threshold *plus* the size of the NIC buffer. This creates an effectively far-too-high ECN threshold, resulting in persistent unfairness, poor performance, and even packet loss of DCTCP traffic as the congestion window overshoots the target.

The effectively larger buffer for a host is not as severe a problem for Cubic alone: it appears as a single large buffer shared across the hosts. However, a single host's traffic can still dominate the buffer, and since only the switch has the ability to prioritize traffic, unfairness between buffer-hungry services and services that need highly-reliable delivery led to shifting the buffer to the switch via per-host queuing.

On racks with this type of MH NIC, the ToR switch creates a separate, rate-limited queue for each downstream host. This rate limit is based on the host's "share" of the NIC bandwidth, i.e., 1/2 or 1/4. This queue is then configured with ECN mark and drop thresholds from Section 4.3. This queue-per-host design does not remedy all interactions between different jobs sharing the same NIC: for example, a slow kernel processing small packets can still fill the NIC buffer, leading to some pause frames to the ToR, but while the link is unpaused, the ToR can round-robin among the other hosts to limit performance degradation. Important for DCTCP however is that in the expected case, packets are marked when the effective queue, entirely on the ToR, reaches the intended threshold.

This queue-per-host feature played an important role in getting performance and fairness out of DCTCP on MH NIC systems. Testing focused on fairness and performance of flows to individual hosts; the performance of concurrent, production-like traffic to different hosts on the MH NIC was not easily observed. We were fortunate that the queue-per-host feature in our ToRs was rolled out in time for DCTCP deployment.

4.4.1 Database clients in particular

Here is a specific example of how queue-per-host became necessary to deploy DCTCP. During our first region rollout, it turned out that many in-region connections *establishments* were timing out. After one second, the Database client code making this connection would time out and report an error. Packet loss and retransmissions overall were down, utilization up; the typical signals of network performance looked good.

We have a set of tools that instrument retransmissions generated by the Linux kernel using eBPF "tracepoints" and "kprobes." While the kernel's built-in counters can track how often retransmissions happen, with eBPF we can classify what generated the retransmission (timeout? duplicate ACK?) what was retransmitted (a SYN? SYN/ACK? a tail loss probe?), and which services were the endpoints of these retransmissions. We observed both SYN retransmissions toward the Database server and SYN/ACKs in return.

The issue was that while SYNs and data packets were marked as being ECN-capable, potentially being marked, the listening socket did not mark the SYN/ACK as being ECN-capable, directing it into the Cubic connection packet drop profile. With DCTCP acting aggressively due to the too-high effective threshold, DCTCP traffic from—potentially from other hosts in the MH NIC—would fill the space between its mark threshold and the Cubic drop threshold, causing the "Cubic" seeming SYN/ACK to be discarded.

We were fortunate that this problem affected this Database client, which had a hard-coded application level timeout just long enough that SYNs and SYN/ACKs would be retransmitted. With that information, it was easy to find endpoints that saw SYN/ACK loss. Many other services abandon connections after a short timeout, and thus do not retransmit SYN packets; such services also reuse connections, making them less sensitive to problems in the three way handshake.

Enabling the queue-per-host feature immediately solved

this problem, though we also prepared BPF filters that would look at each packet to ensure that DCTCP-negotiating SYN/ACK packets would be ECT marked, just in case.

4.5 Experience with different switch ASICs

We discussed in §4.3 how we tuned our ECN and Droptail thresholds for the ToR layer. At the time of initial deployment, the vast majority of ToR switches had the same ASIC, which we refer to as Asic-A1. However, our production network continued to evolve, with newer ASICs from the same manufacturer (Asic-A2), and a new ASIC from a new vendor (Asic-B). There are two major issues to consider about switch ASICs when we rely on them for congestion marking: buffer sizing and ECN implementation.

In our case, initially, the newer ASICs, Asic-A2 and Asic-B inherited the thresholds that we developed for Asic-A1. Asic-A2 had the same architecture as Asic-A1, though it had 4x the buffers. With host NIC speeds also evolving, the newer switches also served faster NICs (2-4x). Put together, although the original thresholds were not always optimal, they still resulted in reasonably good performance.

However, Asic-B had a fundamentally different approach to buffer design and management making our thresholds behave differently from Asic-A family. For service operators, ToR placement is considered to be transparent: however, with Asic-B, performance was potentially now dependent on the ToR hardware; we explain the differences in ASIC architecture that proved consequential for threshold tuning next.

Queue management. Asic-B used separate Virtual Output Queues (VOQs) for ECN-Capable Transport (ECT) traffic and non-ECT traffic, which facilitated isolating these two classes of traffic, even when they are destined to the same host. For Asic-B, we could separate out DCTCP and Cubic buffer threshold tuning as two independent problems. However, this raised unexpected, somewhat intractable issues due to how the shared buffer was allocated in Asic-B.

Buffer allocation. Asic-B's shared buffer space is divided into separate "slices", and buffer thresholds are applied *independently* in each slice. This is in contrast to Asic-A family where the shared buffer pool is broken down into ingress traffic managers (ITMs), but the buffer threshold for a specific VOQ is applied as the sum of buffer to that VOQ across all the ITMs. This distinction was particularly important for any traffic pattern more than a single flow: any such traffic could end up consuming effectively more buffer space on Asic-B–across all slices–compared to Asic-A family for the same buffer thresholds. Needless to say, naively reducing thresholds by a factor of number of slices was not an option because that would affect individual flows that get mapped to a single slice when we did not have incast.

Quantized thresholds. Further complicating threshold deployment, the Asic-B architecture used "quantized" regions, resulting in a small number of actual threshold values, which meant that it did not support arbitrary thresholds, and any configured threshold between two quantized values would be applied on the lower value. For example, if the quantized regions are at 100KB and 200KB, the configuration would accept 120KB, but it would actually apply the threshold at 100KB. This quantized buffer management reduced the effective parameter space; although this had the potential to simplify search, it also reduced tuning flexibility.

Quantized drop probabilities. Similar to thresholds, Asic-B also has quantized drop probabilities (important for schemes such as Drop Tail and WRED. This made it hard to model WRED's performance in our fleet as it was unclear if WRED would perform consistently for different sets of flows with the same thresholds and drop probabilities.

Drop decisions. Although all our ASICs have a notion of dynamically shared buffers, Asic-A family and Asic-B use very different logic to share available buffer. The Asic-A family use the α parameter to share available buffer, while Asic-B uses a function based on the total buffer use, the VOQ size, and the delay experienced by the last packet sent from the VOQ. These values were quantized and used to index into a lookup table. Compared to the Asic-A family's single-parameter shared-buffer model, this mechanism is substantially more complicated, with many knobs to tune. This was further compounded by the fact that the value of these knobs were not always known to us.

These differences are irreconcileable—it is impossible to guarantee that a specific traffic pattern will see the same buffer and marking or drop probabilities across all platforms, or even just across Asic-A2 and Asic-B, which we consider to be generationally equivalent. Even with only a single parameter to tune—ECN—the above challenges underscore the difficulty and complexity of parameter tuning, particularly for more complex parameter-sensitive protocols, in an increasingly heterogeneous network similar to ours.

5 Visibility for Operations

We approached visibility from two perspectives. First, we looked at metrics from each of the layers in the network—from switches and their counters through to services and their query times. Second, we looked at covering different time scales—fine-grained debugging at RTT scale, looking at packet traces, out to long term trends in network metrics such as retransmissions. Visibility and monitoring are important components to any deployment effort. We needed to ensure that our regular monitoring systems were enhanced to account for CCA where possible, and DCTCP-specific counters, such as packet counts with ECT/CE bits set.

5.1 eBPF for monitoring

We used eBPF-based instrumentation extensively to monitor the DCTCP deployment. Similar to our enablement efforts, we found that kernel-maintained counters are not always enough: they do not separate bytes sent with DCTCP and with Cubic, or in-region and cross-region traffic. Similarly, counters of retransmissions have major limitations, as we describe later.

Each TCP connection has a field that stores the CCA for that connection. This allows us to track, using our fbflow [25] packet sampling implementation, the specific CCA that governed that transmission. With this, we can quickly confirm that in-region flows are using DCTCP when we expect them to, and that the overall bit rate of in-region traffic is about the same before and after enablement.

We also instrument and log retransmissions with an eBPF-based system that traps calls to the `tcp_retransmit_skb` function, and annotates the retransmission event with the type (timeout, fast, syn, and synack), CCA, information about the endpoints and the services involved. The CCA field doesn't always have a well defined answer, since a SYN packet can be retransmitted before ECN capability has been negotiated.

5.2 The puzzle of more retransmissions

We observed unexpected increases in retransmissions, both in kernel netstat counters (RetransSegs), and in our eBPF-based pipeline. Packet discard counters at switches were down; so why would the kernel need to retransmit more often?

This increase in retransmissions turned out to be a result of tail loss probes (TLP) [11]. TLP is a means of guarding a TCP connection from a packet loss that otherwise needs a complete RTO to recover. The sender eagerly retransmits as soon as an ACK is overdue, in order to repair the missing packet, to receive an ACK that identifies the missing packet, or to confirm that the original was delivered.

The algorithm for deciding when to send a TLP implemented in Linux is to set a timer after each transmission, set to expire after two times the RTT plus two "jiffies" (i.e., milliseconds when the constant HZ is 1000). We observed that DCTCP reduced queueing and RTTs; on hosts that were somewhat busy and needed a couple milliseconds to answer a query, this reduction was enough to shift the connection from not seeing a TLP (a 3ms RTT would lead to an 8ms TLP timer, and 8ms was plenty to generate the response) to seeing TLPs frequently (a 0ms RTT leads to a 2ms TLP timer). Although there is a counter of transmitted TLPs, it includes both new data and retransmissions, since both can be used in a TLP; the count of necessary retransmissions is not easily recovered.

We adjusted our instrumentation to identify this class of retransmission, allowing us to largely ignore them to focus instead on other retransmissions when debugging performance. TLPs may be wasteful in the common case of a DCTCP connection, where losses are infrequent and RTTs short, but we have not yet experimented with disabling it; the overhead of sending the TLP is low and it may help in certain situations.

5.3 Metrics we monitored for sanity checks

We also monitored existing network metrics that tell us the network state of the fleet. These included metrics from the hosts (e.g., throughput, socket counts, RTOs, TCP memory, CPU utilization), from switches (link utilizations, buffer utilization, congestion discards, queue lengths). To this existing set of metrics, we also added ECT and CE marked packets. These data provided us with baseline assurances that DCTCP was not unnecessarily throttling links, and that it was indeed reducing buffer utilization and packet discards in switches, and that the ECN signaling was working as expected.

We also focused on service monitoring identifying top services in the region of rollout and proactively alerting their oncalls to the rollout. In addition we worked on aggregating metrics to allow both problem identification and the ability to dig into them. The aggregations allowed users to go from a single host to a service to the entire region to see what the scope of an anomaly was and vice-versa from a region level anomaly to a host facing the issue.

Another effort we undertook was to identify if the network improvements were attributable only to DCTCP rollout or some other parallel network change caused by say a higher surge or users in the region. To perform this we created a background signal using all the non-rollout regions and compared it with the signal from the rollout one. We were able to ascertain a statistically significant correlation between the rollout of DCTCP and the improvements in the metrics.

5.4 Metrics that helped us troubleshoot issues

Troubleshooting performance is important—whenever a change of this magnitude is made to the network, any performance degradation seen by services are attributed to the network, whether deserving or not. In such scenarios, the ability to confirm that the network is at fault, or not, can make the difference between a successful versus an unsuccessful deployment. There were several metrics that helped us troubleshoot issues, and blame our rollout as appropriate. We list two in this section.

Connection set up failures: For the Database issue we discussed in 4.4.1, we noticed that connection set up failures spiked at the time of the rollout. This metric was a fleet-wide existing counter; it eventually led us to the root cause, when we saw with the retransmissions data (5.1) that SYNs were affected. Ultimately, we needed `tcpdump` to identify that the ECT bits were missing in the SYNACK.

Hardware tagging: During the initial wave of rollouts, we started noticing that a particular service in a few regions were seeing increased fast retransmissions and timeouts. This led

us to initially wonder if the service was bursty, and whether DCTCP was unable to handle the bursty traffic. However, NIC vendor tagging in our retransmissions data isolated the issue to only one particular vendor, and service tagging in the same dataset told us that other services were affected as well, just not to the same extent. We also built tools for burst visibility that helped us root cause the issue to a driver bug.

The general takeaway is that in a vast and heterogeneous network such as ours, we need extensive monitoring of a variety of network metrics. Even if most of them are not used day-to-day, or point to symptoms rather than the cause, they help isolate the issue and focus the effort to root cause, saving hours or more of engineer time.

6 Kernel and driver trouble

DCTCP exposes a set of interactions with features in the kernel and NIC driver that can lead to undesirably poor or uneven performance. At a high level, the smaller congestion windows of DCTCP mean that CPU-efficiency techniques for segmentation offload ("TSO") and reassembly offload ("GRO") behave a little differently, perhaps adding delay waiting for more data to work with, or simply requiring more operations to send the same number of packets.

Others have noticed performance issues that result from interactions between the kernel's typical use of Cubic, with large windows and large backoff, and DCTCP. For example, Misund [23] notes an interaction between DCTCP, proportional rate reduction [10] and segmentation offload.

6.1 Delayed ACKs

DCTCP appeared in the Linux kernel in 2014 [7], with substantial fixes to delayed ACK handling in 2018. The central bug was that when the sender has a congestion window of 1, the receiver did (but should not) delay its ACK [9]. The delayed ack timeout was 40*ms* by default, resulting in connection having a CWND of 1 stalling that long. We had to backport this change to a significant set of hosts running an older kernel. Although we try to upgrade to the newest kernels whenever possible, sometimes there are specific regressions or driver issues that give older kernels extended life.

6.2 GRO creates unfairness

In addition to issues with delayed acks, a certain vendor NIC delayed delivering packets that it expected to be able to reassemble, and this delay led to wild imbalance in throughput in small scale testing. In particular, an established test flow would reach 91% of link rate, while a second flow would only get 2%. Of course, the fraction of packets being ECN marked was comparable, so one would expect the two flows to converge as they would on a different NIC. With much testing, sending small RPCs that were not delayed, and `tcpdump` at both ends, we found that the NIC was applying the following rule for its GRO. The NIC would deliver if it could reassemble ten packets, if it saw a push bit, or after 1 millisecond.

In practice, this rule meant that flows with a CWND below ten packets would see an extra millisecond added to their delay, and flows having larger windows would not. This GRO rule probably didn't affect long RTT flows, where the 1ms timer was relatively small. But inside the same datacenter, this is much larger than the base RTT. This difference in effective RTT reinforced the unfair distribution of bandwidth between them. To fix, we had to disable hardware GRO for this NIC. There were other alternatives (e.g., to force setting push bits on segments that would not otherwise merit them), but the complexity did not seem worth it.

6.3 New eBPF

In Section 3, we described how BPF provided our best means to express policy about which connections should use DCTCP. However, this left us some additional problems.

First, we had to fix issues with getting ECT marked on SYN/ACKs based on the decision to use DCTCP. This was possible (the decision is made by the BPF code before the SYN/ACK is sent) but was not the default behavior.

Second, to give more flexibility in how DCTCP adapts to different signals, we reimplemented DCTCP in BPF. New features in Linux allow BPF-based congestion control, and we can use the same logic as before to attach a BPF congestion control algorithm to a new socket. However, we also want to be able to upgrade the BPF-based congestion control algorithm "on the fly," replacing the algorithm used by an existing connection. Although it isn't practical to "upgrade" Cubic to DCTCP (if ECN wasn't negotiated, the signal won't be there), replacing one "version" of DCTCP with another allows us to keep fewer versions in use. The key feature here is "bpf-iter," which allows running a loop over all sockets in the system. With this loop, we can replace the congestion control algorithm on every active socket. This is far better than alternatives (drain a datacenter, terminate connections, or wait until all the old connections disappear).

Implementing eBPF CCA. We leverage `struct-ops` [19], an eBPF interface to implement DCTCP through specific kernel function pointers, to create an eBPF program that provides a `tcp-congestion-ops` structure [1] implementation to the TCP subsystem. This capability allows us to manage CCAs similar to all the other eBPF programs we already manage in the fleet. Our DCTCP eBPF implementation closely matches the kernel eBPF example [18].

Managing eBPF CCAs. We built *NetEdit* [16], an agent that orchestrates the composition, deployment and life-cycle management of network eBPF programs across our fleet of servers. NetEdit supports implementation, experimentation, testing and rollout of custom CCAs. This allows us to select

fleetwide defaults for different RTT and also run active experiments at desired scales (specific services or data center). We push a new version of NetEdit almost every week. This allows fast iteration on CCA changes.

7 Ongoing Work, Limitations, Enhancements

This paper up to this point has been primarily about enabling DCTCP and making it work for in-region traffic, and enabling ECN on ToR-switch downlinks. This was a large first step. In this section, we describe some of the follow-on steps: enhancements we made to signaling, and CCA development based on our experience with the deployment. We also list the limitations of DCTCP, in particular for our traffic, and ongoing and future work to mitigate the limitations.

7.1 ECN marking on other hotspots

Our focus on marking from the ToR down ignored all the other links in the network maximizing benefit by targeting where most congestion occurs: most services overload their inbound network connection. However, there were a few cases where the ToR downlink was not the major bottleneck.

The first instance was when the ToR saturated its uplinks; we saw this situation in cases where there were several write-heavy services concentrated on racks, or when the rack did not have its entire capacity available due to maintenance. For such cases, we enabled ECN on the ToR link uplinks

The second instance was when ToR downlink congestion bled over up into the fabric: this happened when there were several read-heavy services concentrated on a rack, with their incoming traffic bursts synchronized at millisecond timescales. This resulted in high contention in both the ToR buffer, as well as the fabric switch immediately uplink of the ToR. To mitigate these cases, we deployed ECN on the fabric switch down links. For both this case as well as the ToR uplinks, we reused the original ECN/DropTail thresholds, which worked well enough. We saw a reduction in uplink queue length and buffer watermark on the switches as well as incoming retransmissions on the hosts.

The third—and surprising—instance was when we saw significant packet discards on the host NIC. This happened on the newer generation faster NICs; our hypothesis is that the host CPU is unable to keep up with faster bursts, resulting in the local NIC buffer overflowing. ECN on the NIC buffer is available to us on a subset of our vendor NICs, however, only one vendor implementation allows us to turn on this feature without rebooting the NIC. None of the vendors provided the means to tune the threshold—in fact, even the marking thresholds are not public information. These limitations meant that we could not deploy NIC ECN marking; however, limited testing showed promise in reducing NIC drops when we enabled it on one vendor NIC with just the default threshold.

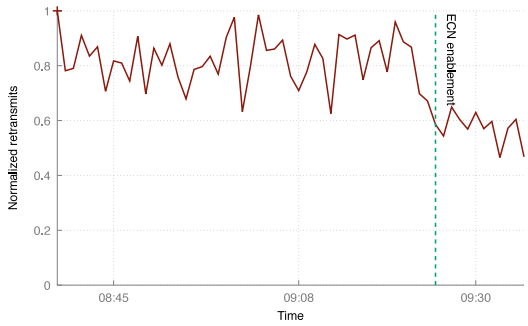


Figure 4: ECN marking on NICs reduces retransmissions.

We tested this feature on one host that was experiencing substantial NIC buffer drops. Turning on this feature reduced incoming retransmissions by about 30% (Figure 4). As host NICs become faster, challenging the CPU's ability to keep up with the incoming traffic, we expect that host-network congestion will become a larger problem for us to resolve.

7.2 Limitations of DCTCP

DCTCP does not solve all congestion issues. Figure 1 showed that DCTCP significantly reduced retransmission rates; it also shows that the gains are uneven across regions, and there is a significant amount of retransmissions that DCTCP is unable to solve. Some of it was due to other hotspots in the network and NIC that we observed and in some cases fixed (Section 7.1). A significant amount, however, is caused by traffic and network characteristics, which DCTCP, in its current form, is unable to solve. Our traffic is characterized by *short* and *heavy* incast bursts [13]. Short bursts do not allow the incast senders to converge, resulting in high buffer usage even with ECN. Heavy of the incast means that a CWND of 1, the lowest DCTCP can maintain, is too high. Other work has tackled high-degree incast [2, 14, 17, 21]; they require enhanced hardware support and a new networking stack. Our bursts can also synchronize across hosts in a ToR: with our ToR buffers being shallow and dynamically shared, synchronized bursts result in contention and high buffer pressure on the ToR, meaning that bursts can receive variable buffer allocation depending on the degree of synchronization.

Although DCTCP by itself is unable to solve the above challenges, we are working on complementary systems and better CCAs that can. One area of ongoing work is receiver-based flow control to help senders converge faster and more reliably [22]; this has resulted in significant wins. Buffer tuning to manage contention is a promising area of research as well. Ultimately though, ECN is a coarse signal which has its limitations; we are working on using delay-based signals that offer more responsive congestion control.

Jumbo frames. Jumbo frames are more efficient on the network; however, enabling Jumbo faced challenges to enabling DCTCP—different kernels and NICs support larger packets

differently; there are bugs in the handling of, say, large TCP fast open SYN packets or in handling a mixed configuration of MTU sizes on MH NICs. We note that open questions exist on how jumbo and non-jumbo frames interact, and whether our empirically derived thresholds need revisiting as frames are larger. We leave that for future work.

7.3 DCTCP implemented in BPF

We reimplemented DCTCP using the recent "bpf-cc" system, a task made a bit easier by the software that used BPF to enable DCTCP. This alternate implementation of the same CCA allows us to make modifications both minor—such as experimenting with different parameters—and major—using the established framework of DCTCP's response to ECN as a foundation, but building-in a response to delay or loss that differs slightly for our environment. Finally, our BPF implementation of DCTCP allows us to log internal CCA state (e.g., to log cwnd every RTT) rather than infer it from packet capture analysis.

We ran extensive tests comparing in-BPF to in-kernel implementations of DCTCP, using long flows as well as small and large RPCs. CPU use at sender and receiver was about the same (perhaps dominated by other parts of packet processing). Both implementations were fair to each other, getting roughly the same throughput alone and in contention.

8 Takeaways and Conclusion

We leave the community with a few observations about *deploying* CCAs in a large scale production network, which we hope will influence CCA research.

Deploying a CCA in production is not a flip of a switch. Safely and incrementally deploying changes leads to a transition period where there is a mixture of CCAs in use. This is not just due to hosts and network devices that have not picked up the changes, but also due to existing connections that have not (and might not be able to) flip over. The resultant transition period could vary depending on the complexity of the switch as well as the nature of the connections / traffic. This means that we have to consider performance during transition as well—if the stable states has excellent performance, but the (long) transition period could have significantly degraded performance, the switch will be more complex. Much of CCA research focuses on the stable state after the transition; insights into how the transition period could affect performance would be immensely useful to plan the deployment.

Data centers are complex and heterogeneous. CCAs must be simple, and forgiving. The mixture of hardware (NIC vendors, NIC speeds, switch vendors, switch ASICs, queues), software (OS kernels, driver versions), and applications (bursty, variable RTT, latency/throughput/tail sensitive), and the combinations thereof can be daunting—making any planned deployment a logistical challenge. While it may be impossible to account for every eventuality prior to deployment (indeed, we discovered a good number of issues only in retrospect), simpler CCAs can be easier to reason about and plan for. This means that new requirements either in the network (ECN, network telemetry) or in the host (hardware timestamps) need to be as minimal and simple as possible. The *tuning* of the new features must also be as forgiving as possible. Much work has gone into identifying ideal ECN thresholds—however, those assume ideal cases where there is no sharing with other CCAs, and the threshold search space is continuous, and not quantized. We were unable to deploy those ideal thresholds, instead having to do extensive testing to find "good-enough" thresholds that resulted in reasonable, though not ideal performance. A CCA that relies heavily on tuned parameters without graceful degradation is harder to deploy successfully in a large scale data center.

Expecting the unexpected. Sometimes long-deployed (and forgotten) configurations or optimizations can be exposed with new CCAs. An ideal CWND size that is only large enough to make full theoretical use of the network link, for example, might not be large enough to trigger the NIC to deliver a reassembled collection of packets, resulting in increased latency, or worse, breaking fairness. Simplicity of CCAs can also reduce probability of bad interactions with other components such as host and NIC optimizations, but it may not be possible to account for every eventuality.

Hotspots may occur in unexpected places. CCAs must have good fallbacks. CCAs moderate how concurrent flows share a *known* bottlenecked resource, but the location of the bottleneck (in-network, host-side, multi-host NICs) is not necessarily clear. A bottleneck in an unexpected location, which is not amenable to deploying the signal that the CCA relies on can be problematic. For example, when we found NIC bottlenecks, we realized that we could not deploy ECN there; therefore packet loss in NICs continued to occur, with DCTCP reacting suboptimally to such losses, being designed to respond optimally to ECN and not loss.

Ultimately, a CCA that might work well analytically and in simulation might not work well in practice—we hope that our experience guides researchers avoid common pitfalls, and design CCAs with an eye towards real-world deployability. Our experience with DCTCP has also guided our own evaluation of the potential of more advanced CCAs with reliance on wider set of signals: in-network telemetry, fine grained hardware timestamps, or early congestion signaling from switches.

Acknowledgments. This paper presents the work of several teams at Meta involved in successfully testing and deploying DCTCP. We thank Andrey Ignatov, Igor Pozgaj, James Zeng, Kiran Palan, Luwei Cheng, Mario Sanchez, Martin Lau, Nivin Lawrence, Omar Baldonado, Petr Lapukhov, Rajiv Krishnamurthy, Rob Sherwood, Rohit Puri, Russell Cloran, Sankaralingam Panneerselvam, Srikrishna Gopu, Takshak Chahande and the NSDI reviewers for their insightful feedback.

References

[1] tcp-congestion-ops. https://elixir.bootlin.com/linux/v5.5/source/include/net/tcp.h#L1043. [Online; accessed 25-February-2024].

[2] ABDOUS, S., SHARAFZADEH, E., AND GHORBANI, S. Burst-tolerant datacenter networks with Vertigo. In *CoNEXT* (2021).

[3] AHUJA, S. S., DANGUI, V., PATIL, K., SOMASUNDARAM, M., GUPTA, V., SANCHEZ, M., YAN, G., NOORMOHAMMADPOUR, M., RAZMJOO, A., SMITH, G., ET AL. Network entitlement: contract-based network sharing with agility and SLO guarantees. In *SIGCOMM* (2022).

[4] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center TCP (DCTCP). In *SIGCOMM* (2010).

[5] ARSLAN, S., LI, Y., KUMAR, G., AND DUKKIPATI, N. Bolt:sub-rtt congestion control for ultra-low latency. In *NSDI* (2023).

[6] BIRO, R., VAN KEMPEN, F. N., EVANS, M., MINYARD, C., LA ROCHE, F., HENDRICK, C., TORVALDS, L., COX, A., DILLON, M., GULBRANDSEN, A., AND CWIK, J. tcp_output.c. https://elixir.bootlin.com/linux/latest/source/net/ipv4/tcp_output.c#L339. [Online; accessed 3-September-2023].

[7] BORKMANN, D., WESTPHAL, F., AND JUDD, G. tcp_dctcp.c. https://elixir.bootlin.com/linux/latest/source/net/ipv4/tcp_dctcp.c#L100. [Online; accessed 3-September-2023].

[8] BRAKMO, L. bpf: BPF support for socket ops. https://lwn.net/Articles/725722/, 2017. [Online; accessed 3-September-2023].

[9] BRAKMO, L., BURKOV, B., LECLERCQ, G., AND MUGAN, M. Experiences evaluating DCTCP. In *Linux Plumbers Conference* (2018).

[10] DUKKIPATI, N., MATHIS, M., CHENG, Y., AND GHOBADI, M. Proportional rate reduction for TCP. In *SIGCOMM* (2011).

[11] FLACH, T., DUKKIPATI, N., TERZIS, A., RAGHAVAN, B., CARDWELL, N., CHENG, Y., JAIN, A., HAO, S., KATZ-BASSETT, E., AND GOVINDAN, R. Reducing web latency: the virtue of gentle aggression. In *SIGCOMM* (2013).

[12] GAO, P. X., NARAYAN, A., KUMAR, G., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *CoNEXT* (2015).

[13] GHABASHNEH, E., ZHAO, Y., LUMEZANU, C., SPRING, N., SUNDARESAN, S., AND RAO, S. A microscopic view of bursts, buffer contention, and loss in data centers. In *IMC* (2022).

[14] HANDLEY, M., RAICIU, C., AGACHE, A., VOINESCU, A., MOORE, A. W., ANTICHI, G., AND WÓJCIK, M. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM* (2017).

[15] JUDD, G. Attaining the promise and avoiding the pitfalls of TCP in the datacenter. In *NSDI* (2015).

[16] KANNAN, P., AND GUPTA, P. Netedit: Fine-grained network tuning at scale. https://atscaleconference.com/videos/netedit-fine-grained-network-tuning-at-scale-prashant 2022. [Online; accessed 25-February-2024].

[17] KUMAR, G., DUKKIPATI, N., JANG, K., WASSEL, H. M., WU, X., MONTAZERI, B., WANG, Y., SPRINGBORN, K., ALFELD, C., RYAN, M., ET AL. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM* (2020).

[18] LAU, M. K. bpf: Add bpf-dctcp example. https://lore.kernel.org/all/20191221062620.1184118-1-kafai@fb.com/, 2019. [Online; accessed 25-February-2024].

[19] LAU, M. K. Introduce BPF STRUCT-OPS. https://lore.kernel.org/bpf/20200109003453.3854769-1-kafai@fb.com/, 2020. [Online; accessed 25-February-2024].

[20] LI, Y., MIAO, R., LIU, H. H., ZHUANG, Y., FENG, F., TANG, L., CAO, Z., ZHANG, M., KELLY, F., ALIZADEH, M., ET AL. HPCC: High precision congestion control. In *SIGCOMM* (2019).

[21] LIU, K., TIAN, C., WANG, Q., ZHENG, H., YU, P., SUN, W., XU, Y., MENG, K., HAN, L., FU, J., ET AL. Floodgate: Taming incast in datacenter networks. In *CoNEXT* (2021).

[22] MADHAVAN, B., AND DHAMIJA, A. Tackling dc congestion and bursts. https://atscaleconference.com/videos/netedit-fine-grained-network-tuning-at-scale-prashant 2022. [Online; accessed 25-February-2024].

[23] MISUND, J., AND BRISCOE, B. Disentangling flaws in linux DCTCP. *arXiv preprint arXiv:2211.07581* (2022).

[24] MONTAZERI, B., LI, Y., ALIZADEH, M., AND OUSTERHOUT, J. Homa: A receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM* (2018).

[25] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network's (datacenter) network. In *SIGCOMM* (2015).

[26] VEERARAGHAVAN, K., MEZA, J., MICHELSON, S., PANNEERSELVAM, S., GYORI, A., CHOU, D., MARGULIS, S., OBENSHAIN, D., PADMANABHA, S., SHAH, A., ET AL. Maelstrom: Mitigating datacenter-level disasters by draining interdependent traffic safely and efficiently. In *OSDI* (2018).

[27] ZHANG, Q., LIU, V., ZENG, H., AND KRISHNAMURTHY, A. High-resolution measurement of data center microbursts. In *IMC* (2017).

TECC: Towards Efficient QUIC Tunneling via Collaborative Transmission Control

Jiaxing Zhang[1,2], Furong Yang[1*], Ting Liu[1,2*], Qinghua Wu[2,3✉], Wu Zhao[1], Yuanbo Zhang[1]*
Wentao Chen[1], Yanmei Liu[1✉], Hongyu Guo[1], Yunfei Ma[1], Zhenyu Li[2,3]
[1]Alibaba Group, [2]University of Chinese Academy of Sciences, [3]Purple Mountain Laboratories, China

Abstract

In this paper, we present TECC, a system based on collaborative transmission control that mitigates the mismatch of sending behavior between the inner and outer connections to achieve efficient QUIC tunneling. In TECC, a feedback framework is implemented to enable end hosts to collect more precise network information that is sensed on the tunnel server, which assists the inner end-to-end connection to achieve better congestion control and loss recovery.

Extensive experiments in emulated networks and real-world large-scale A/B tests demonstrate the efficiency of TECC. Specifically, compared with the state-of-the-art QUIC tunneling solution, TECC significantly reduces flow completion time. In emulated networks, TECC decreases flow completion time by 30% on average and 53% at the 99th percentile. TECC also gains a reduction in RPC (Remote Procedure Call) request completion time of 3.9% on average and 13.3% at the 99th percentile in large-scale A/B tests.

1 Introduction

Internet privacy has become a more and more serious concern for all Internet residents today. Although policymakers have been making regulations (e.g., GDPR [1]) on Internet privacy stricter than ever to protect users from the abuse of their personal information, these regulations may only be respected by legitimate organizations (e.g., large service providers such as Google), not by malicious hackers on the Internet. Therefore, many knowledgeable people resort to privacy-preserving technologies to further secure their surfing on the Internet.

iCloud Private Relay [2] (PR) is a new privacy protection service announced by Apple at its developer conference (WWDC) in June 2021, which aims to protect customers' Internet activities initiated via the Safari web browser. It uses an architecture with two layers of proxies to guarantee that no one in the middle can directly correlate the user and the target service accessed by the user. In PR's architecture, users' traffic is first routed to an Apple-controlled ingress proxy and the ingress proxy then forwards the received traffic to an egress proxy hosted by third parties (e.g., Cloudflare, Akamai, etc) that finally initiates connections to the target hosts. As Apple has a considerable amount of share on the global smartphone market and PR has a much lower entry barrier compared with traditional privacy-preserving technologies such as Virtual Private Networks (VPNs), Proxies, and The Onion Router (Tor), a significant usage increase of PR is envisioned [3, 4].

To steer user traffic to an egress proxy via an ingress proxy, a tunneling protocol called MASQUE (Multiplexed Application Substrate over QUIC Encryption) is used in PR. In this MASQUE use-case, a client first initiates a QUIC-based HTTP/3 connection to a tunnel server (an ingress proxy) and uses an extended HTTP CONNECT method [5] to open a UDP tunnel towards a target server (an egress proxy). Afterward, the client can communicate with the target server using QUIC (UDP-based) through the opened tunnel. This ultimately leads to a QUIC-in-QUIC communication pattern, as shown in Figure 1.

However, as disclosed by recent studies [4, 6], such a QUIC-in-QUIC communication pattern enabled by MASQUE faces several performance challenges. MASQUE only provides a tunneling mechanism and leaves the impact of tunneling on the performance of end-to-end (E2E) connections unexplored. There are also a few recent studies [7–11] that discussed using MASQUE or similar QUIC tunneling schemes to enhance the performance of E2E connections, e.g., accelerating the recovery of packets lost at the last-mile wireless link. Nonetheless, there remains a lack of systematic measurement of MASQUE, which is of great importance for designing efficient QUIC tunneling. To fill the gap, we first conducted a thorough measurement study to dissect the performance of MASQUE in depth. We then identify two fundamental problems that lead to sub-optimal performance:

- *Retransmission in the tunnel:* While retransmission in the tunnel is crucial for accelerated loss recovery, it may cause duplicated retransmissions by both the tunnel server and the server, leading to bandwidth waste and exacerbated network congestion.
- *Congestion control (CC) in the tunnel:* CC in the tunnel is necessary as it can reduce the tunnel's retransmission rate and alleviate network congestion. However, it causes a sending rate mismatch between the inner and outer connection, which can severely degrade E2E throughput.

To tackle these issues, we propose TECC which makes the end hosts and the tunnel server collaboratively control their transmission. In a nutshell, TECC works as follows: network information collected on the tunnel server is fed back to the server through the "Tunnel Server->Client->Server"

* Co-first authors
Corresponding authors: Qinghua Wu (wuqinghua.mail@gmail.com), Yanmei Liu (miaoji.lym@alibaba-inc.com)

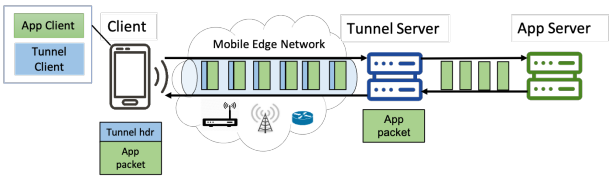


Figure 1: The QUIC-in-QUIC communication pattern enabled by QUIC tunneling

path, which aligns the server's retransmission and sending behavior with the tunnel server. At the same time, TECC offloads the server's CC to the tunnel server, reducing the control loop from the original E2E path to the path between the tunnel client and tunnel server.

We implement TECC and integrate it with the Taobao Android app. Extensive experiments in emulated network environments and real-world large-scale A/B tests demonstrate its efficiency. To sum up, our contributions are:

- We conducted a set of experiments to systematically evaluate MASQUE. An in-depth analysis based on the results reveals the impact of transmission modes (using QUIC streams or datagrams for tunneling E2E packets), retransmission, and CC on the performance of MASQUE.
- Motivated by our key observations on the fundamental limitations of MASQUE, we propose a collaborative transmission control system, TECC, which coordinates the retransmission and CC of the inner and outer connections to achieve enhanced performance.
- We carried out extensive experiments in networks emulated by Mahimahi [12] and large-scale A/B tests in which over 10 million users were involved. Compared with the vanilla MASQUE, TECC decreases the flow completion time (FCT) by 30% on average and by 53% at the 99th percentile in emulated networks. In production networks, TECC reduces the FCT of RPC requests by 3.9% on average and 13.3% at the 99th percentile.

2 Background

2.1 MASQUE overview

MASQUE is a set of specifications developed at the IETF MASQUE working group, which defines new tunneling mechanisms that can leverage QUIC to tunnel E2E UDP and IP packets. Compared with previous tunneling mechanisms (e.g., TCP-based or IP-based tunneling), it benefits from the superior features of QUIC, e.g., 0-RTT connection establishment, more accurate RTT estimation and loss detection, and elimination of head-of-line (HoL) blocking between independent data streams.

Currently, there are two released RFCs in MASQUE: RFC 9297 of "HTTP Datagrams and the Capsule Protocol" [13] and RFC 9298 of "Proxying UDP in HTTP" [5]. The former defines HTTP Datagrams that are the carriers of E2E packets in the tunnel. HTTP Datagrams can be used for conveying multiplexed and potentially unreliable datagrams (e.g., UDP or IP packets) inside an HTTP connection (a tunnel connection). Note, if the tunnel connection is based on HTTP/3 over QUIC, HTTP Datagrams can be transmitted in two different modes: **stream mode** and **datagram mode**. In the stream mode, QUIC streams are utilized to reliably transmit HTTP Datagrams, whereas QUIC datagrams [14] are leveraged to unreliably convey HTTP Datagrams in the datagram mode. The latter specification defines the signaling protocol to set up UDP tunnels inside an HTTP connection. Specifically, depending on the HTTP version of the tunnel connection, a tunnel client can request the tunnel server to open a UDP tunnel via sending an extended HTTP CONNECT request [15,16] or an HTTP Upgrade request [17] with the IP and port of the target server encoded in the request. If the tunnel server accepts such a request, the tunnel is established and the packets of the E2E connection can be transmitted over the opened tunnel. As the tunnel server knows about the addresses of both the client and server, it can forward E2E packets to the correct targets. At the time of writing, there are also a few other specifications under development, e.g., the signaling protocol to establish IP tunnels [18] and the QUIC forwarding mode [19] to reduce the wire and encryption overhead.

In this paper, we mainly focus on the QUIC-in-QUIC use-case enabled by MASQUE UDP tunnels, which means both the tunnel connection and the E2E connection are based on QUIC. We refer to the stream mode and the datagram mode as **MST** (MASQUE stream tunnel) and **MDT** (MASQUE datagram tunnel) in the following paragraphs respectively.

2.2 The potential issues of MASQUE

Although the MASQUE specifications provide functional mechanisms for QUIC-in-QUIC tunneling, the impact of such tunneling mechanisms on the performance of the E2E QUIC connection remains unclear.

First, the two tunnel modes, MST and MDT, could have different impacts on the E2E performance. In MST, each UDP tunnel uses one QUIC stream for the underlying transmission. As the QUIC stream is reliable, if the tunneled E2E packets are lost in the tunnel segment, they can be recovered by the tunnel connection, which is faster than the E2E retransmission. This benefit is not provided by MDT as QUIC does not retransmit QUIC datagrams in which E2E packets are encapsulated. But, MST also has a disadvantage compared with MDT. Due to the QUIC stream only delivering in-order data to its upper layer, MST has a significant HoL blocking problem where a lost packet leads to all subsequent packets being blocked in the tunnel. In MDT, as the QUIC datagrams are independently delivered to the upper layer, the HoL problem does not manifest. Consequently, there are some attempts

(e.g., [7]) trying to combine the advantages of MST and MDT by enabling retransmission for MDT (**RMDT**). Nevertheless, a quantitative comparison between these tunnel modes is essential to understand how they perform in practice.

Second, in MST and RMDT, there may be excessive retransmissions as both the tunnel connection and the E2E connection can retransmit packets lost in the tunnel segment. Excessive retransmissions may exacerbate the congestion in the tunnel and waste bandwidth. Applying CC on the tunnel connection may alleviate this problem, but, it also introduces two layers of nested CC, as the E2E connection also performs CC. The nested CC may lead to unwanted sending behavior mismatches between the tunnel connection and the E2E connection.

3 A deep dive into MASQUE

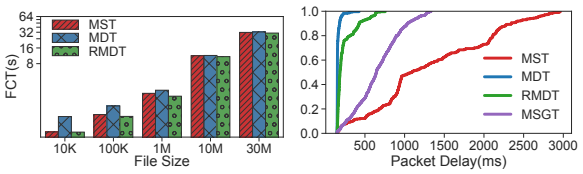
In this section, we first evaluate the performance of different tunnel modes. Then, the necessity of CC in the tunnel is assessed. Finally, as both the tunnel connection and the E2E connection have their own CC, the impact of nested CC is studied. All experiments in this section are conducted in networks emulated by Mahimahi [12].

3.1 The performance of MASQUE

Observation 1: HoL blocking in MST degrades throughput and bloats packet delay. Retransmission in the tunnel is crucial for reducing the complete time of short flows.

To evaluate the performance of different tunnel modes, we transferred files of different sizes through MST, MDT, and RMDT tunnels and calculated the average FCT of the E2E transfers. In the experiments, the E2E RTT and the RTT of the tunnel segment are 200 ms and 100 ms respectively. The loss rate of the tunnel segment is 10%, which emulates a congested link. As the E2E QUIC connection is certainly congestion-controlled (using BBR [20]), the CC of the tunnel connection is disabled in the experiments.

The results are shown in Figure 2a. As observed, MST achieves better performance for short flows compared with MDT, as MST can recover packets lost between the tunnel client and the tunnel server faster than MDT via local retransmission of the tunnel connection. However, as the file size increases, the performance gain of MST is gradually diminished. The diminished return is a result of the HoL blocking in MST as we will explain in the next paragraph. RMDT does the same as MST regarding faster loss recovery, but it performs better than MST as it does not suffer from HoL blocking. We also measured the E2E packet delay of the three tunnel modes, and the results are shown in Figure 2b. Compared with that of MDT and RMDT, the packet delay of MST is higher, which comes from the impact of HoL blocking. Because RMDT also retransmits lost packets, the delay of some packets appears longer from the server's view, as these



(a) FCTs of E2E transfers (b) CDF of E2E packet delay

Figure 2: The performance of different tunnel modes

packets have been retransmitted in the tunnel connection. This explains that the packet delay of RMDT is higher than MDT.

Analysis of HoL blocking in MST: When an E2E packet is lost in an MST tunnel, all subsequent E2E packets that have been sent via the tunnel are blocked in the QUIC stream and thus cannot be decapsulated until the lost packet is retransmitted by the tunnel connection. Consequently, the sender of the E2E connection may consider all packets sent after the lost packet as lost, which appears as a huge burst of packet losses. The burst of losses misleads the sender to reckon that it is in a persistent congestion event and it should significantly cut off its sending rate. As there are more packets in large transfers (thus more losses) and large transfers are more sensitive to throughput, the impact of MST's HoL blocking is more evident. Note that the failure of timely decapsulation of packets blocked in the QUIC stream also leads to increased E2E packet delay. Using multiple QUIC streams for the underlying transmission of an MST tunnel can alleviate the impact of HoL blocking, but, our experience indicates that it is still worse than RMDT. Specifically, we extend MST to use a group of QUIC streams (MASQUE stream group tunnel, MSGT). The packet delay of MSGT is between that of RMDT and MST, as shown in Figure 2b.

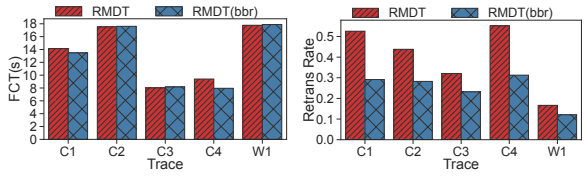
In conclusion, the earlier retransmission in the tunnel provided by MST and RMDT is indeed useful, especially for short flows. As MST suffers from the HoL blocking problem, RMDT seems the more desirable tunnel mode in practice.

3.2 The necessity of CC in the tunnel

Observation 2: Applying CC on the RMDT tunnel significantly reduces retransmissions and bandwidth waste. But, the E2E throughput does not benefit from the reduction in retransmissions.

In RMDT, packets lost in the tunnel could be retransmitted by both the tunnel connection and the E2E connection. Thus, there is a risk that the tunnel connection sends excessive packets to exacerbate the congestion in the tunnel segment as well as waste bandwidth resources. Ideally, enabling CC on the tunnel connection can reduce such a risk and boost performance.

We conducted experiments to compare the performance of RMDT with BBR and RMDT without CC. In the experiments, the E2E RTT and the RTT of the tunnel are 140 ms and 40 ms.



(a) FCT (b) Retransmission rate

Figure 3: The comparison between RMDT with and without CC

The loss rate of the tunnel is 10% and the bandwidth bottleneck is put on the tunnel segment. The bottleneck bandwidth is emulated by Mahimahi which replays traces collected from real-world networks. More details about the traces can be found in §5.1. The flow size in the experiments is 30 MB.

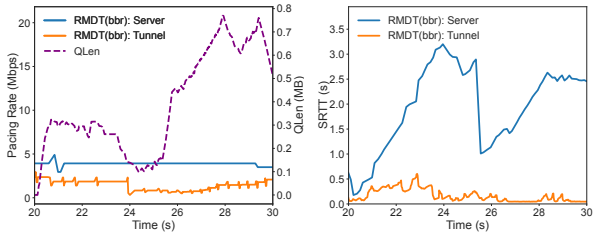
The average FCT and the overall retransmission rate (including retransmissions of the tunnel and E2E connection) are reported in Figure 3b. Indeed, RMDT with BBR significantly reduces retransmissions compared to RMDT without CC. When CC is disabled on the tunnel connection, all lost packets are retransmitted immediately into the network that may already be congested, which in turn triggers more retransmissions and aggravates network congestion. Despite the reduced retransmissions, no significant gain in FCT is observed in Figure 3a. We will see later that this is an outcome of the side effects of nested CC.

3.3 The impact of nested CC

Observation 3: Nested CC causes mismatched sending rates between the tunnel connection and the E2E connection, leading to an increase in the queue length at the tunnel server and bloated E2E smoothed RTTs (SRTTs).

Using the same emulated network environments as in §3.2, we launched long-lived transfers to investigate the impact of nested CC. Note that both the tunnel connection and the E2E connection employed BBR as their CC algorithm. During the transfers, the pacing rate and the SRTT of both the tunnel and E2E connection were recorded. In addition, the sending queue length of the tunnel connection (QLen) was also logged. We present the result of one of the transfers in Figure 4, as the conclusion drawn from the results is generic across different runs.

In Figure 4, we can clearly observe that there is a sending rate mismatch between the tunnel server and the server, leading to persistent queuing at the tunnel server and bloated E2E SRTTs. When packets are sent from the server to the client, the CC of the tunnel server controls the sending rate of the tunnel connection. However, if the sending rate of the tunnel server is lower than that of the server, the tunnel server will inevitably queue packets. The queuing time will increase as the rate difference grows. At the 24th second, the tunnel segment experiences a sudden drop in bandwidth without significant



(a) Pacing rate (b) SRTT

Figure 4: The impact of nested CC: the pacing rate and SRTT of the tunnel and E2E connection are depicted.

changes in SRTT. The tunnel server can quickly detect this drop in bandwidth and decrease the pacing rate accordingly, whereas the server cannot. This is mainly because the increase in SRTT prolongs the bandwidth updating cycle of BBR in the E2E connection.

3.4 Summary and implication

Our experiment results indicate that RMDT with CC is the more desirable tunnel mode in practice. First, it provides better loss recovery via retransmission in the tunnel while not suffering from the HoL blocking problem of MST. In addition, it limits the risk of sending excessive retransmissions to exacerbate network congestion in the tunnel segment by applying CC on the tunnel connection. However, the two layers of nested CC introduce sending behavior mismatches between the tunnel connection and the E2E connection, which ultimately leads to sub-optimal performance. This calls for coordinating the sending behavior of the two connections to eliminate such mismatches.

In fact, the tunnel server provides opportunities for the coordination of the tunnel and E2E connection. As the tunnel server is closer to the client than the server, it is able to sense the network conditions of the tunnel segment, which is usually the bottleneck (the last-mile access link resides here), in a more accurate and timely manner. The sensed network conditions can be fed back to the server, which assists the server in aligning its sending behaviors with the tunnel server.

4 TECC design and implementation

In this section, we present TECC, a collaborative transmission control system, where the server updates its sending rate based on the feedback from the tunnel server, enabling the efficient coordination of the sending behaviors of the server and the tunnel server.

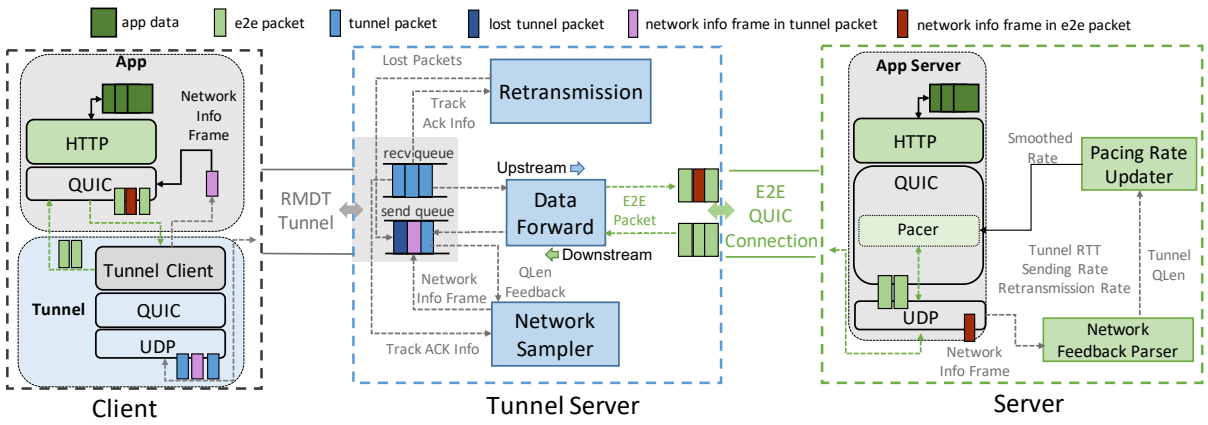


Figure 5: The overview of TECC

4.1 Overview

First, we present an overview of TECC as shown in Figure 5. TECC is based primarily on the RMDT mode and includes three components: the client, the tunnel server, and the server:

Client: Client application modules are mainly responsible for sending and receiving E2E connection data packets. This application uses the QUIC protocol, and the generated QUIC packets will be sent to the tunnel module for encapsulation preparation for entering the tunnel connection. Conversely, QUIC packets received from the tunnel connection will also be sent to the application module. The tunnel module will pass the collected tunnel network info frames to the application module, which will then encapsulate the frame into a QUIC network info frame and pass it to the E2E server application module.

Tunnel Server: The tunnel server mainly employs RMDT. There are three main modules in the tunnel server as shown in Figure 5. The data forward module is responsible for forwarding E2E packets to the client and the server. The retransmission module detects whether the tunnel packets are lost and retransmits those lost packets. The network sampler module is placed in the tunnel server in order to collect important information about tunnel networks and transmission status. The module subsequently generates and transmits network info frames containing the collected information to the client.

Server: The server is mainly responsible for maintaining E2E application modules, and its sending behavior is controlled by the feedback information of the tunnel network. When receiving network feedback information from the client, it parses different network information and uses it for sending rate updates.

4.2 Collaborative transmission control

TECC is based on the RMDT framework. The tunnel server uses the tunnel connection to provide feedback on detected tunnel network information, as well as its own sending and queuing status to the client. After receiving the feedback, the client inserts it into the E2E QUIC frame. Finally, the server updates its sending rate through network feedback of the QUIC frame. TECC achieves two major goals: 1) By quickly collecting bottleneck bandwidth in the tunnel server, the server can respond promptly to dynamic network changes. 2) Through feedback on the retransmission and queue status in the tunnel server, the server can decrease its rate for draining queues, thus reducing queuing delay.

4.2.1 Inaccurate CC in the server

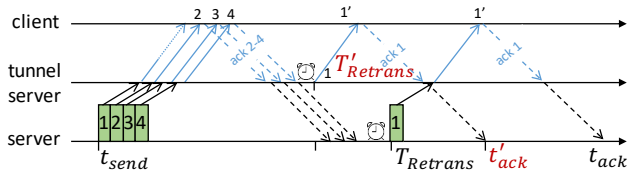


Figure 6: Illustration of retransmission with Tunnel

ACK packets play a vital role in providing network state measurements for server congestion control algorithms. However, due to the impact of RMDT, these packets can inevitably suffer from delays. As shown in Figure 6, assuming the RTT from the client to the server is T_s, and the tunnel is T_t. The retransmission detection time in the server and tunnel is $T_{Retrans}$ and $T'_{Retrans}$, respectively. At the time $t_{send} = 0$, the server transmitted packets with sequence numbers 1-4, but the packet with sequence number 1 was lost in the tunnel link. If the tunnel server does not retransmit, the server receives the ACK packet with sequence number 1 at $t_{ack} = T_{Retrans} + T_s$. However, if the tunnel server retransmits the packet, the server receives the retransmitted ACK packet at $t'_{ack} = 1/2(T_s - T_t) + T'_{Retrans} + T_t + 1/2(T_s - T_t) = T'_{Retrans} + T_s$. However, since the server does not have the knowledge of whether the tunnel server has retransmitted the lost packets, the delays calculated for the packet(s) at the

time t'_{ack} is $Delay(t'_{ack}) = t'_{ack} - t_{send} = T'_{Retrans} + T_s$. And for packets retransmitted by the server, the delay is $Delay(t_{ack}) = t_{ack} - T_{Retrans} = T_s$. In addition, the delay in the tunnel server also increases extra response time. Therefore, these can result in an incorrect evaluation of delay by the server, which affects the accuracy of the server-side congestion control.

4.2.2 Tunnel feedback

To solve the problem of inaccurate CC, we propose a collaborative feedback control framework that offloads the server-side CC to the tunnel server. This framework proactively perceives network information in the tunnel link close to the client and provides feedback to the server for sending rate estimation.

There are two main factors that affect the server's network assessment in the Tunnel Server: retransmission and congestion control. Retransmission causes the lost packets in the tunnel server to be delayed at least T_t after the client responds with an ACK packet, and the signal of packet loss is transformed into a signal of delayed packet delay. Therefore, we need to collect the proportion of retransmitted packets in Tunnel Server $r(t)$ at time t and the min RTT T_t of the tunnel server. Meanwhile, the delay in ACK perception will lead to a slower response of the server to the changes in the tunnel network. Therefore, transmitting the tunnel server's sending rate $Tr(t)$ can more quickly perceive the probe of the tunnel network. The congestion control of the tunnel server makes the queuing situation more severe. Therefore, recording the length of the queue within the connection in the Tunnel Server $q(t)$ can explicitly reduce the server's rate and empty the queue.

Feedback cycle: To determine the optimal tunnel feedback cycle τ, we may need to conduct experiments and evaluate the trade-offs between the accuracy of feedback and the overhead of sending feedback messages. The RTT between the client and the tunnel server is T_t. In our extensive experiments, setting τ to T_t yields high accuracy at a low cost, therefore we use T_t as the cycle τ for tunnel feedback.

4.2.3 Server's pacing rate updating rule

The server in the cooperative scheme employs a pacing-based CC algorithm to regulate its sending rate. It makes sense to use the pacing mechanism in the tunneling scenario to prevent the accumulation of tunnel server queues caused by burst traffic. Therefore, we use the server's sending rate $Sr(t)$ at time t as an indicator to control the sending of packets. In addition to the sending rate, window-based congestion control can also be supported, with the send control window, $W(t) = Sr(t) \cdot T_s$, limiting the total number of packets sent in an RTT. In order to better understand, the relevant parameters are presented in Table 1.

The congestion control mechanism on the server is triggered when it receives a feedback packet containing tunnel information frame from the tunnel server. After parsing the tunnel information frame, server obtains the latest sending rate $Tr(t)$ of the tunnel server. To ensure that the tunnel server and the server send packets at the same rate, we assign the sending rate $Sr(t)$ of the server to $Tr(t)$.

Parameter	Explanation
$Sr(t)$	server's sending rate
$Tr(t)$	tunnel server's sending rate
$U(t)$	proportional parameter, $U(t) \in [0,1]$
r_{ai}	additive increase part
$q(t)$	queue length of the tunnel server
θ	the time needed for the server to empty its queue
δ	proportional parameter, approximately 1
$r(t)$	retransmission rate of the tunnel server
max_pf	maximum penalty factor on the queue

Table 1: Explanation of parameters

If $Sr(t) > Tr(t)$ and the bottleneck link is the link between the client and the tunnel server, the queue at the tunnel server will be built up rapidly and grow gradually. However, if $Sr(t) < Tr(t)$, the tunnel server is limited by the number of packets sent by the server and may not have enough packets to utilize the available bandwidth. Therefore, to avoid queue creation while maximizing the utilization of bottleneck bandwidth, it is important to ensure that $Sr(t) = Tr(t)$.

Additionally, in traditional congestion control algorithms, the sending rate rapidly increases at the beginning of connection establishment to occupy the bandwidth. However, by congestion control in tunnel, servers pacing as $Tr(t)$ can avoid the data limitation at the early stage of connection establishment to probe the bandwidth at the tunnel server. The server pacing rate is:

$$Sr(t) = U(t) \cdot Tr(t) + r_{ai} \tag{1}$$

Due to bandwidth changes or response delays, the tunnel will inevitably build up a queue. To ensure a consistent rate, it will be impossible to empty the queue completely. Therefore, we use $U(t) \in [0,1]$, which changes in real-time according to the tunnel link, as shown in Equation 1. Additionally, we use the multiplicative-increase/multiplicative-decrease (MIMD) strategy to enable the server to quickly respond to tunnel link changes, and add an additive increase (AI) part to ensure fairness, which decouples link control and fairness. This approach is inspired by HPCC [21].

4.2.4 Penalty for building up queue

In a tunneling scenario, a tunnel server hosts multiple E2E connections simultaneously, which makes it more prone to queuing than traditional E2E connections. This is due to a reduced bandwidth of the tunnel link. Therefore, a penalty mechanism for queue establishment is needed in the tunneling scenario to reduce the sending rate and empty the queue in

time. This ensures that the queue at the tunnel server remains almost empty.

There are many traditional congestion control algorithms that guarantee low queue length. For example, DCTCP uses ECN markers to feedback on the queueing of the switch and reduces the sending window by ECN feedback [22]. However, these algorithms have limited transmission information and their performance is limited by ECN thresholds, which makes them inflexible. Additionally, HPCC algorithms can transmit detailed intermediate link load information, which can detect the network queueing situation promptly and accurately for more precise regulation [21]. However, they have complex software and hardware implementations, which makes them difficult to deploy. In the tunneling scenario, the feedback framework can easily feed back the network status and queuing situation to the server for congestion control.

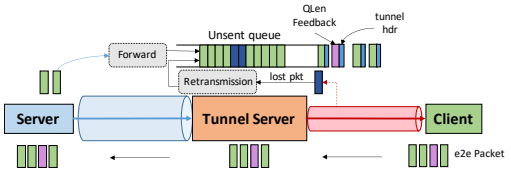


Figure 7: Queueing in the tunnel server

Estimate server pacing rate based on QLen: The tunnel server is responsible for transmitting E2E packets from the server to the client in the downlink, which can result in the buildup of the queue when the server's transmission rate $Sr(t)$ surpasses the rate of transmission $Tr(t)$. As shown in Figure 7, the minimum observed queue length (QLen) within a feedback cycle τ is denoted as $q(t)$, representing the QLen at time t.

The server calculates the pacing rate to drain the queue as:

$$Sr(t) = Tr(t) - \frac{q(t)}{\theta} \tag{2}$$

The parameter θ represents the time needed for the server to empty its queue and is useful in preventing abrupt changes to the sender rate. Usually, θ is set as a fraction of T_s. However, setting a small θ value can lead to inadequate bandwidth for packet probing in the tunnel server, while a large θ value may cause a delay in clearing the queue. To find an appropriate value of θ, we conducted lots of experiments with various values of θ. Eventually, we discovered that it achieves better performance when θ is around $2/3T_s$. Therefore, our research set $\theta = 2/3T_s$.

Alleviate congestion caused by retransmissions: Although retransmission from the tunnel server might decrease the time required for loss recovery, it can increase congestion on the tunnel link. This occurs because retransmitted packets are inserted into the queue of the tunnel server. Therefore, when determining the appropriate queue length, it is important to consider the retransmission of lost packets.

$$Sr(t) = Tr(t) - \frac{q(t) + \delta r(t) Tr(t)}{\theta} \tag{3}$$

The retransmission rate of the tunnel server is represented by $r(t)$. We assume that $r(t)$ remains constant throughout the feedback delay $T_s + d_q$, resulting in $r(t + T_s + d_q) = \delta r(t)$, where δ is approximately 1 (e.g., $\delta = 0.95$). At time $t + T_s + d_q$, the queue length is the sum of retransmitted packets $\delta r(t) \cdot Tr(t)$ and the queue length $q(t)$ at time t.

From Equation 1 and Equation 3, we get

$$U(t) = 1 - \frac{q(t) + \delta r(t) Tr(t)}{\theta Tr(t)} \tag{4}$$

When there is a sudden drop in the bandwidth of the tunnel link, the queue will experience significant growth. Therefore $U(t)$ may decrease to near 0. In order to prevent a rapid decrease in the server rate, We place constraints on the variable $U(t)$ to remain within a specific range:

$$U(t) = \max\{1 - \frac{q(t) + \delta r(t) Tr(t)}{\theta Tr(t)}, 1 - max_pf\} \tag{5}$$

The symbol max_pf represents the maximum penalty factor on the queue to prevent the server rate from dropping too fast. In later experiments, we set max_pf to 1/2.

Queue length noise filtering: To mitigate real-time queue length fluctuations caused by measurement errors, network delays, and other transient factors, we implement a queue length noise filtering procedure. Such fluctuations may cause the server to overreact and lead to substantial changes in transmission rates, subsequently impacting client application performance. We employ two primary methods. Firstly, we perform multiple detections at the tunnel server to obtain an accurate measurement of the queue length, allowing us to filter out invalid information. Secondly, we use Exponentially Weighted Moving Average (EWMA) to filter the queue ratio $U(t)$ before updating the server's transmission rate. This strategy ensures a smoother, more gradual adjustment of the server's send rate while minimizing the impact on client applications.

4.2.5 Retransmission trigger on the server

There are primarily two ways to trigger packet retransmission on the server: timeout retransmission and fast retransmission. The fast retransmission includes time-based fast retransmission and packet-based fast retransmission. The first type is triggered based on a time threshold, which we set as 9/8 SRTT. The second type is triggered based on receiving subsequent packet ACKs, with an initial threshold value of 3. Due to the retransmissions of the tunnel server, the SRTT of the server should increase. Thus, to suppress duplicated retransmissions, we increase the time-based fast retransmission threshold of the server by T_t. Regarding packet-based fast retransmission,

since retransmission by the tunnel server exacerbates the out-of-order client ACK packets, the threshold for reordering increases gradually as spurious retransmission by the server is triggered, leading to a decrease in the probability of server retransmission. Therefore, the server's rate of duplicated retransmission is relatively small and does not exceed 1% in our experiments.

4.2.6 Fairness

In tunnel scenarios, multiple flow connections go through the same tunnel server to different servers on the same bottleneck link. Unlike traditional E2E congestion control algorithms, the bottleneck link detection for tunnel connections relies on the sending rate of the tunnel server, which is dependent on the information of the tunnel connections. However, because the pacing rate update rule uses MIMD strategy, fairness cannot be provided among multiple competing tunnel flows. To achieve fairness among multiple flows, we add an additive-increase (AI) term to the update of pacing rate, making it a multiplicative-and-additive-increase/multiplicative-decrease (MAIMD) scheme. This approach theoretically provides better fairness among multiple competing flows [23–26]. The additive-increase (AI) is set as follows:

$$r_{ai} = \frac{MSS}{T_s} \tag{6}$$

$$Sr(t) = U(t) \cdot Tr(t) + r_{ai} = U(t) \cdot Tr(t) + \frac{MSS}{T_s} \tag{7}$$

The server increases the pacing rate every time it updates by one Maximum Segment Size (MSS) in one server round-trip time, which is a small increase designed to control rate convergence while preventing network congestion. Achieving equilibrium requires that multiple competing tunnel flows pass through the same bottleneck link, resulting in the bottleneck bandwidth being constant between flows. Consequently, the server's sending rate is then modified based on the bottleneck bandwidth such that the sending rates of multiple flows can achieve equilibrium. By reaching equilibrium, fairness is maintained among competing flows.

The server's sending rate depends on the detected bandwidth of the tunnel server. Therefore, TECC's RTT fairness depends on the tunnel server's original congestion control algorithm (wherein we have used BBR [20]). Consequently, the collaboration approach and BBR have varying throughput rates under different tunnel RTT flows [27]. In tunnel scenarios, the original congestion control algorithm's unfairness can be mitigated even further. In §5.4, we demonstrate this through experiments including multiple tunnel flows with various RTTs.

Algorithm 1 Sender algorithm

Data: Tunnel Server Feedback: $Tr(t)$, $q(t)$, T_t
Result: Server: $Sr(t)$

function `UpdateSenderRate()`:
- $e(t) \leftarrow \frac{q(t)+\delta r(t) Tr(t)}{\theta Tr(t)}$
- $U(t) \leftarrow \max\{1-e(t), 1-max_pf\}$
- $U \leftarrow (1-ewma_weight) \cdot U + ewma_weight \cdot U(t)$
- $Sr(t) = U \cdot Tr(t) + \frac{MSS}{T_s}$
- **return** $Sr(t)$

The sender's sending rate must consider not only the tunnel server's sending rate but also the current queue length of the tunnel server and fairness. Ultimately, the complete logic of the sender's sending rate is illustrated in Algorithm 1.

4.3 Implementation

This section mainly discusses the methods we deployed TECC as well as the optimizations we made for large-scale deployment.

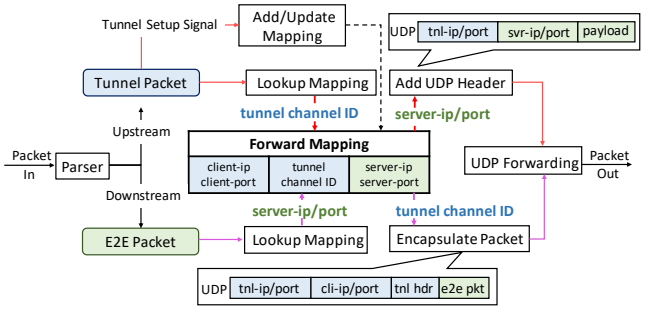


Figure 8: Data forwarding logic in the tunnel server: forwarding downstream/upstream E2E packets

Forwarding in the tunnel server: The packet forwarding mechanism in the tunnel server both encapsulates and decapsulates the E2E packets and directs them to their respective destination addresses, as shown in Figure 8. When a client establishes a tunnel connection to the tunnel server, the server's IP and port are forwarded to the tunnel server as well. After receiving the tunnel setup signal, the tunnel server generates a mapping of the tunnel channel ID that associates the client's IP and port and the server's IP and port. As a result, the E2E packets from the corresponding tunnel channel ID are decapsulated and transmitted to the UDP socket that is bound to the same server IP and port. Correspondingly, the E2E packets from the equivalent socket are encapsulated into the corresponding datagram. Furthermore, the tunnel server facilitates updating forwarded information. If clients require rebinding the forwarding mapping, they can retransmit the

tunnel setup signal on the relevant tunnel server.

Retransmission: Retransmission plays a crucial role in enhancing client performance in tunnels. In the downstream link, retransmission can significantly decrease packet loss recovery time in the tunnel connection. Furthermore, the tunnel server can preemptively detect packet loss from the client in the upstream link, enabling packet loss recovery ahead of time on the client side. We follow the retransmission mechanism in RMDT to eliminate blocking and ensure improved performance. Datagram retransmission requires the tunnel to track the lower-layer QUIC datagram's delivery status closely. Selective retransmission of data packets is adopted with indications of packet loss. The experiment in this paper mainly focused on download scenarios that necessitate retransmitting all data packets in the tunnel connection. Nevertheless, some time-sensitive video scenarios permit partial packet loss. Therefore, leveraging the client's application information allows us to retransmit crucial or time-constrained frames selectively, thereby enhancing bandwidth utilization.

Feedback implementation in the tunnel: The feedback in TECC relies on the support of both the QUIC protocol stack and the tunnel protocol stack. To accomplish this, we have developed the TUNNEL_INFO frame and QUIC_INFO frame in the tunnel protocol and the QUIC protocol. These frames are used to transmit feedback information from the tunnel server to the tunnel client and from the app client to the app server. Because both the tunnel client and app client are deployed on our APP, the tunnel client can directly transfer feedback information to the app client when receiving feedback information.

Tunnel server selection: It's important to select the appropriate edge tunnel server since users can experience better performance by selecting the edge tunnel server closest to them. In our app, the client obtains real-time information about the tunnel server from our own DNS. The tunnel client needs to obtain tunnel information, including the protocol version and the tunnel server information from our DNS first. It then establishes a tunnel connection to complete the transmission of the E2E connection through the tunnel packets. Our DNS owns the information of all tunnel servers and selects the tunnel servers that are close to the user and have sufficient resources.

Load balancing: Load balancing (LB) in the tunnel server is essential because the tunnel server has to handle a large number of tunnel requests and forward massive E2E connection packets. In practical deployments, a bidirectional LB strategy is required, which includes packets from the tunnel client and the server. To achieve this, we use multiple workers with NGINX. Due to the address migration in QUIC, the QUIC CID is considered instead of traditional network five-tuples [28]. Since "tunnel connection CID" and "E2E connection CID" don't follow the same generation rules, it can be challenging to balance the packets to the same worker. Therefore, during actual deployment, the CIDs in tunnel QUIC connections and E2E QUIC connections follow the same consistency encoding rules. This allows the same client's tunnel connection and E2E QUIC connection packets to be routed to the same worker.

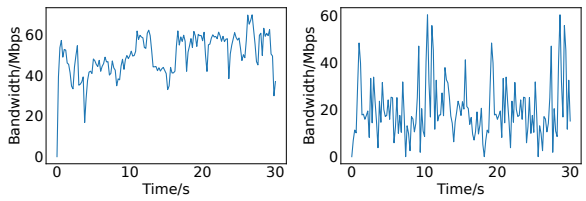
5 Evaluation

In this section, we present the evaluation of the TECC, which consists of two parts: online evaluation and simulated environment evaluation.

Online evaluation: In this part, we collected anonymous data from mobile users who upgraded our app with "QUIC Tunnel". To validate the improvement of TECC compared to the MST, MDT, and RMDT, we conducted a series of large-scale A/B experiments. The user scale of the experiment exceeds ten million. The experimental results show that the TECC significantly reduces completion time.

Evaluation in emulated networks: The Mahimahi-based emulation environment was established to evaluate the performance of the QUIC Tunnel. The test environment comprises three principal containers: a client running HTTP/3 and the tunnel stack, a tunnel server running the tunnel stack, and a server running HTTP/3. Mahimahi emulates the links from the client to the tunnel and from the tunnel to the server. We presume that the link from the client to the tunnel is the bottleneck link, so all test cases mainly involve client requests for a specific file. This download scenario is widely used in several applications, such as web browsing and video downloading.

5.1 Experiment setup



(a) Cellular Home (b) Wi-Fi Home

Figure 9: Bandwidth variation of different traces

Baseline: In the non-tunnel solution, the tunnel server acts as an intermediate forwarding node based on iptables, forwarding packets between the client and the server. MDT and RMDT come from vanilla MASQUE and are detailed in Sec 2. We also compared the forwarding mode (MFT) suggested by Apple to proxy QUIC in MASQUE similar to non-tunnel methods [29]. Since the performance of the non-tunnel solution is similar to MFT, we will refer to both as MFT later. In addition, the RTT between the tunnel server and the server is set to 100ms.

Traces: We used five real-world mobile network traces, four from cellular and one from Wi-Fi, namely: cellu-

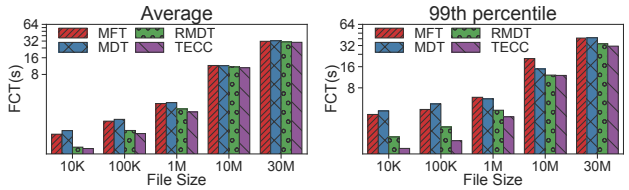


Figure 10: FCT of different solutions under various flow sizes

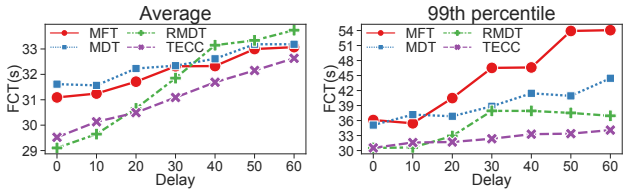


Figure 11: FCT of different solutions under various delays

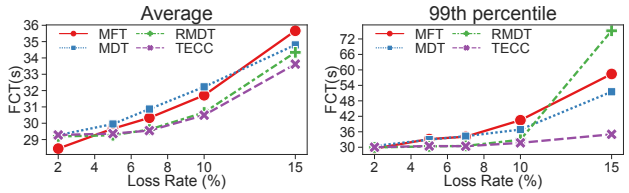


Figure 12: FCT of different solutions under various loss rates

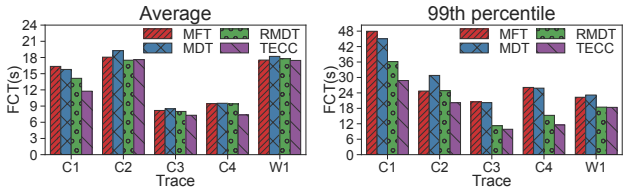


Figure 13: FCT of different solutions in mobile networks

lar_driving (C1), cellular_outdoor (C2), cellular_home (C3), cellular_subway (C4), wifi_office (W1). Among them, the detailed bandwidth information for C3 and W1 is shown in Figure 9. We also measured the performance at stable bandwidth, the bandwidth was set to 10Mbps.

5.2 End-to-end performance

Flow size: *Tunnel is an effective way to reduce the flow completion time, particularly for short flows.* We run the request for different size files from the server on 10Mbps bandwidth, the result shown in Figure 10. The tunnel solutions, including RMDT, and TECC, outperform non-tunneling MFT methods in the case of short flows (flow length less than 10M). The performance of MDT is similar to that of MFT. For short flows, packet loss delays can seriously reduce the overall flow completion time, while the tunnel's early retransmission mechanism can effectively reduce the packet loss recovery time and improve overall performance. As the flow length increases, the delay of a small number of lost packets has less impact on the overall flow completion time, so the benefit of the tunneling mechanism is relatively reduced. In TECC, the end-tunnel cooperative scheme outperforms RMDT. RMDT solves the HoL blocking problem by reducing the latency of blocking but leads to low bandwidth utilization due to a mismatch of the sending rate and bandwidth.

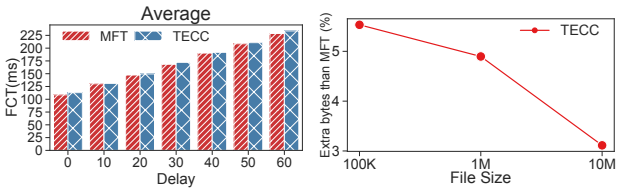
Client-Tunnel delay: *The benefits of the tunnel vary with the change of Client-Tunnel delay and lower delay results in higher benefits.* In our experiment, we transmitted thousands of 30M files with various delays on a 10Mbps network, and the results are presented in Figure 11. As the delay increases, the benefits provided by the tunnel gradually decrease. Compared to MDT, RMDT performs better and still has over 2% optimizations even at 50ms. However, due to rate mismatch, its performance declines when compared to MFT at higher delays. TECC is significantly optimized when compared to the others, and it still maintains good optimization effects even at 60ms. These findings indicate that the TECC continues to deliver certain optimization effects even in mobile edge networks with high RTT.

Loss rate in Client-Tunnel link: *The benefits provided by the tunnel increase as the loss rate increases, especially in tail optimization.* We conducted experiments involving the transmission of thousands of 30M files with varying loss rates on a 10Mbps network, and the results are depicted in Figure 12. When the packet loss rate is low, the additional overhead of the tunnel results in lower performance compared to MFT. However, as the packet loss rate increases to over 5%, the FCT of MDT and MFT are similar. Simultaneously, RMDT and TECC display superior performance compared to MFT. At the 99th percentile, when the packet loss rate exceeds 10%, TECC can reduce FCT by over 21.7%. These findings suggest that in mobile edge networks with a high loss rate, TECC can provide significant throughput enhancements.

Mobile networks: *TECC exhibits strong optimization effects in mobile networks, achieving an average reduction of over 15% at the 99th percentile.* To evaluate the performance of the tunnel in mobile networks with dynamic bandwidth fluctuations, we conducted tests on a large number of 30MB files under various mobile network traces. A comprehensive introduction to the trace can be found in the appendix. The result is shown in Figure 13. In terms of both average FCT and 99th percentile FCT, TECC outperforms MFT and other tunnel solutions across different network traces. In some cases, the average FCT of TECC was reduced by over 30%, and the 99th percentile saw a reduction of more than 53%. Although RMDT has limitations, they generally perform better than MFT in mobile network environments. This evidence suggests that TECC can perform well in different mobile network scenarios.

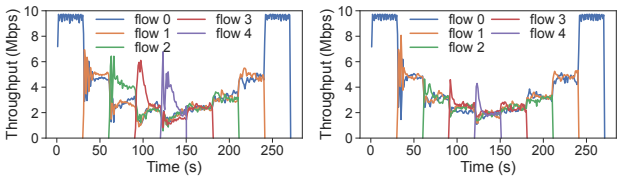
5.3 Tunnel overhead

Compared to not using the tunnel, the tunnel incurs additional overhead, primarily involving the setup delay in connection establishment and the extra packet overhead of packet encap-



(a) Setup Delay (b) Extra Tunnel Header

Figure 14: Tunnel overhead in networks without random losses



(a) TECC w/o AI (b) TECC with AI

Figure 15: Fairness among competing tunnel flows

sulation. Consequently, we respectively tested the overhead of the tunnel under different client-tunnel delays and different file request sizes, the result shown in Figure 14.

Setup delay: *The setup delay of the tunnel is higher, but overall it does not exceed 2%.* Owing to the processing overhead of the tunnel protocol stack and the delay in establishing the tunnel connection, the setup delay is slightly elevated. As the delay fluctuates, the increase in setup delay time remains relatively constant, suggesting that the setup delay does not exhibit significant changes with increasing delay.

Extra packet overhead: *The tunnel incurs additional packet overhead, which overall does not exceed 4% in long flow.* Due to the tunnel packet encapsulation in the QUIC datagram, TECC adds a datagram header, QUIC header, and other QUIC frame overhead compared to UDP. With the increase of data flow, the relative proportion of data packet overhead gradually decreases, from only 5% in the 100K to 3% when it reaches 10M.

5.4 Fairness

	MFT (BBR)	TECC (BBR)
Different Server Delay	0.8039	0.9916
Different Client Delay	0.6759	0.8620

Table 2: Jain's Fairness Index among different RTT flows

Flows with the same RTT: First, we demonstrate that the TECC can converge to fair bandwidth allocation for flows with the same RTT upon arrival and departure. We first start a client-to-server flow through a tunnel server, with a fixed 10Mbps bandwidth in the tunnel link. Then, we incrementally add a flow from a different client accessing a different server through the same tunnel server. Finally, we start tearing down flows one by one. As shown in Figure 15, without additive increase, the MIMD strategy may exhibit throughput jitter upon flow arrival, as flows in the startup phase of BBR [20] initially preempt the bandwidth, leading to a rapid increase in throughput for newly arrived flows. By adding an additive increase, the competition flow can converge more smoothly, making it less likely for existing flows in the link to be preempted by new flows, resulting in better fairness. By calculating Jain's fairness index, the value is greater than 0.99, ensuring fairness for TECC when tunnel flows with the same RTT enter the same tunnel link.

	mean	p95	p99	p999
MST	2.7%	1.9%	6.0%	17.5%
TECC	3.9%	4.5%	13.3%	36.0%

Table 3: Improvements of MST and TECC

RTT unfairness: In tunnel scenarios, including the two delays of client-to-tunnel server and tunnel server-to-server links, we conducted fairness experiments on flows with different delays in each link. For different server delays, we launched 8 different flows with server delays of 10-80 ms on the same 10Mbps bandwidth tunnel link, with a fixed client-to-tunnel delay of 10ms. Based on the average throughput of the 8 flows after convergence, we calculated the corresponding Jain's fairness index values. We conducted the same experiment for different client delay, with 8 different client delays of 10-80, and the results are shown in Table 2. We compared the cooperative solution's CC with MFT. Each experiment was run for 10 rounds, and the average index value was taken in the table. As the results show, compared to MFT, the CC of the cooperative solution can achieve RTT fairness on different server delay, because the tunnel detection by the server is mainly completed through the tunnel node, and the bandwidth allocation and detection frequency are related to the tunnel's RTT, thus achieving fairness in flows with different server delays. Different client delays are related to the original tunnel detection congestion control algorithm (BBR [20, 27]), but because the original end-to-end control loop is shortened to the client-to-tunnel, the impact of different RTTs on throughput is reduced, thus mitigating the RTT unfairness problem of the original congestion control algorithm.

5.5 Real-world A/B tests

We compared the completion times for RPC requests from over 10 million users who use vanilla MASQUE and TECC. To investigate the reduction of FCT with TECC, we deployed three different types of tunnels (MST, MDT, and TECC) to users in the same region. At the same time, the tunnel server

is deployed at the edge nodes closer to the selected users. In order to measure the effectiveness of TECC, we used the completion time of user requests with MDT as the baseline. The recorded relative improvements of the MST and TECC in reducing completion time are presented in Table 3.

MDT does not optimize the mobile edge network and is equivalent to direct forwarding similar to MFT. Our results demonstrate that, regardless of average or tail completion times, both MST and TECC outperform MDT. Furthermore, due to the HoL blocking problem and mismatch in CC actions in MST, TECC achieved a 7.3% improvement in the 99th percentile and an 18.5% improvement in the 999th percentile compared to MST.

6 Discussion

Multiple E2E connections in tunnel: In real-world network, it is common for a bottleneck link to transmit multiple network connections simultaneously. These different connections compete for bandwidth through their respective bandwidth probing techniques. While using a tunnel to transport multiple E2E connections, in addition to priority rules, the TECC tunnel can dynamically control the bandwidth probing of each E2E connection based on the type of each connection. This can achieve better traffic fairness on mobile edge networks.

Congestion control in tunnel: The congestion control at the tunnel server can prevent traffic congestion by restricting the sending rate of packets. However, this may result in longer delay of packets that have arrived at the tunnel server. In our work, we only compared the results when using CC or not. It is beneficial to design a CC framework which also considers the sojourn time of packets at the tunnel server, to balance the tradeoff between efficient congestion control and short delay. This is left as our future work.

Potential enhancements for MASQUE: Presently, the research efforts in the MASQUE working group primarily concentrate on providing generic solutions to proxy different protocols. However, employing these techniques maybe degrade user performance. TECC is dedicated to addressing the performance issues of MASQUE's generic solutions, but achieving an effective universal standard necessitates the support of MASQUE.

7 Related work

Last-mile performance optimization: For most applications where remote servers are involved, optimizing the wireless network close to the client is crucial for last-mile performance. CDNs cache user-requested resources in edge servers closer to user networks; however, the efficiency of CDNs decreases for processing dynamic data [30–32]. An alternative method is Performance Enhancing Proxy (PEP), which optimizes different feature links by breaking the E2E connection [33]. PEP mostly relies on TCP protocols to achieve performance gains in mobile and satellite networks [34–36]; however, it is not applicable to the header encryption and verification mechanism of QUIC packets. Sidecar [37] proposed an ACK-based protocol that enables the proxy to perceive E2E encrypted data, but servers need to explicitly perceive and communicate with the intermediate proxy, making deployment more difficult and increasing communication overhead. TECC leverages QUIC tunnel's own detection and feedback mechanisms to mitigate PEP's damage to E2E connection semantics and ensure data privacy by sending feedback information to the sender in the direction of the receiver.

Middlebox feedback for rate control: Feedback-based middleware has gained widespread application in optimizing server congestion control algorithms [21, 26, 38–40]. Zhuge [38] shortened the control loop by increasing the delay of ACK packets in wireless Access Points. ABC [26] controls server window size by explicitly marking "accelerate" or "brake" signals based on router's perception of network conditions. HPCC [21] obtains accurate load information and rate control via in-network telemetry (INT). However, these congestion control protocols are mostly based on router or switch information package condition detection of a network. In a tunnel, the middle server can actively detect network conditions and control packet transmission rate, thus providing servers with more rate information.

8 Conclusion

In this paper, we present TECC to optimize the E2E performance of the QUIC tunnel. Deployed at the tunnel server which is closer to the client, TECC provides faster and more precise network feedback to assist servers with a more accurate estimation of the available bandwidth. In emulated networks, TECC decreases flow completion time by 30% on average and 53% at the 99th percentile. TECC also gains a reduction in RPC (Remote Procedure Call) request completion time of 3.9% on average and 13.3% at the 99th percentile in large-scale A/B tests.

Acknowledgements

We thank our shepherd Vaibhav Bajpai and the anonymous reviewers for their constructive feedback. This work was supported in part by the National Key R&D Program of China (2022YFB2901800), Natural Science Foundation of China (U20A20180, 62072437), and Beijing Natural Science Foundation (JQ20024).

References

[1] General Data Protection Regulation (GDPR) – Official Legal Text. https://gdpr-info.eu/.

[2] Apple Inc. iCloud Private Relay Overview. https://www.apple.com/kr/icloud/docs/iCloud_Private_Relay_Overview_Dec2021.pdf, 2021.

[3] Patrick Sattler, Juliane Aulbach, Johannes Zirngibl, and Georg Carle. Towards a tectonic traffic shift? investigating Apple's new relay network. In *Proceedings of the 22nd ACM Internet Measurement Conference*, IMC '22, pages 449–457, New York, NY, USA, October 2022. Association for Computing Machinery.

[4] Martino Trevisan, Idilio Drago, Paul Schmitt, and Francesco Bronzino. Measuring the Performance of iCloud Private Relay. In Anna Brunstrom, Marcel Flores, and Marco Fiore, editors, *Passive and Active Measurement*, Lecture Notes in Computer Science, pages 3–17, Cham, 2023. Springer Nature Switzerland.

[5] David Schinazi. Proxying UDP in HTTP. RFC 9298, August 2022.

[6] Mirja Kühlewind, Matias Carlander-Reuterfelt, Marcus Ihlar, and Magnus Westerlund. Evaluation of quic-based masque proxying. In *Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC*, pages 29–34, 2021.

[7] Furong Yang, Yanmei Liu, and Yunfei Ma. A Configurable Retransmission Extension for HTTP/3 Datagrams. Internet-Draft draft-yang-masque-dgram-retrans-01, Internet Engineering Task Force, March 2023. Work in Progress.

[8] Zsolt Krämer, Mirja Kühlewind, Marcus Ihlar, and Attila Mihály. Cooperative performance enhancement using quic tunneling in 5g cellular networks. In *Proceedings of the Applied Networking Research Workshop*, pages 49–51, 2021.

[9] Michele Luglio, Mattia Quadrini, Cesare Roseti, Francesco Zampognaro, and Simon Pietro Romano. A quic-based proxy architecture for an efficient hybrid backhaul transport. In *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pages 144–146. IEEE, 2020.

[10] A Abdelsalam, Michele Luglio, Mattia Quadrini, Cesare Roseti, and Francesco Zampognaro. Quic-proxy based architecture for satellite communication to enhance a 5g scenario. In *2019 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–6. IEEE, 2019.

[11] Mike Kosek, Hendrik L. Cech, Vaibhav Bajpai, and Jörg Ott. Exploring proxying quic and http/3 for satellite communication. *2022 IFIP Networking Conference (IFIP Networking)*, pages 1–9, 2022.

[12] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: accurate {Record-and-Replay} for {HTTP}. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, 2015.

[13] David Schinazi and Lucas Pardue. HTTP Datagrams and the Capsule Protocol. RFC 9297, August 2022.

[14] Tommy Pauly, Eric Kinnear, and David Schinazi. An Unreliable Datagram Extension to QUIC. RFC 9221, March 2022.

[15] Patrick McManus. Bootstrapping WebSockets with HTTP/2. RFC 8441, September 2018.

[16] Ryan Hamilton. Bootstrapping WebSockets with HTTP/3. RFC 9220, June 2022.

[17] Roy T. Fielding, Mark Nottingham, and Julian Reschke. HTTP Semantics. RFC 9110, June 2022.

[18] Tommy Pauly, David Schinazi, Alex Chernyakhovsky, Mirja Kühlewind, and Magnus Westerlund. Proxying IP in HTTP. Internet-Draft draft-ietf-masque-connect-ip-13, Internet Engineering Task Force, April 2023. Work in Progress.

[19] Tommy Pauly, Eric Rosenberg, and David Schinazi. QUIC-Aware Proxying Using HTTP. Internet-Draft draft-ietf-masque-quic-proxy-00, Internet Engineering Task Force, August 2023. Work in Progress.

[20] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 14(5):20–53, 2016.

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

[22] Mohammad Alizadeh, Albert Greenberg, Dave Maltz, Jitu Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Dctcp: Efficient packet transport for the commoditized data center. 2010.

[23] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Comput. Netw. ISDN Syst.*, 17(1):1–14, jun 1989.

[24] Aditya Akella, Srinivasan Seshan, Scott Shenker, and Ion Stoica. Exploring congestion control. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2002.

[25] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 514–528, 2020.

[26] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. Abc: A simple explicit congestion controller for wireless networks. 2020.

[27] Mario Hock, Roland Bless, and Martina Zitterbart. Experimental evaluation of bbr congestion control. In *2017 IEEE 25th international conference on network protocols (ICNP)*, pages 1–10. IEEE, 2017.

[28] Martin Duke, Nick Banks, and Christian Huitema. QUIC-LB: Generating Routable QUIC Connection IDs. Internet-Draft draft-ietf-quic-load-balancers-16, Internet Engineering Task Force, April 2023. Work in Progress.

[29] Tommy Pauly, Eric Rosenberg, and David Schinazi. QUIC-Aware Proxying Using HTTP. Internet-Draft draft-pauly-masque-quic-proxy-06, Internet Engineering Task Force, March 2023. Work in Progress.

[30] Volker Stocker, Georgios Smaragdakis, William Lehr, and Steven Bauer. The growing complexity of content delivery networks: Challenges and implications for the internet ecosystem. *Telecommunications Policy*, 41(10):1003–1016, 2017.

[31] Erik Nygren, Ramesh K Sitaraman, and Jennifer Sun. The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

[32] Vytautas Valancius, Nikolaos Laoutaris, Laurent Massoulié, Christophe Diot, and Pablo Rodriguez. Greening the internet with nano data centers. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 37–48, 2009.

[33] Jim Griner, John Border, Markku Kojo, Zach D. Shelby, and Gabriel Montenegro. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135, June 2001.

[34] Carlo Caini, Rosario Firrincieli, and Daniele Lacamera. Pepsal: a performance enhancing proxy for tcp satellite connections. *IEEE Aerospace and Electronic Systems Magazine*, 22(8):7–16, 2007.

[35] Ye Li, Liang Chen, Li Su, Kanglian Zhao, Jue Wang, Yongjie Yang, and Ning Ge. Pepesc: A tcp performance enhancing proxy for non-terrestrial networks. *IEEE Transactions on Mobile Computing*, 2023.

[36] Xing Xu, Yurong Jiang, Tobias Flach, Ethan Katz-Bassett, David Choffnes, and Ramesh Govindan. Investigating transparent web proxies in cellular networks. In *Passive and Active Measurement: 16th International Conference, PAM 2015, New York, NY, USA, March 19-20, 2015, Proceedings 16*, pages 262–276. Springer, 2015.

[37] Gina Yuan, David K Zhang, Matthew Sotoudeh, Michael Welzl, and Keith Winstein. Sidecar: in-network performance enhancements in the age of paranoid transport protocols. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 221–227, 2022.

[38] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. Achieving consistent low latency for wireless real-time communications with the shortest control loop. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 193–206, 2022.

[39] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 89–102, 2002.

[40] C-H Tai, Jiang Zhu, and Nandita Dukkipati. Making large scale deployment of rcp practical for real networks. In *IEEE INFOCOM 2008-The 27th Conference on Computer Communications*, pages 2180–2188. IEEE, 2008.

iStack: A General and Stateful Name-based Protocol Stack for Named Data Networking

Tianlong Li[†] Tian Song[‡,*] Yating Yang[‡]
[†]*School of Computer Science and Technology*
[‡]*School of Cyberspace Science and Technology*
Beijing Institute of Technology

Abstract

Named Data Networking (NDN) shifts the network from host-centric to data-centric with a clean-slate design, in which packet forwarding is based on names, and the data plane maintains per-packet state. Different forwarders have been implemented to provide NDN capabilities for various scenarios, however, there is a lack of a network stack that is integrated with operating systems (OS) for general purpose. Designing a stateful and entirely name-based protocol stack in OS kernel remains a challenge due to three factors: (i) an in-kernel name resolution architecture for packet demultiplexing is necessary, (ii) an entirely name-based stack requires to be compatible with the current address (MAC/IP/port)-based architecture in OS kernel, and (iii) maintaining per-packet state introduces a trade-off between performance and resource consumption.

In this paper, for the first time, we take NDN into OS kernel by proposing iStack, an Information-Centric Networking (ICN) protocol stack. The main innovations of iStack are threefold. First, we propose a name resolution architecture to support both network-layer forwarding and local packet demultiplexing. Second, a two-layer face system is proposed to provide abstraction of address-based network interfaces. Third, we design socket-compatible interfaces to keep the uniformity of current network stack in OS. Besides, we design compact forwarding data structures for fast packet processing with low memory footprint. We have implemented prototypes on multiple platforms. The evaluation results show that iStack achieves 6.50 Gbps throughput, outperforming the NDN-testbed forwarder by a factor of 16.25x, and reduces 46.08% forwarding latency for cached packets with its in-kernel packet caching. iStack is not just another forwarder for NDN, but a step forward for practical development of ICN.

1 Introduction

Named Data Networking (NDN) [1] has been regarded as a leading Information-Centric Networking (ICN) design, which addresses data directly with hierarchical names rather than hosts. Its novel architecture enables a number of valuable features, including in-network caching [2, 3], securing data directly [4, 5], mobility support [6, 7], and native multicast support [8, 9]. Furthermore, a gateway for integrating NDN into the Internet has been designed [10]. Many studies have explored NDN's advantages in diverse scenarios [11–14].

NDN Forwarding Daemon (NFD) has been implemented and intended to be used as a general-purpose forwarder since 2014 [15, 16]. It is effective in supporting architectural research and being used on the public NDN project testbed [17]. However, it is a user-space forwarder rather than an operating-system resident. In comparison, in the following three ways, an in-kernel network stack like TCP/IP gains additional benefits for practical deployment with operating systems support.

First, a stack integrated into the operating system is designed to work with the socket mechanism. It can provide system-level network functionality and be shared by multiple applications with varying purposes and requirements. Second, an in-kernel stack runs as kernel threads, which gain system-level security from the OS kernel protection. Typically, the x86 architecture has 4 rings of protection, with the kernel running in ring 0 with the greatest privileges and user programs running in ring 3. Because user-space programs cannot access kernel-space memory, the in-kernel networking architecture is naturally isolated and protected from applications. Third, an integrated stack can evolve with operating systems for the long term, providing a stable foundation for network communication. As operating systems are updated or new hardware is introduced, an in-kernel network stack can adapt to these changes seamlessly. Hence, it is worthwhile to take NDN, a clean-slate networking architecture, into operating systems.

However, there are three challenges for designing an in-kernel stack for NDN. First, name-based packet demultiplexing requires a well-thought-out design. A general stack needs to distinguish and deliver packets among local applications, which is done by transport-layer ports in TCP/IP stack. Specifically, an application creates a socket and binds it to a port. The port number is carried in packets and identifies the appli-

*Corresponding author: Tian Song (songtian@bit.edu.cn)

cation to the receiving end, whereas NDN is entirely based on names. An in-kernel stack for NDN requires a name resolution architecture for packet demultiplexing.

Second, an in-kernel NDN stack should be compatible with the existing TCP/IP stack in OS for dual-stack usage. The traditional network stack relies on address (MAC/IP/port) based communication. In contrast, NDN has no endpoints or host addresses. Therefore, an abstraction of network interfaces to support network processing is necessary. NFD incorporates a face mechanism intending to achieve this [16]. However, an in-kernel stack needs to interact with both application sockets and link-layer physical/virtual devices, presenting significant differences from a user-level forwarder.

Third, a general-purpose network stack for NDN should balance its performance and resource consumption. Unlike IP's stateless data plane, NDN adopts a stateful forwarding model. For the first time, a network stack requires large memory for in-network caching and involves multiple table lookups, which remains a challenge for an efficient stack design.

In order to address these challenges, we propose iStack, a general and stateful name-based protocol stack in OS kernel for NDN. The stack itself is native to OS kernel and compatible with other in-kernel network protocol suites. Meanwhile, iStack is an information-centric stack that is entirely based on names and enables kernel-level NDN functionality. Our major contributions are listed as follows:

First, we propose iStack, which for the first time, takes NDN into OS kernel to support application hosts. iStack has a fully name-based architecture. It adopts NDN forwarding model and introduces a name resolution architecture for demultiplexing. Its architecture can support applications with the native socket mechanism in OS.

Second, we design a two-layer face system as the network interface abstraction, which maps both of high-layer socket interfaces and low-layer network devices into logical *FaceIDs*. Hence, iStack uses logical *FaceIDs* for network processing.

Third, we design high-speed forwarding data structures, which efficiently maintain per-packet state and cache packets directly in the kernel. Furthermore, we also implement a fast two-level lock mechanism to ensure multi-threading safety while leveraging parallel processing for higher performance.

Besides, we implemented iStack prototypes in two different Linux kernel versions, four different Linux distributions (Ubuntu, CentOS, Raspberry Pi OS Lite, and OpenWrt), and four different platforms (x86 server, personal computer, Raspberry Pi and edge router (MT7621AT)). Our evaluation results show that iStack achieves 6.50 Gbps throughput, measured with the standard size (1500 bytes) of Ethernet frames. Its throughput outperforms the general NDN forwarder, NFD, by a factor of 16.25x. Moreover, iStack takes in-network caching into kernel space, reducing 46.08% forwarding latency for cached packets. Unlike the previous forwarders, iStack focuses on providing NDN functionality on end hosts for applications rather than mainly supporting high-speed routing/forwarding. Our prototypes demonstrate that iStack is promising to advance ICN deployment.

The rest of this paper is organized as follows. Sec. 2 introduces the background as well as the related work. Sec. 3 details the design and architecture of iStack. Sec. 4 describes a suite of efficient extensions and miscellaneous considerations for iStack. Sec. 5 presents the evaluation results of our implemented prototypes and includes lessons learned. Sec. 6 finally concludes this paper and discusses our future work. This work does not raise any ethical issues.

2 Background

2.1 Named Data Networking

NDN is one of the most important instances of ICN [1]. It uses hierarchically structured names to address the content. NDN uses '/' to indicate the boundaries of name components. For example, segment 2 of version 1 of a video provided by Alice may have the name: */Alice/videos/DemoA.flv/1/2*, which contains three parts: where to forward it (*/Alice*), which application to deliver it (*/videos*), and application-specific information (*/DemoA.flv/1/2*). As NDN matters the content rather than where it is from and caches *Data* in the network, security is built into content itself [4]. It adopts a stateful forwarding model that primarily consists of three structures: Pending Interest Table (PIT), Forwarding Information Base (FIB) and Content Store (CS).

In NDN, a consumer (client) requests content by sending a packet with the content name, namely *Interest*, to the network. Any node receiving it tries to find the corresponding *Data* identified by that name in its CS. If it is found, the *Data* will be returned. Otherwise, the node inserts this *Interest* into PIT and then forwards it based on both the matching result from FIB and the forwarding strategy. When the *Interest* finally meets the corresponding *Data* at the producer (server) or a CS on the *Interest* forwarding path(s), the *Data* is sent back in the reversed path of the *Interest* and meanwhile removes those records in PITs. In particular, different *Interests* with the same name can be merged into a single PIT entry without redundant forwarding. In this case, the merged PIT entry records all *Interests* incoming interfaces. When the *Data* is back, it is forwarded via multicast based on that PIT record. With this design, NDN gains a number of benefits such as network-layer request aggregating and native multicast support.

2.2 Related Work

In the last decade, NFD, the most important NDN forwarder, has been designed and implemented for architectural research [15]. Besides, several dedicated forwarders are proposed to meet the requirements of different scenarios.

NFD is designed modularly and extensively to support diverse NDN technology experimentation [15]. It is written

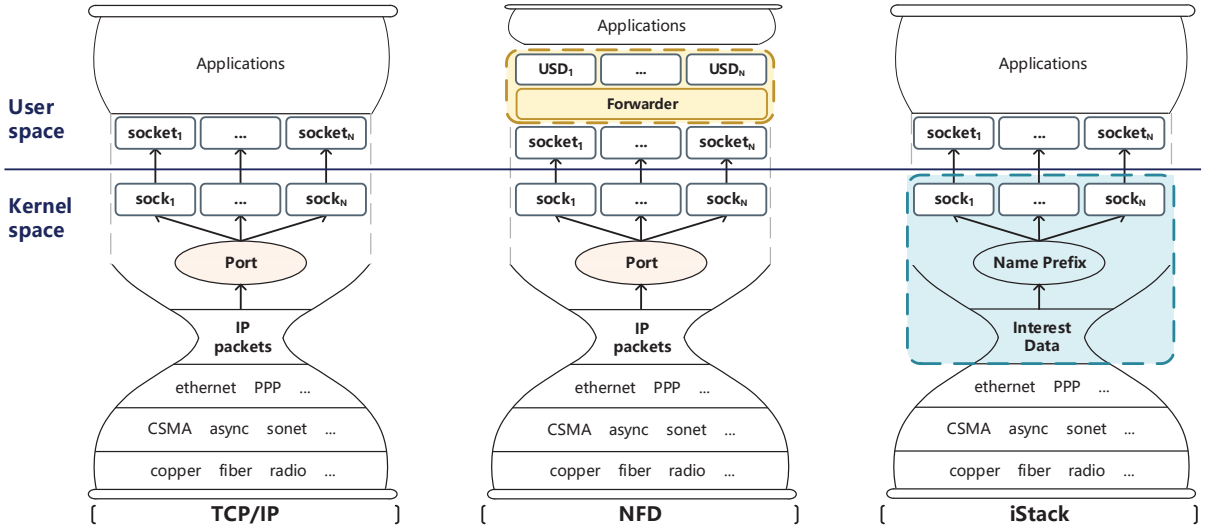


Figure 1: Comparison among TCP/IP stack, NDN user-space forwarders, and iStack from a high-level perspective

in C++ language and has a well-modular design. Although it intends to be used for general purposes, NFD is a user-space forwarder rather than an in-kernel stack. A user-space forwarder must use an inter-process communication (IPC) mechanism (specifically in NFD, Unix domain socket [16]) to communicate with local applications. It does not provide system-level support for applications.

NDN-RIOT is proposed as a lightweight implementation for RIOT-OS [18]. RIOT is a micro-kernel operating system for resource-limited IoT devices. It does not support virtual memory and separate of user and kernel space, which is common in modern general OS. Hence, although NDN-RIOT integrates the core NDN forwarding logic into the RIOT-OS kernel and demonstrates the feasibility of bringing NDN's data-centric communication and security model to constrained IoT platforms, it is a dedicated solution for particular scenarios rather than a general forwarder.

For further supporting embedded/constrained environments, a low-overhead forwarder, NDN-Lite is carried out [19]. Unlike NFD, NDN-Lite simplifies the forwarder design and is written in C language. However, NDN-Lite has to integrate with the thread of a NDN application and can only support a single application simultaneously. This is suitable for low-end platforms but meanwhile limits its generality.

In order to achieve high-speed NDN forwarding, NDN-DPDK is proposed [20] and reaches a peak forwarding rate of 1.84 MPPS[1] by leveraging the fast user-space packet processing framework Data Plane Development Kit (DPDK) [21]. For paralleling, NDN-DPDK dispatches the incoming *Interests* to the private sharded data structures of threads by hashing their name prefixes. For dispatching retrieved *Data* to the correct thread, this work proposes PIT token, a small hop-by-hop header field added to each packet. Due to its high performance, NDN-DPDK may be suitable for throughput-sensitive cases such as backbone routers and data centers. However, the DPDK-based solution limits the generality of it.

MW-NFD is proposed to provide a high-performance forwarder without DPDK support [22]. It splits the forwarding pipeline into multiple threads and follows the packet dispatching method of NDN-DPDK. Its polling-based design has a 100% CPU use rate, which is not suitable for general purposes.

The forwarders discussed above either achieve high performance but are unsuitable for general purpose or vice versa. Given the opportunity that it lacks a more efficient NDN forwarder for edge, YaNFD is proposed [23]. YaNFD is a multi-threaded alternative to existing software packet forwarders for the NDN architecture. The performance of it is higher than that of NFD and lower than that of MW-NFD, while it keeps a reasonable resource usage for generality. Nevertheless, it is written in Go language as a user-space forwarder, which suffers from the garbage collection overhead.

From the practical usage perspective, the genuine requests and content are finally from and to applications running in OS. Hence, a forwarder enabling NDN packet forwarding may be suitable for supporting architectural research purposes, whereas a network stack integrated with OS kernel is still required to provide system-level supporting for real applications. This is also our motivation for proposing iStack. Fig. 1 illustrates the comparison among TCP/IP stack, NDN user-space forwarders, and iStack from a high-level perspective.

2.3 Network Stack in OS Kernel

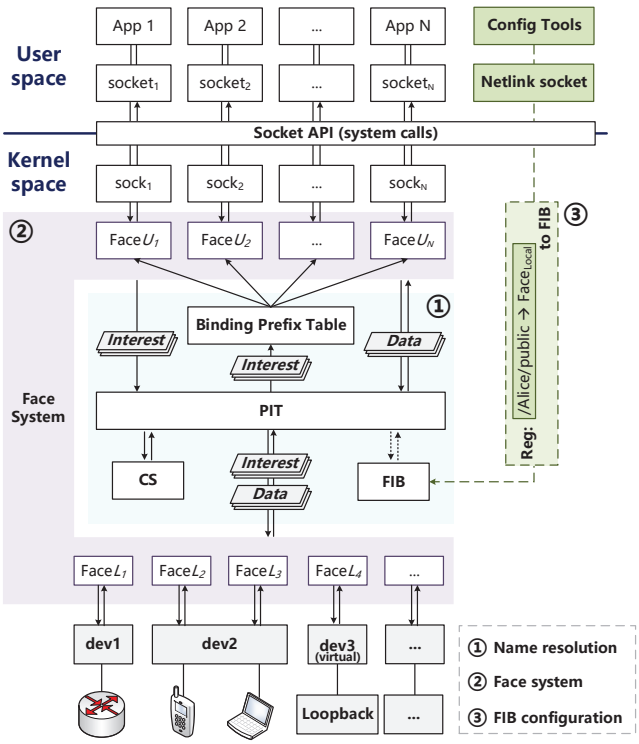
Modern operating systems implement network protocols from the link layer to the transport layer. They also implement

[1] In form of throughput, it achieves 22 Gbps with 1500 Bytes standard Ethernet frame and 108 Gbps with 8000 Bytes jumbo frames.

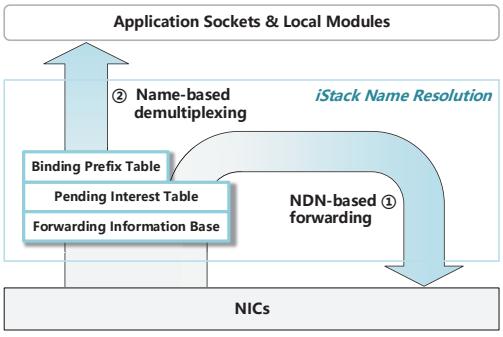
a version of Berkeley socket as application programming interfaces (API). Different layers in a network stack have their own abstractions. Taking the Linux kernel network stack for example, in its link layer, NICs and their drivers are abstracted as *net_device*(s), bound with their hardware MAC addresses.

In the network layer, *address_family* is used to specify the network stack suites, such as AF_INET and AF_INET6 for IPv4 and IPv6, respectively. As the transport layer provides end-to-end services for endpoints of network communication, it is integrated with the network socket. A socket created by an application has two parts: *socket* as an abstraction in user space for the application and *sock* as the in-kernel network representation. The transport-layer functionalities are integrated with the *sock* and specified by *sock_type*. For example in IP stack, SOCK_STREAM and SOCK_DGRAM represent for TCP and UDP, respectively. Hence, the network layer is shared in a host, whereas the transport layer is isolated by different applications with their created sockets separately. Notably, a non-port-based raw type socket allows applications to directly access network/link-layer packets. It catches all packets and shifts the responsibility of (de)multiplexing from the kernel-space network stack to the user-space forwarders/applications. Yet a general-purpose in-kernel network stack requires suitable (de)multiplexing models.

As presented in Fig. 1, the current network stack is address-based: packets go through a process of filtering by network-layer addresses, which are then distinguished by transport-layer ports before finally being delivered to their intended sockets of applications. In contrast, NDN takes all functionalities into name prefixes and there is no port. Therefore, an in-kernel network stack for NDN requires a name resolution architecture for network-layer forwarding and local packet demultiplexing. The current NDN forwarder-based solutions use IPC mechanisms for applications, leaving system-level support for applications as an open issue. iStack is designed to address this and takes NDN into OS kernel.



(a) iStack architecture



(b) Two paths of name resolution

Figure 2: iStack architecture and name resolution

3 iStack Design and Architecture

3.1 Design Overview

As an in-kernel network stack, iStack operates between the applications sockets and the link-layer devices. As illustrated in Fig. 2(a), iStack primarily consists of three parts: a name resolution architecture, a two-layer face system, and socket-compatible interfaces for applications and configuration.

The name resolution architecture enables packet demultiplexing for local applications with two packet paths as shown in Fig 2(b). For supporting name-based *Interests* demultiplexing, it introduces a Binding Prefix Table (BPT) which records the relationship between application sockets and their prefixes. iStack separates application binding prefixes from registering routable prefixes to the network.

In order to integrate the name resolution architecture with OS kernel compatibly, iStack involves a two-layer face system, which abstracts upper layer handles and lower layer communication channels as general interfaces. It separates the upper and lower layer due to the fact that different layers treat semantics of names differently.

iStack provides interfaces based on socket mechanism. Applications can use network sockets to communicate with iStack, following a practice similar to TCP/IP. The only difference is that in TCP/IP, a socket needs to bind an *<address, port>* pair, whereas in entirely name-based iStack, there are no addresses or ports. Instead, an iStack socket binds a local *faceID* to represent where packets are from. A provider-side application socket also needs to bind a prefix to notify iStack

where *Interests* should be delivered to. iStack configuration is isolated from the common network socket communication for system security consideration. Specifically in the Linux kernel, it is through Netlink socket with superuser privilege.

3.2 Name Resolution Architecture

In iStack, name resolution utilizes NDN-semantic names, which carry the information of forwarding and demultiplexing. In this paper, we use the term *routable prefix* to represent a prefix that can be forwarded to, and the term *application-specific prefix* to represent an extension of the routable prefix used for distinguishing applications. For example with the name: */Alice/videos/DemoA.flv/1/2*, its routable prefix and application-specific prefix could be */Alice* and */Alice/videos*, respectively. According to the two types of prefixes, the packet processing path is divided into *network-layer forwarding path* and *local delivering path* as shown in Fig. 2(b).

For the network-layer packet forwarding, iStack follows the basic design of NDN. *Interests* are pending in PIT and forwarded based on FIB and forwarding strategies. *Data* packets are forwarded along the reverse path of the pending *Interests*. The main difference is the serving scope of FIB. In previous design of NDN forwarders like NFD, a local application registers the content prefixes with its identification into FIB, hence the forwarder can deliver the corresponding *Interests* to it. In contrast, FIB in iStack serves for the network layer only, which means it only records routable prefixes rather than a vast number of application-specific prefixes.

FIB in iStack can only be updated by (i) routing protocols running in kernel threads and (ii) user-space configuration tools. The former one updates FIB with routing information learned from the network and the later one is used by superusers to register routable prefixes of providers on this host. The network layer only determines whether *Interests* should be delivered to this host based on routable prefixes in FIB and hands over these *Interests* to BPT. FIB is agnostic to application-specific prefixes and local packet distinguishing.

The local part of name resolution is situated above the network layer and provides local packet distinguishing functionality. It maintains BPT, which records the mapping between application sockets and application-specific prefixes. Indicated by its name, an application registers its application-specific prefix through socket API *bind*.

There are two rules for applications to bind their prefixes. First, in BPT, any prefix *must not* be identical to or be a proper prefix[2] of another prefix. Otherwise one application may hijack the *Interests* intended for other applications. Second, an application-specific prefix *should* be an extending string of a routable prefix that has been pre-registered in FIB. Otherwise, the application may never receive any *Interests* as the network layer does not pass *Interests* with that prefix to local host. The local name resolution provides flexibility for applications by using finer-grained and more complex prefixes without messing up FIB. As needed, an application can create multiple sockets and bind them to different application-specific prefixes which share a common routable prefix.

[2]In this paper, we use *proper prefix* to indicate a component-level prefix of a name, such as */A/B* or */A* for */A/B/file*

3.3 Two-Layer Face System

The face system is designed to (i) hide details of various interfaces and (ii) ensure thread safety against the system events such as interface adding/removing. It provides *Faces*, the generalization of different network interfaces.

From the layered networking perspective, different layers have different targets and communicating scopes. In iStack, for packets travelling down to the lower layer, i.e. the link layer, the semantics of name/prefix is not concerned. Whereas for packets travelling up to the upper layer like applications, the name/prefix semantics may still be concerned. Due to this difference, iStack contains a two-layer face system to provide the generalization of interfaces from different layers.

3.3.1 Face Upper Layer

The face upper layer targets to abstract the local handles upon the network layer, which commonly are application sockets. An upper face consists of a unique *FaceID*, a handler, and a prefix. The handler is composed of an entity pointer, which points to the upper entity represented by the face, and a function entry pointer which points to the handler function of that upper layer entity. If an entity wants to receive *Interests*, then the prefix field of its face has to be specified. Otherwise this field is null. In the network layer, PIT and FIB only use *FaceIDs* to represent network interfaces. Face upper layer maintains a face upper table to map local handles to faces.

In the network part of iStack name resolution, incoming *Interests* are filtered by FIB, and those intended for local applications are passed to BPT. In iStack network layer, there are only *FaceIDs* for interfaces and entrances of BPT is also encapsulated as faces. Face upper layer provides a permanent face, $Face_{local}$, as the local entrance. Routable prefixes that can be served by the host are registered to FIB with $FaceID_{local}$. When provider-side applications bind their sockets with application-specific prefixes, the prefixes are mapped to their faces and then the faces are put into BPT. When an *Interest* comes, it is firstly passed to $Face_{local}$ based on FIB. Then the handler of $Face_{local}$ queries BPT and returns the corresponding face. Finally the handler of the returned face delivers the *Interest* to the application socket.

3.3.2 Face Lower Layer

The face lower layer targets to abstract the link layer communication channels. It uses a face lower table to map link-layer interfaces to faces. A lower face consists of a unique *FaceID*,

a device, and a field for transmitting information. The device can indicate either physical hardware devices like ethernet adapters or virtual devices such as loopback device. Specifically in implementation, all link-layer devices for network accessing in the Linux kernel are abstracted as *net_device* and hence, the device of a lower face is a pointer of that.

A single device is not enough to transmit link-layer frames. A link-layer device may be connected with different remote devices at different time or even with multiple remote devices at the same time, like the wireless NIC of a WLAN access point. Besides, the link-layer transmission includes unicast, multicast and broadcast. Only broadcast needs no remote information for transmitting while the other two needs the destination address(es) of the remote host(s) or group(s). Thus, the field for transmitting information is used for recoding remote host information which typically is a destination MAC address. Obviously, a lower face represents for the channel between a local device and a remote host/group.

3.3.3 Thread-Safe Face Operations

The principle of face system is that the control path should not expose details to the data path and the data path processing should not block the control path operations. To achieve these, the face structure has a field of user count and thread-safe function pairs are provided: *face_register* and *face_release* for the control path; *face_hold* and *face_put* for the data path.

In the control path, *face_register* is used to create a new face and bind a network interface to it. Meanwhile, the user count of it is set to 1. For interface removing operations, such as closing a socket or taking a network adapter offline, the control path uses non-blocking function *face_release*. The function decreases the user count value of the face and its further behaviour depends on the result. If the decreased user count is 0, which means the face is not in use, then *face_free* is called to free the face and recycle its *FaceID*. In the other case, a positive user count means that the face is still referenced by at least one thread, then *face_orphan* is called. The function *face_orphan* releases everything in the face except the shell of it and meanwhile, it set the *FaceID* to 0 and the handler to *face_output_blackhole*, which drops any packets passed to it. The release of the face shell is delayed to the last data path calling *face_put*. Hence, the data path threads do not block the control path operations such as closing a socket.

In the data path, when a *FaceID* is transformed to the face, *face_hold* is called to increase the user count of it. When the packet processing is finished, *face_put* is called to decrease its user count. Specially, if the decreased user count is 0, which means the real interface of the face has been dead, then *face_free* is called to free the shell of this face. The function pair *face_hold* and *face_put* only involves an integer increasing/decreasing operation in common cases and a memory free operation in the case of interfaces removing. Thus the performance influence from control path events is minimized.

3.4 Socket-Compatible Interfaces

3.4.1 Forwarding Configuration

As shown in Fig. 2(a), configuration of iStack is isolated from common network socket communication. iStack utilizes a specific communication mechanism between kernel and user space supported by operating systems to provide control functionalities such as configuring FIB and exporting information.

Specifically in the Linux kernel, iStack exploits two mechanisms: the *procfs* and the Netlink socket. *Procfs* is a virtual filesystem which allows the Linux kernel to export internal information to user space. Netlink socket is more powerful than *procfs*. *Procfs* is able to read/write kernel exported data as operating files, while the Netlink socket communicates with kernel using socket programming model and moreover, the later one can initiate a transmission rather than only waiting for responses to user-space requests. Hence, iStack uses *procfs* to export its running status and statistical information in form of files as convenient interfaces. Meanwhile, we develop some utility tools with the Netlink socket for configuration requirements such as FIB configuration and CS management.

3.4.2 Application View

The common communication between applications and iStack is entirely based on the network socket mechanism. An application uses iStack via creating a socket with the address family AF_NNET. Then, the application can send/receive NDN packets with the socket. We provide three socket types to support different application requirements (detailed in Sec. 4.3.3). Note that a socket of iStack can support both consumer-side and provider-side applications at the same time. Just for ease of understanding, we introduce them separately.

For a consumer-side application, it can send out *Interest* and receive the corresponding *Data* once the socket is created. When *send* is called for the first time, the face system allocates a *FaceID* and automatically *bind* the socket to it. Then the *Interest* goes to the network layer through that face and is inserted into PIT and then forwarded based on FIB and forwarding strategies. When the corresponding *Data* is retrieved, it matches the record in PIT and then is passed to the socket via the *FaceID*. The *Data* is put in the socket receiving queue and waits until the application fetches it with *recv*, or in other cases, the application is blocked with *recv* until the *Data* queues in the socket and triggers a software interrupt request to retrieve it. Note that an application can send out a batch of *Interests* and receive the *Data* packets one by one. The order can be implied by the segment numbers contained in names. The socket owns the face until it is closed.

For a provider-side application, it needs one more step to receive *Interests*. That is, using *bind* to bind a prefix. The face system allocates a *FaceID* and maps the binding prefix to it in BPT. When an intended *Interest* arrives, it is inserted into PIT and then finds out that the outgoing interface recorded in FIB

is $FaceID_{local}$. After matching the application-specific prefix in BPT, the *FaceID* of that application's socket is returned and the *Interest* is delivered to that socket. Finally the application receives the *Interest* and sends out the corresponding *Data* which will go along the reverse path of that *Interest*.

iStack adopts the sockets API rather than the established NDN API. Consequently, it is presently incompatible with existing NDN applications. However, rectifying this disparity through API code modification is a straightforward process. Our forthcoming endeavor involves streamlining the transition of existing NDN applications to iStack. To achieve this, we will provide a user-space library that seamlessly translates the current NDN API into iStack socket API. Furthermore, we are proactively engaged in the development of an information-centric programming model, complemented by a dedicated set of primitives and a socket-based API tailored for the advancement of future NDN applications.

4 Efficient Design and Extension for iStack

4.1 Compact Forwarding Data Structures

4.1.1 Design Rationale

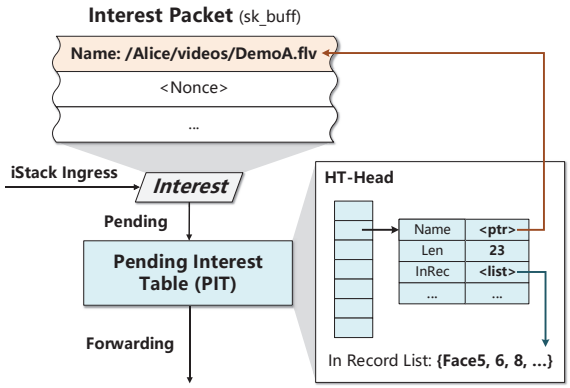
In NDN design, each *Interest/Data* carries a variable-length hierarchical name. As PIT maintains per-packet state and CS caches content in the network, it takes much more memory than traditional stateless network stacks. An in-kernel general-purpose network stack should take the trade-off between performance and resource consumption. In order to reduce the kernel-space memory consumption while keeping fast packet processing, we design compact forwarding data structures for PIT and CS in iStack.

The compact design ingeniously utilizes the fact that the life time of a PIT/CS entry is the same as that of an *Interest/Data* packet. In iStack, each PIT entry keeps at least one *Interest* packet pending for the potential retransmission in its lifetime and obviously, the entry name is carried by that *Interest*. For CS, it caches *Data* packets in kernel directly and each of its entry name is carried by the corresponding *Data*.

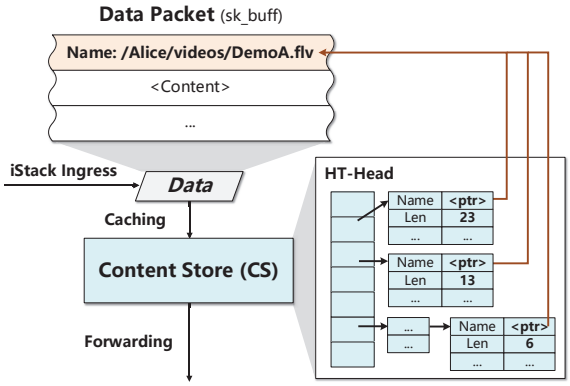
4.1.2 Compact Hash-Based Data Structures

PIT needs to support both exact name matching (ENM) and longest prefix matching (LPM). Hash table is naturally fit for ENM. LPM in PIT is for the situation where an *Interest* is marked with *CanBePrefix* flag. Such an *Interest* can be satisfied by the *Data* with the same name prefix. As PIT only records *Interest*, LPM needs no modification of the hash table, but requires a particular lookup procedure for incoming *Data*.

The compact PIT structure is shown in Fig. 3(a), which is a separated chaining hash table and takes *Interest* names as keys. For an incoming *Interest* whose name has not been recorded before, PIT creates an entry and pends the packet directly. The entry uses a pointer to reference the name string



(a) Compact hash-based data structure for PIT



(b) Compact hash-based data structure for CS

Figure 3: Compact forwarding data structures

in the payload of that *Interest* and an unsigned integer to record the name length. Note that an iStack PIT entry keeps one and only one *Interest*. In case that more *Interests* with the same name arrive before the corresponding *Data* is retrieved, PIT records the incoming *FaceIDs* of them in the existing PIT entry and updates the entry life time. Then the duplicate *Interests* are dropped.

When a *Data* packet arrives, PIT performs LPM in two steps. First, it queries the full *Data* name to find the matched entry and forwards it to the incoming *faces*. Then, each proper prefix of the *Data* name are queried for those*CanBePrefix*-flag marked entries. Note that forwarding *Data* to multiple *faces* is done by the kernel network function *skb_clone* which increases the packet reference count and only copies the *sk_buff* structure for management. Hence, iStack multicast has no additional overhead for duplicate *Data* packets.

CS needs to support both of ENM and all sub-name matching (ASNM) [24], which means all the proper prefixes of a *Data* name need to be indexable. CS caches *Data* packets in kernel directly and the compact structure of CS is shown in Fig. 3(b). In order to extend supporting from ENM to ASNM, CS creates a series of entries for a cached *Data* packet. Each of the entries uses a pointer to reference the name string in

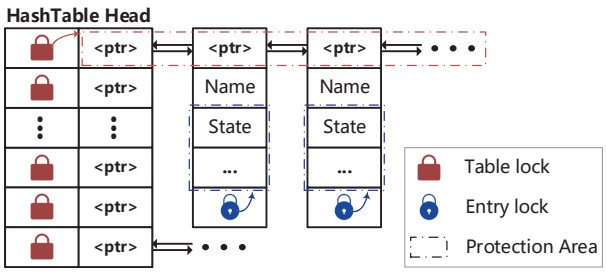


Figure 4: Fast two-level locks structure

the payload of that *Data* and an unsigned integer to record the length of the whole name or one of its proper prefixes.

With this design, CS in iStack supports NDN name discovery: an *Interest* with *CanBePrefix* flag can be satisfied by cached *Data* with an extended name. Moreover, entries in CS may be evicted based on the replacement policies. In order to evict the series of entries of a cached *Data* packet efficiently, the entry of CS is designed with another dual pointer field which forms a doubly linked list associating the series of entries for a *Data* packet. With this, evicting a cached *Data* packet from CS is simple: find the entries with the specific name and then remove all the entries in its doubly linked list and finally release the referenced packet.

The compact hash-based data structures in iStack take zero memory copy for names and prefixes and have no packet duplication, which reduce the memory consumption. Moreover, as both of the data structures and packets are in kernel space, there is also no memory copy and context switch overhead between kernel and user space.

4.2 Fast Locks for Multi-Threading Safety

There are forwarders which choose sharded data structures to improve multi-thread performance, whereas keeping private instance of PIT/CS per thread may break down NDN name discovery. The proposed solution is introducing *PIT token* which is the hash value of the entire name or a fixed-length prefix of *Interest* name and is carried hop-by-hop (NDN-DPDK, YaNFD [20, 23]). As a general-purpose network stack, iStack avoids involving dependency on under-layer protocols. Hence, the three key NDN data structures in iStack are shared by parallel kernel threads. In order to ensure multi-threading safety while achieving high performance, we carry out fast two-level locks structure which reduces locking overhead.

As illustrated in Fig. 4, there are two-level locks for iStack compact hash-based data structures, the table locks and the entry locks. Instead of competing for a single lock, each bucket of the separate chaining hash table is equipped with an individual table lock. Since parallel operations are naturally distributed to different chains by hashing, the blocking overhead is limited to the concurrent operations on entries with a same name and with hash collisions.

As name lookup without table modification takes a significant proportion of name-based forwarding processing, iStack adopts readers-writers locks as the table locks [25]. The critical area of reader locks is a traversal of a hash table chain and writer locks protects for entry insertion and removal. To minimize the impact on performance, the critical area only covers direct operations on the corresponding hash table chain while other processing including entry initialization and memory free are done before and after the critical area, respectively.

The entry lock is used for safely updating the state of an entry such as adding incoming faces to an existing PIT entry. Due to most of the entry related operations are mutually exclusive and likely to be finished in very short period, iStack takes spinlocks as the entry locks.

The fast two-level locks structure reduces performance degradation of parallel processing while ensures multi-threading safety with little cost of memory consumption. iStack only equips per-bucket table locks for three tables: PIT, CS, and FIB. Given the assumption that setting those table heads length to 65536, the extra memory for table locks is 1.5 MB in total. The entry locks are equipped on all table entries. Nevertheless, PIT entries are temporarily pending in the table. For CS and FIB, the size of entry lock is negligible compared with that of cached *Data* and stored routable prefix strings.

4.3 Miscellaneous Design

4.3.1 In-Network Caching

As iStack is an in-kernel stack, it naturally takes CS into the kernel. For generality, two issues should be considered, namely the kernel-space memory consumption for storage and the capacity of CS. For the former one, iStack provides two parameters for management. They are maximum number of entries and maximum size. The two limitations take effect simultaneously. The maximum size is to limit the kernel-space memory consumption. As CS in iStack creates entries for each proper prefix of a *Data* packet, the maximum number of entries is used to prevent from a large amount of entries for *Data* with an extremely long name overwhelming CS and decreasing its performance. For the latter, we design hierarchical structure for CS. The in-kernel part is used for *Data* with the most popular prefixes. The out-kernel part utilizes hard disk space to provide vast size of storage. We implement in-kernel CS with configurable parameters in our prototypes and leave the hierarchical CS implementation as future work.

4.3.2 Loadable Kernel Module

There are two ways to implement iStack in the Linux kernel: i) add iStack codes to the kernel source tree and recompile the kernel or ii) add iStack as kernel loadable module to the running kernel. As an in-developing novel network stack, the main part of iStack is implemented as a loadable kernel module (LKM). But some socket-like APIs remain to be

completed. With the current socket APIs, the parameter of *bind* is limited to 128 bytes long, which limits the maximum binding prefix length. Besides, with Berkeley socket API, the programming model of iStack includes merely *send* and *recv*. We also design information-centric programming model and implement new socket-like APIs for iStack to be published. Practically, there is a static size array for the system call table and adding new system calls needs to modify the architecture-specific table and recompile the kernel.

In principle, NICs provide *receive side scaling* (RSS) on hardware for dispatching incoming packets to different CPU cores. Nevertheless, the typical filter used in RSS is a 4-tuple hash over IP addresses and TCP ports of a packet. Based on our observations from implementing, most of the commercial off-the-shelf NICs do not perform RSS for packets with unknown ether types and they just send all packets with unknown type to a single RX queue. Hence, we implement a software-based dispatcher to enable multi-threading packet processing in iStack with two dispatching strategies: dispatching based on hashing entire names and dispatching randomly.

4.3.3 Reliability and Security

Although the raw type of socket provides maximum flexibility for applications, a general-purpose network stack should provide interfaces with different levels of granularity in terms of reliability. Hence, besides the basic socket type SOCK_RAW and SOCK_DGRAM, we also implement the type SOCK_SEQPACKET in iStack. SOCK_RAW only provides a basic interface for applications to send and receive packets. SOCK_DGRAM provides packet encapsulating and decapsulating functions with which applications can focus on names and content rather than manually deal with packet format. SOCK_SEQPACKET provides more reliability. In the *sock* part of this type of socket, we implement slice-window flow control, retransmission, and name-based reordering. To save space, more details are not elaborated here.

It is worth noting that iStack represents an in-kernel design and prototype for NDN. Its primary value lies in its robust framework. It is designed to offer fundamental NDN functionalities and facilitate integration of additional features and strategies, rather than presenting a finalized production-ready solution. Given that ICN/NDN fundamentally transforms the network communication paradigm from host-dependent to host-independent, and introduces state into the network, the required transport abstractions may differ significantly from the established end-to-end protocols in IP realm. Ongoing research has been dedicated to transport-layer services [26], congestion control [27–29], and etc. Furthermore, in NDN, each *Data* carries a unique name, resulting in distinct transport service requirements compared to traditional models. Consequently, iStack currently incorporates essential transport functionalities for application development, rather than fully piggy-back established end-to-end transport abstractions like TCP/QUIC. We will keep the evolution of iStack.

NDN secures data directly by requiring data producers to cryptographically sign every *Data* packet [1]. There are also literatures investigating security mechanisms for different scenarios [30–32]. Based on the basic *Interest/Data* communication, users can retrieve both content and the corresponding keys and perform verifications. We leave the security operations to user space and plan to carry out function libraries for the convenience of applications in the future.

5 Evaluation and Discussion

5.1 Implementation and Experimentation

We have implemented iStack in the Linux kernel 4.14 and 4.19 and tested on PCs, servers, Raspberry Pies (4B+), and edge routers. We also implemented several applications over iStack, including file transfer, streaming video player, and etc. Our prior test shows that iStack can achieve 1 Gbps line speed on laptops and low-end devices like Raspberry Pies. We built a pure NDN local area network (LAN) with iStack and applications including file transfer and video player ran in the LAN successfully. In this paper, our evaluation focus on two aspects: (i) network performance benchmark and (ii) application-level performance for real edge.

For the network performance measurement, we ran a provider-side file transfer application on a server equipped with dual 6 cores CPUs (hyper to 24 threads in total) and connected it to a packet (*Interest*) generator. As the hyper thread of a core is abstracted as a processor in the Linux kernel, we interchangeably use the terms *core* and *thread* in the following paper. We evaluated the maximum network throughput with different working threads and compared the delay of cached *Data* response between in-kernel and out-kernel CS.

To evaluate application-level performance for real edge, we deployed multiple consumer-side file transfer applications to request different files from the server. All applications use the same format of content names and an example is */Net-A/prdc-01/0/PubFiles/ABCD/1* which consists of three distinct parts:

1) A prefix of the provider (*/Net-A/prdc-01*).
2) A provider application defined prefix part (*/0/PubFiles*).
3) A part of application-specific information including a segment number (*/ABCD/1*, */ABCD/2*).

As the consumer applications run with socket APIs in user space, this scenario can obtain "goodput", application-layer throughput, and monitor CPU usage and memory consumption as the applications are running. For fair comparison with existing forwarders, the throughput and the forwarding rate in the evaluation only accounts for *Data* packets. Hence the total packet forwarding rate (including both *Interest* and *Data*) is at least twice[3] of the results in this paper.

[3]There is potential of *Interest* packet loss and retransmission which makes the total packet forwarding rate may higher than twice of the reported results.

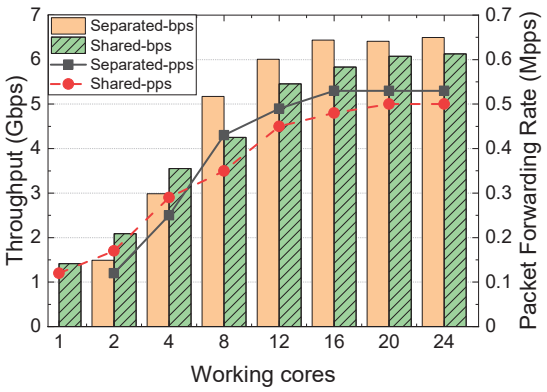


Figure 5: Throughput with different processor affinity settings

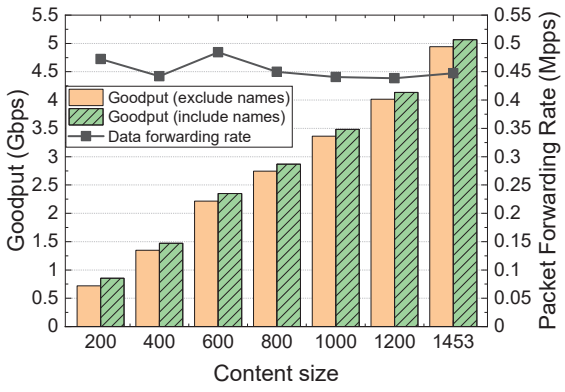


Figure 6: Goodput with different content-payload sizes

Table 1: Delay comparison between in/out-kernel CS

CS size	Avg Delay for Cache Hit (μs)		Reduction Ratio
	In-Kernel	Out-Kernel	
10 MB	3.85	7.14	46.08%
100 MB	4.17	7.69	45.77%
1 GB	**4.55**	8.33	45.38%

5.2 Throughput and Delay

As described in Sec. 4.3, the commercial off-the-shelf NICs do not perform RSS for the packets with unknown ether types. All these packets are passed to a single RX queue and handled by one thread. Hence, in our current implementation, there is only one thread running the dispatcher which distributes all incoming packets to different CPU cores. In the experiments, we used a random dispatching strategy, in which packets are dispatched to different kernel threads randomly and evenly.

Fig. 5 shows that both of the throughput and the *Data* forwarding rate grow almost linearly up to 8 or 12 forwarding cores. After that, the growing slows down and finally achieves 6.50 and 6.13 Gbps with 24 cores. We also evaluated the performance influence of whether using a separated core for the dispatcher. Note in Fig. 5 that the *separated-** represents that the dispatcher never dispatches packets to the core it is running on. The core (separated or shared) for dispatcher is also counted in the number of working cores, hence the result of *separated-** starts from two.

In Fig. 5, we observe that the performance of shared core is higher than that of the separated core with up to 4 cores. This is because the software based dispatcher has not achieved its performance bottleneck yet and separating a core for it means decreasing the number of cores for packet forwarding and finally leads to performance decreasing. However, when there are more than 8 cores, the forwarding throughput is relatively sufficient and the single core dispatcher becomes the bottleneck. In this case, sharing a core decreases the throughput of the dispatcher and finally limits the maximum throughput of iStack. Hence, with a separated core for the dispatcher, the input bottleneck comes later (12 instead of 4 cores as shown in Fig. 5) and the maximum throughput of iStack achieves 6.50 Gbps. Given that the number of cores for packet ingress depends on how many NICs are installed on the system and the hardware RSS of the NICs, we plan to eliminate the dispatching bottleneck in the future work.

In order to evaluate the overhead reduction of taking CS into kernel space, we also implemented an out-kernel version which communicates with iStack through Netlink sockets. CS in user space needs to copy packet buffer from/to kernel space via system calls for *Data* insertion/retrieval. It has significant overhead comparing with the in-kernel version. Table. 1 presents the forwarding delay with capacity varying from 10 K to 1M. The forwarding delay increases slightly with capacity as larger capacity means higher workload. Table. 1 also shows that the in-kernel CS of iStack reduces over 45% forwarding delay for the case of cache hit.

5.3 Application Performance

Fig. 6 shows that varying content size, the goodput grows up linearly from 762.4 Mbps to 5.07 Gbps. Meanwhile the *Data* forwarding rate keeps at about 0.45 Mpps, which proves that the payload length has limited impact on iStack forwarding rate but is important for application goodput. Due to the applications imports additional overhead, the measured maximum *Data* forwarding rate (0.48 Mpps) is slightly lower than that measured with the *Interest* generator, which achieves 0.53 Mpps. Another observation is that comparing with an iStack router, an edge host running applications with iStack can achieve 90.5% of the maximum performance.

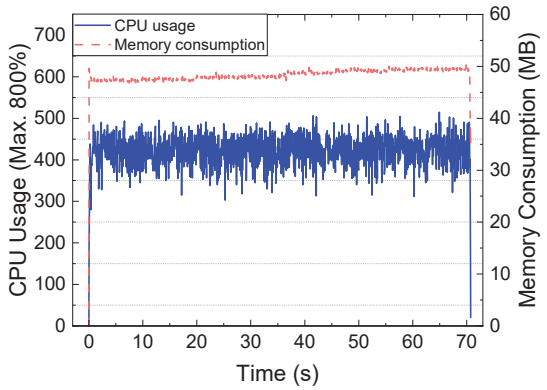
In Fig. 6, the differences between the results of excluding and including names are similar with different content size, whereas it is worth noting that with the content size growing, the proportion of the difference decreases. Hence, if the name does not carry application information, using larger content size and shorter name is more efficient.

5.4 CPU Usage and Memory Consumption

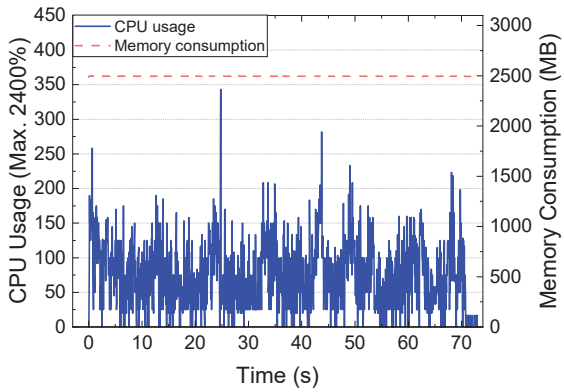
As iStack targets to deploy on edge hosts and routers for general purpose, we evaluated CPU usage and memory consumption of it. The result is measured from the entire system instead of some specific applications. Hence it can better indicate the resource overhead of an end host. In this measurement CS was disabled on end hosts to avoid its influence on memory consumption. All consumer applications continuously ran for 30 times after a warm-up running.

The result is illustrated in Fig. 7. Note that the two hosts were equipped with different cores of CPU and hence the celling of CPU usage in Fig. 7(a) and Fig. 7(b) are 800% and 2400%, respectively. The CPU usage of consumer applications is relatively high, because the consumer-side applications used sockets with the type SOCK_SEQPACKET, each of which ran a TCP-like sliding window algorithm and took care of retransmission based on timeout. The average goodput in this measurement is about 4.97 Gbps. Fig. 7(b) shows that such scenario is an easy task for providers as most of the time the CPU usage is under 200% (maximum 343.3%).

We measured initial system memory consumption before running consumer/provider applications and the result in Fig. 7 is the real-time measurement minus the initial value. Fig. 7(a) shows that the total memory consumption with 15 consumer applications concurrently running is less than 50 MBytes. The memory consumption of both the server and the host vary slightly, which indicates that excepting CS, iStack has little memory consumption and is suitable for edge usage.



(a) Consumers (on PC equipped with 8 cores i7-7700)



(b) Provider (on server equipped with 24 cores Xeon E5-2620)

Figure 7: CPU usage and memory consumption

5.5 Discussion

Our evaluations show that iStack is a promising in-kernel network stack for general purpose. iStack achieves up to 6.50 Gbps throughput and meanwhile keeps low CPU usage and memory consumption. A comparison among iStack and other NDN forwarders is shown in Table 2. NDN-DPDK has the highest throughput. Nevertheless, DPDK-based solutions have limitation of hardware depending and polling-based I/O is not suitable for edge usages, especially for low-end devices.

Among all event-driven forwarders, iStack outperforms in terms of the throughput/goodput with a factor of 6.26x-16.25x. Besides, our in-kernel CS eliminates the overhead of memory copy between kernel space and user space for caching and retrieving *Data*, respectively. The evaluation result shows that in-kernel caching reduces upto 46.08% forwarding latency in case of cache hit. It is worth noting that currently iStack has not been fully optimized and the single-thread dispatcher also limits its multi-threading performance.

iStack is not another NDN forwarder but an ICN protocol stack. The other implementations in Table 2 are all user-space forwarders. In contrast, iStack is designed and implemented integrated with OS kernel and our prototypes can be compiled with the source code of Linux kernel. iStack provides system-level NDN functionality supporting. Applications can access the named network through OS network sockets directly.

5.6 Lessons Learned

Our experiences constructing iStack have revealed important insights into putting NDN into OS kernel. First, we have discovered that from a system perspective, the namespace of NDN should be carefully organized. Allowing applications to register arbitrary prefixes into FIB results in unnecessary overhead of the in-kernel network stack. Hence, iStack classifies prefixes into routable ones and application-specific ones. In the network layer of iStack, FIB only deals with routable prefixes. Applications can binding more specific prefixes which are handled by BPT. Hence, iStack decouples local demultiplexing from network-layer routing and forwarding, while keeping NDN name-based communication model.

Second, in contrast to IP's stateless forwarding design, NDN maintains per-packet state at the network layer. Crafting a robust stateful network stack introduces additional complexity. One challenge arises from the potential for network state to create blockages with certain OS interfaces. Events such as socket closed or network cable unplugged can occur while

Table 2: Comparison with other forwarders

Forwarder/Stack	Type	I/O Driven	Programming Model	Net. Throughput	App. Goodput	Language
NFD [33]	App-level	Event	Specific API	0.4 Gbps	N/A	C++
NFD-Opt. [33]	ditto	ditto	ditto	0.9 Gbps	N/A	ditto
YaNFD [23]	App-level	Event	Specific API	N/A	0.81 Gbps	Go
MW-NFD [22]	App-level	Polling	Specific API	4.26 Gbps	N/A	C++
NDN-DPDK [20]	Bypass kernel	Polling	Specific API	N/A	22 Gbps*	C & Go
iStack	**In kernel**	**Event**	**Network socket**	**6.50 Gbps**	**5.07 Gbps**	**C**

* NDN-DPDK achieves 108 Gbps throughput with Jumbo frames (8000 B). For fairness, we list results that are based on standard Ethernet frames (1500 B).

corresponding Interests are still pending in PIT. iStack mitigates such potential blockages through its thread-safe face system, elaborated in Section 3.3.3. Another consideration is the need for judicious management of overhead. iStack employs streamlined forwarding data structures and avoids packet duplication to minimize the overhead with stateful network processing. Additionally, iStack adopts an interruption-based packet reception approach rather than a polling-based one. Note that in cases of extremely high throughput demands, such as those in data-center server environments, a polling approach like DPDK might be a viable alternative. However, such solutions often lead to elevated CPU usage even during idle periods, potentially monopolizing resources required for non-networking processes. Consequently, iStack ultimately embraces an interruption-based approach.

Third, as NDN enables in-network caching, an in-kernel network stack for it involves storage into the OS kernel. As mentioned in Sec. 4.3.1, taking storage into kernel should consider two issues: memory consumption and capacity. As a packet certainly exists in the network stack before it is cached, an in-kernel stack can directly cache *Data* packets in kernel-space memory for fast response. However, consuming too much memory for fast in-network caching hinders other tasks running on the host. Hence, iStack carefully limits the in-kernel memory usage for CS and utilizes external storage to extend CS capacity. The Linux kernel disapproves operating files directly and from our developing experience, there are two promising methods to achieve this. One is building a user-space CS and exchange packets with in-kernel stack through socket interfaces. Another one is designing a new storage mode for the rising in-network caching involved by NDN.

Network stacks in current OS is built to support the existing node-centric networking model, with the intrinsic notion of socket-connection-interface mapping. An in-kernel stack communicates with user-space applications through system calls. Specifically for networking, there are socket-based system calls. Although presently the socket API of iStack is able to support essential NDN communication functionalities, it is a send/receive programming model and lacks the aware of symmetrical flow, the unique naming of each packet, and other features included by NDN. Hence, designing an information-centric programming model and the corresponding API for NDN applications is an attractive research topic.

6 Conclusion and Future Work

This paper describes iStack, a general-purpose information-centric network stack, taking NDN into OS kernel, for practical deployment and usage. Our work contains both intellectual and practical contributions. On the intellectual side, iStack shows how to integrate a stateful, entirely name-based protocol stack into OS kernel and be compatible with the socket mechanism. On the practical side, iStack demonstrates itself as a practical kernel-level protocol stack for NDN.

We implement iStack prototypes on different four kinds of platforms from high performance servers to low-end devices. Evaluation shows that iStack achieves 6.50 Gbps throughput and meanwhile keeps reasonable CPU usage and memory consumption, which indicates that iStack is sufficient for the requirements of general usage and practical deployment.

To the best of our knowledge, iStack is the first in-kernel stack for NDN and achieves both generality and high performance. In terms of generality, iStack supports multiple applications that have different purposes and requirements with network socket mechanism implemented by operating systems directly. In terms of performance, iStack outperforms the NDN-testbed forwarder by a factor of 16.25x and even faster than the polling-based MW-NFD by a factor of 1.52x.

Our future work includes the following aspects. First, we also design a series of information-centric socket primitives and the corresponding programming models for iStack to be published. Second, we are working on porting iStack to other platforms including Andriod and Windows. Third, with further developing and extensive testing in more cases, we will provide a well-polished version for community as well as long term support and evolution in the near future[4]. We believe that iStack is not just another forwarder for NDN, but a step forward for the development of ICN.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under grant No. 92067203. We would like to thank anonymous reviewers and our shepherd James Mickens for thoughtful feedback.

[4]We provide long term support through the website: *name-ip.cn*

References

[1] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, KC Claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. Named Data Networking. *ACM SIGCOMM Computer Communication Review*, 44(3):66–73, 2014.

[2] Yating Yang, Tian Song, and Beichuan Zhang. OpenCache: A lightweight regional cache collaboration approach in hierarchical-named ICN. *Computer Communications*, 144:89–99, 2019.

[3] Tian Pan, Xingchen Lin, Enge Song, Cheng Xu, Jiao Zhang, Hao Li, Jianhui Lv, Tao Huang, Bin Liu, and Beichuan Zhang. NB-Cache: Non-Blocking In-Network Caching for High-Performance Content Routers. *IEEE/ACM Transactions on Networking*, 29(5):1976–1989, 2021.

[4] Zhiyi Zhang, Yingdi Yu, Haitao Zhang, Eric Newberry, Spyridon Mastorakis, Yanbiao Li, Alexander Afanasyev, and Lixia Zhang. An Overview of Security Support in Named Data Networking. *IEEE Communications Magazine*, 56(11):62–68, 2018.

[5] Tian Song, Yating Yang, and Tianlong Li. Rethinking Caching Security of Information-Centric Networking: A System Recovery Perspective. *IEEE Communications Magazine*, 57(10):104–110, 2019.

[6] Yu Zhang, Hongli Zhang, and Lixia Zhang. Kite: A Mobility Support Scheme for NDN. In *Proceedings of the 1st ACM Conference on Information-centric Networking*, pages 179–180, 2014.

[7] Xavier Mwangi and Karen Sollins. MNDN: Scalable Mobility Support in Named Data Networking. In *Proceedings of the 5th ACM Conference on Information-Centric Networking*, pages 117–124, 2018.

[8] Peter Kietzmann, Cenk Gündoğan, Thomas C Schmidt, Oliver Hahm, and Matthias Wählisch. The Need for a Name to MAC Address Mapping in NDN: Towards Quantifying the Resource Gain. In *Proceedings of the 4th ACM Conference on Information-Centric Networking*, pages 36–42, 2017.

[9] Tianlong Li, Tian Song, Yating Yang, and Jike Yang. iCast: Dynamic Information-Centric Cross-Layer Multicast for Wireless Edge Network. In *Proceedings of the 9th ACM Conference on Information-Centric Networking*, pages 137–147, 2022.

[10] Ran Zhu, Tianlong Li, and Tian Song. iGate: NDN Gateway for Tunneling over IP World. In *2021 International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9. IEEE, 2021.

[11] Safdar Hussain Bouk, Syed Hassan Ahmed, Dongkyun Kim, and Houbing Song. Named-Data-Networking-Based ITS for Smart Cities. *IEEE Communications Magazine*, 55(1):105–111, 2017.

[12] Yating Yang and Tian Song. Local Name Translation for Succinct Communication Towards Named Data Networking of Things. *IEEE Communications Letters*, 22(12):2551–2554, 2018.

[13] Yating Yang, Tian Song, Weijia Yuan, and Jianping An. Towards reliable and efficient data retrieving in ICN-based satellite networks. *Journal of Network and Computer Applications*, 179:102982, 2021.

[14] Chavoosh Ghasemi, Hamed Yousefi, and Beichuan Zhang. Internet-Scale Video Streaming over NDN. *IEEE Network*, 35(5):174–180, 2021.

[15] NFD: Named Data Networking Forwarding Daemon, 2014. https://named-data.net/doc/NFD/current.

[16] Alexander Afanasyev, Junxiao Shi, Beichuan Zhang, Lixia Zhang, Ilya Moiseenko, Yingdi Yu, Wentao Shang, Yi Huang, Jerald Paul Abraham, Steve DiBenedetto, et al. NFD Developer's Guide. *Technical Report NDN-0021, NDN*, 2014.

[17] NDN Project. NDN Testbed, 2017. http://ndndemo.arl.wustl.edu/.

[18] Wentao Shang, Alex Afanasyev, and Lixia Zhang. The Design and Implementation of the NDN Protocol Stack for RIOT-OS. In *2016 IEEE Globecom Workshops*, pages 1–6. IEEE, 2016.

[19] Zhiyi Zhang, Edward Lu, Yanbiao Li, Lixia Zhang, Tianyuan Yu, Davide Pesavento, Junxiao Shi, and Lotfi Benmohamed. NDNoT: A Framework for Named Data Network of Things. In *Proceedings of the 5th ACM Conference on Information-Centric Networking*, pages 200–201, 2018.

[20] Junxiao Shi, Davide Pesavento, and Lotfi Benmohamed. NDN-DPDK: NDN Forwarding at 100 Gbps on Commodity Hardware. In *Proceedings of the 7th ACM Conference on Information-Centric Networking*, pages 30–40, 2020.

[21] DPDK Project. Data Plane Development Kit (DPDK), 2015. http://www.dpdk.org.

[22] Sung Hyuk Byun, Jongseok Lee, Dong Myung Sul, and Namseok Ko. Multi-Worker NFD: an NFD-compatible High-speed NDN Forwarder. In *Proceedings of the 7th ACM Conference on Information-Centric Networking*, pages 166–168, 2020.

[23] Eric Newberry, Xinyu Ma, and Lixia Zhang. YaNFD: Yet another Named Data Networking Forwarding Daemon. In *Proceedings of the 8th ACM Conference on Information-Centric Networking*, pages 30–41, 2021.

[24] Zhuo Li, Yaping Xu, Beichuan Zhang, Liu Yan, and Kaihua Liu. Packet Forwarding in Named Data Networking Requirements and Survey of Solutions. *IEEE Communications Surveys & Tutorials*, 21(2):1950–1987, 2018.

[25] Pierre-Jacques Courtois, Frans Heymans, and David Lorge Parnas. Concurrent Control with "Readers" and "Writers". *Communications of the ACM*, 14(10):667–668, 1971.

[26] Mauro Sardara, Luca Muscariello, and Alberto Compagno. A Transport Layer and Socket API for (h)ICN: Design, Implementation and Performance Analysis. In *Proceedings of the 5th ACM Conference on Information-centric Networking*, pages 137–147, 2018.

[27] Giovanna Carofiglio, Massimo Gallo, and Luca Muscariello. Joint Hop-by-hop and Receiver-Driven Interest Control Protocol for Content-Centric Networks. *ACM SIGCOMM Computer Communication Review*, 42(4):491–496, 2012.

[28] Milad Mahdian, Somaya Arianfar, Jim Gibson, and Dave Oran. MIRCC: Multipath-aware ICN Rate-based Congestion Control. In *Proceedings of the 3rd ACM Conference on Information-Centric Networking*, pages 1–10, 2016.

[29] Sichen Song and Lixia Zhang. Effective NDN Congestion Control Based on Queue Size Feedback. In *Proceedings of the 9th ACM Conference on Information-Centric Networking*, pages 11–21, 2022.

[30] Yanbiao Li, Zhiyi Zhang, Xin Wang, Edward Lu, Dafang Zhang, and Lixia Zhang. A Secure Sign-On Protocol for Smart Homes over Named Data Networking. *IEEE Communications Magazine*, 57(7):62–68, 2019.

[31] Tianyuan Yu, Hongcheng Xie, Siqi Liu, Xinyu Ma, Xiaohua Jia, and Lixia Zhang. CertRevoke: A Certificate Revocation Framework for Named Data Networking. In *Proceedings of the 9th ACM Conference on Information-Centric Networking*, pages 80–90, 2022.

[32] Davide Pesavento, Junxiao Shi, Kerry McKay, and Lotfi Benmohamed. PION: Password-based IoT Onboarding Over Named Data Networking. In *ICC 2022-IEEE International Conference on Communications*, pages 1070–1075. IEEE, 2022.

[33] Yaoqing Liu, Anthony Dowling, and Lauren Huie. Benchmarking Network Performance in Named Data Networking (NDN). In *2020 29th Wireless and Optical Communications Conference*, pages 1–6. IEEE, 2020.

Cloudcast: High-Throughput, Cost-Aware Overlay Multicast in the Cloud

Sarah Wooders
UC Berkeley

Shu Liu
UC Berkeley

Paras Jain
Genmo AI

Xiangxi Mo
UC Berkeley

Joseph E. Gonzalez
UC Berkeley

Vincent Liu
University of Pennsylvania

Ion Stoica
UC Berkeley

Abstract

Bulk data replication across multiple cloud regions and providers is essential for large organizations to support data analytics, disaster recovery, and geo-distributed model serving. However, data multicast in the cloud can be expensive due to network egress costs and slow due to cloud network constraints. In this paper, we study the design of high-throughput, cost-optimized overlay multicast for bulk cloud data replication that exploits trends in modern provider pricing models along with techniques like ephemeral waypoints to minimize cloud networking costs.

To that end, we design an optimization algorithm that uses information about cloud network throughput and pricing to identify cost-minimizing multicast replication trees under user-given runtime budgets. Our open-source implementation, Cloudcast, is used for cloud overlay multicast that supports pluggable algorithms for determining the multicast tree structure. Our evaluations show that Cloudcast achieves 61.5% cost reduction and 2.3× replication speedup compared to both academic and commercial baselines (e.g., AWS multi-region bucket) for multi-region replication.

1 Introduction

Increasingly, data in the cloud must be replicated to multiple cloud providers and different regions within each provider. For example, geo-distributed applications like model serving require model weights or features computed in a single region to be replicated to multiple geographic regions to reduce serving latency for users accros the globe [22,44,51]. Data sharing between collaborating organizations using different providers similarly requires replicating data to multiple locations. Finally, the growth of multi-cloud applications that leverage resources from multiple providers is dependent on application data being available across provider boundaries [13,51,52].

Of course, data replication and multicast are not new. Both topics have been extensively studied to optimize throughput and scalability in the context of IP networks, peer-to-peer overlays [12,14,19,22,33], and inter-DC networks [20,36,47,54].

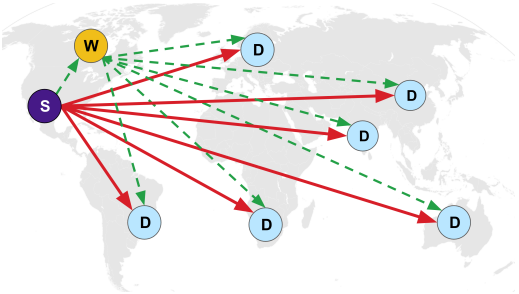


Figure 1: Direct replication from a source region (purple) to destination regions (blue) may traverse expensive or slow links, which can be avoided via *waypoint* regions (yellow).

However, replication between cloud regions and providers introduces first-order concerns beyond just throughput and scalability. In particular, the *monetary cost* of the transfer is a critical factor and one that (as we show later in this paper) is poorly handled by existing techniques for optimizing throughput [33,36,54]. While some existing works consider the monetary cost for multicast, they either ignore the throughout [24] or assume a capacity-based pricing model [35] which is inconsistent with today's cloud. In contrast to capacity-based pricing, cloud providers charge per-GB network egress fees for data transferred out of a given region to another region or cloud provider. Per-GB egress fees introduce a multiplicative term into the transfer cost—(egress price)×(amount transferred)—making it significantly more difficult to optimize throughput and cost.

Egress costs can vary by orders of magnitude depending on the source and destination [42], as well as the capacity of cross-region links. As a result, the structure of the multicast replication tree (i.e., what data is replicated along which paths) can dramatically affect the end-to-end throughput and monetary cost of replication. As a concrete example, consider replication from a GCP source region to six AWS regions (Figure 1). Direct replication of the data between the source and each destination region (shown in red arrows) would cost $720 per TB. Instead, replicating to an AWS region with the lowest cross-region egress fees once and multicasting

data from that AWS region to other regions (shown in dotted green arrows) would reduce the price to $240 per TB. Further modifying the multicast tree to utilize high-throughput links and offload egress bandwidth from the source node can also improve throughput.

In this work, we solve the problem of *high-throughput, cost-optimized cloud multicast* in which we minimize the cost of data replication while achieving a target replication time (across all destinations) for bulk multicast replication. Cloud multicast incurs costs from network egress fees and compute resources needed to mediate the transfer. In addition, cloud multicast must meet application Service Level Objectives (SLO) for the replication time, such as providing freshness guarantees on replicated data.

We design an optimizer to determine a multicast tree structure given a user-specified source region, destination regions, and target replication time. By providing varying target replication times, our optimizer can generate a Pareto-curve (shown in Figure 8) that improves replication cost and throughput compared to prior approaches for cloud multicast [23, 24]. We achieve this by leveraging techniques such as striping, VM parallelism, and overlay networking, while also accounting for the cloud providers' network characteristics, resource constraints, and per-GB network pricing model.

Designing this optimization is challenging for two main reasons. First, the optimizer must account for path-specific pricing models, resource constraints, and varying performance across cloud providers. Existing techniques that formulate the optimization problem in terms of bandwidth allocation cannot be adapted to account for per-GB network pricing without making the problem non-linear (described further in Section 3). Second, the optimization search space is combinatorially large, as the optimizer must determine both the set of overlay waypoint regions (regions which are neither the source nor destination) as well as how data will be routed along the overlay network. Unlike the traditional overlay settings, the cloud offers significantly more flexibility in the number and the location of overlay nodes, as cloud VMs can dynamically be instantiated in specified cloud provider regions. Furthermore, replicating subsets of data (i.e., stripes) via different paths is critical for achieving high-throughput [12]. We introduce several approximations (e.g., pre-selecting the regions and limiting path lengths) to reduce this search space and enable the optimizer to run within seconds.

To run overlay multicast across clouds, we develop *Cloudcast*, a system for bulk data overlay multicast across GCP, AWS, and Azure. Cloudcast has a centralized control plane that supports pluggable algorithms for determining the number and location of overlay nodes and replication trees for multiple segments of data. We implement our optimizer as well as several baseline algorithms as part of Cloudcast's control plane. We run system experiments to multicast data across clouds and show that Cloudcast is able to achieve up to 62.4% cost savings and 2.84× replication speedup depending on the control plane algorithm (Figure 10).

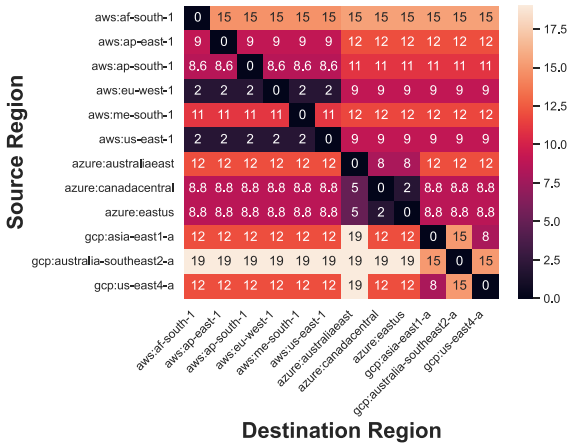


Figure 2: Egress fees between regions (in cents per GB).

We run an end-to-end system evaluation comparing Cloudcast with BitTorrent [19] and AWS's commercial offering for multi-region bucket replication [50], which, like most cloud data replication offerings, only supports replication into or within that cloud. We find that Cloudcast achieves 7.7× replication speedups and 28.4% cost savings compared to BitTorrent (Figure 12). Compared to multi-region bucket replication, we find that Cloudcast achieves up to 61.5% cost reduction and 2.3× replication speedup (Figure 11).

To summarize, we make the following contributions:

1. We design an optimizer for minimizing replication cost under replication time constraints.
2. We introduce several approximations to reduce the search space for the optimizer, reducing the solver runtime from hours to seconds.
3. We build Cloudcast, an open-source system for cloud overlay multicast with pluggable data transfer policy.

2 Problem Setup

We frame the problem of cloud multicast in terms of constructing an overlay network for replicating data, which involves defining: (1) the set of *overlay nodes* (i.e., cloud VMs) and (2) the *paths* between those nodes that will be included in a multicast replication tree. Cloudcast eventually divides the target data into multiple *stripes* (i.e., partitions), so concurrent replication trees may be used in a single transfer. Our optimization objective is to minimize the monetary cost of replication while also meeting a replication time constraint.

2.1 Egress Costs

A unique aspect of multicast in the cloud is the effect of egress costs incurred for data transferred across cloud regions. Cloud providers charge for wide-area data transfer *per-GB* of data transferred. Egress prices—as a method of keeping data within the provider's regions without disincentivizing

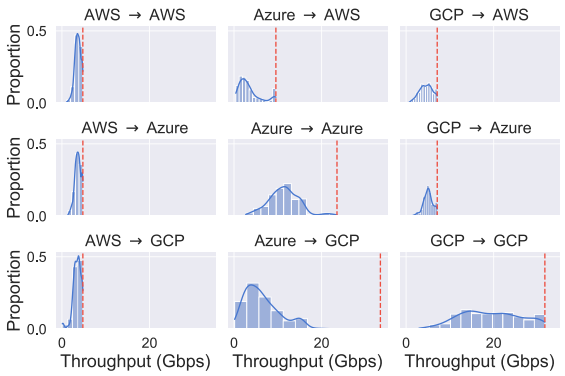


Figure 3: Bandwidth distribution (in Gbps) between regions. Per-VM egress limits are marked in red dotted lines.

migration into the provider—dominate data movement costs in the cloud and fundamentally change the multicast problem. Figure 2 visualizes the pricing for 11 regions across AWS, Azure, and GCP. Prices vary depending on the source and destination cloud or region, with differences of up to 23× across region pairs. Along those lines, one particularly important axis is whether the transfer stays within a given cloud provider or crosses provider boundaries, as inter-cloud egress costs are generally higher than intra-cloud egress.

Intra-cloud egress (data movement between geographically separated datacenters in the same cloud provider) is priced between $\$0.01 - \$0.19$ per GB transferred. Prices typically increase with longer-distance transfers. For example, GCP charges \$0.08 for transfers between continents but only \$0.02 for transfers within the US. Some smaller providers (e.g., IBM, Cloudflare) offer free cross-region egress.

Inter-cloud egress (data movement between different cloud providers) is typically priced at a much higher rate per GB ($\$0.08 - \$0.23$). As such, it is essential to minimize cross-cloud transfers in a multicast replication tree.

2.2 Bandwidth Variability Across Endpoints

Meeting replication time constraints can be challenging due to network bandwidth variability in the cloud. One type of variability arises from cloud providers, who impose constraints on per-VM egress and ingress bandwidth. These constraints differ significantly across providers: for instance, AWS throttles intra-cloud and inter-cloud egress to 5 Gbps per VM, while Azure imposes no VM-level limits. The impact of these egress limits can be observed in Figure 3, where bandwidth is capped at the VM egress limit for AWS and GCP. Limited node egress poses a particular challenge for cloud multicast, as the source node's egress bandwidth is often the bottleneck.

Even when source-node bandwidth is not the bottleneck, observed network capacity can also vary considerably across cloud region pairs (up to 202×). Note that these networks are relatively stable across time; prior work [28] has found that network throughputs are stable over periods of at least 24 hours. Instead, variations are primarily observed across different source and destination regions. Figure 3 depicts the distribution of profiled bandwidth between VMs running in AWS, Azure, and GCP. Intra-cloud bandwidth is typically (but not always) higher than inter-cloud bandwidth.

2.3 Elasticity of Resources

A major advantage of the cloud is resource elasticity and the ability to flexibly provision VMs across many regions. In the face of the source bottlenecks described above, VM elasticity translates to a corresponding *elasticity of bandwidth*. Allocating multiple parallel VMs enables users to scale throughput beyond per-VM network bandwidth limits.

Unfortunately, adding elastic VM capacity at the source region has limitations. Additional VMs add additional costs due to per-second billing on VMs, which can impact the cost/throughput tradeoff. We note that because the marginal cost of additional VMs is often relatively small compared to egress fees, the tradeoff is often worth making. However even in these situations, bandwidth elasticity has limits: for instance, if a network-based bottleneck is unavoidable or when cloud providers limit the number of vCPUs per region.

Crucially, elastic VM capacity can also be deployed at *waypoint regions* that are neither the source nor the destination. These waypoint regions can help mitigate source VM bottlenecks by distributing load from multicast fan-out across multiple separate regions. Waypoint regions also mitigate points of congestion by routing data around slow paths.

2.4 Illustrated Example

Selecting overlay nodes and replication trees to optimize cost and throughput is challenging. Consider the toy example in Figure 4 for a 2 GB replication with two 1 GB stripes. Assuming a 4 Gbps bandwidth limit for all nodes and one VM per region, the source ("S") and destination ("D") nodes have fast but expensive outgoing paths, capable of sending at 2 Gbps but costing 10¢ per GB transferred. Other regions have cheaper but slower outgoing paths, capable of sending at 1 Gbps but costing 2¢ per GB transferred. In a simple direct replication scenario, the replication will be bottlenecked by the source node's egress limit (4 Gbps). With two copies of data to send, the total transfer time will be 8 seconds.

Like many bandwidth-optimized techniques [12, 23, 33], we offload egress bandwidth by sending a single data copy from the source and leveraging multiple replication trees. Replication cost is reduced by replicating to a waypoint, and then multicasting to destinations. This doubles replication time to 16 seconds due to stripes being replicated via the slower path (dotted arrows). An 8-second replication SLO is met by transferring just one stripe via the cheaper waypoint.

This simple example presents a large search space for possible replication trees, and real-world cloud networks present

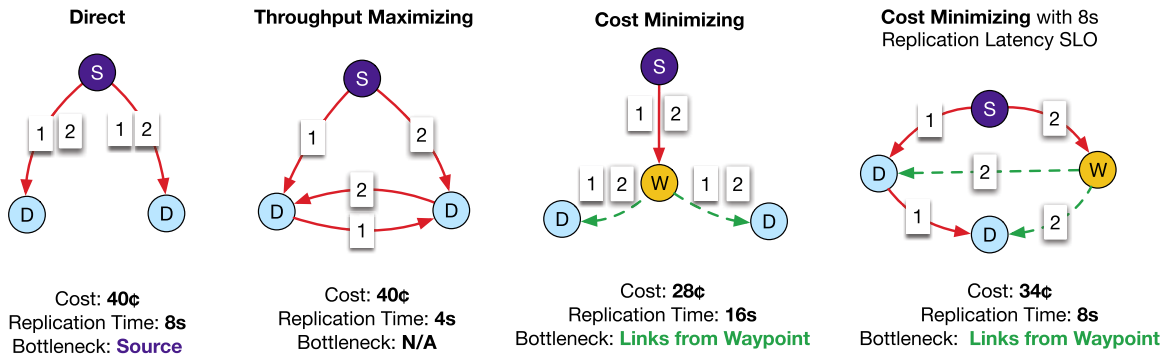


Figure 4: Overlay node set and distribution trees for a toy example. The source and destination nodes are marked 'S' and 'D' respectively, while waypoint nodes in yellow are marked 'W'. Expensive, fast paths ($0.1 per GB, 2 Gbps) are shown in solid red, while slow, cheap paths ($0.02 per GB, 1 Gbps) are shown in dashed green.

additional parameters such as choosing the number of VMs per region and many possible waypoint regions.

3 Cost Optimization in Cloudcast

We design an optimizer to minimize replication cost while meeting a replication time SLO (i.e., a constraint on the maximum replication time to a destination). Our optimizer has two main contributions. The first is a Mixed-Integer Linear Program (MILP) formulation of the cost-aware multicast problem, jointly selecting overlay nodes and replication trees. While others [23, 54] have used MILP formulations for multicast overlay design, they formulate the optimization problem in terms of *bandwidth*. Extending these formulations to accommodate per-GB costs would violate linearity as data transfer volume (cost) is proportional to the product of the key decision variables: allocated bandwidth and replication time. As a consequence, we propose a new formulation that reframes the optimization in terms of data volume. Our new formulation assigns discrete subsets of data (i.e. stripes) to replication paths in the network while ensuring that a complete copy of the data arrives at all destinations. Unfortunately, solving this MILP formulation can be intractable for larger numbers of destinations. Our second contribution is an approximation of the MILP formulation that significantly reduces solve time without significantly degrading the solution quality.

3.1 Egress Cost Minimization Algorithms

The challenge with our optimization problem stems from having to consider *both* throughput and cost. Without replication time constraints, we observe that the Steiner Tree [27] minimizes egress cost. A Steiner Tree is a set of cost-minimizing edges that form a tree that connects a subset of nodes within a graph. If we do not allow the use of waypoint regions, the cost-minimizing tree is a Minimum Spanning Tree (MST). While solving for the MST can be done in linear time, the Steiner Tree problem is NP-hard, though many approximations exist [43]. We cannot use the Steiner Tree to account for replication throughput or instance costs, since it only optimizes total edge cost, but we expect our optimizer's solution to be similar to a Steiner Tree in cases where the replication time constraint is loose.

Inputs	
$\text{TRANSFER-SIZE} \in \mathbb{R}$	*Transfer size in GB*
$\text{TIME} \in \mathbb{R}$	*Replication time constraint*
$\text{STRIPES} \in \mathbb{Z}_+$	*Number of data stripes*
Decision Variables	
$P \in \{0,1\}^{\lvert\text{STRIPES}\rvert \times \lvert V\rvert \times \lvert V\rvert}$	*Path indicator variable*
$N \in \mathbb{Z}_+^{\lvert V\rvert}$	*Number of VMs per region*
$F \in \mathbb{R}_+^{\lvert\text{STRIPES}\rvert+1 \times \lvert V\rvert \times \lvert V\rvert}$	*Flow feasibility variable*
Constants: Cross-Region Paths (edges)	
$\text{BANDWIDTH}^{path} \in \mathbb{R}_+^{\lvert V\rvert \times \lvert V\rvert}$	*Bandwidth profile matrix (Gbps)*
$\text{COST}^{path} \in \mathbb{R}_+^{\lvert V\rvert}$	*Network cost ($/Gbit)*
Constants: VM Instances (nodes)	
$\text{EGRESS}^{VM} \in \mathbb{R}_+^{\lvert V\rvert}$	*Per region per VM egress limit (Gbps)*
$\text{INGRESS}^{VM} \in \mathbb{R}_+^{\lvert V\rvert}$	*Per region per VM ingress limit (Gbps)*
$\text{COST}^{VM} \in \mathbb{R}_+^{\lvert V\rvert}$	*Per region per VM cost ($/s)*
$\text{LIMIT}^{VM} \in \mathbb{Z}_+^{\lvert V\rvert}$	*Max number of VMs per region*

Table 1: Symbol table for Cloudcast's ILP formulation.

3.2 Profiling Cross-region Bandwidth

The bandwidth of paths between cloud regions (both intra-cloud and inter-cloud) is determined by the number of VMs in each region, each VM's egress and ingress limits, and the profiled bandwidth. As discussed in Section 2.2, cross-region bandwidth per VM can be estimated by profiling the bandwidth between region pairs using `iperf3`. Egress and ingress limits vary across cloud providers but are static and can be determined by cloud providers' documentation [8, 10, 16]. We utilize these profiles as an estimate of expected network bandwidth for the duration of a transfer. Profiling results are included as part of our open-source repository and shared across all users of Cloudcast.

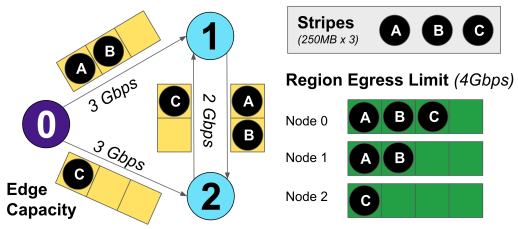


Figure 5: Stripes transferred from the source (purple) to destinations (blue) are placed by the solver along edges depending on edge capacity (yellow) and node capacity (green).

3.3 Optimizing Cost with Time Constraints

In order to minimize replication price while meeting runtime requirements and cloud resource constraints, we frame a MILP on a directed graph representing the entire cloud topology. The input to the optimizer is the transfer size: TRANSFER-SIZE, the runtime budget: TIME, and the number of stripes: STRIPES to divide the data into.

To formulate the optimization problem as a MILP, formulate the problem in terms of allocating data *volume* to edges rather than bandwidth, with allocation units per stripe. We translate cross-region bandwidth and per-region egress/ingress limits into volume capacities, as shown in Figure 5, this determines how many stripes can fit along each edge. This makes the MILP similar to a bin packing problem, where we aim to pack stripes into edges such that all destinations receive all stripes. The volume-based representation allows cost to be computed as a function of the number of stripes placed on each edge.

Next, we formally describe the MILP decision variables, objectives, and constraints. The cloud regions and cross-region paths are represented as $G = (V, E)$, where V denotes the set of cloud regions and E denotes paths between regions. We provide a reference table for the notation in Table 1.

3.3.1 Decision variables

The MILP formulation consists of three decision variables. The path indicator variable $P_{s,(v,u)}$ indicates whether a stripe s is sent between regions $(u,v) \in V$. The paths selected by P make up the multicast replication tree for each stripe. The decision variable N_v represents the total number of overlay routers in the region v. An additional flow variable $F_{s,(u,v)}$ ensures valid paths when constructing the multicast tree. It ensures that the paths selected by P do not contain cycles and are connected, by allowing flow to be pushed from the source to all destinations for each stripe (see Section 3.3.3).

3.3.2 Objective: minimizing price under a deadline

To minimize the price of a multicast transfer while meeting replication time constraints, we use a two-part objective function. The first part optimizes the number of virtual machines (VMs) per region, represented by N, and the second part optimizes the distribution trees per stripe, represented by P. The objective is formulated as follows:

$$\arg\min_{P,N} \quad \underbrace{\text{TIME} \times \langle \text{COST}^{VM}, N \rangle}_{\text{Instance Cost}} \tag{1}$$

$$+ \quad \underbrace{\frac{\text{TRANSFER-SIZE}}{\text{STRIPES}} \times \sum_{s \in \text{STRIPES}} \langle \text{COST}^{path}, P_s \rangle}_{\text{Egress Cost}} \tag{2}$$

The price of a data transfer is the sum of the instance fee and the egress fee. The instance fee depends on the number of VMs running per region, the job completion time, and the per-region VM fee. The egress fees are determined by the data distribution path and the amount of data traversed through the path, as defined by P. We note that the instance cost is also an upper bound as it can be potentially overestimated if the data transfer is completed in less than the user-defined time budget. However, this is necessary to ensure linearity.

3.3.3 Constraints

We represent cross-region bandwidth, node egress/ingress bandwidth, per-region VM limits, and replication tree structure requirements as constraints within the MILP.

Representing Inter-Region & Inter-Cloud Bandwidth. Cross-region bandwidth is represented as the *per-GB capacity* given the run-time budget, i.e., how many stripes can fit along an edge. Increasing the number of VMs in the source regions linearly increases the rate at which we can send data. We thus model the bandwidth between two regions as the per-VM bandwidth profiled between those two regions multiplied by the number of VMs in the source region:

$$\text{CAPACITY}^{path} = \langle N, \text{BANDWIDTH}^{path} \rangle * \text{TIME}, \tag{3}$$

and constrain P in terms of the path capacity:

$$\forall (u,v) \in E \qquad \text{SIZE}_{\text{STRIPE}} * \sum_{s} P_{s,(u,v)} \leq \text{CAPACITY}^{path}{}_{(u,v)}. \tag{4}$$

to ensure allocated stripes fit within the capacity.

Representing VM Bandwidth Constraints. Cloud providers impose per-VM bandwidth constraints on network egress, as described in Section 2.2. As such, a major bottleneck of multicast transfer is the source region's limited egress bandwidth. We constrain P in terms of the ingress and egress limits:

$$\forall v \in V \quad \text{SIZE}_{\text{STRIPE}} * \sum_{s} \sum_{u \in V} P_{s,(v,u)} \tag{5}$$

$$\leq \text{EGRESS}^{VM}{}_{v} * N_v * \text{TIME} \tag{6}$$

$$\forall u \in V \quad \text{SIZE}_{\text{STRIPE}} * \sum_{s} \sum_{v \in V} P_{s,(v,u)} \tag{7}$$

$$\leq \text{INGRESS}^{VM}{}_{u} * N_u * \text{TIME} \tag{8}$$

Representing VM Capacity Constraints. We account for per region VM limits by adding the constraint $N \leq \text{LIMIT}^{VM}$.

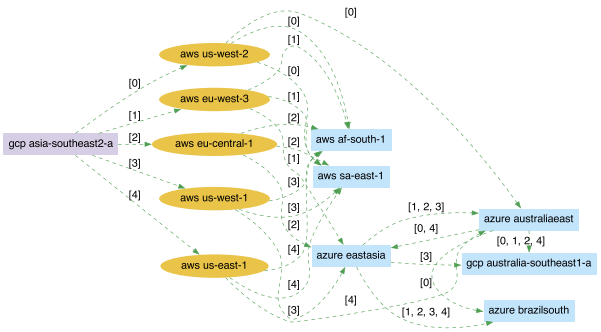


Figure 6: Visualized solver output for inter-cloud replication described in Section 5.1, consisting of source (purple), waypoint (yellow), and destination (blue) regions. The data is divided into 5 stripes (marked on edges).

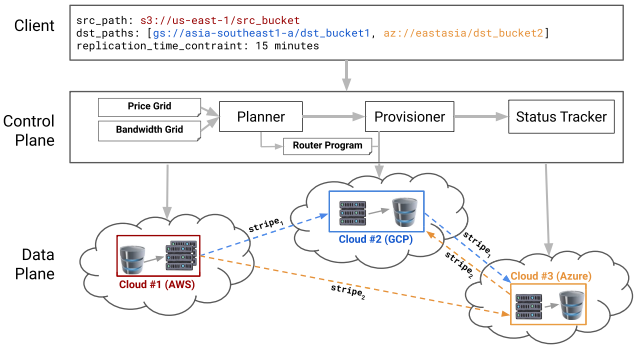


Figure 7: Cloudcast system architecture.

Ensuring Valid Multicast Trees. We use an additional variable F to ensure that the paths selected by P are valid distribution trees, i.e., they are connected and acyclic, and they deliver all data to each destination. At a high level, we ensure that $F_{s,(u,v)} \geq 1$, if $P_{s,(u,v)} = 1$, and impose conservation of flow constraints on F but not P, since P is an indicator variable not a flow variable. We then ensure that flow can be pushed from the source node to destination nodes on F for each stripe, which also ensures that flow can be pushed from the source to destination for the paths selected by P (without having to impose flow conservation on P). We leave details on this part of the formulation for Appendix B due to space.

3.3.4 Solver feasibility

Our formulation so far has a search space of size $O(2^{|V|^2 \times |\text{STRIPES}|})$. With 71 possible regions across GCP, AWS, and Azure and 10 stripes, the search space is, therefore, $O(2^{50410})$, which is infeasible even for advanced solvers to solve within a few minutes, necessitating approximations.

3.4 Reducing Optimizer Runtime

In this section, we describe several mechanisms that we combine to reduce the optimization runtime or an order of seconds, while still maintaining solution quality.

Node Clustering. We observe that many regions across cloud providers share similar characteristics in terms of bandwidth and the costs of their outgoing and incoming paths. A motivating observation was that sub-sampling regions randomly could produce similar solutions with much lower solve time, as shown in Figure 17. At a high level, AWS regions in Europe regions all have similar egress/ingress costs and bandwidth, so only one of those regions needs to be considered as a potential waypoint. Therefore, to reduce the optimizer search space, we cluster regions using their incoming and outgoing path costs and bandwidth as features and select a representative node from each cluster. We empirically find that, with about 20 clusters (i.e. 20 subsampled regions), the optimizer can generate solutions that are reliably similar to the original MILP without approximation (more discussion in Section 5.3.2).

Hop Constraining. To further reduce the optimization space, we only consider a maximum of 2-hop overlay waypoints. Previous research has shown that limited numbers of overlay hops are often sufficient [7, 41, 46]. Our analysis also found solutions using multiple overlay hops to be rare, suggesting that they need not be considered. We implement the hop constraints as an additional constraint on the MILP.

Stripe-iterative Approximation. To make the optimizer runtime linear with respect to the number of stripes (rather than exponential), we design a greedy, stripe-iterative approximation algorithm that solves for one stripe per iteration. We solve for each stripe independently, then update the input graph for the next stripe by reducing the path capacity (CAPACITY^{path}), instance limits, and egress/ingress limits per region (LIMIT^{VM}, EGRESS^{VM}, and INGRESS^{VM}).

3.5 Example Topology

We show an example of the optimizer's output replication tree topology visualized in Figure 6. Due to variability in cloud provider egress pricing and cross-region throughput, our optimizer often finds unexpected solutions, such as routing one stripe (marked `[3]`) from GCP to AWS, AWS to Azure, then back to GCP. Although questionable at first glance, we evaluate this same replication in Figure 10 and demonstrate both cost and replication time improvements over baselines.

4 Architecture of Cloudcast

A key contribution of this work is the design and implementation of the Cloudcast artifact, which provides a practical, performant, and extensible system for studying overlay multicast algorithms in cloud environments. The Cloudcast system simplifies the design and deployment of multicast overlays spanning cloud object stores, and is used to implement the optimizer described in Section 3.

We provide an overview of Cloudcast in Figure 7. Cloudcast is designed with a centralized control plane and a dis-

tributed data plane. The control plane determines the set of overlay nodes and routing paths, and it dispatches and monitors multicast jobs. The data plane consists of *overlay routers*, which we implement as modular software routers running on overlay nodes deployed on cloud VMs. The control plane configures overlay routers using a *router program*, which specifies a graph of modular operators for processing data. Each overlay router gets a unique router program, which, in cooperation with other routers in the system, implements the desired flow of data over the overlay network.

Cloudcast is implemented as part of the Skyplane [28] open source project and consisted of 5K additional lines of Python to implement a design based on overlay routers.

4.1 Control Plane

The control plane contains the planner, which supports pluggable algorithms for determining the placement of overlay routers across cloud providers and paths along which data is replicated (shown in Figure 7). The output of the planner is used to provision VMs to act as overlay routers across cloud regions and to compile a router program for each overlay router that configures its behavior. Finally, the control plane initiates the transfer and monitors its progress.

Planner. The planner is responsible for creating a multicast plan based on a target replication time, source and destination object store paths provided by the user, and profiling data described in Section 3.2. The planner takes as input the algorithm to use for generating a multicast plan, which can be the default Cloudcast optimizer described in Section 3.3 or a custom plan (e.g., a Steiner Tree over the cost graph). The planning algorithm determines how many overlay nodes to create in each region and how each data stripe should be routed through the overlay network. The planner uses the algorithm output to generate a *router program* for each overlay router, which specifies how the overlay router should process a chunk header when received. The Cloudcast default optimizer is implemented using Python's CVXPY library [3] (version 1.3.2) with a Gurobi solver [26].

Provisioner. Once a multicast plan is determined, the provisioner instantiates the overlay routers. The provisioner creates a VPC in each cloud provider and provisions VMs to act as overlay routers within these VPCs. The provisioner also sets firewall rules to allow network traffic between overlay routers, which send and receive data from each other, as specified by the planner-generated router programs. Once a VM has been instantiated, the provisioner installs and launches the router programs as containers on the VMs.

Chunk Dispatching and Status Tracker. The control plane subdivides replication target data into *chunks*, which are at most 64MB in size, to allow for transfer pipelining and parallelism. Each chunk has a *chunk header*, which specifies a key (e.g., object store object, filename), byte range, and an optional multipart ID (required for multipart uploads). The chunk header also contains a *stripe ID*, which specifies which path along the overlay the chunk will take.

The control plane informs each source overlay router (i.e., overlay routers responsible for reading source data) the chunks for which they are responsible by sending the corresponding chunk headers. We refer to this as *registering* a chunk to an overlay router. The control plane's status tracker monitors the status of each chunk by querying the status of chunks on each overlay router.

4.2 Data Plane

The data plane is composed of overlay routers, each running on a single VM. The overlay routers are created and configured by the control plane to execute the transfer according to the multicast plan. Cloudcast supports configurable overlays by defining processing on overlay routers using modular operators, inspired by the design of configurable routers [32].

The router program provided by the control plane specifies a directed acyclic graph (DAG) of *operators* (analogous to elements) and *connections*, all of which run on each overlay router and are used to process incoming chunk headers registered to the overlay router. The DAGs are created at the overlay router's startup time based on the router program, and they allow overlay routers to process chunks without additional coordination with the control plane.

Operators are implemented as a pool of worker processes running processing steps for a chunk, such as reading the chunk from the source object store, relaying the chunk to another overlay router, writing the chunk to a destination object store, or transforming the chunk data (e.g., compression or encryption). Connections pass chunk headers between operators via thread-safe queues, and can be configured to send a chunk header to one or all of multiple downstream operators.

For example, on a source overlay router, chunk registrations from the control plane will provide chunk headers to the first operator in the DAG, which downloads chunk data from an object store. All chunk data is stored in a shared memory filesystem to allow for fast access across operators. Once chunk data is downloaded, the chunk header is passed to the next operator via a connection, which runs LZ4 compression [4]) and secret key encryption [5, 18] on the chunk data. The leaf operators are 'sender' operators, which relay the chunk header and data to other overlay routers.

Chunk data is relayed between overlay routers by a 'sender' operator on the sending router and a 'receiver' operator on the receiving overlay router. When the sender operator is created, it creates parallel TCP connections which are kept open for the duration of the transfer. Before sending chunk data, the sender will attempt to register the corresponding chunk headers with the receiving overlay router to ensure it has space in its shared memory file system to write the chunk data. Once chunks are registered, the sender will send the chunk data over the TCP sockets, and the receiver will wait for the written chunk data size to match the size specified by the chunk header, before

System	Description
Direct	Data is transferred directly from the source to the destination regions.
MDST	Data is transferred along edges selected by a Minimum Directed Spanning Tree (including source and destination regions) computed from network costs.
Steiner Tree	Data is transferred along edges selected by a Steiner tree (including optional waypoint regions) computed from network costs.
SPIDER [23]	Data is transferred according to the plan generated by SPIDER, a system designed for fast bulk replication to multiple destinations.
Skyplane	Skyplane's optimizer is used to select paths for each source-destination pair, which are combined to build the distribution tree.
CloudMPCast [24]	Data is transferred over a set of cost-minimizing edges that meet a minimum bandwidth threshold.
Deadline-aware Inter-DC Multicast [30]	Data is transferred to meet deadlines in the inter-DC context according to [30]. Note that due to scalability issues, we needed to modify the candidate tree generation step to only consider a subset of waypoint regions to achieve tractable runtimes.
AWS S3 Multi-Region Bucket [9]	Vendor product that supports intra-cloud between AWS regions only. We enable Replication Time Control [38].
Bullet [33]	Data is transferred according to the plan generated by Bullet, a high-bandwidth dissemination technique using an overlay mesh.
BitTorrent [49]	Peer-to-peer protocol where peers download data from each other in a decentralized manner.
Cloudcast-Opt (HT)	Data is transferred along the highest throughput (HT) multicast tree generated by our optimizer (tightest time constraint).
Cloudcast-Opt (LC)	Data is transferred along a low cost (LC) multicast tree generated by our optimizer (relatively loose time constraint).

Table 2: All of the systems and variants we evaluate, covering a mix of academic baselines and commercial solutions.

sending chunk headers to the next operator. Successfully sent chunk data is deleted from the shared memory filesystem.

Backpressure. Connections are configured with a maximum size for the underlying queues. If the queue reaches its maximum size, the upstream operator will wait until the queue size decreases sending chunk headers to the connection.

Striping. Registered chunk headers with different stripe IDs are placed in different queues and processed by separate DAGs, so that different stripes can be routed differently.

5 Evaluation

In this section, we evaluate Cloudcast across three metrics: replication cost, replication time, and the optimizer solve time (or simply, runtime). In particular, we show that for intra-cloud and inter-cloud bulk data transfer, Cloudcast is able to achieve up to 61.5% cost improvements under a tight runtime budget when compared to academic, commercial, and open-source baselines. We also show that our approximations to reduce the optimizer solve time (as discussed in Section 3.4) are highly effective by reducing the runtime by, on average, 30.68× for 5-destination replications. To simplify evaluation, we disable compression and encryption in experiments.

The full list of evaluated baselines is shown in Table 2. We note that many algorithms do not determine the number of VMs to use in each region. To present them in the best light possible, we maximize the number of VMs in each region traversed by data, subject to per-region quota limits.

5.1 Comparison to Multicast Algorithms

We compare the replication time and cost of existing multicast algorithms with Cloudcast's optimizer to send 100 GB of data from one source to six destination regions.

Simulation results. Given the above replication scenario, we start by exploring a wide range of algorithmic baselines and

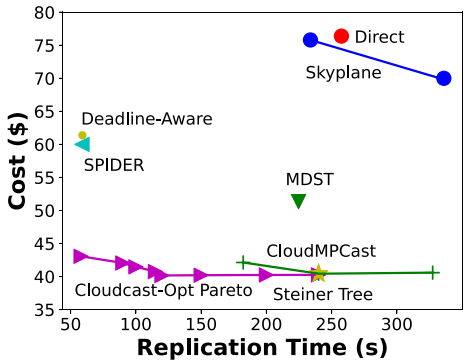
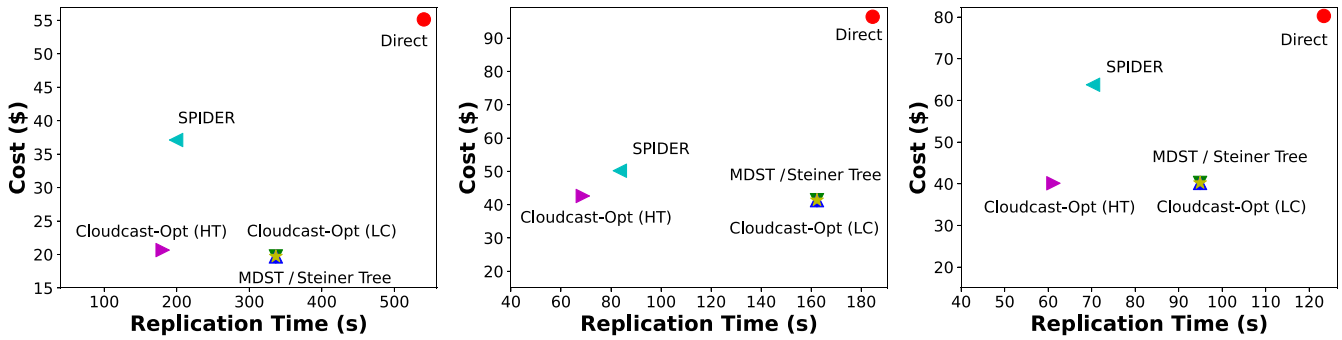


Figure 8: Simulated results for Multicast Algorithms.

Cloudcast parameter settings through simulation. While we tested many configurations through the development of Cloudcast, due to limited space, we present results for a representative configuration[1]. Evaluated systems include Cloudcast-Opt, direct transmission to the destinations, sending along cost-minimizing trees (MDST and Steiner Tree), SPIDER [23], CloudMPCast [24], Skyplane [28], and a deadline-aware inter-DC optimizer [30]. Although Skyplane's optimizer is designed for unicast, not multicast, we adapt the optimizer's solution to multicast by running the optimization for each source-destination pair, and then combining all the graphs to build the distribution tree.

For Skyplane, CloudMPCast, and Cloudcast-Opt, we vary the throughput parameter to evaluate the performance range. For CloudMPCast [24], the optimizer allows for the level of throughput degradation to be controlled by an $\alpha \leq 1$ term, which determines how aggressively edges are filtered out. Our parameter sweep includes $\alpha \in [1, 0.5, 0.1]$, where $\alpha = 1$ maximizes CloudMPCast's throughput. For Skyplane, we vary the target throughput to maximize throughput and minimize cost, and plot both of these points. For Cloudcast-Opt, we show

[1]Simulated Inter-Cloud: from `gcp:asia-southeast1-a` to `azure:eastasia`, `aws:af-south-1`, `azure:brazilsouth`, `aws:sa-east-1`



(a) AWS Intra-Cloud **(b)** Azure Intra-Cloud **(c)** GCP Intra-Cloud

Figure 9: Intra-cloud multicast results for algorithms implemented on Cloudcast.

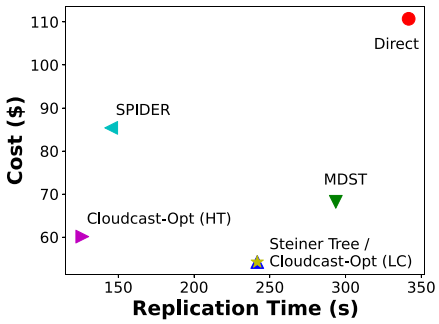


Figure 10: Inter-cloud multicast results for different algorithms implemented on Cloudcast. The Cloudcast replication tree is visualized in Figure 6.

results for several replication time constraints.

In Figure 8, we see that all baselines improve significantly upon direct transmission, and while some can match Cloudcast-Opt's capacity for fast replication time or low cost, no existing baseline can optimize both metrics simultaneously. Rather, Cloudcast-Opt's Pareto-curve can match or beat all baselines on at least one of cost or performance. CloudMPCast, whose α parameter does provide some flexibility, still offers a worse tradeoff than Cloudcast-Opt. Skyplane also has a significantly worse tradeoff curve, as it is not designed for multicast, so does not perform optimizations to alleviate source bottlenecks which are crucial for achieving high throughput. Despite this, even Skyplane's can improve throughput (for the throughput-maximizing solution) and reduce cost (for the cost-minimizing solution) as compared to direct transfers.

Cloud deployments. The remainder of our evaluations present empirical results from real cloud data transfers. Due to the high cost of running data multicast in the cloud ($20–$110 per transfer), we limit our evaluation to four representative configurations and four representative baselines identified by our simulation results. Among the configurations, three are intra-cloud replications corresponding to AWS[2], Azure[3] and GCP[4], and one is an inter-cloud replication workload that covers all three major providers[5]. These configurations are chosen to contain a source region with high egress costs to demonstrate potential cost savings. Among the baselines, we sub-selected the best-performing baselines from our simulation results in terms of throughput (SPIDER) and cost (Steiner Tree), with direct transmission providing a naive baseline.

Figures 9a to 9c show results for AWS, GCP, and Azure intra-cloud replication, and Figure 10 shows inter-cloud results. Across all configurations, given a very tight replication time constraint, Cloudcast-Opt (HT) solution leads to $46-62.4\%$ cost reductions and $2-2.84\times$ replication time speedup compared to sending directly to each destination.

Of the baselines tested, SPIDER [23] consistently demonstrates the lowest replication time, as it did in simulation. However, as SPIDER is not cost-aware, Cloudcast-Opt (HT) can achieve $28.4-44.0\%$ cost savings. Surprisingly, while saving significant cost, Cloudcast-Opt (HT) simultaneously speeds up replication by $1.11-1.35\times$, beating SPIDER on both axes. If, on the other hand, Cloudcast is given a loose replication time budget, i.e., Cloudcast-Opt (LC), it can find the cost-optimal solution in all setups, matching Steiner Tree solutions.

5.2 Cloud Provider and P2P Systems

We run end-to-end evaluation comparing Cloudcast with a commercial baseline (AWS S3 multi-region bucket replication) and P2P systems (BitTorrent and Bullet).

5.2.1 AWS S3 Multi-Region bucket replication

We run an end-to-end comparison between Cloudcast and AWS's S3 multi-region bucket replication [9] for single-provider multicast. AWS supports adding multiple replication rules to a source bucket to specify automatic replication to

[2]AWS Intra-Cloud: from `ap-east-1` to `us-west-1`, `ap-northeast-3`, `eu-north-1`, `ap-south-1`, `ca-central-1`, `ap-northeast-1`

[3]Azure Intra-Cloud: from `brazilsouth` to `westeurope`, `westus`, `koreacentral`, `australiaeast`, `uaenorth`, `centralindia`

[4]GCP Intra-Cloud: from `asia-southeast2-a` to `australia-southeast1-a`, `southamerica-east1-a`, `europe-west4-a`, `europe-west6-a`, `asia-east1-a`, `europe-west2-a`

[5]Inter-Cloud: from `gcp:asia-southeast1-a` to `azure:australiaeast`, `azure:eastasia`, `aws:ap-southeast-2`, `azure:brazilsouth`, `aws:sa-east-1`, `gcp:australia-southeast1-a`

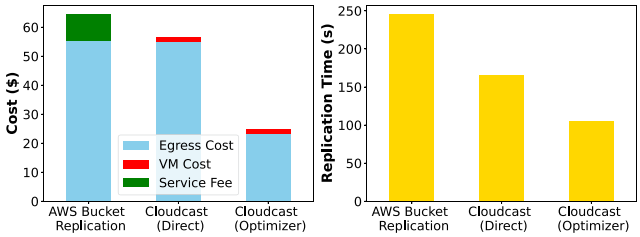


Figure 11: Cloudcast outperforms AWS S3 Replication Time Control while reducing total transfer costs.

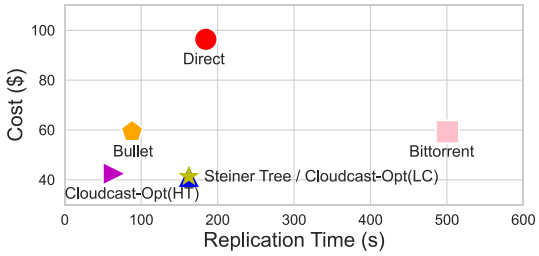
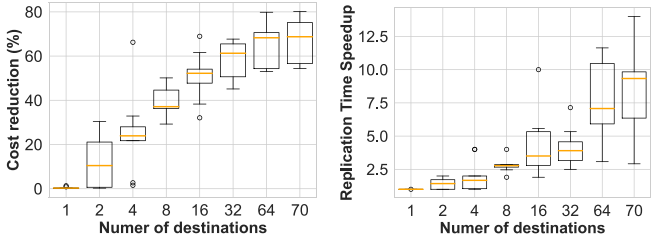


Figure 12: Comparison with BitTorrent protocol on the intra-cloud Azure workload in Figure 9b.



(a) Cost Reduction **(b)** Replication Time Speedup

Figure 13: Cloudcast optimizer's cost and time improvement over direct replication with varying destination numbers.

one or more replication buckets. In the aspect of time control, AWS supports a replication time control with a minimum 15-minute SLO. However, we found that in our experiments, replications typically completed much faster than 15 minutes. Therefore, we use the actual replication time as a point of comparison.

We compare AWS's replication time and cost to Cloudcast with the planner implemented with both direct transfer and the optimizer. We transfer an OPT model [53] with 66 billion parameters (122 GB in total across 9 files) between regions in a single continent[6]. To evaluate AWS replication time and cost, we create buckets with replication rules from a bucket in the source region to buckets in destination regions. Once the replication rules are created, we copy data from a bucket in the same region into the source bucket with 16 VMs. After the write completes, we measure the time until the completion of replication into all destination buckets. We calculate the transfer cost according to AWS's pricing page [39]. We compare AWS multi-region bucket replication to Cloudcast implemented with both the direct and optimizer planner and running. As shown in Figure 11, the direct transfer has the same egress costs as AWS bucket replication, but the VM costs are much less than the service fee charged by AWS for the replication. Overall, Cloudcast with the optimizer is able to achieve 2.3× replication speedup and 61.5% cost savings. This is a result of being able to leverage VM parallelism as well as an overlay network that minimizes total egress costs.

5.2.2 P2P BitTorrent and Bullet

We also compare Cloudcast against P2P systems like BitTorrent and Bullet. We run the same transfer benchmark in Azure in Figure 9b, sending 100GB within Azure to 6 destination regions. We host our own BitTorrent tracker and use aria2 [48] as a BitTorrent client. Since Bullet's implementation is not available, we evaluate Bullet by implementing Bullet's algorithm inside Cloudcast's planner. The result is shown in Figure 12: both BitTorrent and Bullet have lower egress costs than direct but higher than Cloudcast. BitTorrent is the slowest because most clients cannot utilize the full bandwidth. The clients are built for scenarios like background seeding and transfer off the critical path, rather than for bulk data transfer. Interestingly, without a centralized planner, BitTorrent is able to find a low-cost multicast replication tree by inferring the bandwidth among peers and preferring the data from peers who have the highest throughput. However, it is still significantly more expensive than Cloudcast.

[6] from `aws:ap-east-1` to `aws:ap-southeast-2`, `aws:ap-south-1`, `aws:ap-northeast-3`, `aws:ap-northeast-2`, `aws:ap-northeast-1`

5.3 Ablations of Cloudcast's Optimizer

To understand how our optimizer behaves for different selections of source and destination regions and different target replication times, we run simulated ablations.

5.3.1 Varying region selection

We test the generality of our improvements by randomly selecting source and destination regions for varying numbers of destinations. We show aggregated results over 100 samples for different numbers of destinations in Figure 13. Cloudcast is able to improve the runtime and cost of replication consistently across varying numbers of destinations. Cost and throughput improvement increase with more destinations, since more destinations provide a larger optimization space.

5.3.2 Impact of approximations on solutions

We evaluate how the optimizer with and without approximations scales to larger numbers of destinations in Figure 14, by randomly selecting source and destination regions for varying numbers of destination regions. We find that combining all three approximation mechanisms is necessary to scale the optimizer: using no approximations, or only one approximation, takes several minutes for just 10 destinations while using all approximations together reduces solve time to seconds.

We also evaluate how approximations affect the quality of the solution using the monetary cost of the solver-generated

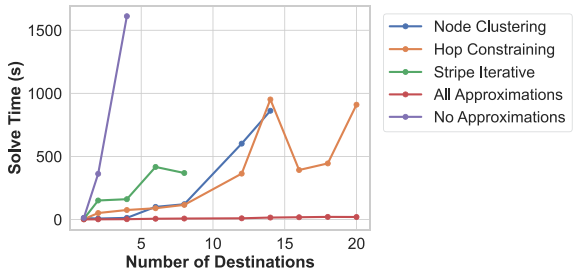


Figure 14: Approximations reduce solver runtime from the cutoff of 30 minutes to seconds for up to 20 destinations.

Method	Mean error	Solver speedup (geomean)
Node Clustering	0.3%	9.04×
Hop Constraining	1.1%	5.72×
Stripe Iterative	0.0%	7.02×
All Approximations	1.1%	30.68×

Table 3: Solve time and solution quality with approximations.

solution. We randomly sample 100 source/destination combinations for 5 destinations and compute the difference in the solution's monetary cost and replication runtime compared to MILP without approximation in Table 3. We find that the difference in cost averages around 1%, and estimate the worst-case approximation ratio to be 1.4. We find that for even just 5 destinations, the approximated solver runs with a geometric-mean speedup of 30.68×.

5.3.3 Accuracy of replication time model

We compare optimizer-modeled throughput and real throughput in Table 4. As transfer size increases, the approximation becomes more accurate. This is because Cloudcast's optimizer, designed for bulk data replication, makes several simplifying assumptions, such as perfectly pipelined stripes. Thus, transient inefficiencies during startup and teardown mean smaller transfers may experience lower throughput than the optimizer expects, but for larger, more expensive transfers, modeled throughput closely matches empirical results.

5.4 When to Use Cloudcast for Multicast?

Cloudcast is designed for bulk multicast replication in the cloud, so should only be used with data sizes are sufficiently large. Since Cloudcast relies on creating VMs in the cloud at transfer initiation time, there is a constant overhead from VM startup time. We calculate the transfer size break-even point (i.e. the minimum data size for using Cloudcast) for varying providers and VM capacity limits (constraining the throughput for the Cloudcast overlay), shown in Figure 15. We approximate the per-destination replication throughput without Cloudcast as equal to the per-VM egress bandwidth limit, ignoring congestion between source and destination VMs. Azure has a higher break-even point than AWS and GCP due to two effects. First, the VM startup time is the highest of all providers (56 seconds). Second, VMs in Azure are not subjected to egress constraints (5 Gbps and 7 Gbps for AWS and GCP, respectively). As a result, the benefits of using Cloudcast's techniques are only realized for larger transfer sizes or larger numbers of destinations.

Transfer Size (GB)	Prediction error
16	16.6%
32	8.51%
64	3.31%
128	1.69%

Table 4: Accuracy of the optimizer's predicted throughput.

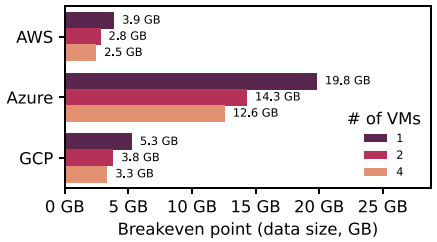


Figure 15: Estimated break-even point for a 6-destination replication based on VM startup times (35, 56, and 34 seconds for AWS, Azure, and GCP, respectively) and VM egress limits.

6 Related Work

Overlay Unicast. A significant body of prior work uses overlay networks to improve the performance and resilience of one-to-one data transfers in the Internet and peer-to-peer networks [7, 12, 33]. In clouds, previous work has also leveraged cloud elasticity to further improve performance [28, 37]. However, they do not consider multicast, and except Skyplane [28], none consider the monetary cost of replication in the cloud. Handling multicast is challenging. For example, while [28] can leverage elastic resources, cloud pricing models, and overlay networking for bulk unicast replication in the cloud, its techniques are not directly applicable to the multicast setting. More specifically, Skyplane's flow-based throughput model results in ambiguous multicast distribution tree solutions as it ignores the identity of data sent along multiple paths. Furthermore, since Skyplane's optimizer is not designed for multicast, it cannot take advantage of techniques such as leveraging multiple distribution trees to alleviate source bottlenecks.

Overlay Multicast. End-system multicast [15] and overlay multicast have been proposed to efficiently disseminate data from a single source to multiple destinations. Many application-level multicast algorithms have been proposed. Algorithms like SPIDER [23], SplitStream [12], Bullet [33], and Overcast [29] are designed for high-bandwidth, cross-internet file distribution with application-level multicast overlays. However, these algorithms ignore monetary costs and focus on techniques to maximize bandwidth.

Inter-DC Replication. Extensive prior work addresses inter-DC replication [21, 22, 34, 44, 54], including bulk multicast [36]. Recent research includes deadline [36] and cost-awareness [20, 21]. However, the cost model in the Inter-DC setting cannot be easily adapted to cloud users, for which network pricing is based on total data volume rather than bandwidth (which introduces a non-linearity for a standard bandwidth-based MILP formulation). Furthermore, existing formulations are not designed for multicast [20, 21] or do not consider more than a few geo-distributed regions [36]. Our work focuses on public clouds, considering unique per-GB network pricing, elastic resources, and cloud-specific resource constraints. Our approximation algorithm is also designed to scale to all regions across multiple cloud vendors.

Traffic Engineering. The classic problem of traffic engineering has also formulated optimization problems for minimizing cost under performance constraints. These techniques have recently been applied to cloud providers and their monetary costs. For example, Entact [55] studied how to optimize costs for online service providers while still minimizing user latency. Similarly, Cascara [45] leveraged latency-equivalent paths to identify cost-minimizing paths for cloud providers. Like Inter-DC Replication, these approaches have been developed from the perspective of the cloud or service provider and, thus, present a materially different optimization problem.

Steiner Trees. The Steiner Tree algorithm has been applied in the multicast setting both to minimize costs in terms of delay [31] and cloud egress costs [24]). CloudMPCast [24] minimizes egress costs in cloud bulk data multicast by constructing a Steiner Tree overlay network that avoids low-throughput cross-region paths. However, CloudMPCast overlooks VM capacity and per-VM egress/ingress limits in its MILP formulation. Also, CloudMPCast aims to achieve *comparable* performance to direct transfers while minimizing cost, unlike Cloudcast, which optimizes throughput.

Geo-Distributed Storage. Geo-distributed storage via data replication is supported by a variety of cloud services, such as AWS Cross-Region Replication [9], AWS Multi-Region Access Points [11], and GCP Multi-region buckets [25]. Cross-region replicated buckets (e.g., S3 replication rules) automatically replicate written data from a bucket in one region to one or more buckets in other regions. However, these services have limited support for cross-cloud data movement and do not minimize egress costs even for intra-cloud data movement. SPANStore [51] designs a system for geo-distributed storage across multiple cloud providers, and also optimizes egress costs of relaying data on PUT requests. However, its relay strategy is optimized for latency, not bandwidth.

Peer-to-peer Multicast. Peer-to-peer systems (P2P) support file sharing among a set of end-user clients. The BitTorrent protocol [17] reduces the network load on the source by allowing clients to upload and download data to each other. BitTorrent is widely used for data multicast in data center environments by Facebook [19] and Twitter [49]. Specialized systems for data multicast that use BitTorrent include Uber's Kraken [40] and Ant Group's Dragonfly [6]. These P2P systems have significant overhead as they are designed for adversarial settings where peers may be unreliable or fail. Moreover, P2P systems must scale to millions of destinations and therefore lack centralized control which prevents custom routing topologies. P2P systems may redundantly send data over expensive links due to a lack of cost awareness.

Method	Multicast	Cloud Pricing	Striping	Resource Elasticity
Unicast overlay networks				
RON [7]	×	×	✓	×
Skyplane [28]	×	✓	✓	✓
COMS [20]	×	×	×	∼
Peer-to-peer				
BitTorrent [17]	✓	×	✓	×
SplitStream [12]	✓	×	✓	×
Bullet [33]	✓	×	✓	×
Inter-DC overlay multicast				
SPIDER [23]	✓	×	✓	×
CodedBulk [47]	✓	×	✓	×
BDS [54]	✓	×	✓	×
Deadline-aware Inter-DC [30]	✓	×	×	✓
Cost optimized overlay networks				
SPANStore [51]	✓	✓	×	×
CloudMPCast [24]	✓	✓	×	×
Jetway [21]	×	✓	✓	×
Cloudcast (ours)	✓	✓	✓	✓

Table 5: Cloudcast builds on prior work by enabling multicast, optimizing cloud costs, and leveraging cloud resource elasticity and multiple distribution trees.

7 Conclusion

In this paper, we explored the problem of cost-optimized cloud multicast by introducing overlay networks of ephemeral VM waypoints that exploit path-specific cloud pricing to significantly reduce cost and improve throughput. We developed a MILP formulation of this problem and introduced approximations that make the solving time feasible for practical applications. Our evaluation against academic and commercial baselines demonstrated up to a 61.5% reduction in cost and a 2.3× improvement in runtime. Cloudcast has been released as part of the Skyplane open source project with pluggable planning algorithms to enable future research in this space.

Acknowledgments

This work was supported by gifts from Accenture, AMD, Anyscale, Google, IBM, Intel, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, SAP, Uber, and VMware. We also thank Asim Biswal for his contributions to the open source artifact, and Daniel Rothchild for providing feedback on writing.

References

[1] Blackblaze. https://www.backblaze.com/b2/cloud-storage-pricing.html. Accessed: 2022-12-08.

[2] Wasabi. https://wasabi.com/. Accessed: 2022-12-08.

[3] Cvxpy: A python library for convex optimization, 2021.

[4] Lz4 - extremely fast compression, 2023.

[5] Pynacl: Python binding to the libsodium library, 2023.

[6] Alibaba. Dragonfly. https://github.com/dragonflyoss/Dragonfly, 2018. Accessed on 12/15/2022.

[7] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 131–145, 2001.

[8] Amazon AWS. Ec2 on-demand instance pricing. https://aws.amazon.com/ec2/pricing/on-demand, 2023.

[9] Aws cross-region replication. https://docs.aws.amazon.com/AmazonS3/latest/userguide/replication.html.

[10] Microsoft Azure. Pricing - bandwidth. https://azure.microsoft.com/en-us/pricing/details/bandwidth/, 2023.

[11] Alex Casalboni. Amazon s3 multi-region access points. https://aws.amazon.com/s3/features/multi-region-access-points/.

[12] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth multicast in cooperative environments. *ACM SIGOPS operating systems review*, 37(5):298–313, 2003.

[13] Sarah Chasins, Alvin Cheung, Natacha Crooks, Ali Ghodsi, Ken Goldberg, Joseph E Gonzalez, Joseph M Hellerstein, Michael I Jordan, Anthony D Joseph, Michael W Mahoney, et al. The sky above the clouds. *arXiv preprint arXiv:2205.07147*, 2022.

[14] Yang Chu, Sanjay Rao, Srinivasan Seshan, and Hui Zhang. Enabling conferencing applications on the internet using an overlay muilticast architecture. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 55–67, 2001.

[15] Yang-hua Chu, Sanjay G Rao, Srinivasan Seshan, and Hui Zhang. A case for end system multicast. *IEEE Journal on selected areas in communications*, 20(8):1456–1471, 2002.

[16] Google Cloud. All networking pricing. https://cloud.google.com/vpc/network-pricing, 2023.

[17] Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72. Berkeley, CA, USA, 2003.

[18] Frank Denis. The sodium cryptography library, Jun 2013.

[19] Facebook uses bittorrent, and they love it. https://torrentfreak.com/facebook-uses-bittorrent-and-they-love-it-100625/. Accessed on 12/15/2022.

[20] Bita Fatemipour, Wei Shi, and Marc St-Hilaire. A cost-effective and multi-source-aware replica migration approach for geo-distributed data centers. In *2022 IEEE Cloud Summit*, pages 17–22. IEEE, 2022.

[21] Yuan Feng, Baochun Li, and Bo Li. Jetway: Minimizing costs on inter-datacenter video traffic. In *Proceedings of the 20th ACM international conference on Multimedia*, pages 259–268, 2012.

[22] Jason Flinn, Xianzheng Dou, Arushi Aggarwal, Alex Boyko, Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, and Fang Zhou. Owl: Scale and flexibility in distribution of hot content. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 1–15, 2022.

[23] Samrat Ganguly, Akhilesh Saxena, Sudeept Bhatnagar, Rauf Izmailov, and Suman Banerjee. Fast replication in content distribution overlays. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 4, pages 2246–2256. IEEE, 2005.

[24] José Luis García-Dorado and Sanjay G Rao. Cost-aware multi data-center bulk transfers in the cloud from a customer-side perspective. *IEEE Transactions on Cloud Computing*, 7(1):34–47, 2015.

[25] Gcp multi-region bucket. https://cloud.google.com/storage/docs/locations#location-mr. Accessed on 12/15/2022.

[26] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023.

[27] Frank K Hwang and Dana S Richards. Steiner tree problems. *Networks*, 22(1):55–89, 1992.

[28] Paras Jain, Sam Kumar, Sarah Wooders, Shishir G Patil, Joseph E Gonzalez, and Ion Stoica. Skyplane: Optimizing transfer cost and throughput using cloud-aware overlays. *arXiv preprint arXiv:2210.07259*, 2022.

[29] John Jannotti, David K Gifford, Kirk L Johnson, M Frans Kaashoek, and James W O'Toole Jr. Overcast: Reliable multicasting with an overlay network. In *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, 2000.

[30] Siqi Ji, Shuhao Liu, and Baochun Li. Deadline-aware scheduling and routing for inter-datacenter multicast transfers. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 124–133, 2018.

[31] Jehn-Ruey Jiang and Szu-Yuan Chen. Constructing multiple steiner trees for software-defined networking multicast. In *Proceedings of the 11th International Conference on Future Internet Technologies*, pages 1–6, 2016.

[32] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[33] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 282–297, 2003.

[34] Nikolaos Laoutaris, Michael Sirivianos, Xiaoyuan Yang, and Pablo Rodriguez. Inter-datacenter bulk transfers with netstitcher. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 74–85, 2011.

[35] Long Luo, Qixuan Jin, Jingzhao Xie, Gang Sun, and Hongfang Yu. Cost-efficient scheduling of multicast transfers with deadline guarantees across edge datacenters. *IEEE Transactions on Services Computing*, 2021.

[36] Long Luo, Yijing Kong, Mohammad Noormohammadpour, Zilong Ye, Gang Sun, Hongfang Yu, and Bo Li. Deadline-aware fast one-to-many bulk transfers over inter-datacenter networks. *IEEE Transactions on Cloud Computing*, 10(1):304–321, 2019.

[37] Miguel Matos, António Sousa, José Pereira, and Rui Oliveira. Clon: Overlay network for clouds. In *Proceedings of the Third Workshop on Dependable Distributed Data Management*, pages 14–17, 2009.

[38] Meeting compliance requirements using s3 replication time control. https://docs.aws.amazon.com/AmazonS3/latest/userguide/replication-time-control.html. Accessed on 12/15/2022.

[39] Overview of data transfer costs for common architectures. https://aws.amazon.com/blogs/architecture/overview-of-data-transfer-costs-for-common-architectures/. Accessed on 12/15/2022.

[40] P2p docker registry capable of distributing tbs of data in seconds. https://github.com/uber/kraken. Accessed on 12/15/2022.

[41] Simon Peter, Umar Javed, Qiao Zhang, Doug Woos, Thomas Anderson, and Arvind Krishnamurthy. One tunnel is (often) enough. *ACM SIGCOMM Computer Communication Review*, 44(4):99–110, 2014.

[42] Matthew Prince and Nitin Rao. Aws's egregious egress. https://blog.cloudflare.com/aws-egregious-egress/, 2021.

[43] Daniel Rehfeldt and Thorsten Koch. Implications, conflicts, and reductions for steiner trees. In Mohit Singh and David P. Williamson, editors, *Integer Programming and Combinatorial Optimization - 22nd International Conference, IPCO 2021, Atlanta, GA, USA, May 19-21, 2021, Proceedings*, volume 12707 of *Lecture Notes in Computer Science*, pages 473–487. Springer, 2021.

[44] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, et al. Ekko: A {Large-Scale} deep learning recommender system with {Low-Latency} model update. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 821–839, 2022.

[45] Rachee Singh, Sharad Agarwal, Matt Calder, and Paramvir Bahl. Cost-effective cloud edge traffic engineering with cascara. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 201–216, 2021.

[46] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 73–86, 2002.

[47] Shih-Hao Tseng, Saksham Agarwal, Rachit Agarwal, Hitesh Ballani, and Ao Tang. Codedbulk: Inter-datacenter bulk transfers using network coding. In *NSDI*, pages 15–28, 2021.

[48] Tatsuhiro Tsujikawa and Nils Maier. aria2 - the ultra fast download utility. https://github.com/aria2/aria2, 2008.

[49] Twitter uses bittorrent for server deployment. https://torrentfreak.com/twitter-uses-bittorrent-for-server-deployment-100210/. Accessed on 12/15/2022.

[50] Marcia Villalba. Amazon s3 replication adds support for multiple destination buckets. https://aws.amazon.com/blogs/aws/new-amazon-s3-replication-adds-support-for-multiple-destination-buckets/, 2020.

[51] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning bmultiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 292–308, 2013.

[52] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. SkyPilot: An Intercloud Broker for Sky Computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI '23)*, April 2023.

[53] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022.

[54] Yuchao Zhang, Junchen Jiang, Ke Xu, Xiaohui Nie, Martin J Reed, Haiyang Wang, Guang Yao, Miao Zhang, and Kai Chen. Bds: A centralized near-optimal overlay network for inter-datacenter data replication. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[55] Zheng Zhang, Ming Zhang, Albert G Greenberg, Y Charlie Hu, Ratul Mahajan, and Blaine Christian. Optimizing cost and performance in online service provider networks. In *NSDI*, pages 33–48, 2010.

A Optimizer Parameters

A.1 Stripe Granularity

The number of stripes is a parameter in the optimizer that determines the unit of data size that the optimizer determines routes. Choosing too small a number of stripes (e.g., 1-4) can result in solution infeasibility, since an individual stripe may be too large to fit along any given link under a replication time constraint. We show the tradeoff in 16 between solution quality (dollar cost) and the number of stripes set of solving a 3-destination topology. Adding more stripes can increase the solver runtime unnecessarily. We use $8-16$ stripes for experiments.

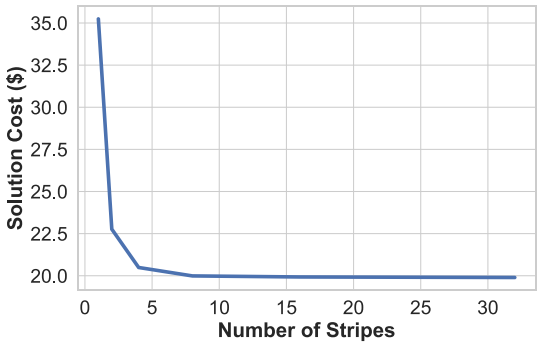


Figure 16: Solution quality v.s the number of stripes. Given a 6-destination intra-AWS transfer job and the same runtime SLO, our optimizer generates different solutions for different numbers of stripes. Small numbers of stripes can result in no feasible solution or solution with worse quality (i.e., higher cost). However, increasing the number of stripes to larger than 10 has diminishing returns.

A.2 Node Sub-Selection

In Figure 3.4, we describe how we cluster nodes to select a subset of nodes for consideration by the optimizer. We motivate this by running an experiment to randomly select a subset of nodes in Figure 17. Generally, there are diminishing returns (beyond 20 nodes) to consider additional nodes. To avoid variability from randomness, we use the techniques described in Figure 17 to select a represented subset of nodes.

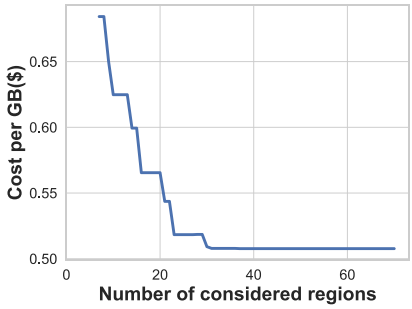


Figure 17: Solution quality versus the number of considered nodes. Considering a larger set of regions has diminishing returns of solution quality but exponentially increases optimizer runtime.

B Formulation Details

B.1 Ensuring Valid Paths

We cannot ensure connectivity of prevent cycles in paths defined by P without adding an exponential number of con-

straints. We define a special flow variable F to add constraints that ensure that flow can be pushed along paths from P from the source to all destinations. The source node is denoted as the source region index in the transfer, the destination nodes are denoted as the destination region indices, and the sink node is denoted as a special node that is only connected to destination nodes. We constrain F to have flow if and only if the corresponding stripe and edge for P is set to 1.

$$F_{s,(u,v)} \geq 1, \text{ if } P_{s,(u,v)} = 1 \tag{9}$$

We ensure zero or negative flow for $P_{s,u,v} = 0$ via capacity constraints. We set special capacity constraints between destination nodes and the sink, to ensure that the sink can only receive sufficient flow if it receives flow from all destinations.

$$F_{s,(u,v)} \leq \begin{cases} 1, & \text{if } u \in \text{DEST}, v = \text{sink} \\ 0, & \text{if } P_{s,(u,v)} = 0 \\ |\text{DEST}|, & \text{otherwise} \end{cases} \tag{10}$$

We impose conservation of flow $\forall s$:

$$\sum_{u \in V} F_{s,(u,v)} = \begin{cases} |\text{DEST}|, & \text{if } v \text{ is the source} \\ -|\text{DEST}|, & \text{if } v \text{ is the sink} \\ 0, & \text{otherwise} \end{cases} \tag{11}$$

If the above constraints are met, this ensures that the stripe paths assigned by P are able to push flow from the source to all destinations.

B.1.1 Full Formulation

We can write a full formulation of an integer linear program as the following:

$$\arg\min_{P,N,F} \text{TIME} * \langle \text{COST}^{VM}, N \rangle + \sum_s \langle \text{COST}^{path}, P \rangle \tag{12}$$

$$N \leq \text{LIMIT}^{VM} \tag{13}$$

$$\text{SIZE}_{\text{STRIPE}} * \sum_s P_{s,(u,v)} \leq \text{CAPACITY}^{path}{}_{u,v} \tag{14}$$

$$\text{SIZE}_{\text{STRIPE}} * \sum_s \sum_{u \in V} P_{s,(v,u)} \leq \text{EGRESS}^{VM} \tag{15}$$

$$\text{SIZE}_{\text{STRIPE}} * \sum_s \sum_{v \in V} P_{s,(v,u)} \leq \text{INGRESS}^{VM} \tag{16}$$

$$F_{s,(u,v)} \geq 1, \text{ if } P_{s,(u,v)} = 1 \tag{17}$$

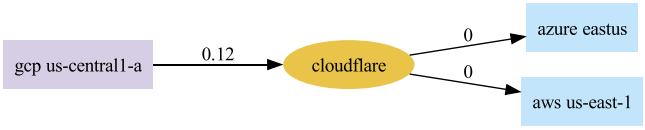
$$\sum_{u \in V} F_{s,(u,v)} = \begin{cases} |\text{DEST}|, & \text{if } v \text{ is the source} \\ -|\text{DEST}|, & \text{if } v \text{ is the sink} \\ 0, & \text{otherwise} \end{cases} \tag{18}$$

$$F_{s,(u,v)} \leq \begin{cases} 1, & \text{if } u \in \text{DEST}, v = \text{sink} \\ 0, & \text{if } P_{s,(u,v)} = 0 \\ |\text{DEST}|, & \text{otherwise} \end{cases} \tag{19}$$

C How does Cheaper Egress Affect Cloudcast's Optimizations?

Some existing cloud providers (e.g., Cloudflare [42], Wasabi [2], and Blackblaze [1]) offer discounted or even free network egress. Interestingly, incorporating free-egress clouds into Cloudcast offers further opportunities to reduce costs. Figure 18a illustrates this effect. This highlights the importance of using a system like Cloudcast, which can adapt replication plans in response to cheaper network offerings.

It is possible that major cloud providers will also adapt free-egress models or, in a less extreme case, make intra-cloud network fees more uniform as they build up additional capacity for inter-region networks with limited bandwidth. In this case, the techniques used by Cloudcast (overlay networking, VM parallelism, and striping) would achieve only substantial throughput improvements but no cost improvement.



(a) Example of routing via no egress fee clouds.

	Direct	Cloudcast w/o Cloudflare	Cloudcast w/ Cloudflare
Cost ($/GB)	0.24	0.2075	0.12

(b) Egress cost is 2× cheaper including Cloudflare in Cloudcast.

Figure 18: Routing data through clouds with no egress fees (e.g., Cloudflare) can reduce inter-cloud replication costs as egress fees need to be paid only *once* rather than a minimum of *twice* to replicate data across AWS, GCP, and Azure.

Understanding Routable PCIe Performance for Composable Infrastructures

Wentao Hou[1], Jie Zhang[2], Zeke Wang[2], and Ming Liu[1]

[1]University of Wisconsin-Madison [2]Zhejiang University

Abstract

Routable PCIe has become the predominant cluster interconnect to build emerging composable infrastructures. Empowered by PCIe non-transparent bridge devices, PCIe transactions can traverse multiple switching domains, enabling a server to elastically integrate a number of remote PCIe devices as local ones. However, it is unclear how to move data or perform communication efficiently over the routable PCIe fabric without understanding its capabilities and limitations.

This paper presents the design and implementation of **rPCIeBench**[1], a software-hardware co-designed benchmarking framework to systematically characterize the routable PCIe fabric. rPCIeBench provides flexible data communication primitives, exposes end-to-end PCIe transaction observability, and enables reconfigurable experiment deployment. Using rPCIeBench, we first analyze the communication characteristics of a routable PCIe path, quantify its performance tax, and compare it with the local PCIe link. We then use it to dissect in-fabric traffic orchestration behaviors and draw three interesting findings: approximate max-min bandwidth partition, fast end-to-end bandwidth synchronization, and interference-free among orthogonal data paths. Finally, we encode gathered characterization insights as traffic orchestration rules and develop an *edge constraints relaxing* algorithm to estimate PCIe flow transmission performance over a shared fabric. We validate its accuracy and demonstrate its potential to provide an optimization guide to design efficient flow schedulers.

1 Introduction

Composable infrastructures–organizing computing, memory, and storage as elastic resource pools–have gained a rising attraction recently. Empowering by emerging cluster interconnects [7, 8, 16], applications running over such a platform can access disaggregated hardware resources natively as local ones, adaptively scale based on workload demands, and achieve fine-grained sharing with co-located tenants, yielding independent scaling capability, high device utilization, and cost-efficiency improvement. We have seen a number of early engineering samples and commodity prototypes, such as GigaIO's FabreX [10], Liqid's SmartStack [13], H3's Falcon [12], Groq's GroqRack [11], and Enfabria's ACF [9].

PCIe (Peripheral Component Interconnect Express) is the defacto interconnect for high-performance intra-host communications. With the introduction of a specialized non-transparent bridge (NTB) device, one can extend the PCIe bus tree and facilitate communications between PCIe devices from different switching domains, enabling inter-host PCIe transactions or *routable PCIe*. Based on this capability, we can interconnect tens to hundreds of PCIe devices using NTB-enabled PCIe switches and adapters, which lays the foundation for many of today's composable infrastructures. More importantly, routable PCIe also serves as the basis for emerging memory fabrics, like CXL [8].

However, our community lacks a systematic understanding of the capabilities and limitations of routable PCIe. Specifically, first, as a routable PCIe fabric introduces extra external hops to PCIe transitions, what are the latency and bandwidth overheads? Second, since the fabric concatenates another PCIe switching domain at the endpoint of the local PCIe bus tree, compared with the intra-host PCIe link, how does the inter-host link behave? How well does it orchestrate co-located flows? Third, the routable PCIe fabric allows concurrent host-device and device-device communications. Since the existing PCIe layered protocol still applies with no changes, when different communication paths interleave, how do they interact with each other? In sum, there is a strong need to characterize the routable PCIe fabric, firmly answer these questions, and derive some design guidelines to assist in building communication sublayers and runtime systems atop routable PCIe-enabled composable infrastructures.

Toward this end, we design and implement a software-hardware co-designed benchmarking framework (called **rPCIeBench**) to help us conduct the characterization study. It consists of three major components: (1) programming APIs that provide various data movement primitives and allow developers to configure arbitrary testing scenarios; (2) host run-

[1]rPCIeBench is available at https://github.com/netlab-wisconsin/rPCIeBench.

time and driver, responsible for both data-plane PCIe transaction delivery as well as control-plane platform management and profiling; (3) FPGA bitstream, realizing the device-side logic and manifesting itself as a reconfigurable target accelerator. Overall, rPCIeBench is generic and device/fabric-independent, enables end-to-end PCIe transaction observability, and allows flexible HW/SW/traffic configurations.

We apply rPCIeBench to GigaIO's FabreX testbed [10] and first examine the performance characteristics of one routable PCIe path. We find routable PCIe indeed incurs performance tax. Its one-way PCIe latency between two endpoints is 868.6 ns, while a local one takes 379.0 ns. The forwarding rate of an external PCIe switch is slower than a server internal one, yielding 30.4% and 6.9% bandwidth degradation for host→device and device→host scenarios. Further, as routable PCIe hinges on the credit-based flow control, the more intermediate hops along a routable PCIe path, the more time it takes to replenish credits, resulting in higher latencies, especially when bandwidth is oversubscribed. Our findings also indicate that an external PCIe link preserves most proprieties of a local one due to the inherent same layered protocol architecture. For example, the bandwidth partition among concurrent PCIe flows over a link depends on the ratio of their outstanding bytes. PCIe is bidirectional, imposing little interference between concurrent reversed flows, regardless of local or external.

We then use rPCIeBench to dissect in-fabric traffic orchestration characteristics and draw three findings. First, in a routable PCIe fabric, each communication port realizes a *credit-by-credit* round-robin scheduling discipline across active lanes, yielding an approximate max-min bandwidth partition. Second, the fabric preserves little buffering at adapters and switches, where the bandwidth availability can be piggybacked via credits and quickly back-propagated from the congestion point to other parts along the path. Third, orthogonal data communication paths over the routable PCIe fabric can be viewed as physically isolated communication domains, imposing little performance interference.

Finally, we formalize the data movement problem over a routable PCIe fabric, encode our empirical findings as traffic orchestration rules, and derive a solution to estimate flow transmission performance. Our *edge constraints relaxing* algorithm takes the underlying fabric topology and PCIe flow properties as inputs, applies iterative reduction by gradually constraining flow bandwidth based on the capacity of oversubscribed links, and outputs the per-flow achieved bandwidth. Our characterization insights make the routable PCIe fabric well-structured and predictable, holding great potential to assist flow scheduling design. We validate the algorithm in three different experimental settings and show that the average performance prediction error rate is 2.9–11.3%.

2 Background

This section provides the necessary background about routable PCIe and the resulting composable infrastructures.

2.1 PCIe Non-Transparent Bridge and Routable PCIe

PCIe [16], introduced in 2003, is an interconnect for communication among processors and peripheral devices. It is a packet-based data communication network and provides point-to-point connections through high-speed serial buses. PCIe is organized into three layers: (a) physical layer, which transmits/accepts packets over a link and performs packet encoding/decoding; (b) data link layer, maintaining data integrity, sequencing packets from the transaction layer, and ensuring reliable delivery via the credit-based flow control protocol [38–40]; (c) transaction layer that realizes different request and completion transaction semantics. Today, PCIe Gen3/4 devices and ecosystems are predominant, Gen5/6 is gaining adoption, and industry standardization of Gen7 is underway and expected to be finalized in 2025.

Generally, a PCIe interconnect network consists of endpoints, switches, bridges, and root complexes, running under one memory domain within a host and supporting the corresponding layer functionalities. A bridge, switch, and root complex forwards and routes packets using memory-mapped I/O (MMIO) addresses or requester IDs. To enable cross-host PCIe communication, a special type of PCIe bridge device–PCIe Non-Transparent Bridge (NTB)–is introduced. A PCIe NTB allows a local host to interact with a remote device via native PCIe transactions by building two memory address mappings: (1) between a remote host and a local NTB: and (2) between an NTB and a local host. As such, one can enable *routable PCIe* traversing through multiple hosts without sharing the same memory domain. To realize scalable deployment, one can integrate a PCIe NTB into an external PCIe switch that interconnects tens of remote PCIe devices. Consequently, these remote PCIe devices will appear in the host PCIe subsystem as a PCIe subtree, laying out the foundation for composable infrastructures. More importantly, routable PCIe has become the basis of emerging memory fabrics (such as CXL [8] and CCIX [7]).

2.2 Composable Infrastructures

Infrastructure composability has gained significant attraction recently because of its independent scaling capability, high device utilization, and improved cost efficiency. By exposing remote accelerators and I/O devices as local, applications can access a large pool of computation/storage resources using native PCIe or other interconnect transactions (without protocol conversion), adaptively scale based on workload demands, and achieve fine-grained sharing with co-located tenants. We have seen a rising number of infrastructure startups delivering a variety of solutions, such as GigaIO's FabreX [10], Liqid's SmartStack [13], H3's Falcon [12], Groq's GroqRack [11], and more. We use the FabreX system as the developing target, and our benchmarking system generally applies to others. Figure 1-a and -b depict our prototyped composable testbed and the architecture of a typical routable PCIe fabric. It encloses (1) a couple of external PCIe switches that realize scalable

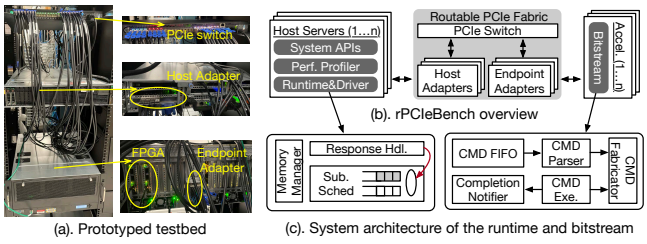


Figure 1: The software architecture of rPCIeBench and its targeted composable hardware testbed.

topologies; (2) host PCIe adapters, offering server-side connectivity; (3) endpoint PCIe adapters, which hold accelerators and I/O devices in standalone chassis. All connections use PCIe copper SFF-8644 cables [18]. There is a fabric manager deployed at one dedicated server, responsible for system management, such as device enumeration, topology configuration, and liveness monitoring.

As discussed above, routable PCIe is the technology enabler to build composable infrastructure. However, our community lacks a systematic understanding and detailed performance characterization of routable PCIe, especially when communicating with composable devices. There are no canonical software utilities, test suites, or referenced hardware platforms. Thus, we fill this gap by developing a benchmarking framework (called **rPCIeBench**). Based on it, we design various experimental composable scenarios, dissect how routable PCIe interacts with remote devices, and analyze its in-fabric traffic characteristics.

3 rPCIeBench Framework

This section first describes the design principles of rPCIeBench, and presents its system design and implementation.

3.1 Design Principles

Our goal is to systematically characterize the performance of routable PCIe and analyze its execution behavior under different composable scenarios. We build rPCIeBench adhering to the following principles:

- **Generality.** rPCIeBench supports any PCIe-based hosts and routable PCIe fabric testbeds, not relying on device-dependent functionalities. We divide the benchmarking functionalities between host servers and target devices;
- **End-to-end operation.** rPCIeBench should capture the communication performance of an entire data movement between the data generator and data consumer. We enable end-to-end tracing and equip a reconfigurable accelerator at the target side to interact with hosts flexibly;
- **Reconfigurability.** rPCIeBench should be able to generate stipulated benchmarking requests based on a traffic profile. We expose a set of programmable APIs, open-source the reference hardware architecture and software implementation, and define pluggable interfaces for module updates.

3.2 Overview

rPCIeBench consists of three components (Figure 1-b), spreading across host servers and remote devices. The first one is programming APIs that allow developers to implement and deploy arbitrary testing scenarios. Users prescribe benchmarking servers and target devices, initialize the system environment, and configure data movement patterns and attributes. The second part is the host runtime and driver, responsible for fabricating and submitting PCIe requests, interacting with the underlying PCIe subsystem and host adapter, handling transaction completions, and conducting performance analyses. The last piece is the bitstream within the FPGA accelerator. An FPGA generally encloses programmable LUTs (lookup tables), DSPs (digital signal processors), domain-specific engines, and heterogeneous memory domains (like block RAMs and HBMs), enabling us to emulate different types of data communications. Specifically, our bitstream sets up the FPGA execution environment, receives data transfer requests, instantiates a series of data transfers via DMA engines over routable PCIe fabric, reads/writes data to memory destinations, and issues completion signals back.

3.3 System APIs

rPCIeBench provides three types of APIs. The first one is used to initialize the execution environment of remote FPGAs, configure the device memory, and set up the host-device address mapping. The second category allows device-side memory management such that one can specify the source and destination of memory locations for a data transfer. The last one offers generic communication primitives, enabling host-device and device-device data movement via the MMIO (memory-mapped I/O) or DMA engine. We equip each primitive with several attributes, such as performing batched communications via a scatter-gather list, enabling flexible load balancing among multiple queues, and more.

3.4 Software Components

rPCIeBench benchmarks and characterizes the routable PCIe fabric using three software subsystems (Figure 1-b).

Performance Profiler. We trace a PCIe transaction's entire lifetime, from when the benchmarking application submits the requests until receiving the completion signals. Our utility timestamps the transaction queueing time at the host server (phase 1), data traversing time over the fabric (phase 2), and command execution at the remote accelerator (phase 3). All timestamps are marked at the nanosecond precision. We use polling to improve the system profiling accuracy. After all stipulated requests are finished, we report (a) the overall bandwidth, queueing, and average/tail latency; (b) the CDF/histogram of each transaction and its individual phases. We follow design strategies (e.g., bitwise recording format, compact data structure, and memory logger) of contemporary perf tools [15, 17] when building the profiler.

Category	API	Description
Device Conf.	`dev_init (pci_bus_addr, bar_addr, size)`	Initialize the FPGA device and map the corresponding BAR address
	`dev_mem_init(pci_bus_addr, region)`	Instantiate the memory manager for the given FPGA memory region
	`dev_setup_mapping(pci_bus_addr, region, hmem_addr)`	Map the FPGA's region to the host and set up the device mapping
Memory Mgt.	`dev_mem_alloc (pci_bus_addr, region, size)`	Allocate the device memory from a given region of an FPGA
	`dev_mem_free (pci_bus_addr, dmem_addr)`	Free the allocated address from an FPGA and clear up the mapping
	`dev_mem_getaddr (pci_bus_addr, hmem|dmem_addr)`	Obtain the memory-mapped host(device) address
Communication	`mmio_rd|wr(pci_bus_addr, hmem_addr, dmem_addr, size)`	Perform an MMIO read/write from the host to a device
	`h2d|d2h(pci_bus_addr, hmem_addr, size, dmem_addr, qnum)`	Move data between the host and the device via a given DMA queue
	`dev2dev_rd|wr(pci_bus_addr, dmem1_addr, size, dmem2_addr, qnum)`	Move data between two FPGA devices via a given DMA queue

Table 1: The rPCIeBench API list. hmem/dmem = Host(Device) memory address. All communication APIs support a batched version.

Runtime & Driver. Our system runtime has three parts: (a) a memory manager that allocates and reclaims host-side memory for data movements; (b) a request submission scheduler, determining the next issuing transaction based on the specified policy; (c) a response handler, which polls the completion vector and wakes up the corresponding submission path. We use the Linux *hugepage* and implement a segment-based memory allocator [23, 24] atop. As shown in Figure 1-c, the scheduler is a multi-queueing system, exposing a programmable interface for users to limit the number of outstanding requests and define the scheduling policy. One can further control the scheduling behavior at a fine granularity for each queue. Besides, our driver layer realizes a slim PCIe subsystem that implements the basic functionalities (such as bus enumeration, device registration, and buffer/engine management) to interact with the device on the control plane (using memory-mapped registers) and data plane (through DMA).

Bitstream. Figure 1-c depicts the circuit diagram of the remote accelerator. It has three 64-bit base address register (BAR) spaces for different roles. BAR0 is used for configuring the DMA engine, and BAR2 enables passing benchmarking parameters. BAR4 is connected with the FPGA's HBM and mapped to the host memory for data movement. One can also use BlockRAM in this case and we present the latency comparison in Appendix B. There are five modules along the command execution pipeline: (a) command FIFO queues, taking user requests via MMIO write, where the host runtime specifies the queue ID; (b) command parser, analyzing the request format, extracting the parameters, and checking the request's validity; (c) command fabricator, which encapsulates PCIe transactions and submits them to the DMA engine; (d) command executor, reading from device-side memory, buffering data temporarily, and issuing PCIe writes to the host memory under host→device communications (device→host works vice versa); (e) completion notifier, writing the completion signal to a predefined memory region. Note that (1) host→device and device→host, albeit exhibiting similar processing paths, use different hardware components; (2) we realize device↔device communications by mapping one FPGA's HBM to the host memory and accessing it via another FPGA's DMA engine, causing data copied from one FPGA to another.

3.5 Command Data Path

rPCIeBench supports three types of communication primitives (Table 1). An MMIO read/write, issued from the host processor, is the first category, generating only one PCIe read/write transaction to access the device memory. The second one is a host-device data movement. As depicted in Figure 2-a/b, it encompasses four steps: (a) passing command arguments via an MMIO write, (b) moving data between host and device, (c) reading/writing to the HBM, and (d) issuing completion signals via another PCIe write, yielding 1 MMIO write and 2 PCIe transfers (which will translate to multiple PCIe transactions based on the command size) in total. The last type is device-device communication (Figure 2-c/d), operating similarly to the host-device case. The difference is that two device memory accesses are triggered at both source and destination. We use a server host to submit requests and catch completion signals. In our implementation, command FIFO queues and data buffer (of the command execution engine) are located in the block RAMs (BRAMs). Under batch execution, the command fabricator within each device (Figure 1-c) formalizes a list of transactions and schedules them concurrently. We trace each primitive between submitting the command and receiving the completion acknowledgment.

3.6 Workflow

Using the rPCIeBench framework requires three basic steps, and we follow them when performing the characterization throughout this paper. First, one should configure the composable testbed based on the experimental data movement flows, considering how host adapters, PCIe switches, and endpoint adapters are connected. The second step is to write profiling applications using our system APIs. This includes determining traffic profiles and benchmarking parameters. Finally, one will deploy the host execution environment, load the bitstream into FPGAs, run the profiling application, and collect performance results.

4 Basic Performance of Routable PCIe

This section examines the performance characteristics of routable PCIe and compares them with the local PCIe case.

4.1 Experimental Methodology

Hardware testbed. Our host servers are 2U Dell R740 boxes, enclosing two 20-core Intel Xeon Gold 6248 processors (running at 2.5GHz), 192GB DRAM, and 1.92TB HDD. We disable both Hyper-Threading and Turbo Boost features. All PCIe lanes of the server are Gen3. We use Xilinx Alveo U55C cards (×16) as the major fabric-attached devices. As discussed above, we choose the GigaIO's Fabrex as the

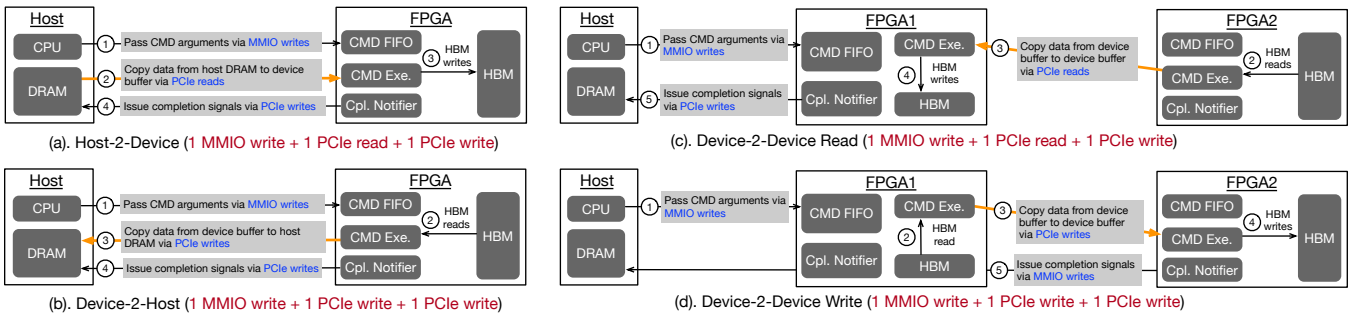
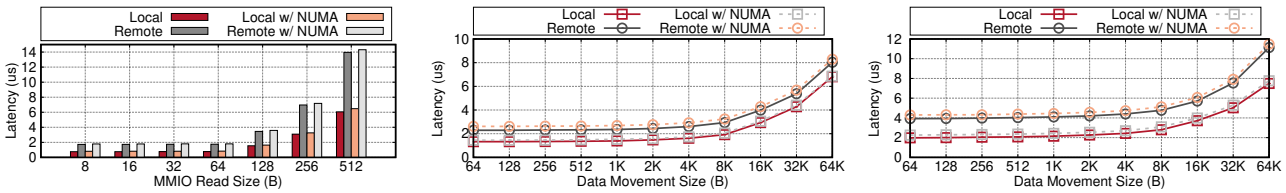


Figure 2: Data path of four communication primitives. We consider the data movement between host DRAM and device HBM.



(a) MMIO. **(b) Device→Host.** **(c) Host→Device.**

Figure 3: Latency of MMIO, Device→Host, and Host→Device communication when varying the data sizes.

routable PCIe-based composable testbed. Its RS4024 switch has 24 ports, where each connects to a PCIe Gen3×4 link.

Terminology. We use a PCIe flow to describe one data transfer from a source entity to a destination entity. Multiple flows can interleave over the same communication path for different data movements. A PCIe transaction layer packet (or packet for short) and a PCIe transaction are used interchangeably, referring to the smallest transmission granularity of a PCIe flow. Our work mainly considers three types of PCIe transactions [16]: memory read, completion with data, and memory write. The first two are non-posted, requiring data responses, while the last one is a simple posted transaction. MPS (maximum payload size) and MRRS (maximum read request size) limit the size of corresponding packets, which are 1024B and 512B in our case.

Experiment configuration. This section focuses on the single communication path. There are three types of communication paths in a composable testbed: host→device (H2D), device→host (D2H), and device→device (D2D). We set up each of them and use the rPCIeBench's communication primitives for traffic generation. We change our traffic profile by varying the number of outstanding PCIe flows, the packet size per flow, and its burstness. rPCIeBench reports average/tail latency and throughput as the major performance metrics.

4.2 Latency

One-way PCIe. We first dissect the one-way PCIe latency between two entities using the rPCIeBench' tracing functionality. When communicating within a server, we find out that the local PCIe one-way latency is 379.0 ns, which matches the number reported in recent literature [1, 2, 56]. However, when traversing across the routable PCIe fabric, the one-way PCIe latency rises to 868.6 ns, adding 489.6 ns (129.2%) overheads! This is non-trivial for small-sized PCIe transfers. We further worked with the device vendor and performed a latency breakdown. We find that (1) the host adapter, switch, and target adapter consume ∼105ns each due to the NTB switching, respectively; (2) the propagation delay of the copper wire is around 5ns; (3) the RS4024 has a 10ns processing delay; (4) the host-side software takes ∼150ns.

DMA-induced PCIe. When PCIe transfers are triggered via DMA, we should include the DMA engine execution cost, including preparing the command, submitting it to the command queue, and catching the completion signal. We examine the hardware module within the accelerator and find out this overhead is around 418.0 ns regardless of local or remote. For example, a 64B PCIe write issued via the DMA engine would take 946.0 ns and 1421.4 ns to complete in the local and remote cases, respectively.

MMIO & H2D & D2H. The latency of an MMIO read depends on the number of generated cache lines. As shown in Figure 3-a, a local 64B PCIe read takes 766.0 ns, while the remote one consumes 1751.0 ns, because one PCIe round trip (two-way) is required. When crossing the CPU socket, we observe there is an additional 67.0 ns and 52.0 ns for the local and remote scenario, contributing 833.0 ns and 1803.0 ns, respectively. When the MMIO read size is 1KB, yielding 16 cache lines, a local PCIe latency rises to 11.9 us, while the remote one increases to 27.8 us.

Both device→host and host→device trigger the same amount of PCIe transactions (Figures 2-a/b). However, in the D2H case, as we overlap the data write and completion acknowledgment, it takes less time to finish. Figures 3-b/c report our results. For example, a 64B D2H data movement consumes 1.3 us, while the H2D takes 2.0 us. When the data size is less than 4KB, routable PCIe adds 69.3% and 91.5%

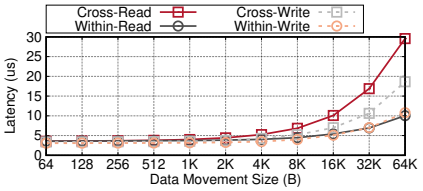


Figure 4: D2D latency when communication within or across a PCIe switch.

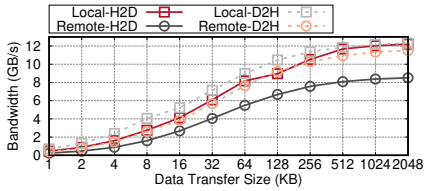


Figure 5: H2D&D2H bandwidth varying the data transfer size.

Figure 6: D2D bandwidth when communicating within or across a PCIe switch.

latency penalties to the D2H and H2D cases due to 3 one-way PCIe. With larger data movement sizes, such overheads diminish considerably. For instance, when performing a 64KB data transfer over the routable PCIe fabric, the D2H/H2D path takes 8.0/11.1 us, adding 17.9%/48.7% (6.8/7.5 us) compared to the local scenario. This emphasizes the importance of batching when building systems over the routable PCIe fabric. Further, NUMA still hurts latency a little bit. On average across all cases, it brings in 10.7%/7.0% overheads for the local H2D/D2H data transfers and adds 7.5%/11.2% more latencies for the remote ones.

D2D. We focus on two types of device-device communication: one is crossing the external PCIe switch; the other is within the PCIe subtree, not across the switch. Clearly, traversing the switch is not free. When the data transfer size is less than 1KB, as shown in Figure 4, crossing the switch incurs 2.2% and 11.0% more latencies for the read and write scenarios, respectively. As the data movement size increases beyond 1KB, we find that the overhead increases significantly. For example, a 64KB data transfer over D2D read/write within the subtree consumes 10.0/10.8 us, but takes 29.6/18.6 us when passing the switch, resulting in 194.4%/72.6% overheads.

Takeaways. Communicating over the routable PCIe fabric (via the switch and adapters) is not as performant as the local case. A one-way PCIe transfer takes 868.6 ns (compared with 379.0 ns in the local case). When using DMA engines for data movements, one should also consider the engine execution cost (which is 418.0 ns in our case). Large data transfer (beyond 4KB) can amortize the routable PCIe-induced latency overheads for H2D and D2H scenarios, suggesting the effectiveness of batching. However, for D2D communication, traversing the external PCIe switch is costly, especially for 4+KB data sizes. This indicates that when building D2D communication subsystems, one should consider not only their positions over the fabric, but also the data transfer granularity.

4.3 Bandwidth

H2D&D2H. We gradually increase the data transfer size and measure the communication bandwidth (Figure 5). Within a server host, H2D and D2H max out their bandwidth with at least 1MB data granularity, achieving 12.2 GB/s and 12.3 GB/s. However, when communicating across the routable PCIe fabric, H2D and D2H drops to 8.4 GB/s and 11.3 GB/s, contributing to 30.4% and 6.9% degradation. We carefully examine each communication entity across the path and find out that the maximum payload size (MPS) and the number of concurrent PCIe transactions are the same in both local and remote cases. This indicates the bandwidth drop mainly comes from the fact the PCIe transaction rate of the external switch is slightly lower than an internal PCIe switch on the server board. The H2D and D2H have different performance degradation because the adapters and switch of our composable testbed use different DMA engines for upstream and downstream links, respectively.

D2D. Next, we present the device→device communication bandwidth. As shown in Figure 6, within a PCIe subtree (not across the remote switch), a read/write D2D transfer achieves 9.8/8.0 GB/s. However, surprisingly, when traversing the switch, the maximum achieved bandwidth is only 2.3/4.3 GB/s! The 4.2/1.8× bandwidth degradation cannot be simply attributed to the additional switching hop across the path (§4.2). By dissecting the data path (Figure 2), we find out that another limitation–root complex contention, happening because all the PCIe transactions (including launching, preparing, and running the command) pass the root of a PCIe bus tree–throttles the number of concurrent cross-switch D2D transfers. However, in the within case, step 3 (Figure 2-c) and steps 3/5 (Figure 2-d) are executed locally within the subtree.

Takeaways. The forwarding rate of an external PCIe switch operates slower than a server internal PCIe switch, yielding 30.4% and 6.9% bandwidth degradation for H2D and D2H scenarios. Device-to-device communications not only traverse the external PCIe switch but might also cause root complex contention (when devices are located in a local-remote hybrid scenario), jeopardizing the maximum achieved bandwidth.

4.4 Latency v.s. Throughput

We examine the latency-throughput relation for each data movement direction. We gradually inject more background traffic (generated via large PCIe transactions) and measure the average latency of 64B PCIe requests. As shown in Figures 7 and 8, the latency starts to rise when approaching the maximum bandwidth because credit starvation happens, causing request stalls. However, we find that it takes more time for a routable PCIe fabric to replenish credits. For example, when achieving 80–90% of the maximum bandwidth, the local H2D and D2H experience 20.2% and 28.7% higher latencies, while the remote ones see 55.3% and 28.1%, respectively. Similarly, within a PCIe subtree, there is an 18.3%/3.9% higher latencies for the D2D read/write case, while the cross fabric case

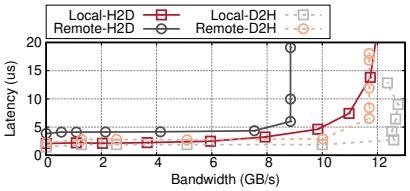


Figure 7: Latency v.s. throughput for local/remote H2D and D2H cases.

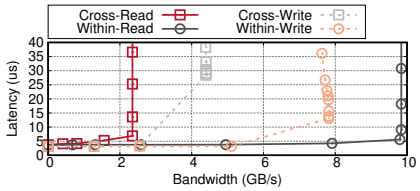


Figure 8: Latency v.s. throughput for within/cross D2D read/write cases.

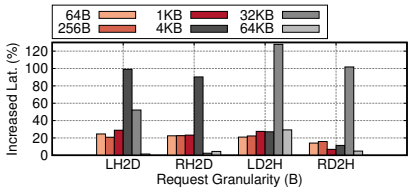
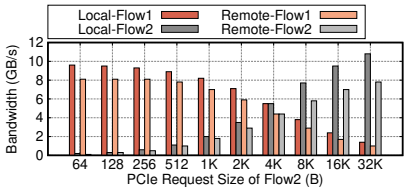
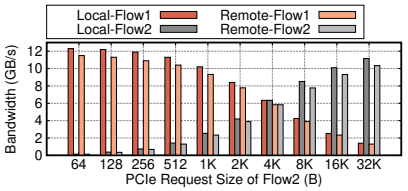


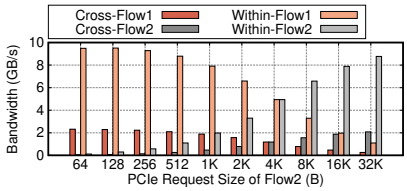
Figure 9: Tail-over-Avg latency increases for local/remote H2D and D2H cases.



(a) Host→Device.



(b) Device→Host.



(c) Device→Device Read.

Figure 10: Bandwidth partition between two concurrent flows of Host→Device, Device→Host, and Host→Device communications.

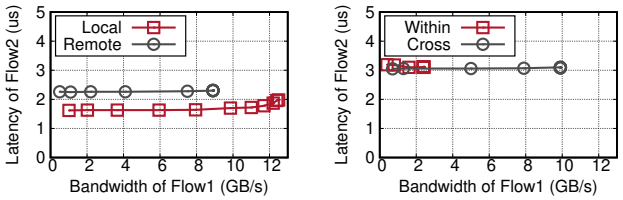
experiences 41.1%/6.7% more. Since the (routable) PCIe fabric applies a hop-by-hop credit-based flow control, the more intermediate entities along the path, the more credit interactions one would observe. When bandwidth is (close to) oversubscribed, a longer communication path needs more credit coordination to deliver a transaction.

Tail latency. Next, we interleave 16 concurrent homogeneous-sized PCIe flows and sweep the request size of each flow from 64B to 64KB. For each data movement direction, when the number of available credits runs out at the link layer, a PCIe transaction would be queued up, increasing the service latency. Hence, we measure the average/P99 latency and use the $\frac{Tail_{lat}}{Avg_{lat}}$ metric to estimate the credit capacity. As shown in the Figure 9, we find that the credit capacity is not consistent for different directions. For example, the H2D experiences the largest ratio under 16 4KB requests, generating up to 512 concurrent transactions, while the D2H direction can sustain 4096 ones (i.e., 16 32KB). This is the same for both local and remote cases, indicating that the routable PCIe fabric has provisioned enough credits (or communication resources) than endpoints. Similarly, D2D reads/writes support 128 and 512 concurrency when staying within and across the fabric, respectively.

Takeaways. Similar to most communication fabrics, one would experience latency rises under high bandwidth utilization. However, the issue stems from the credit-based flow control in the data link layer. It generally takes more time for a routable PCIe fabric to replenish credits because there are more intermediate identities along the path, requiring more credit coordination. The fabric is provisioned with more credits than endpoints, leaving itself from becoming a communication bottleneck from the data link layer perspective.

4.5 Bandwidth Partition

We explore how bandwidth is partitioned across concurrent PCIe flows. Our experiments are configured as follows. For each data movement direction, we consolidate two PCIe flows



(a) Device→Host. **(b) Device→Device.**

Figure 11: Path asymmetry demonstration of Device→Host and Device→Device communications.

that continuously issue one outstanding PCIe request: Flow1 sends a 4KB request; Flow2 increases its transaction size from 64B to 32KB. We find that when concurrent PCIe flows share the same communication path, bandwidth partition among these flows is roughly proportional to the ratio of their outstanding bytes. Take the H2D case as an example (Figure 10-a). When a 4KB flow contends with a 128B one, Flow1 and Flow2 achieve 9.5 GB/s and 0.39 GB/s, respectively, resulting in a 32.8 partition ratio. When two 4KB flows interleave, both sustain at 5.5 GB/s. The remote H2D scenario shows similar results. This observation also holds for the device→host data movement. For example, a 4KB flow achieves 4.3/3.9 GB/s in the local/remote D2H case (Figure 10-b), one-third of the total bandwidth, when intermixing with the 8KB flow. When moving data between two devices, such a bandwidth partition rule still holds. As shown in Figure 10-c, Flow1 only consumes 0.5 GB/s and 1.9 GB/s in the D2D read case when across or within the external PCIe switch, ∼22.0% of the total bandwidth, where Flow2 issues a 16KB request.

Takeaways. Between two endpoints, the bandwidth partition among concurrent PCIe flows mainly depends on the ratio of their outstanding bytes. The defacto transaction layer imposes no fair bandwidth enforcement. The routable PCIe fabric extends the basic scheme of a local PCIe network.

4.6 Asymmetric Communication Path

PCIe is a full-duplex bidirectional network. This section explores whether flows with opposite directions interfere with

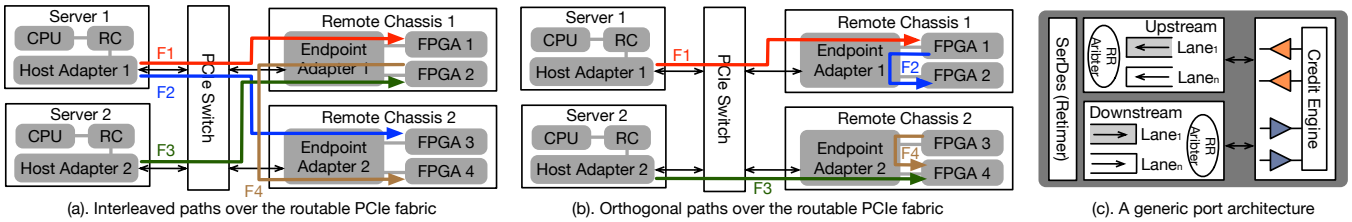


Figure 12: We consider a small deployment with two host servers and two remote chassis, connected via an external PCIe switch. (a) and (b) present in-fabric traffic sharing scenarios. (c) shows the generic architecture of a communication port. RC=Root Complex.

each other. We place a latency-sensitive flow (Flow2) from A to B and a throughput-oriented flow (Flow1) in the reverse direction B→A over one physical communication path, and then analyze how latency varies with the throughput. Figure 11 reports our results. Take the device→host as an example. When maxing out the bandwidth, Flow2's latency only increases from 1.6 us to 1.8 us in the local case, while the remote one stays around 2.2–2.3 us. Similarly, in terms of device-device communications, within a PCIe subtree, Flow2's latency sustains at 3.1 us regardless of how much traffic is injected on the reversed side; across the PCIe switch, Flow2's latency varies between 3.0 us and 3.1 us. Hence, there exists little interference among concurrent flows under opposite directions.

Takeaways. Akin to the local PCIe network, routable PCIe incurs no communication interference among concurrent reverse PCIe flows over one physical path, no matter whether transmitting data is in any of the following directions: host→device, device→host, and device→device.

5 In-Fabric Traffic Orchestration

§4 focuses on understanding different aspects of a single communication path. This section analyzes how multiple paths interact over the routable PCIe fabric, especially at the host adapter, external PCIe switch, and endpoint adapter.

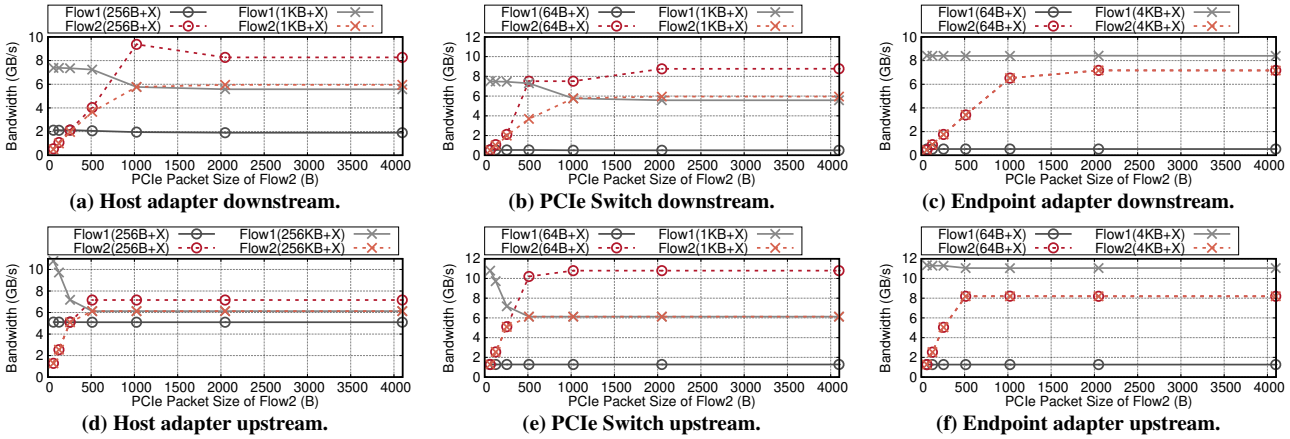
5.1 Max-Min Fair Bandwidth Allocation

Across the fabric, PCIe flows from different communication paths contend for the bandwidth resource of any intermediate transmit points. As shown in Figure 12-a, we configure three path interleaving scenarios that share the host adapter (F1 v.s. F2), switch (F1 v.s. F3), and endpoint adapter (F2 v.s. F4), respectively. In each experiment, we fix the packet size of one flow, gradually increase the packet size of another flow, and explore how bandwidth is partitioned.

Our results show that each communication entity (e.g., an adapter or a switch) realizes an approximate max-min bandwidth allocation scheme. Specifically, when N flows from different paths/lanes share an upstream/downstream port with the following demands $BW_{F_1}, BW_{F_2}, ..., BW_{F_n}$, if the aggregated bandwidth is less than the link capacity, each flow can achieve its desired rate; if the bandwidth is oversubscribed, each flow F_i will receive its max-min share.

For example, as shown in Figure 13-a, when a 256B flow is interleaved with a 4KB one at the downstream path of a host adapter, both max out their bandwidth, resulting in 10.7 GB/s, less than the link capacity. However, in terms of the 1KB and 4KB mixed case, they achieve 7.5 GB/s and 8.5 GB/s when running in a standalone mode, but receive an equal bandwidth share (i.e., 5.6 GB/s). The upstream one presents similar results (Figure 13-d). Regarding the PCIe switch (Figures 13-b/e), when a 64B flow (Flow1) shares with the other one, it can always achieve 0.5/1.3 GB/s along the downstream/upstream path. When Flow1's packet size rises to 1KB, Flow1 sustains at 7.5 GB/s if the packet size of Flow 2 is less than 512B, and drops to 5.6 GB/s, which is the same as Flow2 if the packet size exceeds 1KB. The endpoint adapter behaves similarly. Take the 4KB+X upstream contention as an example (Figures 13-f). Since two flows traverse different communications (one is host→device and the other is device→device), Flow1 and Flow2 achieve 12.6 GB/s and 8.2 GB/s at most if deployed exclusively. When interleaving, Flow2 is able to max out, but Flow1 is limited to 11.3 GB/s due to the link capacity. We also notice that the bandwidth partition at the upstream and downstream points is not always symmetric (Figure 13-c). We believe this is mainly due to the implementation differences across our communication primitives (e.g., the completion delivery step in Figure 2).

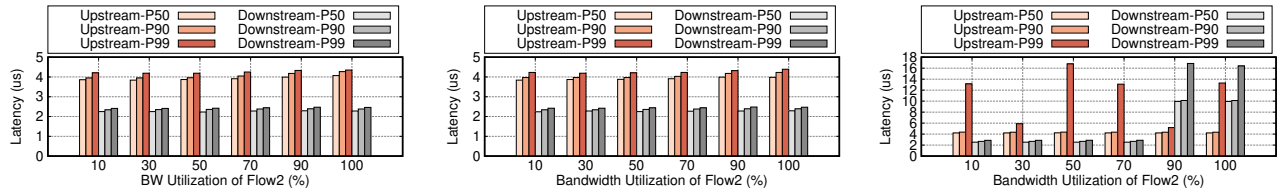
We then drill down to the underlying mechanism to explore how such cross-lane (link) max-min fairness is realized. By walking through the hardware details of the adapter/switch, we find that they all employ a generic port architecture (Figure 12-c), which includes: (a) a SerDes module for data conversion, (b) a upstream and downstream pipeline for packet transmission, and (c) a credit engine to realize the link layer protocol. Some might also include a PCIe retimer to retransmit signals. The reason why max-min fairness across lanes is guaranteed is due to the compounding effect between the credit engine and round-robin arbiter within the pipeline. Specifically, the credit-based flow control enforces an even credit distribution scheme across *active lanes*, whereas the arbiter inside the pipeline schedules each fixed-size PCIe flit in a round-robin fashion. Note that a flit is the basic transmission unit over the PCIe, which is 64B in our case. Therefore, each communication port realizes a *credit-by-credit* (or *flit-by-flit*) round-robin scheduling across all active lanes, resulting in an approximate max-min bandwidth allocation. Even though this is in contrast to the classic *bit-by-bit* round-robin (BR) [25] or *deficit round-robin* (DRR) [53] algorithm, given most PCIe



(a) Host adapter downstream. (b) PCIe Switch downstream. (c) Endpoint adapter downstream.

(d) Host adapter upstream. (e) PCIe Switch upstream. (f) Endpoint adapter upstream.

Figure 13: We report the bandwidth of two PCIe flows contending the upstream/downstream point of the host adapter, PCIe switch, and endpoint adapter, where Flow1 is fixed-size and Flow2 varies from 64B to 4KB. The number of outstanding PCIe transitions is 1.



(a) Host adapter. (b) PCIe Switch. (c) Endpoint adapter.

Figure 14: We report the P50, P90, and P99 latency of Flow1 when varying the bandwidth of Flow2, where two PCIe flows contend the host adapter, PCIe switch, and endpoint adapter, respectively. We consider the upstream and downstream of each scenario.

flows in our context contain a sequence of flits, max-min bandwidth partition is achieved.

Takeaways. In a routable PCIe fabric, any communication port (within a switch or adapter) realizes a *credit-by-credit* round-robin scheduling across different active lanes, resulting in a max-min bandwidth partition. This not only helps us to simplify the performance reasoning under traffic congestion but also assists us in deriving a predictable flow scheduler.

5.2 Fast End-to-End BW Synchronization

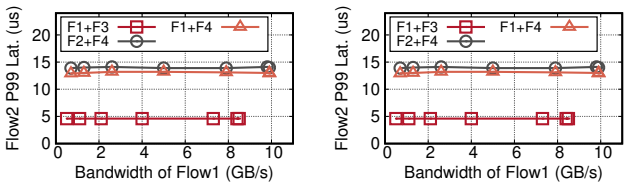
In a shared networking fabric, the link available bandwidth fluctuates considerably with the application behaviors and the underlying topological changes. Such vagaries would cause either traffic congestion (e.g., in-network queue build-up and transmission delay increase) or bandwidth underutilization. In an Ethernet fabric, the congestion control mechanism at the end host will adjust the traffic sending rate accordingly based on stipulated congestion signals. Since the routable PCIe has no such layer, in this section, we'd like to explore how PCIe flow bandwidth is adjusted based on the traffic condition.

We configure three experimental scenarios, where each has two PCIe flows sharing an intermediate communication point from different paths. The first flow is fixed and consumes more than half of the link bandwidth capacity. We then gradually increase the bandwidth utilization of the second flow and measure the P50, P90, and P99 latency of each PCIe transaction of Flow1. We find that the routable PCIe fabric has little queueing effect and the bandwidth demand can quickly propagate from the bottleneck point to upstream entities along the path. Ash shown in Figure 14-a/b, when contending the host adapter or PCIe switch, we observe up to 3.5%/2.3% or 2.6%/2.1% P99 latency increase at the upstream/downstream port. This is mainly because the flow at a congested upstream/downstream port would receive fewer credits than it requires, where such information will be back-propagated to the upstreamed ports until the source host. Since the adapter and switch within the fabric preserve little buffering, the end host could then adjust the flow rate based on how fast the PCIe transactions are delivered to the destination. However, the end host adapter (Figure 14-c) behaves differently, where contention at the upstream/downstream path can use drastic P99 latency increase, more than 10us. This is because our device engine (Figure 1) doesn't implement an auto-pacing module as the host and uses a large command queue inside, yielding significant queueing.

Takeaways. The routable PCIe fabric provides ultra-low latency communication between two endpoints and preserves little buffering at both adapters and switches. The bandwidth availability will be piggybacked over credits, which can be quickly back-propagated from the congestion point to upstream entities until the source node. One can use this as a congestion signal when coordinating concurrent flows.

5.3 Interference-free Orthogonal Paths

Last, we explore how orthogonal communication paths interact with each other over the PCIe fabric since they stay under the same PCIe root complex. As shown in Figure 12-b, we



(a) Downstream. **(b) Upstream.**

Figure 15: Performance interference among orthogonal paths.

enable concurrent data movements across these orthogonal paths (e.g., F1 v.s. F3, F2 v.s. F4, F1 v.s. F4) and explore how latency and bandwidth are affected. Specifically, we increase the bandwidth of PCIe flow1 by increasing the number of outstanding requests and measure the transaction latency of flow2 (which is a 64B flow). As shown in Figures 15, unsurprisingly, orthogonal paths across both upstream and downstream links are completely independent and interference-free.

Takeaways. Orthogonal data paths over the routable PCIe fabric can be viewed as physically isolated communication domains, imposing little performance interference. When reasoning about the fabric performance or designing flow schedulers, one can apply a divide-and-conquer strategy and categorize flows into different isolated domains.

6 Performance Model of the Routable PCIe Fabric: An Optimization Guide

Based on gathered characterization insights, we formalize the data movement problem over a routable PCIe fabric, develop an algorithm to predict the PCIe flow transmission performance, and validate its accuracy in real settings.

6.1 Problem Formalization

We describe a routable PCIe fabric as a directed tree $G = \{N, E\}$, where the host root complex is the root, PCIe endpoints are leaves, and internal/external PCIe switches are branches. Each edge represents a PCIe upstream or downstream link with capacity. The fabric holds a set of active flows $F = \{f_i\}$, where each is described by $f_i = (B_i^{in}, src, dst)$. B_i^{in} is the bandwidth of a flow when running exclusively over the fabric (i.e., standalone BW). *src* and *dst* are the source and destination nodes of a PCIe transfer, which can be a host or PCIe endpoint. We assume there is a unique path between two nodes, which is widely applicable to the PCIe subsystem.

We aim to estimate how much bandwidth a flow is allocated when deploying all the flows concurrently over the routable PCIe fabric. To achieve this, we encode the above characterization insights as the following traffic orchestration rules:

- **Rule 1: Maximum bandwidth bound.** The aggregated bandwidth of co-located flows over an upstream/downstream link should be no larger than the link bandwidth capacity (§4.3);
- **Rule 2: Bandwidth partition of a single link.** Over one PCIe link, the bandwidth partition among concurrent PCIe flows depends on the ratio of their outstanding bytes (§4.5). Besides, there exists no interference between the upstream and downstream direction (§4.6);
- **Rule 3: Approximate max-min fair bandwidth allocation.** Each communication entity guarantees the max-min fairness across active lanes/links due to the *credit-by-credit* round-robin scheduling discipline (§5.1). A PCIe flow can max out its bandwidth when the link is under-utilized and drops to a fair share when oversubscription happens;
- **Rule 4: Isolated communication domains.** There exists no interference among orthogonal paths (§5.3). One can apply it to categorize flows in the first place and then conduct performance analysis hierarchically.

Algorithm 1 Bandwidth Constraints on an Edge

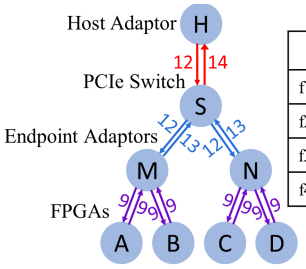
Input: Edge Capacity C, flows $F = \{f_i\}$ and their unconstrained bandwidths $\{B_i^u\}$
Output: The bandwidth constraints of flows $\{B_i^c\}$

1: **if** $\sum B_i^u \le C$ **then**
2: $B_i^c = B_i^u$, for each f_i; ▷ not oversubscribed
3: **else**
4: $n = F.size(); C' = C;$
5: **while** True **do**
6: $m = 0;$
7: **for** each f_i in F **do**
8: **if** $B_i^u \le C'/n$ **then** ▷ less than equal share
9: $B_i^c = B_i^u; C = C - B_i^u;$ ▷ not constrained
10: $F.remove(f_i); m = m + 1;$
11: $n = n - m; C' = C;$
12: **if** m == 0 **then** ▷ all flows exceed equal share
13: break; ▷ must break if oversubscribed
14: **for** each f_i in F **do**
15: $B_i^c = C'/n;$ ▷ equal share on remaining capacity

6.2 Edge Constraints Relaxing Algorithm

We propose a new algorithm (called *Edge Constraints Relaxing*) to solve the problem. The key idea is to apply iterative reduction by gradually constraining flow bandwidth based on the capacity of oversubscribed links. Given the fabric topology and deployed flows as inputs, based on the encoded rules, our algorithm first finds all the oversubscribed edges and their bandwidth constraints, and then updates each flow with its most conservative constraints. Such iterative relaxing allows all flows to converge to one allocation in finite steps where no oversubscribed edge exists. The algorithm requires us to maintain two tables: oversubscribed edges and flow constraints.

Next, we'll describe the algorithm in detail. To begin with, we first initialize all flows with their standalone bandwidths (ALG2 L1). For the oversubscribed edges table, each link is associated with its housed flows (ALG2 L2–L6). Next, for each round, the algorithm traverses each edge and determines if it is under oversubscription or not by comparing the link bandwidth capacity and the aggregated target bandwidth of its housed flows. For all oversubscribed ones, we use the Algorithm 1 based on **Rule 3** to compute the constrained bandwidth of each flow, which is then stored in the flow constraints table (ALG2 L10–L12). After all edges are traversed in this round, flows that have constraints are updated accordingly. The largest constraint of a flow is chosen as its next bandwidth (ALG2 L17). Flows not being captured means they are able to achieve their bandwidth in the current round,



Flows	Standalone
f1: H->C	8 GB/s
f2: B->D	9 GB/s
f3: H->A	7 GB/S
f4: C->D	4 GB/S

	Flows	Current	Constraints	Next
Round 1	f1	8	6, 6	6
	f2	9	6, 5	6
	f3	7	6	6
	f4	4		4

Edge	HS	SN	NC	CN	ND	BM	MS	SM	MA
Capacity	12	12	9	9	9	9	13	12	9
Flow List	f1,f3	f1,f2	f1	f4	f2,f4	f2	f3	f3	f3
Oversubscribed	T	T	F	F	T	F	F	F	F
Constraints	{6,6}	{6,6}			{5, / }				

	Flows	Current	Constraints	Next
Round 2	f1	6		6
	f2	6	5	5
	f3	6		6
	f4	4		4

Edge	HS	SN	ND
Capacity	12	12	9
Flow List	f1,f3	f1,f2	f2,f4
Oversubscribed	F	F	T
Constraints			{5, / }

(a) Testbed Topology and Input Flows **(b) Flow Constraints Table** **(c) Oversubscribed Edges Table**

Figure 16: The workflow of the proposed algorithm (§6.3). (a) shows algorithm inputs, including the topology graph and flows. (b) and (c) depict the flow constraints table and oversubscribed edges table for each execution round. The units of all numbers are GB/s.

Algorithm 2 Edge Constraints Relaxing

Input: Edges $E = \{e_i\}$ and their capacities $\{B_{e_i}\}$, flows $F = \{f_i\}$ and their standalone bandwidths $\{B_i^{in}\}$.
Output: Bandwidth allocation of flows $\{B_i^{out}\}$

1: $B_i = B_i^{in}$, for each f_i in F; ▷ initialize
2: **for** each e_i in E **do**
3: **if** e_i has flows **then**
4: EdgeTable.add(e_i); ▷ initialize Edge Table
5: **for** each f_i in e_i **do**
6: e_i.flow_list.add(f_i);
7: **while** EdgeTable.empty() == False **do**
8: **for** each e_i in E **do**
9: **if** $\sum_{f_j \in e_i} f_j > C_{e_i}$ **then** ▷ an oversubscribed edge
10: $\{B_j^c\} = Algorithm1(C_{e_i}, \{B_j\}), f_j \in e_i$;
11: **for** each $f_j \in e_i$ **do**
12: f_j.constraints.add(B_j^c);
13: **else** ▷ not oversubscribed, delete;
14: EdgeTable.delete(e_i)
15: **for** each f_i in F **do**
16: **if** f_i.constraints.empty() == False **then**
17: $B_i = f_i$.constraints.max(); ▷ flow update
18: $B_i^{out} = B_i$ ▷ loop finishes, output bandwidth

which are the final outputs and will not be updated in later phases. The insight of choosing the most conservative bandwidth constraint is that it can guarantee the flow bandwidth will always be at least its fair share. At the end of the current round, we remove all edges that are no longer oversubscribed from the edge table and use the new bandwidth (ALG2 L14) for the next round to continue the iteration. When there are no more oversubscribed edges, the bandwidths of each flow are the final allocated results (ALG2-L18).

The algorithm is guaranteed to converge because there are only finite edges generating fixed constraints in the system. At any round, every constrained flow will be reduced, indicating that at least one constraint will be eradicated from the list. Thus, the algorithm must converge in finite rounds. Since the number of edges a flow traverses is the maximum number of constraints, the number of steps to converge is bounded by twice the height of the tree. Our algorithm has a $O(N)$ complexity where N is the number of input flows.

6.3 A Walkthrough Example

We now use an example to show how the proposed algorithm works. Figure 16-a shows a PCIe tree topology based on our testbed. Specifically, node H represents the host adapter, node S refers to the external PCIe switch, and nodes M/N are endpoint adapters. Nodes A/B are two FPGAs in chassis 1 and C/D are the other two in chassis 2. The flows and their standalone bandwidths are listed in the right table.

At the beginning, we initialize the flows with standalone bandwidth and fill the flow list in the edge table. Then, we calculate the aggregated bandwidth of each edge and compare it with the link capacity to determine oversubscribed ones. For example, edge HS is an oversubscribed one because its capacity is 12, while the two housed flows f1 and f3 require 8 and 9 transmission bandwidths ($8+9>12$), respectively. Applying the Algorithm 1 to HS, we will obtain a constraint {6,6}, which is inserted into f1 and f3's constraints in the table (Figure 16-b). In the first round, the algorithm decides that edges HS, SN, and ND are oversubscribed ones. As shown in Figure 16-b, HS and SN put 6,6 to the f1's constraints entry, SN and ND insert 6,5 to the f2's constraints, and HS writes 6 to f3's constraints. After finding all constraints, we now use these constrained bandwidths to reduce flows. We update the f1's bandwidth to 6 as its largest constraint is 6. The same logic is applied to f2 and f3. Since f4 has no constraints, as discussed above, it means that f4 can take the original bandwidth as the final one with no bandwidth reduction.

In the second round, only those oversubscribed edges are left in the edge table (Figure 16-c). After the first round, HS and SN are no longer oversubscribed links, except ND. We will then repeat the same procedure to update the flow. The entire process is completed at the end of the second round as all links are not oversubscribed (i.e., the edge table becomes empty). So our final estimated results are: f1, f2, f3, and f4 will achieve 6, 5, 6, 4, respectively.

6.4 Validation and Discussion

We designed three experiments to validate the accuracy of our proposed algorithms. Each experiment targets different oversubscribed links. We use rPCIeBench to figure out the standalone bandwidth of each flow and the link capacity (Table 2-a). Tables 2-b/c/d present the comparison of each experimental scenario (i.e., measured v.s. estimated).

The oversubscribed edge of the first validation experiment is HS, where two downstream PCIe flows from the host fully utilize the bandwidth between the host and switch. The two flows should receive an equal bandwidth share. As shown in Table 2-b, H→C and H→A achieve 5.77 GB/s and 5.43

Edge	Capacity
HS / SH	11.55 / 12.25
SM, SN	15.56
MS, NS	15.46
MA,MB,NC,ND	8.74
AM,BM,CN,DN	11.70

(a) Measured edge capacities.

Flow	Sta.	Est.	Mea.
H->C	8.51	5.78	5.77
H->A	7.21	5.78	5.43
C->B	1.76	1.76	1.70
B->D	7.19	7.19	6.93
A->H	2.54	2.54	2.52

(b) HS is oversubscribed.

Flow	Sta.	Est.	Mea.
H->C	8.56	8.37	7.82
H->A	0.53	0.53	0.47
C->B	1.76	1.76	1.63
B->D	7.19	7.19	7.00
A->H	2.54	2.54	2.53

(c) SN is oversubscribed.

Flow	Sta.	Est.	Mea.
H->C	0.55	0.55	0.49
H->D	8.58	4.37	3.58
C->B	1.76	1.76	1.75
B->D	7.19	4.37	3.59
A->H	2.54	2.54	2.54

(d) ND is oversubscribed.

Table 2: Measured bandwidth v.s. estimated bandwidth for three validation experiments. *Sta.* refers to the standalone bandwidth measured via rPCIeBench. *Est.* means the estimated bandwidth using our Algorithm 2. *Mea.* shows the actual measured bandwidth when all flows are deployed. The unit of all numbers is GB/s.

GB/s, respectively, close to our estimation. SN link is not oversubscribed after H→C is constrained. The average error of our estimation is 2.94%. In the second validation experiment, the oversubscribed edge is SN. Our algorithm suggests that H→C should be reduced for fairness, same as the measured result (Table 2-c). Yet all other flows are affected a little bit. Our modeling indicates that most PCIe flows in this setting have no interactions with each other. But still, the algorithm identifies the most constrained flow (H→C) and delivers a 5.15% estimation error. In the last validation experiment, the oversubscribed point is at edge ND. The computed allocation suggests an equal bandwidth share should happen on the endpoint link while other flows are left unchanged. The actual bandwidth (Table 2-d) is almost the same except the overall link capacity on ND decreases. We suspect this is mainly due to the MMIO contention impact, which bounds the maximum PCIe bandwidth [50]. Because of this, our algorithm is able to predict the right trends, but the estimation error rate is increased to 11.32% due to decreased link capacity.

7 Related Work

PCIe Characterization. People have studied extensively on understanding PCIe for different contexts. Kalia *et al.* [35] explored the interaction between PCIe and RDMA primitives, providing a low-level evaluation and system design guidelines to optimize RDMA-based systems. Researchers [50] proposed a theoretical model of PCIe and developed the `pcie-bench` to systematically measure the host PCIe substrate. NetTLP [37] enhances the observability of PCIe transactions by separating the PCIe transaction layer into a software layer and connecting it to the hardware root complex. Wei *et al.* [56] characterized an off-path SmartNIC when running distributed applications and unearthed the peculiarities of the SmartNIC PCIe subsystem. Unlike these studies that predominantly consider intra-host PCIe, we focus on understanding the performance implications of routable PCIe when holding composable infrastructures.

System Benchmarking. Our study benefits from prior pioneering efforts in developing benchmarking systems for different computing domains, such as single-/multi-core processors [21, 33], domain-specific accelerators [32, 41, 52], cloud applications [26, 36], microservices/serverless functions [28, 54], interconnects [35, 50], storage systems [22, 31, 43, 48, 49], and programmable networking devices [27, 30, 44, 46, 51, 56, 58, 59]. We follow similar design principles when building the rPCIeBench framework: hardware/software open-source across the system stack, end-to-end visibility, elastic modularity for upgrading/replacing sub-components, and parameterized deployments with reconfigurability.

Memory Fabrics. The past few years have seen rising interest from industry [3–8, 14, 19, 20] and academia [29, 34, 42, 45, 47, 55, 57] in developing this new cluster interconnect. Memory fabrics (such as CXL [8] and CCIX [7]), providing the load/store interface, allow tight integration of cross-server computational resources, yielding next-generation system composability. However, under the hood, the memory load/store instructions are carried over a PCIe-like substrate. Therefore, our experimental methodology, performance analyses, and findings would be generally applicable.

Discussion. Compared with an intra-server PCIe switch, the external PCIe one offers higher scalability, allows elastic resource management, and can assign remote endpoint PCIe devices to different server hosts. However, its routing table is still constructed during the bus enumeration phase when booting the server host. Our characterization results and findings (such as max-min bandwidth fairness and fast bandwidth synchronization) are applicable to other routable PCIe testbeds, not only GigaIO Fabrex. Future PCIe Gen5/6 devices would see a latency and throughput improvement.

8 Conclusion

This paper presents **rPCIeBench**, a software-hardware co-designed benchmarking framework to characterize the performance of routable PCIe, the underlying cluster interconnect for building emerging composable infrastructures. Using rPCIeBench, we first examine the performance of one routable PCIe path and then dissect the in-fabric traffic orchestration behaviors. Based on the gathered insights, we develop an edge-constrained relaxing algorithm to accurately predict the communication performance of each PCIe flow over a shared routable PCIe fabric.

Acknowledgement

We would like to thank the anonymous reviewers and our shepherd, Anuj Kalia, for their comments and feedback. This work is supported in part by NSF grants CNS-2106199 and CNS-2212192 and Intel CAD SRS award.

References

[1] A new twist on PCI-Express switching for the datacenter. https://www.nextplatform.com/2019/10/02/a-new-twist-on-pci-express-switching-for-the-datacenter/, 2019.

[2] Pushing PCI-Express fabrics up to the next level. https://www.nextplatform.com/2020/03/27/pushing-pci-express-fabrics-up-to-the-next-level/, 2020.

[3] Micron Compute Express Link™ (CXL™) memory expansion for the next-generation data center. https://www.micron.com/solutions/server/cxl, 2022.

[4] Samsung Electronics Introduces Industry's First 512GB CXL Memory Module. https://news.samsung.com/global/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module, 2022.

[5] SK hynix Develops DDR5 DRAM CXLTM Memory to Expand the CXL Memory Ecosystem. https://news.skhynix.com/sk-hynix-develops-ddr5-dram-cxltm-memory-to-expand-the-cxl-memory-ecosystem/, 2022.

[6] The XMM CXL E3.S from SMART Modular Technologies. https://www.smartm.com/product/xmm-cxl-e3s, 2022.

[7] CCIX Specifications. https://www.ccixconsortium.com/library/specification/, 2023.

[8] CXL Specifications. https://www.computeexpresslink.org/download-the-specification, 2023.

[9] Enfabrica's Accelerated Compute Fabric. https://blog.enfabrica.net/press-release-enfabrica-raises-125-million-series-b-to-fuel-ramp-of-ai-infrastructure-networking-a8a0b21653d2, 2023.

[10] GigaIO's FabreX System. https://gigaio.com/products/fabrex-system-overview/, 2023.

[11] GroqRack Compute Cluster. https://groq.com/wp-content/uploads/2022/10/GroqRackâĎć-Compute-Cluster-Product-Brief-v1.0.pdf, 2023.

[12] H3's Falcon System. https://www.h3platform.com/solution/composable-ai, 2023.

[13] Liqid's SmartStack System. https://www.liqid.com/products/gpu-on-demand, 2023.

[14] Omega Fabric from IntelliProp. https://www.intelliprop.com/products-page, 2023.

[15] OProfile: a Statistical Profiler for Linux Systems. https://man7.org/linux/man-pages/man1/oprofile.1.html, 2023.

[16] PCIe Specifications. https://pcisig.com/specifications/pciexpress/, 2023.

[17] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page, 2023.

[18] Small Form Factor (SFF) Specifications. https://www.snia.org/technology-communities/sff/specifications, 2023.

[19] The Intel® Agilex™ 7 FPGA I-Series Development Kit. https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/agilex/i-series/dev-agi027.html, 2023.

[20] The Leo Memory Accelerator Platform from Astera Labs. https://www.asteralabs.com/products/cxl-memory-platform/, 2023.

[21] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[22] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC'10)*, pages 143–154, 2010.

[23] F. J. Corbató and V. A. Vyssotsky. Introduction and Overview of the Multics System. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, page 185–196, 1965.

[24] Robert C Daley and Jack B Dennis. Virtual memory, processes, and sharing in Multics. *Communications of the ACM*, 11(5):306–312, 1968.

[25] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Symposium Proceedings on Communications Architectures & Protocols*, page 1–12, 1989.

[26] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, page 37–48, 2012.

[27] Alex Forencich, Alex C Snoeren, George Porter, and George Papen. Corundum: An open-source 100-gbps nic. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 38–46, 2020.

[28] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, page 3–18, 2019.

[29] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, 2022.

[30] Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. LogNIC: A High-Level Performance Model for SmartNICs. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*, pages 916–929, 2023.

[31] Zerui Guo, Hua Zhang, Chenxingyu Zhao, Yuebin Bai, Michael Swift, and Ming Liu. LEED: A Low-Power, Fast Persistent Key-Value Store on SmartNIC JBOFs. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM'23)*, pages 1012–1027, 2023.

[32] Johann Hauswald, Michael A Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, et al. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–238, 2015.

[33] J.L. Henning. SPEC CPU2000: measuring CPU performance in the New Millennium. *Computer*, 33(7):28–35, 2000.

[34] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search. In *2023 USENIX Annual Technical Conference (USENIX ATC'23)*, pages 585–600, 2023.

[35] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, 2016.

[36] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, 2016.

[37] Yohei Kuga, Ryo Nakamura, Takeshi Matsuya, and Yuji Sekiya. NetTLP: A Development Platform for PCIe devices in Software Interacting with Hardware . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 141–155, 2020.

[38] HT Kung, Trevor Blackwell, and Alan Chapman. Credit-based flow control for ATM networks: Credit update protocol, adaptive credit allocation and statistical multiplexing. In *Proceedings of the conference on Communications architectures, protocols and applications (SIGCOMM'94)*, pages 101–114, 1994.

[39] HT Kung and Koling Chang. Receiver-oriented adaptive buffer allocation in credit-based flow control for ATM networks. In *Proceedings of INFOCOM'95*, volume 1, pages 239–252, 1995.

[40] HT Kung and Robert Morris. Credit-based flow control for ATM networks. *IEEE Network*, 9(2):40–48, 1995.

[41] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan Tallent, and Kevin Barker. Tartan: Evaluating modern gpu interconnect via a multi-gpu benchmark suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 191–202, 2018.

[42] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–587, 2023.

[43] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator. In *16th USENIX Conference on File and Storage Technologies (FAST'18)*, pages 83–90, 2018.

[44] Ming Liu. *Building Distributed Systems Using Programmable Networks*. University of Washington, 2020.

[45] Ming Liu. Fabric-Centric Computing. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS'23)*, page 118–126, 2023.

[46] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*, pages 318–333. 2019.

[47] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 742–755, 2023.

[48] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM'21)*, pages 106–122, 2021.

[49] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. {eZNS}: An elastic zoned namespace for commodity {ZNS}{SSDs}. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*, pages 461–477, 2023.

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

[51] Yiming Qiu, Qiao Kang, Ming Liu, and Ang Chen. Clara: Performance clarity for SmartNIC offloading. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets'20)*, pages 16–22, 2020.

[52] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. MachSuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 110–119, 2014.

[53] M. Shreedhar and George Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, page 231–242, 1995.

[54] Akshitha Sriraman and Thomas F Wenisch. μ suite: a benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12, 2018.

[55] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. Demystifying cxl memory with genuine cxl-ready systems and devices, 2023.

[56] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing Off-path SmartNIC for Accelerating Distributed Systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 987–1004, 2023.

[57] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. Overcoming the Memory Wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC'23)*, pages 601–617, 2023.

[58] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 34(5):32–41, 2014.

[59] Noa Zilberman, Yury Audzevich, Georgina Kalogeridou, Neelakandan Manihatty-Bojan, Jingyun Zhang, and Andrew Moore. NetFPGA: Rapid prototyping of networking devices in open source. *ACM SIGCOMM Computer Communication Review*, 45(4):363–364, 2015.

Words/Burst	Channel (#)	Throughput (GB/s)	Latency (ns)
1 (32B)	1	6.8	326.3
2 (64B)	1	13.4	330.3
4 (128)	1	13.7	649.5
8 (256B)	1	13.7	1301.9
16 (512B)	1	13.0	1677.9
1 (32B)	32	216.2	326.4
2 (64B)	32	427.4	330.5
4 (128)	32	438.2	648.7
1 (256B)	32	439.5	1297.6
1 (512B)	32	416.1	1671.5

Table 3: Throughput and latency of HBM data read when varying the number of channels.

Granularity	BRAM Latency (ns)	HBM Latency (ns)
8B	627	762
16B	632	763
32B	640	768
64B	644	766
128B	1264	1551
256B	2511	3091
512B	4993	6055
1KB	9992	11961

Table 4: MMIO read latency comparing between BRAM and HBM when varying the request size.

A HBM Performance Characterization

We characterized the latency and throughput of the enclosed HBM of U55C. Table 3 presents our results.

B BlockRAM MMIO Performance

We compared the MMIO latency between BlockRAM (BRAM) and HBM. In this experiment, we configure the BlockRAM as the target device. Table 4 presents our results.

Alea-BFT: Practical Asynchronous Byzantine Fault Tolerance

Diogo S. Antunes, Afonso N. Oliveira, André Breda,
Matheus Guilherme Franco, Henrique Moniz, Rodrigo Rodrigues[*]
Instituto Superior Técnico (ULisboa) and INESC-ID

Abstract

Traditional Byzantine Fault Tolerance (BFT) state machine replication protocols assume a partial synchrony model, leading to a design where a leader replica drives the protocol and is replaced after a timeout. Recently, we witnessed a surge of asynchronous BFT protocols, which use randomization to remove the need for bounds on message delivery times, making them more resilient to adverse network conditions. However, existing research proposals still fall short of gaining practical adoption, plausibly because they are not able to combine good performance with a simple design that can be readily understood and adopted. In this paper, we present Alea-BFT, a simple and highly efficient asynchronous BFT protocol, which is gaining practical adoption, namely in Ethereum distributed validators. Alea-BFT brings the key design insight from classical protocols of concentrating part of the work on a single designated replica and incorporates this principle in a simple two-stage pipelined design, with an efficient broadcast led by the designated replica, followed by an inexpensive binary agreement. The evaluation of our research prototype implementation and two real-world integrations in cryptocurrency ecosystems shows excellent performance, improving on the fastest protocol (Dumbo-NG) in terms of latency and displaying good performance under faults.

1 Introduction

The history of Byzantine fault tolerant (BFT) replication has gone through different stages throughout the years, from the initial exploration of the topic in the 1980s [34] to the start of a series of practical protocols that achieve good performance in the late 1990s [16], and more recently the real-world adoption of this class of protocols in the context of cryptocurrencies and blockchains [55].

BFT protocols must carefully navigate the constraints of the FLP impossibility result [23]. This result states that no deterministic algorithm can guarantee consensus (or, equivalently, agreement on the outcome of a client request within a replicated state machine) in a fully asynchronous system where even a single process might experience a crash failure. For many decades, the almost universally accepted way to circumvent this hurdle was by assuming a partial synchrony model, where the network is assumed to be initially asynchronous but, after an unknown point in time, delivers and processes messages within a certain time bound [21]. This model leads to a class of protocol designs where a leader can drive the execution of the protocol. In this case, after a timeout indicating that the protocol is not making progress, all replicas must cooperate in picking a new leader.

Recently, researchers picked up a different line of research that had been somewhat dormant for many years: asynchronous BFT protocols [9]. These protocols are safe and live irrespectively of any timing assumptions being met, but at the cost of probabilistic guarantees, i.e., they are provided with very high probability. Removing these timing assumptions improves protocol resilience against replica and network delays, which may be due to reasons ranging from network problems to malicious activity [17, 50]. The recent surge of interest in asynchronous BFT came after the publication of a protocol called HoneyBadgerBFT (HBBFT) [40]. Since its publication, several other protocols appeared [20,25–27,29,35,36,54], making tremendous progress in the properties of these protocols, namely their performance and asymptotic complexity.

However, while these proposals succeeded in showing that asynchronous BFT algorithms can perform well, they have yet to gain practical adoption in production systems. In our view, this can largely be due to the fact that existing protocols fall short of striking a virtuous combination of good performance and simple protocol design. The academic research community often overlooks the latter, but it can be a decisive factor in practical adoption. An illustrative example of this point, from the partially synchronous arena, is the work of Istanbul BFT [41] (also known as QBFT [7]). This protocol is widely adopted by the blockchain community [6], to a large extent due to it being simple to understand and implement, and despite it being published more than two decades after PBFT [16] and its many successors.

[*]A. Oliveira is now with Three Sigma. M. Franco is now with ssv.network.

In this paper, we present Alea-BFT, the first protocol for asynchronous BFT state machine replication that brings together top-notch performance – in terms of throughput, latency, and asymptotic complexity – with a simple and elegant design and practical adoption in real-world systems.

The main insight in Alea-BFT is that it selectively brings a key design feature from classical partially synchronous protocols, namely having a per-request designated leader replica that drives the protocol execution for that request. To avoid resorting to timeouts for leader replacement, the choice of leader can constantly rotate among all replicas, as previously done in the crash [39] and Byzantine [53] models. Then, by splitting the request execution into two phases and placing on this replica the responsibility of initiating the broadcast phase to disseminate client requests, Alea-BFT avoids redundant instances of expensive building blocks present in existing asynchronous protocols and also avoids the use of threshold cryptography to encrypt proposals replicated across processes. However, this also introduces challenges, namely that there is no guarantee that the broadcast by the leader will reach a sufficient number of replicas in time for the subsequent agreement phase. We address this challenge by including an agreement phase, pipelined with the broadcast phase, whose goal is to allow replicas to agree on whether it is safe to execute the client request. The execution can proceed if sufficient replicas received the request to reconstruct it. Otherwise, the request is locally stored in one of the queues of pending requests. This leads to a design featuring a novel combination of existing building blocks, namely using VCBC as a broadcast primitive and ABA as the driver for agreement, which are judiciously joined together to provide a simple and performant protocol.

We report on three implementations of Alea-BFT: a research prototype and two real-world implementations, one of them in the context of the SSV Ethereum distributed validator (the key technology behind staking pools), which is currently being considered to replace QBFT as its main consensus protocol in the near future [51], and another in the context of an experimental consensus layer for the subnets of Filecoin [45]. More recently, a second Ethereum distributed validator incorporated Alea-BFT in its protocol roadmap [32].

Our experimental evaluation of these three prototypes shows that Alea-BFT has excellent performance, namely with comparable throughput and better latency than the fastest available asynchronous BFT from the recent literature [24]. This combination of excellent performance, protocol elegance, and real-world adoption makes Alea-BFT a practical solution for asynchronous BFT.

The remainder of the paper is organized as follows. Section 2 surveys related work. Section 3 describes the system model and building blocks. Section 4 presents the design of Alea-BFT, and we optimize it in Section 5. Section 6 analyses its asymptotic complexity. Section 7 sketches a correctness proof. Section 8 describes our various implementations, which are evaluated in Section 9. We conclude in Section 10.

2 Related Work

The Byzantine consensus problem was formulated by Lamport et al. [34], and, over time, accumulated a large body of research in the area [4, 16, 30, 31, 38, 48]. BFT recently gained adoption in cryptocurrencies and blockchains, with several new protocols for those deployments [41, 55].

From these, the protocols that implement a form of consensus – namely state machine replication protocols [49] – face the FLP impossibility of consensus in asynchronous systems [23]. To circumvent this result, most BFT systems rely on timing assumptions such as partial synchrony [21] for liveness. This is the case, for instance, of systems such as PBFT [16] and also more recent proposals such as HotStuff [55], Kauri [44] or ISS [52]. Partially synchronous protocols can, however, be sensitive to conditions like a primary that deliberately slows down the system [17] or situations where replicas are correct but the network is unreliable [50].

As an alternative to assuming partial synchrony, randomized protocols circumvent FLP by guaranteeing the liveness property with high probability. The design for this class of protocols runs the main algorithm through multiple rounds until its nondeterministic nature allows the probability of not having liveness to be irrelevant. These protocols can then operate over a fully asynchronous model, eliminating the need for timing assumptions.

Existing asynchronous BFT protocols do not simultaneously achieve the goals of simplicity and performance, which are key for practicality. In particular, the initial asynchronous BFT protocols [10, 11, 14, 42, 47] are very elegant (sometimes described in less than 10 lines of pseudocode [11]) but have high communication costs and expected termination time. More recently, several new randomized protocols appeared. At the core of this new line of proposals is an asynchronous binary agreement (ABA) primitive, in which processes decide on the value of a single bit. These ABA protocols are then used as building blocks for atomic broadcast and state machine replication solutions. After a small set of initial proposals, namely HoneyBadgerBFT (HBBFT) [40], BEAT [20], EPIC [35], and Dumbo [27], a large number of proposals emerged over the last few years [25, 26, 29, 36, 54]. Given the relatively large literature, we only describe in detail two of these proposals, namely the pioneering work of HBBFT [40] and a recent proposal with excellent performance named Dumbo-NG [24].

HBBFT [40] is based on the observation that atomic broadcast can be built on top of an asynchronous common subset (ACS) framework by combining it with a threshold encryption scheme. In ACS, every party proposes an input value and outputs a common vector containing the inputs of at least $N - f$ distinct parties. HBBFT constructs ACS from the composition of two phases: reliable broadcast (RBC) and asynchronous binary agreement (ABA). During the broadcast phase, every replica starts an RBC instance to disseminate its proposal to all other replicas. Then, in the agreement phase,

N parallel ABA instances are invoked to decide on an N-bit vector, where the i-th value indicates whether or not to include the proposal from replica P_i in the final ACS output. Here, threshold encryption prevents an adversary from selectively censoring requests by selecting which proposals to include in the ACS output vector.

To our knowledge, the best performing and state-of-the-art proposal in this area (outperforming its competitors by several-fold) is Dumbo-NG [24]. This protocol decouples a continuously running broadcast phase from a sequence of multi-valued Byzantine agreement (MVBA) instances. The broadcast phase uses a custom protocol that resembles VCBC (see Section 3), whereas the MVBA phase reuses an existing protocol, whose validity predicate is fine-tuned to check for valid threshold signatures and other protocol-specific conditions. The presence of an MVBA protocol introduces an $O(n^3)$ message complexity, which contrasts with Alea-BFT's use of a round-robin ABA, with only $O(n^2)$ complexity.

Generally, we can categorize previous proposals as either suffering from high communication costs (pre-HBBFT protocols) or having a more complex design that hinders practical adoption (new generation, starting from HBBFT). In contrast, Alea-BFT brings together a simple and elegant design with excellent performance and is now being adopted in real-world systems, namely Ethereum distributed validators. This might be in part due to the simplicity of the protocol and its components – for instance, while Dumbo-NG uses an MVBA, which is complex in both the provided guarantees and its implementation, Alea-BFT leverages a much simpler ABA primitive, resulting in an overall protocol that is easier to understand and implement. Furthermore, Alea-BFT improves on most prior asynchronous protocols through its near quadratic message complexity. Note that while quadratic protocols have been theoretically proposed [5], we do not know of any protocol with such characteristics that was implemented.

3 Basics

In this section, we present the system model and precisely define the basic blocks upon which Alea-BFT is built.

3.1 System model

We consider a distributed system composed of N processes, also called replicas, uniquely identified from the set $S = \{P_0, ..., P_{N-1}\}$ and an arbitrary number of clients.

We assume a Byzantine failure model where up to $f = \lfloor \frac{N-1}{3} \rfloor$ replica processes can fail arbitrarily during the execution of the protocol. The remaining processes follow the protocol specification and are termed correct. Alea-BFT is adaptively secure against an adversary that dynamically determines the replicas to compromise. That said, it reuses two classes of protocols described later in this section, which can have either statically secure or adaptively secure instantiations. As such, choosing a statically secure subprotocol would downgrade the solution to be statically secure.

The system is asynchronous, with the message delivery schedule under adversarial control, and without bounds on communication delays or processing times. Processes are fully connected by channels, providing guarantees that messages are not modified in transit and are eventually delivered. In practice, this requires message retransmission and point-to-point authentication, but by considering this network model, we can omit these from the protocol description.

Lastly, the adversary is assumed to be computationally bound and thus unable to subvert cryptographic primitives.

3.2 Specification

We specify Alea-BFT as an atomic broadcast protocol, which is a common abstraction for implementing state machine replication. Intuitively, this allows a process (e.g., a proxy replica) to broadcast a message (e.g., a client request[1] to be executed on the state machine) to all processes, ensuring that all processes deliver all messages in the same order (executing all client requests in the same order and therefore transitioning through the same sequence of states). Formally, atomic broadcast is defined as follows (with the standard assumption that messages include a per-sender id and sequence number to make them unique) [28]:

- *Validity.* If a correct process broadcasts a message m, then some correct process eventually delivers m.
- *Agreement.* If any correct process delivers a message m, then every correct process delivers m.
- *Integrity.* A message m appears at most once in the delivery sequence of any correct process.
- *Total order.* If two correct processes deliver messages m and m', then both deliver m and m' in the same order.

3.3 Building blocks

Alea-BFT is designed in a modular way by reusing several subprotocols to carry out certain tasks. In this modular architecture, upper-level protocols provide inputs and receive outputs from subprotocols at the lower layers. Next, we present the precise specification of these underlying primitives.

3.3.1 Verifiable Consistent Broadcast Protocol

Verifiable consistent broadcast (VCBC) is a broadcast variant that was first proposed by Cachin et al. [14]. It can only guarantee that all correct replica processes deliver the broadcast value if the sender is correct; however, it always ensures that no two correct processes deliver conflicting messages. Additionally, it allows any party P_i that has delivered message m to inform another party P_j about the outcome of the broadcast execution, allowing it to deliver m immediately and terminate the corresponding VCBC instance. More formally, a VCBC protocol ensures the following properties [14]:

- **Validity:** If a correct sender broadcasts m, then all correct parties eventually deliver m.
- **Consistency:** If a correct party delivers m and another correct party delivers m', then $m = m'$.

[1] Client requests are also referred to in the literature as state machine commands. Throughout the paper, we will use only the term *request*.

- **Integrity:** Every correct party delivers at most one message. Additionally, if the sender is correct, then it previously broadcast the message.
- **Verifiability:** If a correct party delivers a message m, then it can produce a single protocol message M that it may send to other parties such that any correct party that receives M can safely deliver m.
- **Succinctness:** The size of the proof σ carried by M is independent of the length of m.

In Alea-BFT, we use a VCBC implementation consisting of extending an echo broadcast protocol [14] with threshold signatures to generate the proof σ. In short, the protocol consists of the distinguished sender process sending m to all processes and collecting a Byzantine quorum of $\lceil \frac{n+f+1}{2} \rceil$ signature shares in the replies, allowing the sender to combine these shares and convey the signature to all processes in the final message step. Using threshold signatures keeps the message size constant, ensuring succinctness. The message complexity of the VCBC protocol we use is $O(N)$ and its communication complexity is $O(N(|m|+\lambda))$, assuming the size of a threshold signature and share is at most λ bits.

3.3.2 Asynchronous Binary Agreement

An asynchronous binary agreement (ABA) protocol allows correct processes to agree on the value of a single bit. Each process P_i proposes a binary value $b_i \in \{0,1\}$ and decides a common value b from the set of proposals by correct processes. Formally, a binary agreement protocol can be defined by the following properties:

- **Agreement:** If any correct process decides b and another correct process decides b', then $b = b'$.
- **Termination:** Every correct process eventually decides.
- **Validity:** If all correct processes propose b, then any correct process that decides must decide b.

Given the FLP theorem [23], no deterministic algorithm can satisfy all the previous properties in the asynchronous model of Alea-BFT. As such, we use a randomized solution with the following termination property:

- **Termination:** The probability that a correct process is undecided after r rounds approaches 0 as r approaches ∞.

This way, even though the number of rounds required to reach agreement is unbounded, the probability that the protocol does not terminate converges to zero.

We instantiate this primitive via the Cobalt ABA [37] protocol, a modified version of the proposal by Mostéfaoui et al. [43]. The protocol relies on a common source of randomness, i.e., a "common coin", realized from a threshold signature scheme by signing a unique bit string corresponding to the name of the coin and combining the signature shares to generate a random seed [15]. The protocol proceeds in rounds, each consisting of the following all-to-all message communication steps: INIT, conveying the most recent proposal (0 or 1) of each process, followed by AUX and CONF, trying to confirm the existence of strong support (i.e., a Byzantine quorum) in the previous step for a value. At the end of these exchanges, processes either decide a value (if that support was gathered) and convey it through a FINISH message or otherwise move to the next round, changing their proposal to the value of the common coin whenever both final outcomes are considered possible. This protocol provides optimal resilience, $O(N^2)$ expected message complexity, $O(\lambda N^2)$ expected communication complexity and terminates in $O(1)$ expected time.

4 Alea-BFT

This section presents Alea-BFT, starting with a broad overview, followed by a detailed description and pseudocode.

4.1 Overview

One of the central insights of Alea-BFT is to have a single replica propose a value per consensus instance, similar to what happens in leader-based protocols in the partially synchronous model, and all others agree on whether to deliver it or not. Departing from a design where all replicas try to insert each client request in the total order enables us to remove an all-to-all communication phase and only have a single ABA execution per client request (or batch of requests). This insight then leads to the following initial design.

Strawman Proposal. The first design consists of adapting the ACS construction of HBBFT but, instead of having all replicas simultaneously propose values, a single replica is selected as the proposer for each consensus round. The role of the proposer is to choose a value (or batch of values) from its buffer of pending requests to serve as a proposal and broadcast it to all replicas, using a broadcast primitive that ensures that all replicas receive the same value – if they output a value at all – a property ensured by consistent broadcast [13]. Correct replicas would then proceed to execute a *single* ABA to determine whether to deliver the proposed value for that round (if enough replicas have received it to ensure it persists despite faults or asynchrony) or not deliver anything. Additionally, the proposer is deterministically rotated upon every ABA execution to address the scenario where the proposer is faulty without introducing a fail-over sub-protocol, similar to what happens in other protocols for the partially synchronous model (both with crash [39] and Byzantine [53] faults) that incorporate leader rotation into the normal operation, such that it is constantly changing.

This strawman protocol, however, raises an immediate problem. In previous protocols based on an ACS framework, replicas are guaranteed to receive proposals from at least $N-f$ correct replicas. Therefore, they can wait until this threshold is met before deciding which values to input for the subsequent agreement stage. In contrast, in our strawman protocol, only a single replica takes the role of the proposer at any given time, so there is no way to determine whether the current proposer is faulty or not, thus making it difficult to decide which value to input into the ABA without resorting to some timeout, which contradicts the asynchronous model.

Final design. The impossibility of waiting for some threshold to be met before deciding the value to input to the ABA stage

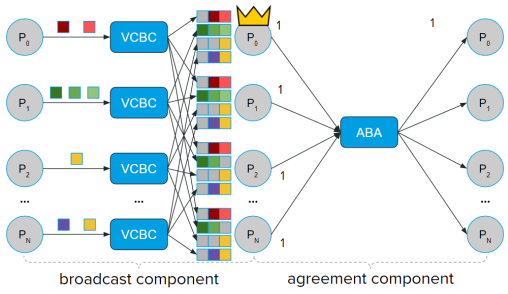


Figure 1: Overview of Alea-BFT. Requests go through a single broadcast primitive (VCBC), are inserted in a priority queue at each replica, determining the final ABA input.

leads us to the insight of not waiting at all and instead allowing undelivered proposals to exist, which are then carried over across rounds. In other words, every time a particular replica is reelected as the proposer, the corresponding ABA execution will decide over its queue of pending proposals instead of a single newly proposed value (or batch of values). This way, replicas can submit their input to start the ABA for a new round as soon as they conclude the previous round, since even if the decision is 0 (i.e., not deliver any proposal in the round), the same proposal will be eventually revisited when the same replica becomes the leader and a larger threshold of replicas become aware of the proposal, guaranteeing convergence to an ABA decision of 1 over time. The ABA execution also serves as a synchronization mechanism between replicas since no replica can progress to a round until it concludes all ABA instances for previous rounds.

In Alea-BFT, we leverage this idea to decompose the monolithic architecture of previous ACS-based protocols, in which a binary agreement instance actively waits for the corresponding broadcast to terminate, into a two-stage pipeline, where the results of the first phase (broadcast component) are queued to be eventually processed, either by the current or by a subsequent execution of the second phase (agreement component). Very importantly for performance, these two phases are executed in parallel, allowing for efficient pipelining.

Figure 1 depicts the resulting overall protocol flow. It starts with the broadcast component of the Alea-BFT pipeline, where replicas receive client requests, (optionally) batch them by storing these in a pending buffer of size B, and, when the buffer is full, disseminate its contents via a VCBC primitive tagged with an incremental sequence number s. The output of VCBC at each replica is stored in a buffer and only removed upon a decision of 1 in the subsequent phase. The broadcast stage produces an instance of an ordered queue of undelivered proposals at each replica. These instances are then used as input to the next component of the pipeline. Note that every replica maintains N queues of undelivered proposals, one for each replica in the system, and these grow and shrink over time depending on how efficiently the agreement component can process them.

The next stage is the agreement component, which iteratively selects one of the queues and decides whether to deliver the oldest proposal. To this end, replicas participate in a single ABA execution, voting 1 if their queue contains this proposal or 0 otherwise. If the decision is 1, then a sufficient threshold of correct replicas are aware of the proposal and may safely deliver it, as explained next. Otherwise, if a decision is 0, the agreement component simply moves on to the next queue, repeating the same process.

As mentioned, since the broadcast's VCBC primitive may terminate at different times in different processes, we need to address the scenario where a correct process outputs an ABA decision of 1, but does not yet know the corresponding proposal. In this scenario, such a correct process requests the missing proposal from the other processes that voted 1. This recovery mechanism is guaranteed to work for the following reasons. Since ABA decided 1, at least one correct process voted for 1. Therefore, this process has the required VCBC proof (as guaranteed by VCBC's verifiability property) and can forward it to the requesting process.

4.2 Detailed description

Processes in Alea-BFT maintain two state variables shared between the two components of the pipeline: variable S_i, consisting of the set of all messages delivered by the protocol, which is initialized as empty upon a call to the START procedure, and updated during the execution of the agreement component; and variable $\texttt{queues}_i$, comprising an array of N priority queues, each corresponding to a distinct replica $P_x, \forall x \in \{0,...,N-1\}$. Algorithm 1 is responsible for initializing the shared state variables and starting the pipeline components upon a call to the START procedure. In the remainder of this section, we begin by specifying the data structure of the priority queues and then describe the two components of the pipeline in turn.

Algorithm 1 Alea-BFT - Initialization (at P_i)

```
1:  constants:
2:      N
3:      f
4:  state variables:
5:      S_i ← ∅
6:      queues_i ← ∅
7:  procedure START
8:      queues_i[x] ← new pQueue() , ∀x ∈ {0,...,N−1}
9:      async BC-START()
10:     async AC-START()
```

4.2.1 Priority queues

A priority queue is a custom data structure for storing elements sorted according to their priority values. We refer to each position in a priority queue as a slot, uniquely identified by a priority value associated with it, where the lower-numbered priority values represent the elements that must be

processed first. Only a single element can be inserted in a given slot, even after being removed, as the slot is permanently labelled as used and cannot store another element. A special slot called the head slot always points to the slot with the lowest-numbered priority whose value has not been removed yet. The pointer to the head slot progresses incrementally, conditioned by the removal of elements from the queue. A priority queue exposes the following attributes:

- **id:** The unique identifier of the queue (static).
- **head:** The priority value associated with the head slot of the queue (dynamic).

Additionally, a priority queue provides an interface for accessing and modifying its contents as described below:

- **Enqueue** (v,s)**:** Add an element v with a given priority value s to the queue (ignored if the corresponding slot is not empty).
- **Dequeue** (v)**:** Removes all instances of the specified element v from the queue if it is present.
- **Peek** $() \rightarrow \{v, \perp\}$**:** Retrieve, without removing, the element v in the head slot of the queue or $\perp$ if the slot is still empty (because no **Enqueue** for that slot has been invoked yet).

As we will see, Alea-BFT leverages the properties of this structure to mediate the communication between the broadcast and agreement components of the protocol pipeline. In particular, each of the N priority queues that each replica maintains keeps track of the undelivered proposals originating from the other replicas, ordered by the priority value assigned to those proposals.

4.2.2 Broadcast Component

The broadcast component is responsible for establishing an initial local order over the client updates received and propagating that order to other replicas. Every replica process maintains two local state variables, a buffer of pending client requests buf_i, and an integer value $priority_i$, indicating the next sequence number it should assign to a proposal. The main logic of this component, illustrated in Algorithm 2, is split between two upon rules:

Upon rule 1 (lines 9 to 15): The first rule is triggered at process P_i upon receiving a client message m to be broadcast in total order. It is responsible for waiting until a batch of B requests has been accumulated, assigning it a local sequence number, and VCBC-broadcasting it to all replicas. In more detail, process P_i proceeds as follows:

- If the set of delivered messages S_i does not contain the client message m, append it to the buffer buf_i, or ignore it otherwise (lines 10 to 11).
- If the size of buf_i reached a threshold B, input buf_i to a VCBC instance tagged with ID $(i, priority_i)$, indicating that P_i assigned the local priority value $priority_i$ to a proposal consisting of the current buffer contents (lines 12 to 13).
- Increment $priority_i$, so that it can be assigned to the next proposal from P_i, and clear the buffer (lines 14 to 15).

Upon rule 2 (lines 16 to 20): The second rule is triggered at process P_i upon the delivery of a proposal m for a given VCBC instance tagged with ID $(j, priority_j)$, where j corresponds to the identifier of the replica P_j that proposed m, and $priority_j$ to the sequence number assigned to it by P_j. Process P_i proceeds as follows:

- Insert the delivered proposal m into the slot $priority_j$ of the priority queue Q_j, mapping to P_j (lines 17 to 18). This corresponds to P_i updating its view on the state of P_j's pending requests.
- If the set S_i contains m, indicating that it had already been delivered, then process P_i immediately removes it from Q_j to prevent a duplicate delivery that would violate the integrity property (lines 19 to 20).

Algorithm 2 Alea-BFT - Broadcast Component (at P_i)

```
1:  constants:
2:      B
3:  state variables:
4:      buf_i
5:      priority_i
6:  procedure BC-START
7:      buf_i ← ∅
8:      priority_i ← 0
9:  upon receiving a message m, from a client do
10:     if m ∉ S_i then
11:         buf_i ← buf_i ∪ {m}
12:         if |buf_i| = B then
13:             input buf_i to VCBC (i, priority_i)
14:             buf_i ← ∅
15:             priority_i ← priority_i + 1
16: upon outputting m for VCBC (j, priority_j) do
17:     Q_j ← queues_i[j]
18:     Q_j.Enqueue(priority_j, m)
19:     if m ∈ S_i then
20:         Q_j.Dequeue(m)
```

4.2.3 Agreement Component

The agreement component presented in Algorithm 3 establishes a total order among client requests. Requests are ordered through a succession of agreement rounds that iterate through the various priority queues and decide whether to insert the head of that queue in the total order or skip it. Processes maintain a single state variable r_i, serving as a unique identifier for the current agreement round. The execution of the agreement component starts with a call to the AC-START procedure (line 3), which initializes the local variable r_i to 0 and begins executing the agreement loop.

Agreement loop (lines 5 to 16): For each iteration r_i of the agreement loop, the queue of proposals pertaining to a certain replica is selected. This replica is a designated round leader, chosen through a deterministic function of the round number F (e.g., by rotating through all replicas). Let P_a denote the current round leader, and Q_a the corresponding priority queue at each replica r_i. Process P_i proceeds as follows:

- Run an ABA instance with id (r_i) to determine whether the *value* in the head slot of Q_a should be delivered in this round. Process P_i, inputs 1 to ABA if its local Q_a contained *value* in the head slot, or 0 otherwise (lines 6 to 9).
- If the ABA execution decided for 0, indicating that no proposal should be delivered for the current round r_i, simply proceed to the next loop iteration, otherwise:
  - If process P_i input 0 to ABA, send a FILL-GAP message to all processes that voted for 1. This step is required because, at this point in time, P_i is unaware of the value to deliver for r_i and, therefore, must request it from another process. (lines 12 to 13).
  - Block execution until the head slot of Q_a contains a value to be delivered via a call to the AC-DELIVER procedure (line 14). The value of the head slot can be updated by the delivery of a pending VCBC instance, either through "normal" execution or as a result of the reception of a FILLER message.

In addition to the main agreement loop, the agreement component also defines two upon rules associated with the recovery sub-protocol to handle the reception of valid FILL-GAP and FILLER messages:

Upon rule 1 (lines 17 to 21): The first rule is triggered by any correct process P_i upon the reception of a valid $\langle \texttt{FILL-GAP}, q, s \rangle$ message from P_j, where q identifies a priority queue Q_q, and s specifies the current head slot of Q_q in P_j. Process P_i then proceeds as follows:

- Check if its local queue pertaining to P_q is more advanced than the one of P_j, by comparing the head pointer of its Q_q against s (line 19). If it is lower, P_i cannot satisfy the FILL-GAP request and thus ignores it. Otherwise:
  – Compute and store in *entries* a verifiable message M for all VCBC instances originating from P_q tagged with a priority comprised between the value s, requested by P_j, and the current head slot of Q_q in P_i (line 20).
  – Send a FILLER message to P_j containing all the VCBC verifiable messages M, computed in the previous step (line 21).

Upon rule 2 (lines 22 to 24): The second rule is triggered by any correct process P_i upon receiving a valid $\langle \texttt{FILLER}, entries \rangle$ message. This message is received as a response to a FILL-GAP request. It contains the required information necessary for P_i to progress in the execution of the protocol by completing pending VCBC instances after blocking in line 14. Process P_i proceeds as follows:

- Deliver all M messages in *entries* to the corresponding VCBC instances. Note that the verifiability property of VCBC ensures that it immediately terminates upon the reception of M, therefore triggering the second upon rule of the broadcast component.

Finally, the AC-DELIVER procedure (line 25), called during the execution of the agreement loop, is responsible for delivering the contents of *value*, a batch of totally ordered messages m, to the application layer (line 31). Additionally, this procedure also removes *value* from all priority queues and appends its contents to the set of delivered requests S. Note that if batching is naively used, this scheme would likely lead to some redundant work being done by the replicas, as large batches differing only in a few requests could not be removed from the priority queues (in line 27), and therefore redundant operations would go through agreement and only be removed before attempting to execute them (line 29). To avoid this, we steer the protocol towards all replicas having the same batches by having the client optimistically submit requests to a single replica. If, after a timeout, the client does not receive a response, then it resubmits to all replicas. Furthermore, a real-world implementation would place an upper bound on the number of broadcast but not delivered requests, which implies that requests are not batched as soon as they are received but instead stay in a pool until the protocol progresses. Because of this, deduplication can be made before the batch is created, avoiding the redundant work problem.

Algorithm 3 Alea-BFT - Agreement Component (at P_i)

```
1:  state variables:
2:      r_i
3:  procedure AC-START
4:      r_i ← 0
5:      while true do
6:          Q ← queues_i[F(r_i)]
7:          value ← Q.Peek()
8:          proposal ← value ≠ ⊥ ? 1 : 0
9:          input proposal to ABA (r_i)
10:         wait until ABA (r_i) delivers b then
11:             if b = 1 then
12:                 if Q.Peek() = ⊥ then
13:                     broadcast ⟨FILL-GAP, Q.id, Q.head⟩
14:                 wait until (value ← Q.Peek()) ≠ ⊥ then
15:                     AC-DELIVER(value)
16:         r_i ← r_i + 1
17: upon receiving a valid ⟨FILL-GAP, q, s⟩ message from P_j do
18:     Q ← queues_i[q]
19:     if Q.head ≥ s then
20:         entries ←VCBC(queue, s').M ∀_{s'} ∈ [s, Q.head]
21:         send ⟨FILLER, entries⟩ to P_j
22: upon delivering a valid ⟨FILLER, entries⟩ message do
23:     for each message M ∈ entries do
24:         deliver M to the corresponding VCBC
25: procedure AC-DELIVER(value)
26:     for each Q ∈ queues_i do
27:         Q.Dequeue(value)
28:     for each m ∈ value do
29:         if m ∉ S_i then
30:             S_i ← S_i ∪ {m}
31:             output m
```

5 Optimizations

As we implemented and tested Alea-BFT, we developed the following optimizations to improve its performance.

Input unanimity. When a replica observes all N replicas providing as input the same value v to an ABA instance, then it is guaranteed that the ABA will decide v. To leverage this observation, we added an early termination path to the ABA protocol. This is achieved by modifying the `INIT` message (which is only sent once at the start of the protocol) to convey the input of each replica. Then, when a replica receives N modified `INIT` messages with the same value v, it immediately delivers v and broadcasts `FINISH` (if not broadcast yet). Crucially, it continues executing the ABA protocol normally until it receives $2f+1$ `FINISH` messages, as only then it is guaranteed that all correct replicas can eventually terminate.

Pipelining prediction. To maximize the chances of a successful outcome of the ABA stage, replicas keep statistics about the time to complete previous VCBC and ABA executions and use that information to fine-tune the pipeline and adapt it to the network conditions. In particular, replicas delay negative votes for an ABA when a VCBC for the slot being voted is still in progress but is expected to end soon (according to the current estimate), with the expectation that the time to complete the broadcast is smaller than the cost of a negative ABA result. Additionally, replicas anticipate batch formation (and consequently the start of VCBC) when deemed useful to minimize the chance of a negative ABA result. This is achieved by attempting to time the start of the broadcast, such that it ends right before the corresponding ABA.

Leader prediction. Latency can be improved if the client sends the request to a replica that is about to become a leader: if that happens, that replica will quickly include it in the next batch to be processed and delivered. In situations with low load and where a single client issues a sequence of requests, we found that using a round-robin approach is very effective because the rate of requests followed the leader rotation. Alternatively, clients can receive periodic hints from the replicas about the rotation schedule or rely on the replica they contact to redirect the request to a faster replica.

6 Analysis

This section analyzes the asymptotic efficiency of the Alea-BFT protocol according to time, message, and communication complexity metrics. The results of this analysis are summarized in Table 1.

To analyze Alea-BFT we observe that message exchanges occur in three places for each proposal payload to be delivered. First, during the execution of the broadcast component, a replica initiates a VCBC instance to disseminate the locally ordered proposal. Second, all replicas participate in successive ABA executions to decide whether or not to deliver the proposal in a particular slot. Here, we denote by σ the average number of ABA instances executed over a given slot to reach a positive decision. Finally, a fetch request is triggered by replicas that did not VCBC-deliver the proposal before the corresponding ABA decided 1.

6.1 Time Complexity

Time complexity is defined as the expected number of communication steps from a client request to its output. In the case of Alea-BFT, the first and third steps terminate in constant time $O(1)$. In contrast, the total number of rounds required for the agreement component to decide depends on the value of σ, therefore giving an expected time complexity of Alea-BFT of $O(\sigma)$.

6.2 Message Complexity

We measure message complexity as the expected number of messages generated by correct replicas to execute a single client request. In Alea-BFT, the VCBC instance from the broadcast phase generates $O(N)$ messages; then, every ABA instance exchanges $O(N^2)$ messages in expectation; and finally, the third recovery phase incurs an overhead of $O(N)$ messages per replica that triggers this fallback protocol. Hence, the expected message complexity of Alea-BFT is $O(\sigma N^2)$, due to the σ ABA instances that are executed per priority queue slot before delivery, which is close to the quadratic lower bound on message complexity shown by Dolev and Reischuk [19].

6.3 Communication Complexity

Communication complexity consists of the expected total bit-length of messages generated by correct replicas during the protocol execution. Let $|m|$ correspond to the average proposal size and λ the size of a threshold signature share. The execution of VCBC incurs a communication complexity of $O(N(|m|+\lambda))$. Each ABA instance requires correct replicas to exchange $O(\lambda N^2)$ bits in expectation, and finally, each replica that triggers the recovery phase adds communication cost of $O(N(|m|+\lambda))$ bits. This results in an expected total communication complexity of $O(N^2(|m|+\sigma\lambda))$ due to σ ABA executions and up to N recovery rounds being triggered.

Table 1: Complexity of Alea-BFT decomposed by stages.

Stage	Message	Communication	Time
Broadcast	$O(N)$	$O(N(\|m\|+\lambda))$	$O(1)$
Agreement	$O(\sigma N^2)$	$O(\sigma\lambda N^2)$	$O(\sigma)$
Recovery	$O(N^2)$	$O(N^2(\|m\|+\lambda))$	$O(1)$
Total	$O(\sigma N^2)$	$O(N^2(\|m\|+\sigma\lambda))$	$O(\sigma)$

6.4 Estimating σ

As previously mentioned, Alea-BFT does not guarantee a constant-time execution, which could negatively affect the protocol latency. In particular, this is because multiple zero-deciding ABA instances could be executed over the same priority queue slot until its contents are considered totally ordered. However, we argue that, despite being theoretically

unbounded, the value of σ (the number of ABA instances required for a decision) is, in practice, close to the optimal value of 1. This is justified by the observation that, in a round-robin leader assignment, each queue is revisited every N rounds, meaning that $N-1$ other ABA instances were executed by the time a given queue is revisited. Considering ABA's validity property, which states that the decided value must have been proposed by a correct process, the termination of a VCBC instance by $N-f$ correct replicas guarantees that the next ABA execution pertaining to it will decide for 1. Therefore, for the value of σ to increase by a single unit, correct replicas would, on average, have to complete N sequential ABA executions for every single VCBC instance. In our experiments, we validated that the value of σ was very close to 1 in practice.

7 Correctness

Next, we sketch the correctness of the protocol, and we provide a complete proof in a separate arXiv e-print [8]. The complete proof follows the structured proof format by Lamport [33] for increased preciseness.

Safety. The two safety properties that need to be proven are integrity (each message m appears at most once in the delivery sequence of correct process i) and total order (any two messages m and m' are delivered in the same order by any pair of correct processes i and j). Integrity is derived from the fact that once a message is delivered, it is added to the set of delivered messages, dequeued from all queues and never enqueued again. The total order property follows from the fact that delivering two different messages at different replicas in the same slot would lead to a violation of the consistency property of VCBC.

Liveness. The liveness properties build mostly on the liveness guarantees of the protocols used as building blocks. Given these guarantees, it suffices to follow the protocol steps to prove that we eventually satisfy the preconditions for the building block protocols to produce the necessary outputs to decide a value, namely that the messages that were broadcast reach a sufficient number of correct processes.

Censorship resilience. Prior asynchronous BFT protocols include mechanisms to enforce that Byzantine replicas cannot significantly delay the delivery of any particular message (i.e., a fairness property). This is required, in particular, for protocols that use an asynchronous common subset (ACS) to agree on a subset of the various proposals from different replicas to deliver, since a Byzantine replica can bias the choice of proposals to be included in the output of ACS. In Alea-BFT, however, censorship resilience is easily achieved by construction, given that any replica can initiate a VCBC for a client request. Thus, clients can broadcast their requests to $f+1$ or more replicas (possibly after a wait, to optimistically check if sending to a single replica suffices). This guarantees that at least one of these replicas is non-faulty and will drive the request execution.

8 Implementations

We implemented Alea-BFT in three open-source prototypes: an initial research implementation, and then two real-world integrations, namely with the SSV Ethereum distributed validator (where it is being considered to replace QBFT as its main protocol in the near future [51]) and with an experimental consensus layer for subnets in the Filecoin network [1].

Research prototype. Our first prototype implementation of Alea-BFT, which is available as open source [2], comprises 20,000 lines of Java code. The source code is organized in a modular manner, with the main logic of Alea-BFT leveraging different subprotocols (namely broadcast and binary agreement) as building blocks. Reliable point-to-point links were implemented using TCP streams, similar to prior work [24, 27, 40]. Additionally, we implemented HBBFT using the same codebase as a starting point to use it as one of the comparison baselines.

Ethereum distributed validator. Decentralized Validator Technology (DVT) is a technology to improve the security, robustness, and openness of the Ethereum network [12, 22]. Using distributed validators, several non-trusted parties cooperate to logically act as a single validator, and this way each participant is able to overcome the need to commit 32 ETH (over USD 3,700 as of this writing) to enter the network. To act as a single logical entity, once a distributed validator is called to conduct a validation task (called a duty in Ethereum), the various parties that form the validator run a BFT consensus protocol to decide on the input to the duty (which can be, for instance, a block or a pointer to the head of the chain, depending on the type of duty) and the respective outcome.

We have been collaborating with ssv.network for over one year [51] to implement Alea-BFT in the SSV codebase, with the goal of offering stronger resilience in the presence of adverse network conditions or Byzantine behavior. Their current plan is to incorporate Alea-BFT in their production codebase in the near future. The repository for this implementation is available as open source [3].

The main integration challenge came from the fact that consensus is used as a standalone instance, instead of a replicated state machine that executes a command sequence, which would be more aligned with the abstraction offered by Alea-BFT. Therefore, we adapted Alea-BFT with the following design features and optimizations to fit this specific context.

Adapting Alea-BFT to one-shot consensus. In a distributed validator, even though the duty is the same across the processes that comprise the validator, because it is known a few epochs in advance, the input to that duty needs to be agreed upon because each process may retrieve it from a different source (called a beacon client in Ethereum, with several possible providers). To reach consensus on that input, each process will attempt to send its input using Alea-BFT's atomic broadcast protocol, and the first to be delivered by the protocol is the output of consensus. Note that only one instance of VCBC per process is needed to implement this one-shot consensus, thus

simplifying the implementation. A possible concern is that the validity condition for this consensus implementation allows for a single divergent opinion to be the final output. However, this is safe because the inputs are coming from sources that are outside of the distributed validator system. Therefore their correctness is beyond the scope of the distributed validator. (In addition, some basic validation checks can also be conducted.) Additionally, for fairness, the round-robin rotation of protocol leaders in different consensus instances is based on a pseudorandom sequence, allowing the advantageous roles to even out over time.

Early consensus termination. As an optimization, if a replica receives a VCBC proof for the same value for every participant, it knows in advance that the corresponding value is the only possible output of consensus and can return it immediately. However, it continues to run the consensus protocol in case other replicas do not receive the same view. This is particularly useful in distributed validators because replicas have a high chance of proposing the same value. This is because, for most validations, different inputs only occur when different replicas have a divergent view of the current state of the blockchain, which is rare. Note that this optimization differs from the first optimization described in Section 5, which refers to ABA instead of VCBC instances.

The current implementation consists of 5,000 lines of Go code, integrated as a subset of the large codebase of SSV.

Consensus layer for Filecoin subnets. We also implemented Alea-BFT as part of Mir/Trantor [45, 46], an experimental framework for distributed protocols, which is meant to become a new consensus layer for the subnets of Filecoin [18, 45]. This framework already supports the ISS-PBFT [52] protocol, and an upcoming implementation of a new protocol, apart from Alea-BFT. We implemented Alea-BFT in 4,000 lines of Go code and are currently merging it into the main repository [45]. It is nonetheless already freely available as open source [1]. The subprotocols used by Alea-BFT – ABA and VCBC – are implemented as independent modules, allowing their reuse by other protocols to be implemented within the same framework. In addition, we added support for threshold cryptography and BLS signatures within the framework, to be available to other protocols.

We improved the performance of this implementation through the parallel execution of agreement rounds. In Alea-BFT, the system can only order one batch of requests per agreement round, capping the system throughput to $BatchSize \times ABARate$. To overcome this limit, we allow multiple agreement rounds to make progress in parallel, but they buffer the delivery until it is guaranteed that the instances are delivered in order. However, we have to be careful so that this parallelization does not overload the network, while also remaining effective. To this end, we limit parallelization to the next N agreement rounds and restrict ABA execution as follows. Before all the preceding agreement rounds deliver, ABA instances are only allowed to make progress using the unanimity optimization and otherwise need to wait for their turn. Under this restriction, the eager execution of ABA instances only broadcasts up to two messages (INIT and FINISH), thus limiting its network impact.

9 Evaluation

We evaluated the three implementations of Alea-BFT under several scenarios. The following questions guide our evaluation. (1) How does the performance of Alea-BFT compare to other asynchronous BFT protocols across different configurations? (2) How robust is Alea-BFT to faults? (3) How do the real-world implementations of Alea-BFT compare to the previous protocols that they employed?

9.1 Experimental environment

Baselines. The research prototype of Alea-BFT was evaluated against two baselines. First, we compared it to our own implementation of HBBFT [40], the first protocol in the new generation of asynchronous BFT, and where, to obtain an apples-to-apples comparison, we started from the same codebase as Alea-BFT and tried to optimize the implementation of HBBFT to the fullest extent. Second, we compared to Dumbo-NG [24], which is the state-of-the-art asynchronous BFT protocol with outstanding performance (several-fold better than the direct competitors, according to their results [24]), and, in this case, we deploy their unmodified codebase. For the two real-world implementations, we used the protocols that those systems originally supported as baselines.

Setup. We deployed Alea-BFT in a cluster where the replica and client instances ran on machines equipped with AMD EPYC 7272 12-Core Processors. In addition, Docker was used to limit each replica's CPU usage to 4 cores and the Java VM was capped at 10GB of memory. These machines were connected to the same local network with 1Gb connections. To evaluate the effects of deploying Alea-BFT in a wide-area network, we emulate varying additional inter-replica latency using `netem`. Some deployment characteristics for the real-world prototypes differed whenever noted.

Clients submit requests in an open loop, and we vary the inter-request interval and the number of clients to increase the load. The payload size of the requests and responses is 256 bytes, aligned with the size of Bitcoin transactions (as noted and employed in previous work [40]). Each experiment runs for 2 minutes, and we repeat experiments 5 times and report the average. When measuring latency for Dumbo-NG, its implementation always loads the system, and therefore we would have to modify their codebase to measure the latency of an isolated request. As such, the latency measurements for Dumbo-NG represent the performance at an intermediate load generated by the implementation, where the system is not quiescent but also not as overloaded as during throughput measurements for the other protocols.

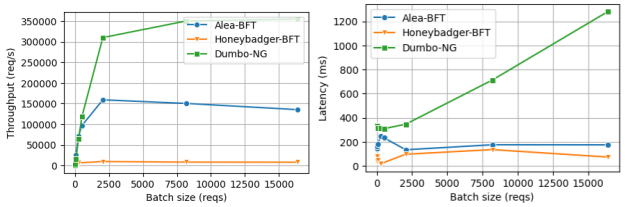
9.2 Performance under different parameters

We start by using our research prototype to measure the latency and throughput of Alea-BFT and the two baselines under different configurations and deployments, namely varying the batch size, the replication factor, and the inter-replica latency. When the sensitivity to one of these parameters is being evaluated, the remaining ones are fixed to a batch size of 1024, n =4 replicas (i.e., $f = 1$), and LAN (minimal inter-node latency), respectively.

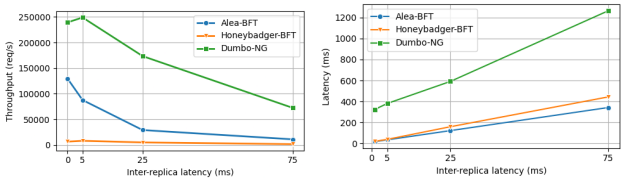
Figures 2a and 2b show the peak throughput and latency while varying the batch size. In this particular setting, the latency at the peak throughput is reported instead of the base latency (measuring an isolated request without any system load). This is because the latter would require issuing only one or a few requests, much less than the batch size. Thus, the measured latency would correspond to either slowly filling up the batch or triggering a batch timeout. The results show that Alea-BFT is competitive with the state of the art (Dumbo-NG) and that both are a significant improvement over their predecessor (HBBFT). While Alea-BFT cannot match the peak throughput of Dumbo-NG, they are both in the same order of magnitude (hundreds of thousands of txs/s), versus ≈15k txs/s for HBBFT. When considering latency, Alea-BFT outperforms Dumbo-NG across all tested batch sizes. We attribute these differences in throughput and latency to the choice of agreement primitive. In particular, Dumbo-NG uses MVBA, which allows for better throughput by accepting several batches at once, but the simpler ABA primitive of Alea-BFT enables a better latency under a comparable load. Note that the fact that Alea-BFT's throughput peaks earlier than Dumbo-NG's in this setting is mainly an implementation artifact – our codebase has an external open loop client that further saturates the network, unlike Dumbo-NG's.

Next, we use `netem` to evaluate the performance under different network conditions (LAN vs. WAN). Figures 2c and 2d show the peak throughput and latency when varying inter-replica latency. The results show that Alea-BFT has the lowest latency of all protocols while achieving a peak throughput in the tens of thousands of requests per second when the inter-node latency is under 25ms. HBBFT witnesses a similar degradation in latency to Alea-BFT because the critical path for a normal-case request execution is the same for both protocols, except for a single protocol step, which explains the slightly higher latency of HBBFT.

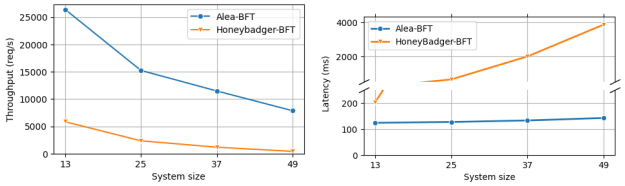
Finally, we scale out the experiments by increasing the number of replicas participating in the consensus. For these experiments, we use 13, 25, 37 and 49 replicas, and, in this case, we use `netem` to simulate a WAN environment, with a 75ms inter-replica latency, corresponding to an RTT of 150ms (approximating a cloud deployment). Furthermore, since the available setup forced some replicas to be co-located on the same machine, to ensure a realistic and uniform bandwidth availability, each instance's bandwidth was capped at 50Mb/s using a token bucket filter.



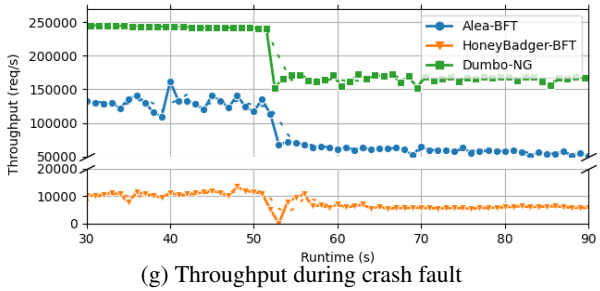
(a) Peak throughput vs batch size (b) Latency at peak throughput vs batch size



(c) Peak Throughput vs inter-replica latency (d) Base latency vs inter-replica latency



(e) Peak throughput vs system size (f) Base latency vs system size



(g) Throughput during crash fault

Figure 2: Prototype implementation evaluation

In this case, we were not able to configure the Dumbo-NG code to use the same replica group sizes as the ones we employed for the other two systems, which explains why there are only two curves. Our results show that Alea-BFT not only has superior throughput but also achieves very good latency in unloaded scenarios, due to the clients being able to predict the current leader and send requests to the replica that will drive the decision the fastest. On the other hand, in unloaded scenarios, HoneyBadger's clients need to contact $2f + 1$ replicas to ensure progress, meaning that, for a single request to go through, $2f + 1$ ABAs need to be executed.

9.3 Performance under faults

We compared the performance of Alea-BFT and the baselines in a scenario where one of the replicas crashes 50 seconds into the trace. We inject a crash fault instead of a protocol-specific Byzantine fault, for a direct comparison between protocols. The results in Figure 2g show that Alea-BFT and HBBFT suffer more with the crash of f replicas because they share the unanimity optimization described in

Section 5 (which cannot be used when a replica is unresponsive), but Dumbo-NG also takes a significant hit (around 30% of throughput) due to bandwidth wasted on the faulty replica.

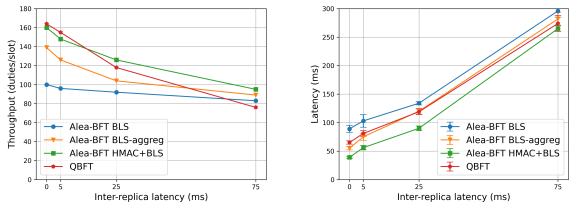
We also evaluate the performance under faults for the other implementations and present the results in the next section.

9.4 Real-world implementations

Ethereum distributed validator. We start this part of the evaluation with the implementation of Alea-BFT in the SSV distributed validator of Ethereum. In this case, the experimental methodology is constrained by the way that SSV operates, namely that the system progresses in a sequence of slots of fixed duration (12 seconds in Ethereum), and during each slot, each validator is assigned a set of duties (tasks such as block proposal and attestation). Thus, to measure the base latency, we set the number of duties per slot to 1 and measure the time to complete it, whereas throughput is measured by increasing the number of duties per slot until this metric peaks. Since the performance is not network-bound and the number of nodes is low, we did not use the bandwidth cap. We used a group of 4 replicas, with no added inter-replica latency, and batching is not applicable in this setting. We tested variants of Alea-BFT that use different message authentication methods and compared these to the existing QBFT-based codebase. In particular, we start with a direct comparison to QBFT (which, in the SSV codebase, uses BLS digital signatures without aggregation), and then add BLS aggregation to Alea-BFT. (This change could also be applied to QBFT, but this was avoided to keep the baseline as the existing codebase.) Then, we replace signatures with HMACs, which is possible in Alea-BFT but would not be directly applicable to QBFT, because of messages conveyed to all processes during round changes. In the case of HMACs, BLS is only used to verify the final VCBC signature and compute ABA's shared coin.

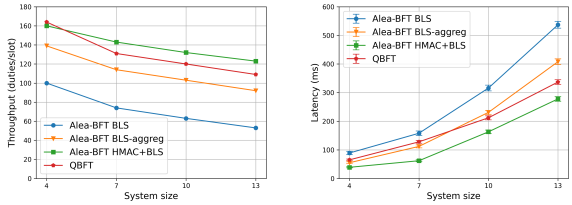
Figure 3 shows the performance of the SSV validator using different protocols. In these plots, the latency and throughput in the most basic setting can be determined by the leftmost point of Figures 3a and 3b. These results show that Alea-BFT with BLS aggregation and with HMACs has similar peak throughput and better latency than the previous codebase that uses QBFT. This highlights how designing Alea-BFT to have a small number of protocol steps, combined with the possibility of using HMACs that comes from not having the view change mechanism from partially synchronous protocols can lead to competitive performance in this setting. We attribute the slightly better throughput of QBFT to the leader-driven protocol allowing for exchanging a smaller number of messages. However, we see the overhead of Alea-BFT as a modest price to pay for not making partial synchrony assumptions.

The effects of varying the inter-replica latency is shown in the remainder of Figures 3a and 3b. The key takeaway is that, for all tested conditions, Alea-BFT closely follows QBFT in terms of base latency and peak throughput and, with the best choice of cryptographic primitives in place, Alea-BFT can



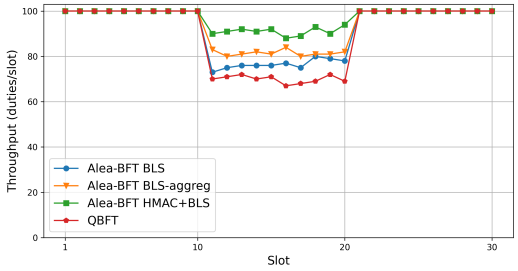
(a) Peak throughput vs inter-replica latency

(b) Base latency vs inter-replica latency



(c) Peak throughput vs system size

(d) Base latency vs system size



(e) Throughput during crash fault

Figure 3: Distributed validator deployment evaluation

even achieve lower latency values. Figure 3b also depicts the change of relative importance of cryptographic primitives as inter-replica latency varies – in a LAN environment, as the network delay is small relative to the cost of cryptography, the several variants have a very noticeable relative difference among them. However, as the inter-replica delay increases, this difference decreases in proportion.

Next, we measured performance as the group size increases (Figures 3c and 3d). Currently, in SSV, a validator can only employ 4, 7, 10, or 13 operators, as defined in its smart contract. In this experiment, as in the previous one, Alea-BFT's latency and throughput follow QBFT's, achieving lower latency and higher throughput values when using HMAC for point-to-point authentication and BLS digital signatures.

Finally, Figure 3e shows the results of an experiment where we crash one of the processes, chosen at random, at the beginning of the 11th slot in the run, then restart it in the 21st slot, and we plot the number of duties that are executed per slot throughout the trace. The results show that Alea-BFT is more resilient to this fault because of the principles behind its design: the fault will affect $1/4$ of the VCBC instances, but these rounds will quickly be skipped and replaced with productive work led by the other replicas. In contrast, QBFT waits for a timeout and a leader change protocol to complete, which slows down the entire system for that duration.

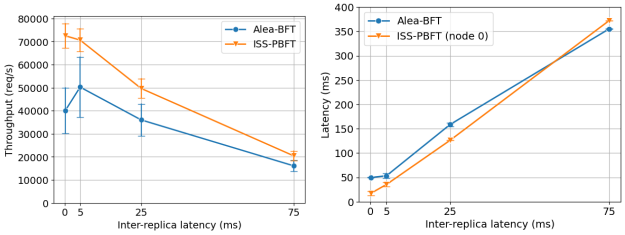
Consensus layer for Filecoin subnets. In the last part of this section, we evaluate Alea-BFT's integration into an experimental consensus layer [45, 46] for Filecoin subnets [18] against the existing implementation of ISS-PBFT [52] in the same codebase. ISS-PBFT uses the same parameters as in its original evaluation [52].

Base latency experiments use two co-located closed-loop clients per replica, ensuring incoming request queues are never empty. This was required because the implementation of ISS-PBFT stalls when there are no requests to propose. However, it has the downside of generating some load that may negatively affect latency. We only present results for replica 0 in the ISS-PBFT latency measurements due to an implementation issue that inflates latencies in other replicas, which would unfairly harm the baseline's performance. Peak throughput is measured in a configuration of $8 * B$ closed-loop clients co-located with each replica, where B is the batch size, which we empirically determined to maximize the throughput.

We begin by evaluating Alea-BFT's performance against ISS-PBFT when varying the inter-node latency. Figures 4a and 4b show that Alea-BFT closely follows the performance ISS-PBFT in wide-area settings, in terms of peak throughput and base latency. Furthermore, while Alea-BFT is initially limited to ≈40k requests/s and ≈50ms base latency, it becomes on par with ISS-PBFT when the limiting factor shifts from threshold cryptography to network latency.

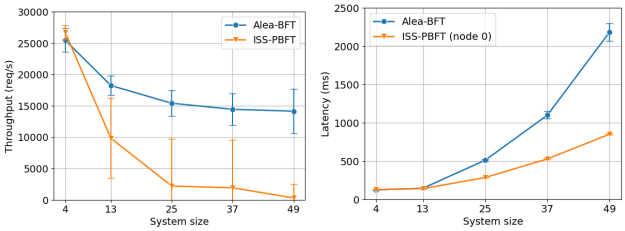
Additionally, we evaluated Alea-BFT's ability to scale against ISS-PBFT, which is relevant given that scalability is a key design goal in ISS. Figures 4c and 4d show peak throughput and base latency measurements for Alea-BFT and ISS-PBFT for a variety of system sizes. Regarding peak throughput (Figure 4c), Alea-BFT's throughput degrades gracefully as the system size increases, fully saturating the (bandwidth-capped) network. In contrast, ISS-PBFT degrades abruptly and stops processing requests altogether after a few seconds for $N = 49$. However, we believe this is an artifact of this implementation of ISS-PBFT, which reacts poorly under strained network conditions and is not intrinsic to the ISS-PBFT protocol. Regarding latency (Figure 4d), both protocols maintain near-constant base latency under system sizes up to $N = 22$, after which it begins to increase. In this case, ISS-PBFT has a lower latency than Alea-BFT because its multi-leader design allows requests to be processed as soon as they reach the PBFT primary replica, whereas in Alea-BFT we have to wait for the designated replica's turn to run its agreement round.

Finally, we studied the impact of crash faults on both Alea-BFT and ISS-PBFT. Figure 4e shows an execution trace of one Alea-BFT and one ISS-PBFT execution with the default settings, where a single replica crashes after 150s (and stays crashed). To aid evaluation, a dotted line was added to both curves, showing a moving average of the system's throughput across all repetitions. In this trace, we first observe a 15-second stall of ISS-PBFT after the crash, waiting for a timeout for the detection of the crashed replica, whereas Alea-BFT can continue uninterrupted (albeit at reduced throughput) thanks to its leaderless design. After this timeout expires, ISS excludes the crashed replica from the set of leaders and continues with a relatively small (≈20%) performance hit. However, Alea-BFT is penalized on two fronts – it both loses a replica proposing requests (like ISS) and the ABA unanimity optimization – leading to a reduction in throughput when compared to the system with all replicas functional.



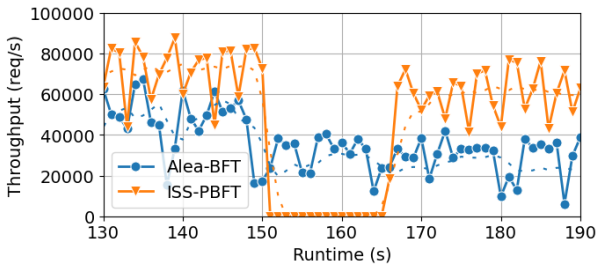
(a) Peak throughput vs inter-replica latency

(b) Base latency vs inter-replica latency



(c) Peak throughput vs system size

(d) Base latency vs system size



(e) Throughput during crash fault

Figure 4: Mir/Trantor deployment evaluation

10 Conclusion

In this paper, we presented Alea-BFT, a practical asynchronous BFT protocol with a design that combines simplicity with performance. Our experimental evaluation shows that Alea-BFT performs better than the top-performing Dumbo-NG in latency, offers comparable throughput, and is resilient to faults. Importantly, Alea-BFT is being adopted in the real world, namely by Ethereum distributed validators.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Zhaoguo Wang, for their helpful feedback. This work was supported by Fundação para a Ciência e a Tecnologia, projects UIDB/50021/2020 and PTDC/CCI-INF/6762/2020, and by the European Union's Horizon 2020 research and innovation programme, under grant agreement No 952226, project BIG.

References

[1] Mir codebase with Alea-BFT. https://github.com/abread/mir/tree/43a82f13b3f5353a80bdfe2fe2613daed0fbf710.

[2] Prototype implemenation of Alea-BFT. https://github.com/diogoantunes25/Alea-BFT.

[3] ssv codebase with Alea-BFT. https://github.com/MatheusFranco99/ssv.

[4] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74, 2005.

[5] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 337–346, 2019.

[6] Enterprise Ethereum Alliance. EEA publishes QBFT blockchain consensus protocol. https://entethalliance.org/23-01-qbft-spec-version-1-released/.

[7] Ethereum Enterprise Alliance. QBFT blockchain consensus protocol specification v1. https://entethalliance.org/specs/qbft/v1/.

[8] Diogo S. Antunes, Afonso Oliveira, André Breda, Matheus Guilherme Franco, Henrique Moniz, and Rodrigo Rodrigues. Alea-bft: Practical asynchronous byzantine fault tolerance. arXiv:2202.02071 [cs.DC] https://arxiv.org/abs/2202.02071, 2022.

[9] James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2):165–175, 2003.

[10] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, PODC '94, pages 183–192, 1994.

[11] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, nov 1987.

[12] Vitalik Buterin. Post by @VitalikButerin on X. https://twitter.com/VitalikButerin/status/1588669782471368704.

[13] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2nd ed.)*. Springer, 2011.

[14] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology — CRYPTO 2001*, pages 524–541. Springer, 2001.

[15] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[16] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation, OSDI 1999*, pages 173–186, 1999.

[17] Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009*, pages 153–168, 2009.

[18] Alfonso De la Rocha, Lefteris Kokoris-Kogias, Jorge M Soares, and Marko Vukolić. Hierarchical consensus: A horizontal scaling framework for blockchains. In *2022 IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 45–52, 2022.

[19] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *Journal of the ACM (JACM)*, 32(1):191–204, 1985.

[20] Sisi Duan, Michael K Reiter, and Haibin Zhang. Beat: Asynchronous bft made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2028–2041, 2018.

[21] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, apr 1988.

[22] ethereum.org. Distributed validator technology. https://ethereum.org/en/staking/dvt/.

[23] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[24] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo-NG: Fast asynchronous bft consensus with throughput-oblivious latency. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1187–1201, 2022.

[25] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *Financial Cryptography and Data Security - 26th International Conference, FC 2022*, volume 13411 of *Lecture Notes in Computer Science*, pages 296–315. Springer, 2022.

[26] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Speeding dumbo: Pushing asynchronous BFT closer to practice. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022*, 2022.

[27] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 803–818, 2020.

[28] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.

[29] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, page 165–175, 2021.

[30] K.P. Kihlstrom, L.E. Moser, and P.M. Melliar-Smith. The securering protocols for securing group communication. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, volume 3, pages 317–326 vol.3, 1998.

[31] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP 2007, pages 45–58, 2007.

[32] Oisín Kyne. The distributed validator protocol roadmap. Obol Network Blog. https://blog.obol.tech/roadmap-the-distributed-validator-protocol/, 2024.

[33] Leslie Lamport. How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*, 11:43–63, 2012.

[34] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[35] Chao Liu, Sisi Duan, and Haibin Zhang. Epic: Efficient asynchronous bft with adaptive security. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2020)*, pages 437–451. IEEE, 2020.

[36] Yuan Lu, Zhenliang Lu, and Qiang Tang. Bolt-dumbo transformer: Asynchronous consensus as fast as the pipelined BFT. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022*, pages 2159–2173, 2022.

[37] Ethan MacBrough. Cobalt: Bft governance in open networks. *arXiv preprint arXiv:1802.07240*, 2018.

[38] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distrib. Comput.*, 11(4):203–213, oct 1998.

[39] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machine for wans. In *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*, pages 369–384, 2008.

[40] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 31–42, 2016.

[41] Henrique Moniz. The istanbul bft consensus algorithm. arXiv preprint arXiv:2002.03613, 2020.

[42] Henrique Moniz, Nuno Ferreria Neves, Miguel Correia, and Paulo Verissimo. Ritas: Services for randomized intrusion tolerance. *IEEE transactions on dependable and secure computing*, 8(1):122–136, 2008.

[43] Achour Mostefaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous byzantine consensus with t< n/3 and o (n2) messages. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, PODC '14, pages 2–9, 2014.

[44] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. Kauri: Scalable bft consensus with pipelined tree-based dissemination and aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 35–48, 2021.

[45] Matej Pavlovic. Mir – the distributed protocol implementation framework. https://github.com/consensus-shipyard/mir/blob/e100175138a4fd8947b6757452334698ee518967/README.md.

[46] Matej Pavlovic. Trantor: Modular state machine replication. https://github.com/consensus-shipyard/trantor-doc/blob/47dfc316a6d81604e1c567b823358f53fdfde4b4/main.pdf, 2023.

[47] Michael O Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 403–409. IEEE, 1983.

[48] Michael K Reiter. The rampart toolkit for building high-integrity services. In *Theory and practice in distributed systems*, pages 99–110. Springer, 1995.

[49] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[50] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. Bft protocols under fire. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 189–204, 2008.

[51] ssv.network. SSV into the future: Asyncrounous BFT protocols. https://ssv.network/blog/technology/ssv-into-the-future-asyncrounous-bft-protocols/.

[52] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, pages 17–33. ACM, 2022.

[53] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 135–144, 2009.

[54] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. DispersedLedger: High-throughput byzantine consensus on variable bandwidth networks. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022*, pages 493–512, 2022.

[55] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC'19, pages 347–356, 2019.

Harmony: A Congestion-free Datacenter Architecture

Saksham Agarwal
Cornell University

Qizhe Cai
Cornell University

Rachit Agarwal
Cornell University

David Shmoys
Cornell University

Amin Vahdat
Google

Abstract

Datacenter networks today provide best-effort delivery—messages may experience unpredictable queueing, delays, and drops due to switch buffer overflows within the network. Such weak guarantees reduce the set of assumptions that system designers can rely upon from the network, thus introducing inefficiency and complexity in host hardware and software.

We present Harmony, a datacenter network architecture that provides "congestion-free" message delivery guarantees—each message, once transmitted by the sender, experiences bounded queueing at each switch in the network. Thus, by design, Harmony ensures that network delays are bounded in failure-free scenarios, and that congestion-related drops are eliminated. We establish, both theoretically and empirically, that Harmony provides these powerful properties with near-zero overheads compared to best-effort delivery networks: it incurs a tiny additive latency overhead that diminishes with message sizes, and achieves near-optimal network utilization.

1 Introduction

Datacenter networks today provide best-effort delivery—it is hard, or even impossible, to bound queueing and delays experienced by messages at switches; even worse, messages may be dropped due to switch buffer overflows, and may need to be retransmitted multiple times before they are delivered to their destination. As a result, messages experience unpredictable and variable network delays and congestion-related drops. Such unpredictability and variability reduces the set of assumptions that system designers can rely upon from the network. Thus, to operate correctly on such best-effort delivery networks, host hardware and software must embrace inefficiency and complexity.

We present Harmony, a distributed packet-switched datacenter network architecture that provides powerful "congestion-free" message delivery guarantees: each message, once transmitted by the sender, is guaranteed to experience a small bounded amount of queueing at each switch along the path(s) it traverses. Thus, by design, Harmony ensures that network delays are bounded in failure-free scenarios, and that congestion-related drops are eliminated. The bounded network delay and zero switch buffer overflow guarantees also allow Harmony to handle inevitable failures efficiently: since delayed and/or undelivered messages are limited to hardware failures, Harmony enables extremely fast failure reaction.

Harmony enables these powerful properties by placing its intellectual roots in the classical Resource ReSerVation protocol (RSVP) [71]. We demonstrate that naïvely using RSVP on datacenter networks enables bounded queueing at each switch; however, throughput can be significantly lower than the optimal. Intuitively, the core reason for the suboptimal throughput is that the RSVP-based design maintains the invariant that each switch observes zero queueing at all times; enforcing zero queueing leads to non-trivial throughput overheads in distributed designs. Harmony combines the idea of virtual channels [23, 60] with RSVP to orchestrate network resources among competing messages while maintaining the invariant that each switch observes bounded, albeit potentially non-zero, queueing. We establish theoretically that allowing a small amount of queueing at the switches enables Harmony to provide bounded queueing and network delay guarantees while achieving near-optimal throughput.

We evaluate an end-to-end implementation of Harmony over a testbed, and over simulations across a variety of settings that mix-and-match multiple workloads, traffic patterns, network topologies, network oversubscription, and network loads with and without background (best-effort) traffic. Our evaluation reveals several interesting phenomenon. First, we find that Harmony achieves network delay and throughput very close to centralized zero-queue network designs [55], while providing the benefits of a completely distributed architecture. Second, rather surprisingly, we find that Harmony's tail latency and throughput is comparable or even better than state-of-the-art distributed datacenter protocols [19, 27, 32, 33, 44, 51] that provide best-effort delivery. Finally, we find that Harmony performance near-perfectly matches our theoretical bounds.

Harmony demonstrates that it is possible to rearchitect dat-

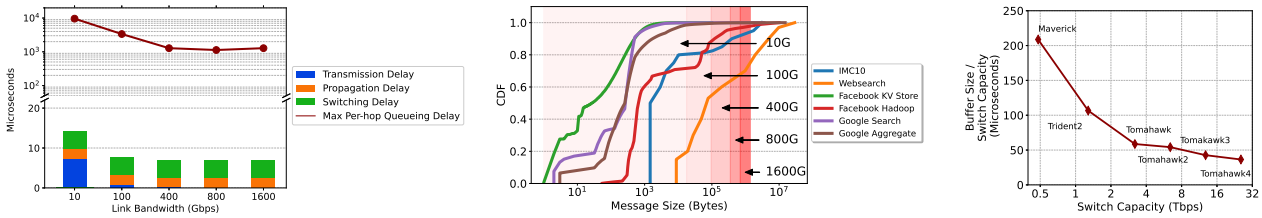


Figure 1: **Motivation for congestion-free datacenter architectures**. (left) modern high-bandwidth datacenter networks have tiny unloaded RTTs, and worst-case per-hop queueing delays can be orders-of-magnitude higher than unloaded RTTs; (center) a large fraction of messages are smaller than a single BDP for high-bandwidth networks, rendering modern best-effort delivery protocols inefficient; (right) switch buffer sizes are growing slower than switch capacities, resulting in increasingly more packet drops for best-effort delivery protocols. Discussion in §2.

acenter networks to provide much stronger guarantees than today's best-effort delivery networks, while maintaining near-optimal performance. There are two important caveats, however. First, our point is not that every application will benefit from Harmony—achieving these guarantees requires a tiny additive latency overhead (that diminishes with message sizes) at low and moderate loads; since not all applications may want to tradeoff such an overhead for strong guarantees from the network, datacenter networks should simultaneously support both Harmony and best-effort delivery, allowing applications to choose between the two depending on desirable goals. Second, the current implementation of Harmony requires support from the network: the two highest priority levels, a small amount of per-port soft state, support for embedding path identifiers in packet headers, routing based on path identifiers in packet headers, and support for packet modifications. While modern (programmable) switch hardware readily support these functionalities, it remains an interesting question whether Harmony guarantees can be achieved without network support. Finally, Harmony provides network-layer guarantees: bounded queueing, delays, and zero switch buffer overflows within the network; additional work is needed at each layer of systems stack to reap Harmony benefits in terms of improved host software and hardware. While it may take longer than the lifetime of a single project to realize systems that efficiently exploit all the benefits of Harmony, we believe the potential benefits make it a worthwhile exploration.

2 Congestion-free Datacenters: Motivation

Our exploration of congestion-free datacenter network architectures is motivated by three datacenter hardware trends[1].

Modern datacenter networks have tiny unloaded[2] round trip times (RTTs); queueing delays and buffer overflow are the root cause of unpredictability. Network hardware has improved over the past few years: multi-hundred gigabit links are already being deployed and Terabit Ethernet deployments are anticipated soon. Thus, datacenter networks will soon, if not already, support single-digit microsecond unloaded RTTs between hosts—as shown in Figure 1(left), transmission delays are reduced to a bare minimum, and unloaded RTTs are now sum of tiny propagation delays and switching delays. Furthermore, emergence and deployment of high-performance host network stacks [21, 38, 46], hardware-offloaded network stacks [2, 26, 58], host congestion control mechanisms [7, 31], and μs-scale host schedulers [21, 37, 53] have reduced host processing variability to a bare minimum. On the other hand, as shown in Figure 1(left), the worst-case queueing delay experienced by a packet is, and will continue to remain, much larger than unloaded RTTs. Put together, these trends result in queueing delays and buffer overflows as the root cause of network unpredictability.

Large bandwidth-delay products (BDPs) in modern datacenter networks make congestion control ineffective. Network RTTs limited by propagation and switching delays means that network BDP now increases linearly with link bandwidth. Such rapid increase in BDP means that most messages in the network now fit within a few BDPs. Figure 1(center) demonstrates this for several production workloads [8, 13, 18, 61, 65]: for 100 and 400Gbps links, more than 54% and 64% of the messages in the websearch workload are less than 1BDP; for the IMC10 workload, the corresponding numbers are 82% and 89%; a recent study from Google presents similar numbers for RPCs within Google [11]. Modern best-effort delivery datacenter transport protocols [19, 27, 32, 41, 44, 51, 72] advocate to blast the first BDP worth of packets into the network during the first RTT; blasting the first BDP worth of data fundamentally means that even optimal congestion response, one that detects congestion and responds perfectly in one network round trip, will not have time to converge to the "right" rates for an overwhelmingly large fraction of the messages resulting in larger amounts of data in flight, and larger unpredictability in queueing and network delays.

[1] We use Maverick (48-port 10G with 12MB buffers), Trident-2 (32-port 40G with 16MB buffers), Tomahawk (32-port 100G with 22MB buffers), Tomahawk-2 (64-port 100G with 42MB buffers), Tomahawk-3 (32-port 400G with 64MB buffers) and Tomahawk-4 (64-port 400G with 113MB buffers) switches [4].

[2] Defined as the maximum, across all sender-receiver pairs, time taken for a single MTU-sized packet to go from the sender to the receiver, and a 40-byte control packet to go from the receiver to the sender in the absence of any other packet in the network. This is a property of the network hardware.

Larger BDPs and relatively stagnant switch buffer sizes make packet drops more likely for best-effort delivery networks. Rapid increase in BDP and switch capacities coupled with relatively stagnant switch buffer sizes (Figure 1(right)) also suggest that increasing link bandwidths will make it easier to overwhelm switch buffers. As a result, best-effort delivery networks will experience an increasingly larger fraction of packets dropped within the network, resulting in even higher network unpredictability and reduced network throughput.

3 Harmony

This section presents Harmony, a distributed packet-switched datacenter architecture that guarantees bounded queueing at each switch in the network, while achieving near-optimal network throughput. We describe the Harmony protocol in §3.1, provide low-level details on Harmony design in §3.2, and establish theoretical properties of Harmony in §3.3.

3.1 Harmony Protocol

Harmony ensures bounded queueing at each switch by placing its intellectual roots in the classical Resource ReSerVation protocol (RSVP) [70, 71], and integrating it with another classical idea: virtual channels [23, 60].

We outline the Harmony protocol below, followed by an intuitive description of how it guarantees bounded queueing while achieving near-optimal throughput. Harmony protocol uses the following two constructs: host slots and virtual links.

Host slots. Each sender and receiver maintains K slots, each allowing a message to be sent and received using bandwidth B/K, where B is the access link bandwidth. Importantly, each slot can be allocated to at most one message; however, a message may be allocated more than one slot (that is, a sender-receiver pair may simultaneously use as many as K slots to send and receive a message).

Virtual links. Each physical link in the network is logically decomposed into virtual links, each of bandwidth B/K. Each virtual link can be allocated to at most one message, allowing the message to be transmitted using bandwidth B/K; however, a message may be allocated more than one virtual link.

Harmony protocol. The Harmony protocol is extremely simple, and works as follows:

- Each sender, immediately upon a message arrival, sends a `request` control packet to the receiver.
- Each receiver keeps track of its free slots. Upon receiving a `request`, if the receiver has one or more free slots, it assigns one of the slots to the message, updates the number of free slots, and sends a `rsvp` to the corresponding sender. If the receiver has no free slots, it adds the `request` to a list of `pending_requests` and starts a `timer` (using its local clock) for the `request`. Whenever a slot becomes free, the receiver picks the `request` with the largest `timer` value, assigns the slot to the message, updates its number of free slots, and sends a `rsvp` to the sender for this `request`.
- Whenever the `timer` for a `request` reaches $\delta_{\text{admission}}$ time, the receiver removes the `request` from the `pending_requests` list and sends a `reject` to the sender indicating that the receiver is unable to admit the message due to high load at the receiver.
- Each switch maintains, for each of its links, the number of free virtual links. Upon receiving a `rsvp`, the switch uniform randomly chooses a free outgoing virtual link along one of the shortest paths to the sender, embeds its identifier into the `rsvp` header, updates the number of free virtual links for the port, and forwards the `rsvp` on to that link. If no free virtual link is available, the switch transforms the `rsvp` into a `reject` packet, and sends it towards both the sender and the receiver using information in `rsvp` headers. `reject` traverses the same set of switches that forwarded `rsvp` so that corresponding links can be freed. Importantly, only shortest paths are used to ensure deadlock freedom.
- Each sender also keeps track of its free slots. Upon receiving a `rsvp`, if the sender has one or more free slots, it assigns one of the slots to the message, updates the number of free slots, and starts transmitting the message at a slot bandwidth. Once the message is finished, the sender sends a `complete` and marks the corresponding slot as free. Upon receiving a `rsvp`, if no free slots are available, the sender sends a `reject` to the receiver. All data and control packets are source routed using switch identifiers in the `rsvp` header.
- Each switch, upon receiving the `reject` or `complete`, marks the corresponding virtual link as free (using identifiers in the header), and forwards it to the next hop;
- Each receiver, upon receiving `reject` or `complete`, marks one of its slots as free; for `complete`, the receiver also sends a `complete` to the sender indicating completion of message transmission.
- Each sender, upon receiving a `reject` or `complete`, marks such for the message.

We describe additional design details for Harmony, including fast failure reaction, handling background (best-effort) delivery traffic, prioritization mechanisms for isolating messages from control packets and best-effort delivery traffic, optimizations like multi-slot allocation to individual messages, etc., in the next subsection. Below, we provide an intuitive description on how Harmony guarantees bounded queue at each switch and how it uses multiple virtual links and host slots to achieve near-optimal throughput. Figure 2 shows an example.

Intuitively understanding the invariants maintained by Harmony. Harmony maintains two invariants. First, for each outgoing link at each switch, the sum of arrival rates of all messages to be transmitted on the outgoing link is no more than the link bandwidth; and second, at all times, a small bounded

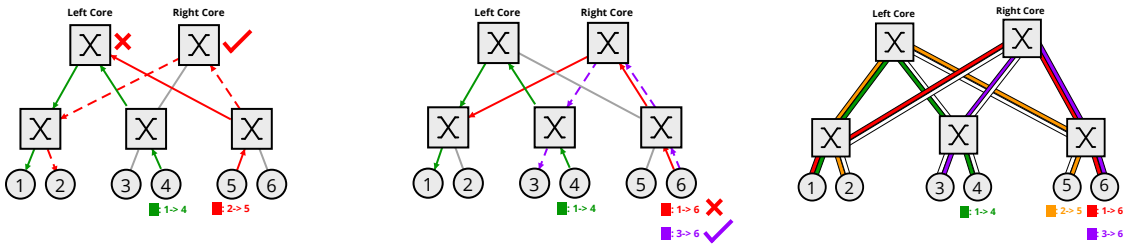


Figure 2: **Understanding benefits of virtual links and host slots. (left)** without multiple virtual links, network may be underutilized due to switches making uncoordinated decisions; and **(center)** without multiple host slots, network may be underutilized due to hosts making uncoordinated decisions; **(right)** using multiple virtual links and host slots, Harmony alleviates network underutilization. Discussion in §3.1.

number of messages use any link in the network. The first invariant ensures zero persistent queueing since the rate at which messages (destined to a particular outgoing link) arrive is at most the link bandwidth. However, while each message uses a different virtual link, multiple messages may now share a physical link; thus, data from multiple messages may arrive at the switch (via different virtual links) at the same time resulting in transient queueing. The second invariant ensures that transient queueing is bounded since a physical link is shared by at most as many messages as the number of virtual links (in §3.3, we will bound worst-case queueing in terms of link bandwidth, number of virtual links and maximum number of switches along any network path). Put together, these two invariants are sufficient for Harmony to guarantee bounded queueing at each switch in the network.

Intuitively understanding the necessity of virtual links. Virtual links in Harmony design are not merely a heuristic—they are key to Harmony achieving near-optimal throughput. Intuitively, in the above design, switches make decisions on forwarding `rsvp` based on purely local information, without any view of the state of links at neighboring switches. Switches making such uncoordinated decisions could lead to requests being rejected even if there is a path available in the network. For instance, consider the example shown in Figure 2(left): here, green message is using a reserved path $1 \rightarrow 4$; when receiver 5 sends a `rsvp` toward 2, its leaf switch uniform randomly chooses the left spine switch and forwards the `rsvp`. Since there is no unreserved outgoing link from the left spine switch to 2, the `rsvp` will be rejected. Had the leaf chosen the right spine switch, it would have successfully reached sender 2, improving network throughput. As shown in Figure 2(right), virtual links help alleviate unnecessary `rsvp` rejections by enabling fine-grained sharing of network resources. Intuitively, since an `rsvp` message can be forwarded along any of the virtual links, and there are more virtual links than physical links, the probability of an `rsvp` being unnecessarily rejected reduces significantly.

Intuitively understanding the necessity of host slots. Host slots, on the other hand, help alleviate network underutilization due to hosts making uncoordinated decisions. Specifically, since receivers do not have information about the network and/or sender state, an incorrect choice on messages to send `rsvp` for can lead to suboptimality: for the last `rsvp` sent by the receiver, it may receive a `reject` if there is no available path to the sender. This, in turn, will result in suboptimal throughput. An example is shown in Figure 2(center): here, receiver 6 has two outstanding requests, one from sender 1 (red) and one from sender 3 (violet). The receiver, however, can choose only one message to send `rsvp` to; if the receiver sends a `request` for 1, it will receive a `reject` since there is no unreserved path between 1 and 6. However, ideally the receiver should have selected 3 for `request` since there is an unreserved path between 3 and 6. As shown in Figure 2(right), with multiple host slots, receivers can now send `rsvp` for multiple messages again increasing the network throughput.

Intuitively understanding the benefits of $\delta_{admission}$. Harmony design allows requests to wait for time $\delta_{\text{admission}}$ at the receiver before they can be rejected. This parameter enables Harmony to handle bursty traffic and/or incasts—with higher $\delta_{\text{admission}}$ values, Harmony will be able to admit larger amounts of bursty traffic and/or larger incasts. Network operator can choose the right $\delta_{\text{admission}}$ to accommodate the desired bursty traffic and/or incast within Harmony.

3.2 Harmony Design Details

We now provide additional details on Harmony design.

Handling control packets. Harmony design uses a number of control packets—{`request`, `rsvp`, `reject`, `complete`}. These control packets utilize non-zero bandwidth, and could potentially interfere with message transmission at switches. To minimize impact of control packets on message transmissions, Harmony uses priorities, a standard technique for traffic isolation in modern datacenters [19, 27, 64]. Specifically, all messages are transmitted using the highest priority, all `rsvp` and `complete` messages are also transmitted using the highest priority, and all other control packets are transmitted using the second highest priority. In [5], we provide details on how this prioritization mechanism maintains Harmony queueing and delay bounds; intuitively, `rsvp` and `complete` packets transmitted using the highest priority does not impact message queueing much: these are tiny packets, and each link forwards a small number of `rsvp` and `complete` packets

(in the worst-case, as many as the number of virtual links). Furthermore, since Harmony bounds the switch buffer occupancy due to data packets at all times, and since control packets are small, Harmony can absorb large number of low-priority control packets within switch buffers—for instance, $16-32$MB buffer space in switches today can hold up to $\sim 400,000-800,000$ concurrent 40-byte control packets per switch; even with Harmony's worst-case queueing bound for data packets, we can still sustain $\sim 300,000-700,000$ concurrent control packets per switch. To ensure that low-priority control packets are not starved by data packets, Harmony also reserves a small amount of bandwidth at each host and switch for control packet transmission.

Exploiting bounded network delays in failure-free scenarios for fast reaction to failures and `reject`*s.* Harmony eliminates congestion-related drops in failure-free scenarios. However, inevitable hardware failures can still lead to data and control packet drops. Harmony uses the key insight that all data packets in Harmony traversing a bounded-queue path means that receivers know exactly when to expect data packets corresponding to a `rsvp` message in a failure-free scenario; similarly, senders know exactly when to expect `complete` or `rsvp` message in a failure-free scenario. Indeed, absence of any of these events must imply a hardware failure or a `reject`. Harmony exploits such a predictability in failure-free scenarios to trigger fast retransmission mechanism—as soon as a sender or a receiver infers a failure or a `reject` has happened, they trigger fast retransmission (receiver sending a `failure` control packet to the sender, and/or sender retransmitting data using a new `request` or using best-effort delivery interface, as described below).

Exploiting bounded network delays in failure-free scenarios for efficient multi-slot allocation to individual messages. To ensure high utilization in practice, Harmony assigns multiple host slots and virtual links to a single message when a receiver has less than K active messages. A message that has been allocated at least one slot is marked as an active message. Whenever receiver has a free slot and no pending requests, an active message is allocated additional slots; to ensure that additional slot allocation does not block future requests, additional slots are allocated for one BDP worth of data only. Harmony receivers always send one `rsvp` per slot, independent of which message the `rsvp` is for. When a new `rsvp` message is received by the sender for an active message, an additional virtual path of bandwidth B/K has been reserved and the sender can now send the message using an additional slot (increasing the transmission rate for this message by B/K). This ensures correctness, while also maximizing network throughput in practice [5].

Exploiting bounded network delays in failure-free scenarios for efficiently handling best-effort delivery traffic. Harmony can be integrated with any existing datacenter transport mechanism to handle best-effort traffic. The only constraint is that bounded-queue traffic and control packets must be isolated from best-effort traffic; Harmony thus uses the third priority level for best-effort traffic. Thus, the only impact best-effort traffic has on Harmony's timely delivery or bounded queueing properties is that latency can increase by as many packet transmission times as the number of switches in the path—each switch can incur one extra packet transmission time due to non-preemptive nature of today's switches (Harmony packet will be transmitted immediately after ongoing transmission of best-effort traffic packet).

We present additional details on Harmony design such as handling drops of control packets under failures, utilizing predictable network delays to optimize Harmony performance via pipelining `rsvp` packets, handling packet reordering, etc., in the technical report [5].

3.3 Harmony Theoretical Properties

We now establish theoretical properties of Harmony under a model similar to previous studies [19, 22, 45, 47, 55, 62].

Network model. For our theoretical analysis, we assume full-bisection bandwidth datacenter networks; this assumption is purely for ease of theoretical analysis—Harmony guarantees bounded queueing even for oversubscribed networks [5]. We model the network as a $N \times N$ crossbar switch, with N inputs (sender hosts) and N outputs (receiver hosts). Each input has a buffer of infinite capacity, and is partitioned into N virtual output queues (VOQ); each output also has a buffer of infinite size. The virtual output queue VOQ_{ij} holds packets arriving at input i and are destined for output j. It is well-known that, under the above assumptions, datacenter network transfers can be modeled as a bipartite matching problem [9, 19, 27, 55].

We assume that time is slotted. Packets are transferred from input buffers to output buffers in scheduling cycles. Each scheduling cycle consists of a matching phase and a data transfer phase. In the matching phase, an algorithm computes a matching between inputs and outputs in a manner that no input may be matched to more than one output, and no output is matched to more than one input. For each input i and output j matched to each other, one MTU-sized packet is removed from VOQ_{ij} and one MTU-sized packet is put into the output buffer of j during the data transfer phase (assuming VOQ_{ij} is non-empty). The network is said to have a speedup of s if, during each time slot, there are s scheduling cycles.

Let the average rate of packet arrival at VOQ_{ij} be λ_{ij}. The input traffic is said to be admissible if $\sum_i \lambda_{ij} < 1$ and $\sum_j \lambda_{ij} < 1$, that is, the load at each input and output is less than 1.

Fundamental limits. It is known that full-bisection bandwidth networks can achieve 100% throughput for any admissible input traffic as long as the underlying protocol computes a so-called maximum matching [22]. However, computing a maximum matching typically requires complex implementations; thus, existing switch fabrics and network protocols typically compute a so-called maximal matching [19, 27, 55].

A classical result establishes that any maximal matching based protocol requires a network with a speed up of 2 to achieve 100% throughput [22, 45] for all admissible input traffic.

Theoretical analysis for Harmony. We now provide bounds on queueing, network delays and throughput for Harmony.

Theorem 3.1 *Let #H be the number of switches along the longest path (across all sender-receiver pairs), K be the number of virtual links per physical link, B be the minimum bandwidth across physical links, and p be the maximum packet size. Then, maximum queueing observed by any packet at any switch in Harmony is bounded by*

$$Q_{Harmony} \leq \#H \cdot (K-1) \cdot p$$

Moreover, the total queueing delay (across all switches) incurred by any packet in Harmony is bounded by:

$$\delta_{queueing} \leq \frac{\#H \cdot (\#H+1) \cdot (K-1)}{2} \cdot \frac{p}{B} \qquad (1)$$

We provide the full proof in [5]. Intuitively, given that we have K virtual links sharing the physical link, we would expect a queue bound of $(K-1) \cdot p$; however, our proof demonstrates that, in the worst-case, packet transmissions across multiple switches can (mis)align in a manner that the worst-case queueing bound worsens with more switches along the path. The bound on queueing delay follows by aggregating the precise amount of queueing delay at each switch along the path. These bounds hold independent of the underlying network topology (since they are parameterized by number of switches along the path and K).

Let $S^\star$ be the minimum buffer size across all switches, let $\#\ell$ be the number of ports at the switch, and let $K^\star = S^\star/(\#H \cdot p \cdot \#\ell)$. Then, the first part of the theorem shows that for $K \leq K^\star$, Harmony will ensure that queueing at each switch never grows beyond the switch buffer size, and that congestion-related drops are eliminated. For modern datacenter hardware, $K^\star$ turns out to be large enough for all practical purposes. For instance, for a datacenter organized around a FatTree topology using 100Gbps Tomahawk switches (32 ports and 22MB switch buffer size) and with 1.5KB packet sizes, we get $K^\star \approx 91$. For a FatTree topology with 400Gbps Tomahawk-4 switches (64 ports and 113MB buffers), we get $K^\star = 235$.

Theorem 3.1 not only guarantees that congestion-related drops are eliminated for $K \leq K^\star$, but also bounds the queueing delay seen by any packet: for the above two topologies and for $K = 8$, we get that $\delta_{\text{queueing}} = 12.6\mu$s and $\delta_{\text{queueing}} = 3.15\mu$s, respectively. Thus, Harmony guarantees tiny worst-case queueing delays. The bound on queueing delays can be trivially used to bound the delay between the sender starting to transmit a message m (upon receiving an `rsvp`) until it receives the `complete` for the message [5]. Extending Harmony design to guarantee end-to-end delays (that is, including processing delays at the sender and at the receiver, and queueing delays at the sender) is an intriguing open question (§6).

Harmony uses a distributed protocol for allocating host slots and virtual links to senders and receivers, with hosts and switches making decisions based on purely local information. It, thus, becomes an interesting question to characterize the quality of successful allocations—requests for which the receiver sends a `rsvp` that is not rejected within the network. The following theorem characterizes the slot and virtual link allocation efficiency of Harmony for a full-bisection bandwidth leaf-spine topology with respect to an ideal centralized algorithm that performs K maximal matching for allocating host slots, and perfectly allocates virtual links to matched slots (see [5] for a basic extension of [55] that realizes such an ideal centralized algorithm). Let $\Theta^\star$ denote the number of successful slot allocations by such an ideal algorithm.

Theorem 3.2 *Let K be the number of virtual links per physical link and C be the number of core switches in the topology. Then, the expected number of successful slot allocations by Harmony, randomized over choice of* `rsvp` *forwarding decisions, is:*

$$\mathbb{E}[\Theta_{Harmony}] = f(K,C) \cdot \Theta^\star, \qquad \textit{where,} \qquad (2)$$

$$f(K,C) = \frac{\sum_{i=0}^{KC} \min(i,K) \binom{KC}{i} \left(\frac{1}{C}\right)^i \left(1-\frac{1}{C}\right)^{(KC-i)}}{K}$$

We provide a full proof in Harmony technical report [5]. At a high-level, our proof for Theorem 3.2 uses an argument similar to the classical balls-and-bins problem [59] to establish the slot allocation efficiency in Harmony. This result, similar to most prior analytical results [19, 55], assumes two-tier full-bisection bandwidth leaf-spine network topology; it is an intriguing open question to generalize our bounds to FatTree, expander-based and oversubscribed network topologies. Nevertheless, this result provides us several insights on the behavior of Harmony. First, for the special case of $K = 1$ (which is roughly equivalent to RSVP adapted to datacenter networks), the above expression leads to $\mathbb{E}[\Theta_{\text{Harmony}|K=1}] = (1-1/e)\Theta^\star$, where $e \approx 2.72$ is base of the natural logarithm. That is, for $K = 1$, the slot allocation efficiency is merely ~63% of the optimal. Second, Figure 4 confirms the intuition discussed earlier: increasing the number of virtual links and host slots (increasing K) results in significantly improved efficiency of Harmony, converging to the optimal $\Theta^\star$. Harmony evaluation in §5.1 demonstrates that, in practice, Harmony achieves significantly better performance even for small values of K.

4 Harmony Implementation

We have done prototype implementation of Harmony using readily available programmable switches and DPDK-based hosts. This section provides some of the most interesting implementation details. We provide more details in [5].

The Harmony interface. Harmony currently implements a RPC interface [38, 66]: messages are submitted to a submission queue and the message delivery status is updated in

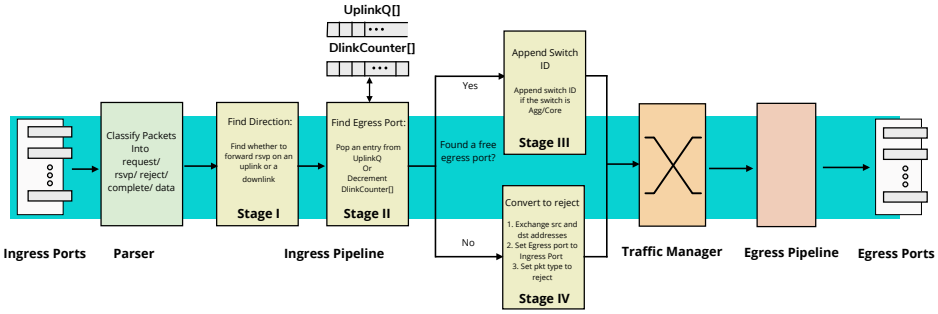


Figure 3: Harmony can be realized using programmable switches supporting PSA architecture. More description on §4.

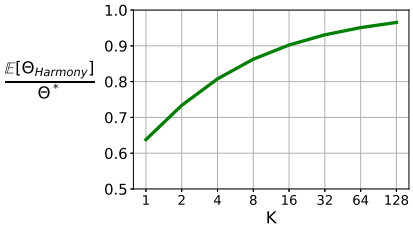


Figure 4: Harmony's performance in terms of successful host slot allocations—$\mathbb{E}[\Theta_{Harmony}]$—converges to that of an ideal centralized maximal matching based algorithm with increasing values of K. We use a 64-port leaf-spine full-bisection bandwidth topology with 32 core switches in this figure.

the completion queue (using a `complete` or `failure` flag). The only difference between Harmony and standard RPC interfaces is that the completion queue in Harmony has one additional flag: `reject`, which indicates that Harmony is unable to guarantee bounded queueing for that message (*e.g.*, due to failures, high receiver load, high network load, etc.). If the flag is either `reject` or `failure`, the message can be resubmitted using the Harmony interface, or using the standard best-effort delivery interface.

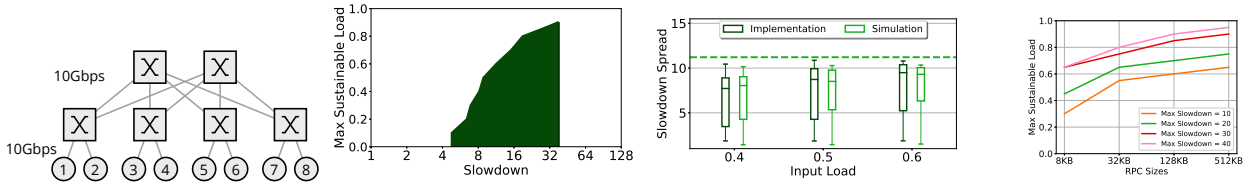
Harmony host implementation. Harmony maintains the invariant of bounded queueing at switches thus alleviating the need for congestion control on the fast path at hosts; as a result, Harmony can be easily integrated with existing high-performance userspace or in-kernel network stacks [21,38,46], as well as existing accelerator based network stacks [2,26,58]. To implement Harmony-specific functionality, much of our host implementation uses modules from existing network stacks. For example, generating and responding to `rsvp` messages is similar to the grant mechanism in receiver-driven transport protocols [19, 27, 32, 33, 51]. As another example, Harmony design requires senders to transmit messages at a rate that is dependent on the number of slots allocated to that message; such a rate limit functionality at the sender is already implemented in almost all network stacks. Overall, our Harmony host implementation uses ∼3107 lines of code.

Harmony switch implementation. Harmony switch implementation uses programmable switches [3]. The data plane of these switches employs Portable Switch Architecture [3], and is composed of a parser, an ingress and an egress pipeline and a traffic manager (Figure 3). Upon a packet arrival on an ingress port, Harmony parser extracts the header, identifies the packet type (`rsvp`, `reject`, `complete`, or data packet), and forwards the packet to the ingress pipeline. The ingress pipeline decides the egress port to forward the packet to, embeds switch identifiers within the packet header (using techniques from path tracing mechanisms on programmable switches [67, 68]), and then passes the packet to the traffic manager (which then forwards the packet to the desired egress port). Harmony's implementation of the ingress pipeline uses two data structures: A FIFO `UplinkQ`, and a counter `DlinkCounter`. `UplinkQ` maintains the available virtual links which can be reserved by `rsvp` packets while traversing uplink (toward the spine, or the core switch). `DlinkCounter` maintains the number of reserved virtual links at the downlink path (away from the core switches). In addition, Harmony switches maintain a small constant amount of state to avoid blocking of virtual links during control packet drops [5].

5 Harmony Evaluation

We now evaluate Harmony performance over a small-scale testbed (§5.1), and over large-scale packet-level simulations (§5.2). We start by describing our evaluation setup.

Network Topologies. We use a testbed with 8 servers organized along a two-tier topology with 10Gbps links (as shown in Figure 5a). For our simulations, we use the same default setup as in [9, 19, 27, 44]: the standard 144-node leaf-spine topology with 9 top-of-rack (ToR) switches, each connected to 16 hosts with 100Gbps access link bandwidth. The propagation and switching delay are 200ns and 450ns, respectively. We use switch buffer capacity of 32MB. The unloaded RTTs and bandwidth-delay product values for this topology are 4.6μs and 56KB, respectively. We also use oversubscribed and FatTree topologies. For the former, we vary the oversubscription (1:1 to 2:1 and to 4:1) in our leaf-spine topology by correspondingly reducing the leaf-spine link bandwidths; we reduce the load based on oversubscription factor to ensure that the traffic remains admissible. Our three-tier FatTree topol-



(a) Testbed Topology (b) Feasible region (c) Slowdown Spreads (d) Varying RPC Sizes

Figure 5: **Evaluation of Harmony implementation on a small-scale testbed.** (a) The testbed used in this evaluation; (b) Harmony's feasibility region in terms of achievable delays and maximum sustainable loads (the green dotted line represents Harmony's delay bound for this topology); (c) Harmony evaluation over the testbed almost perfectly matches the theoretical bounds, as well as simulation results in Harmony's packet-level simulator; (d) for a fixed delay bound, Harmony sustains larger loads with larger RPC sizes (as expected). Discussion in §5.1.

ogy contains 1024 hosts, 64 core switches, 128 aggregation switches, 128 ToRs and 100Gbps access link bandwidth. We use the same propagation delay, switching delay and switch buffer size as above. The unloaded RTT and bandwidth-delay product for this topology are 7.6μs and 93KB, respectively.

Evaluated schemes. Existing distributed datacenter transport protocols are not designed to achieve congestion-free communication; thus, comparison with Harmony would be unfair to any choice of protocols. Nevertheless, we evaluate Harmony performance against state-of-the-art sender-driven (HPCC [44]) and receiver-driven (dcPIM [19]) datacenter transport protocols. We use the default parameters from HPCC and dcPIM papers. We also extend the centralized Fastpass scheduling algorithm to support message-level scheduling[3] to provide an ideal baseline for Harmony. Unless specified otherwise, we use the following values for Harmony parameters: $K = 8$ and $\delta_{\text{admission}} = \sim 63\mu$s (a factor 5 of Harmony worst-case queueing delay bound); we also perform sensitivity analysis against these two Harmony parameters.

Workloads and traffic patterns. We used 128KB RPCs (as in [49, 72]) as our default workload since it allows us to provide in-depth insights due to each message having the same latency bound, but also present results for RPC sizes varying from 0.5KB to 512KB. For evaluation of Harmony with background (best-effort) traffic, we use a mix of RPCs (for traffic desiring congestion-free guarantees), and the standard web-search datacenter workload [9, 19, 27, 44] (for the best-effort traffic). We use the standard methodology of generating message arrival times using Poisson arrival process [8, 9, 19, 27, 44] and use an all-to-all traffic pattern by default. To avoid degenerate scenarios, we ensure that the generated load for each sender and each receiver is less than 1 for every $\delta_{\text{admission}}$ duration of time.

Evaluation metrics. Harmony guarantees bounded queueing at each switch in the network, thus ensuring bounded queueing delays for each message. However, Harmony does not bound processing delays at the sender and at the receiver, and queueing delays at the sender. Nevertheless, in our evaluation, we use Harmony queueing delay bound as the overall end-to-end bound. We use two metrics—maximum sustainable load (defined as the input load for which no message is rejected) and latency. For latency, we use the standard slowdown metric, defined as the ratio of the message completion time under loaded and unloaded scenarios; we present slowdown spread—{minimum, mean, 99%-ile, 99.9%-ile, and maximum}—using lower whisker, lower box edge, mid-line in the box, upper box edge, and upper whisker, respectively.

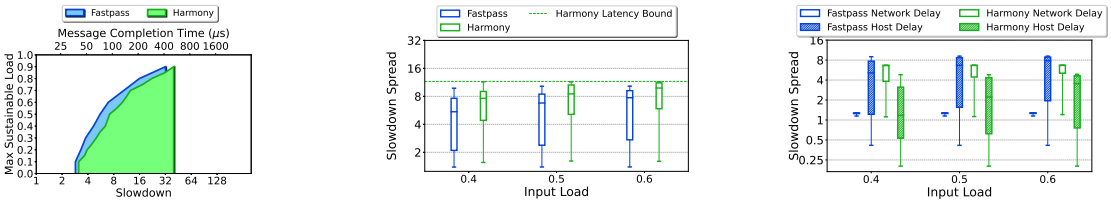
[3]The original Fastpass algorithm performs per-packet scheduling; this provides delay guarantees for each individual packet, but not for each message as in Harmony. We make two extensions: (a) to enable message-level delay guarantees, we ensure that all packets for a message are scheduled in consecutive timeslots; and (b) to enable a fair comparison in terms of maximum sustainable load, we allow a message request to wait for $\delta_{\text{admission}}$ time before it is rejected.

5.1 Harmony Testbed Evaluation

We start with evaluation of Harmony implementation over a small-scale testbed. Our testbed results are primarily to demonstrate the feasibility of Harmony—it is an extremely simplified setup with no contending applications, dedicated cores to ensure minimal host processing delays, no host congestion, etc.; more work is needed to evaluate end-to-end performance of Harmony over large-scale deployments.

Our first result, shown as the shaded region in Figure 5b, shows that Harmony enables a unique trade-off space between the delay bound and maximum sustainable load. Specifically, the two parameters in Harmony—K and $\delta_{\text{admission}}$—allow it to tradeoff the delay bound for higher maximum sustainable loads. Here, we vary K and $\delta_{\text{admission}}$, and plot the pareto curve on the observed worst-case delay and sustainable loads. This tradeoff essentially follows the intuition discussed earlier—with a larger bound on target message delay (via increasing K and $\delta_{\text{admission}}$), Harmony sustains higher loads. Figure 5d shows this tradeoff for varying RPC sizes: we observe that Harmony can sustain as high as 0.9 load for 128KB RPCs, and 0.65 load for 8KB RPCs.

We also use our testbed to verify the simulator fidelity; in particular, we incorporate the measured testbed parameters (link propagation, switching and average PCIe delays) into our simulator and show the corresponding results alongside the testbed results in Figure 5c. The testbed latency results near-perfectly match our simulator (which, in turn, matches the analytical bounds in Theorem 3.1 and Theorem 3.2).



(a) Feasible region (b) Slowdown spreads (c) Harmony latency breakdown

Figure 6: **Evaluation of Harmony and Fastpass using large-scale simulations**. Harmony achieves feasibility region surprisingly similar to Fastpass—for any given latency bound, it can sustain loads as high as Fastpass (and vice versa). Discussion in §5.2.

5.2 Harmony Large-Scale Simulation

We now evaluate Harmony over simulations, both in absence (Figure 6) and in presence (Figure 7) of best-effort delivery traffic. For the former, we compare Harmony with Fastpass; for the latter, we compare Harmony with HPCC and dcPIM.

[Figure 6, no best-effort delivery traffic]. Figure 6a shows maximum sustainable loads for varying target delay bound (or, slowdown) for both Harmony and Fastpass; each point in the feasibility region of Figure 6a is achieved using a combination of K and $\delta_{\text{admission}}$.

We observe that, despite using a purely distributed design, Harmony's feasibility region closely matches that of Fastpass. This result may be surprising. Digging deeper, we found that this result is due to Fastpass and Harmony making significantly different tradeoffs. Fastpass enforces zero queueing at each switch by transmitting each message on a dedicated pre-reserved path—the arrival rate for each message thus perfectly matches the outgoing bandwidth available for that message at each switch. However, this invariant—enforcing zero queueing at each network switch—incurs high overheads for a distributed protocol like Harmony. Instead, the key insight in Harmony design is that it is feasible to achieve bounded (but not necessarily zero) queueing at each switch using a completely distributed design, while maintaining high throughput. Figure 6c provides more insights on how Harmony achieves performance so close to an ideal centralized architecture even with non-zero queueing at switches. Essentially, by enabling bounded queueing (rather than zero queueing) within the network, Harmony significantly reduces the amount of time a message spends waiting at the sender before it can be scheduled and can start transmitting packets; this reduced host-side delay easily compensates for the increased queueing delay in the network when compared to Fastpass. Overall, this turns out to be a good tradeoff—Harmony, despite its distributed nature, achieves delays and maximum sustainable loads comparable to an ideal centralized datacenter architecture.

Harmony does make a different tradeoff compared to Fastpass: as shown in Figure 6b, it achieves $\sim 2\times$ higher mean and $\sim 1.3\times$ higher P99 latencies, while enjoying the benefits of a distributed datacenter architecture.

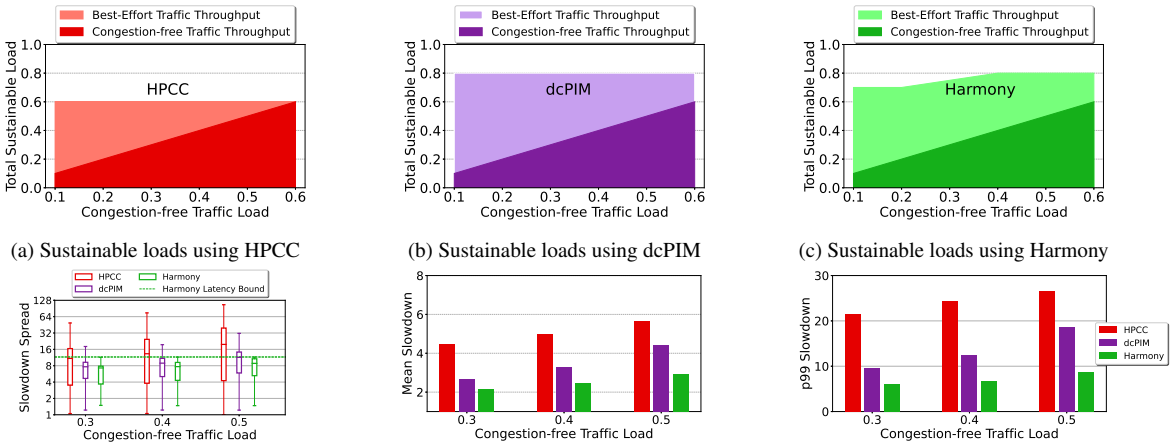
[Figure 7, Harmony performance with best-effort traffic]. We now discuss Harmony performance for scenarios where congestion-free traffic (using 128KB RPCs) coexists with best-effort delivery traffic (generated using websearch workload). We vary the load of congestion-free traffic from 0.3 to 0.5, and respectively vary the load of co-existing best-effort delivery traffic from 0.3 to 0.1, such that the total offered load is 0.6 and hence remains sustainable for all protocols.

Figure 7 shows a surprising result: when compared to existing state-of-the-art sender-based and receiver-based best-effort delivery protocols, Harmony sustains similar or higher loads for best-effort traffic (with varying congestion-free traffic load, as shown in top row), achieves significantly better tail latencies for congestion-free traffic, and significantly better mean and tail latencies for best-effort traffic! Upon digging deeper into this surprising result, we found the following two interesting phenomena.

First, as discussed earlier, Harmony ensures that the rate at which congestion-free traffic arrives at any switch perfectly matches the bandwidth available to forward this traffic; as a result, congestion-free traffic incurs a small amount of queueing at each switch. The result is that, despite lower priority, best-effort delivery traffic experiences minimal contention with congestion-free traffic in Harmony. For other protocols, however, these two traffic create queueing for each other resulting in lower sustainable loads and higher latencies. The problem exacerbates at higher loads.

Second, for HPCC, we observed that RPC traffic creates congestion in the network core despite a full-bisection bandwidth topology. This is because the transmission time for a 128KB RPC on a 100Gbps link is $\sim 10.2\mu$s, which is $\sim 2.2\times$ the unloaded network RTT. Thus, even a small amount of queueing can result in congestion control being ineffective—by the time congestion signal arrives, the network condition has changed. For dcPIM, we found that 128KB RPCs were larger than a single BDP and thus required matching before they can start transmitting; this leads to larger mean and tail latencies. Harmony avoids both of these problems—it ensures congestion-free network for the RPC traffic, and minimizes queueing experienced by best-effort delivery traffic.

[Figure 8, Sensitivity analysis for Harmony]. Figure 8 demonstrates that Harmony performance is robust across all evaluated scenarios—as long as the load is sustainable, Harmony achieves low tail slowdown and bounded worst-case message delays. We discuss three important observations.



(a) Sustainable loads using HPCC

(b) Sustainable loads using dcPIM

(c) Sustainable loads using Harmony

(d) Slowdown spread (Congestion-free traffic)

(e) Mean slowdowns (Best-effort delivery traffic)

(f) 99p slowdowns (Best-effort delivery traffic)

Figure 7: **Evaluation of Harmony with background (best-effort delivery) traffic.** Figures in the top row show maximum sustainable loads for traffic desiring best-effort delivery (lighter color) and traffic desiring congestion-free guarantees (darker color) for HPCC, dcPIM and Harmony. Bottom row shows the slowdown spread for congestion-free traffic (left), mean latency for best-effort traffic (center) and tail latency for best-effort traffic (right).

First, as expected, Harmony sustains low load for small-sized RPCs; for instance, Figure 8a shows that when RPC size is 4KB (that is, each RPC is $\sim$14$\times$ smaller than the network BDP), Harmony requires a target delay that is 20$\times$ larger than the unloaded RTT to sustain 0.5 load. For such small RPCs, modern best-effort delivery datacenter transport protocols will simply send all the packets within the first RTT, rendering congestion control essentially irrelevant. As a result, depending on the network load, network topology, oversubscription ratios, load balancing mechanisms and/or traffic patterns, queueing delays may or may not dominate network delays; thus, Harmony may or may not be the best choice for such small transfers. Recent studies from datacenter networks suggest that RPCs are much larger in practice (*e.g.*, a recent study from production datacenters demonstrates that median RPC size is greater than 40KB [11]; similar observations have been made in prior studies [49, 72]); for such real-world scenarios, Harmony guarantees small worst-case delays while sustaining high loads.

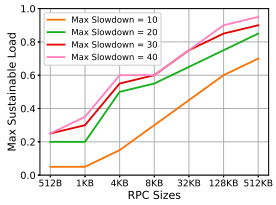
Second, Harmony maintains its performance for oversubscribed topologies. This is not surprising—as discussed in §3.3, for all sustainable loads, Harmony delay bounds hold even for oversubscribed topologies. This is because each message is still transmitted over a dedicated set of virtual links. Moreover, the worst-case queueing delay bounds remain near-identical with increasing oversubscription because (i) the maximum possible per-hop queueing decreases; and, (ii) the per-packet transmission delay increases, by roughly the same amount (the delay bound now uses the minimum link bandwidth in the topology). These two factors essentially balance out resulting in near-identical delay bound. Finally, since load is now defined with respect to the minimum link bandwidth in the topology, Harmony is also able to sustain similar loads as in full-bisection bandwidth topologies.

We present results for sensitivity analysis of Harmony performance over the two Harmony parameters—K and $\delta_{\text{admission}}$—in [5]. To briefly summarize, the results confirm the intuition from our theoretical analysis: with increase in K, Harmony delay bounds are higher but maximum sustainable load is also higher. Increase in $\delta_{\text{admission}}$ has the same effect; however, $\delta_{\text{admission}}$ has lower impact than K in terms of the magnitude of change in delay bounds and maximum sustainable load. Moreover, since these parameters can be set independently, any operating point in the feasibility region for Harmony (shown in Figure 6) can be achieved.

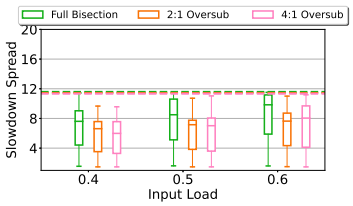
6 Harmony benefits

Harmony enables bounded queueing and network delays, and eliminates congestion-related drops for each message. This leads to some immediate benefits. For instance, Harmony has the potential to enable efficient RDMA over converged Ethernet deployments. Specifically, most current RDMA deployments use Priority Flow Control (PFC), a mechanism to enable lossless network fabrics (no buffer overflows and packet drops within the network fabric) [1, 44, 49, 72]. While PFC enables lossless network fabrics, it suffers from several undesirable problems: head-of-line blocking, inevitable deadlocks, congestion spreading, reduced network utilization and increased tail latencies [34–36, 44, 50, 72]. Harmony, by design, eliminates congestion-related drops thus enabling lossless networks without the need for PFC.

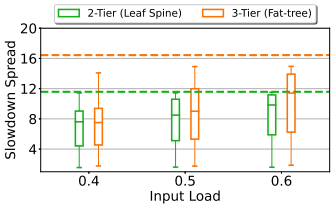
However, to truly realize Harmony benefits, Harmony must be extended to provide end-to-end delay guarantees. Specifically, we need extensions in Harmony to bound host-side processing delays (*e.g.*, due to slow software [20]), and to



(a) Varying RPC sizes



(b) Varying Oversubscription



(c) Varying Topology Types

Figure 8: **Sensitivity of Harmony performance with (a) varying RPC sizes (b) varying oversubscription ratio in network topologies; and (c) varying scale of topologies (two-tier versus three-tier)**. Discussion in §5.2.

eliminate packet queueing and delays at the host (*e.g.*, due to host congestion [7,31]). To bound host-side processing delays, we observe that Harmony maintains the invariant of bounded queueing at switches thus allowing to move congestion control out of the fast path; as a result, Harmony can be easily integrated with existing high-performance userspace network stacks [38,46], in-kernel network stacks [21], and accelerator based network stacks [2,26,58]. These stacks often use dedicated cores and/or specialized hardware for packet processing, reducing host processing overheads to a tiny fraction of the network RTT. To eliminate queueing and packet drops at the host due to host congestion, Harmony can be easily integrated with recent mechanisms for host congestion control [7]. Finally, to extend Harmony's bounded delay guarantees all the way to the application layer, Harmony can be integrated with μs-scale network, CPU and storage schedulers [21,37,53]. We leave these extensions to the future; however, we demonstrate several applications that may benefit from such extensions.

To demonstrate potential application-layer benefits of Harmony, we use the DPDK-based prototype implementation of Harmony from §4, and use three dedicated cores—one for pacing data, one for control packet processing, and one for implementing the remainder of the logic—to minimize host processing delays. We integrate this implementation with several applications, while ensuring that network layer processing is isolated and host does not observe congestion. In extremely rare scenarios where Harmony incurs worst-case network-layer delays as well as host processing delays, applications may get a `reject` message—this does not violate any correctness guarantees, and simply requires applications to retry; nevertheless, to account for these delays, we simply add 1μs to Harmony's end-to-end delay bound. This prototype implementation thus offers bounded message delays between sender-side and receiver-side Harmony interfaces. We integrate it with three applications, that we discuss next.

CPU-efficient storage stacks. Disaggregated storage has become common in today's datacenters. As a result, modern storage stacks have been integrated with network transports (e.g., NVMe-over-Fabrics) in order to facilitate access to remote storage devices. Today's storage stacks rely on one of the two designs: polling-based or interrupt-based mechanisms. Polling-based designs provide extremely good latency when applications are run in isolation, but suffer when applications share CPU resources. Interrupt-based designs work better in the shared scenario, however have suboptimal CPU-efficiency due to frequent context-switches. Harmony's predictable delay guarantees enable a new point in the design space of CPU-efficient storage stacks: improved CPU efficiency without sacrificing tail latency. Specifically, storage stack running atop Harmony can use predictable network delays in Harmony to avoid both interrupts or polling—applications can be scheduled more precisely using the delay bound. In [5], we demonstrate the feasibility using the Harmony prototype: we find that Harmony enables new operating points in terms of tail latency and system throughput.

Efficient host packet processing pipelines. Recent work [20, 28] has demonstrated that CPU-efficiency of host packet processing pipelines can degrade significantly when multiple applications/connections running on a single CPU core contend for cache capacity, leading to higher cache misses. Recent work on Reframer [28] tries to reduce these misses by deliberately buffering packets, and waiting for a fixed amount of time for a batch worth of packets to arrive per application, before allowing them to be processed by the CPU. The choice of timeout value presents an inherent tradeoff: a larger timeout value results in better CPU-efficiency due to more packets getting batched at the cost of higher latency, while a smaller timeout value results in lower latency but worse CPU-efficiency. Determining the ideal timeout value is difficult in best-effort networks due to unpredictable network delays. Harmony's predictable network delay guarantees enable overcoming the above tradeoff: in absence of failures, receivers receive data at a fixed bandwidth, allowing to set a timeout to reap the maximum benefits of Reframer. Evaluation of our prototype, integrated with Reframer, achieves 2$\times$ higher throughput than the original Reframer [5].

Efficient failure detectors. Detecting host failures is a fundamental problem in distributed systems. Designing a failure detector that is reliable (i.e. provides small false positive probability) and fast (provides failure notification within a small delay) is a hard problem, especially in best-effort networks where network delays are unpredictable [43]. Harmony, using its bounded delay guarantees, has the potential to allow

achieving both these goals simultaneously. In [5], we present a prototype failure detection built on top of Harmony. The failure detector probes the target host by issuing a `request` and waiting for an `rsvp`. Given Harmony's predictable delay guarantees, the failure detector knows exactly when to expect the corresponding `rsvp` message. The absence of such a message implies there was a failure, either at the host or in network hardware. We show analytically that, for modern datacenter networks, the probability of the probe encountering a network failure is relatively small (and decreases with the number of tries) due to large path diversity; based on this, we are able to demonstrate the feasibility of designing low-latency host failure detectors that also have low false positive rates.

7 Related Work

Our exploration of congestion-free datacenter architecture is related to three key areas of research.

Pre-datacenter network designs (RSVP, virtual circuit switching, ATM networks, etc.). There have been several attempts to designing Internet architectures with predictable performance, *e.g.*, using Resource ReSerVation protocol (RSVP) [70, 71], virtual circuit switching [12, 40, 56], and hop-by-hop flow control in ATM networks [17, 54], to name a few. Realizing predictable performance on the Internet faced several challenges: large RTTs, the lack of a single administrative entity precluding support from hosts and routers, and potential deadlocks due to policy-driven routing [17, 54]. These challenges proved to be insurmountable in the Internet context; however, the equation is quite different for modern datacenter networks: they support small RTTs, operate within a single administrative domain allowing us to leverage both host and switch support, and are already exploring programmable switches and custom-designed network interface cards (NICs) with more powerful interfaces than commodity hardware. Our work builds upon decades of work on predictable Internet architectures, but advances them significantly: combining the idea of virtual channels with RSVP to avoid throughput loss in datacenter networks and presenting analytical bounds on bounded queueing in the datacenter context.

Circuit-switched networks. Circuit-switched datacenter network architectures [10, 25, 42, 48, 57, 63], by establishing an end-to-end dedicated path prior to data transmission, enable bounded queueing at each switch in the network. Our goals are aligned with those in circuit-switched networks; unsurprisingly, some of our ideas resemble the techniques used in circuit-switched networks, *e.g.*, wavelength-switching [14, 16, 52] and packet-based optical switching [24, 30]. However, there are two—fundamental—differences. First, our work demonstrates that it is possible to achieve powerful guarantees using distributed packet-switched networks, the primary deployment scenario in today's datacenter networks; recent results [10] demonstrate that it is impossible to simultaneously achieve low latency and high network utilization for all workloads using distributed circuit-switched networks. Second, unlike circuit-switched networks that assume host and network hardware clocks to be perfectly synchronized (that is known to be hard at the datacenter scale [69]), our design does not make any assumptions on clock synchronization.

Predictable and low-latency datacenter network designs. Most of the existing datacenter network designs focus on best-effort delivery [6, 8, 9, 19, 27, 29, 32, 33, 44, 51]. There are two exceptions. The first exception is the recent work on lossless network designs [1, 44, 72]; while these techniques ensure that packets are never dropped due to buffer overflow, by design, they can suffer from packet stalls—packets can be queued in switch buffers for an unpredictable amount of time due to PFC pause frames [1, 44]. Thus, they do not guarantee bounded network delays. Harmony enables lossless networks by design (in failure-free scenarios), thus offering a way to realize RDMA over converged Ethernet without PFC.

Another closely related line of work [15, 39, 55, 69] is on predictable network performance using centralized schedulers. These designs suffer from the usual centralized design limitations, namely scalability and availability, especially for high-bandwidth links. Harmony focuses on achieving predictable network performance over distributed packet-switched networks. Indeed, our evaluation results suggest that Harmony achieves performance similar to centralized designs while enjoying the benefits of a distributed datacenter architecture.

8 Conclusion

Existing datacenter networks provide best-effort delivery, that is, messages may experience unpredictable queueing, delays, and congestion-related drops due to switch buffer overflows within the network. System designers are thus forced to rely on minimal assumptions from the network, resulting in inefficiency and complexity in host hardware and software. This paper argues for datacenter architectures that provide stronger guarantees by design. We have presented Harmony, a datacenter architecture that provides congestion-free message delivery guarantees—each message experiences bounded queueing at each switch in the network. Harmony thus ensures that network delays are bounded in failure-free scenarios, and that congestion-related drops are eliminated. We establish that Harmony provides these properties with near-zero overheads compared to best-effort delivery networks.

Acknowledgements

We would like to thank our shepherd, Ravi Soundararajan, and NSDI reviewers for insightful feedback. We would also like to thank Midhul Vuppalapati for many useful discussions during this project. This research was in part supported by NSF grants CNS-2047283 and a Sloan fellowship.

References

[1] 802.1Qbb – Priority-based Flow Control . https://1.ieee802.org/dcb/802-1qbb/.

[2] Annapurna Labs. http://www.annapurnalabs.com.

[3] Portable Switch Architecture (PSA) . https://p4.org/p4-spec/docs/PSA.html.

[4] Switch Buffer Size. https://people.ucsc.edu/~warner/buffer.html.

[5] Harmony Technical Report. https://github.com/communication-harmony/tech-report.

[6] V. Addanki, O. Michel, and S. Schmid. PowerTCP: Pushing the performance limits of datacenter networks. In *USENIX NSDI*, 2022.

[7] S. Agarwal, A. Krishnamurthy, and R. Agarwal. Host congestion control. In *ACM SIGCOMM*, 2023.

[8] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *ACM SIGCOMM*, 2011.

[9] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *ACM SIGCOMM*, 2013.

[10] D. Amir, T. Wilson, V. Shrivastav, H. Weatherspoon, R. Kleinberg, and R. Agarwal. Optimal oblivious reconfigurable networks. In *ACM STOC*, 2022.

[11] S. Arslan, Y. Li, G. Kumar, and N. Dukkipati. Bolt: Sub-rtt congestion control for ultra-low latency. In *USENIX NSDI*, 2023.

[12] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line Routing of Virtual Circuits with Applications to Load Balancing and Machine Scheduling. In *JACM*, 1997.

[13] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS*, 2012.

[14] I. Baldine, G. N. Rouskas, H. G. Perros, and D. Stevenson. JumpStart: A Just-in-time Signaling Architecture for WDM Burst-switched Networks. In *IEEE communications magazine*, 2002.

[15] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predictable Datacenter Networks. In *ACM SIGCOMM*, 2011.

[16] D. Banerjee and B. Mukherjee. Wavelength-routed Optical Networks: Linear Formulation, Resource Budgeting Tradeoffs, and a Reconfiguration Study. In *IEEE/ACM ToN*, 2000.

[17] C. Basso, J. Calvignac, D. Orsatti, and F. Verplanken. Hop-by-hop Flow Control in an ATM Network, 1998. US Patent.

[18] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *ACM IMC*, 2010.

[19] Q. Cai, M. T. Arashloo, and R. Agarwal. dcPIM: Near-optimal Proactive Datacenter Transport. In *ACM SIGCOMM*, 2022.

[20] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal. Understanding Host Network Stack Overheads. In *ACM SIGCOMM*, 2021.

[21] Q. Cai, M. Vuppalapati, J. Hwang, C. Kozyrakis, and R. Agarwal. Towards μs Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. In *ACM SIGCOMM*, 2022.

[22] J. G. Dai and B. Prabhakar. The throughput of data switches with and without speedup. In *IEEE INFOCOM*, 2000.

[23] W. J. Dally, P. P. Carvey, L. R. Dennison, and P. A. King. Router with Virtual Channel Allocation, 2003. US Patent.

[24] T. S. El-Bawab and J.-D. Shin. Optical Packet Switching in Core Networks: between Vision and Reality. In *IEEE Communications Magazine*, 2002.

[25] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: a Hybrid Electrical/optical Switch Architecture for Modular Data Centers. In *ACM SIGCOMM*, 2010.

[26] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, et al. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX NSDI*, 2018.

[27] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. pHost: Distributed Near-optimal Datacenter Transport Over Commodity Network Fabric. In *ACM CoNEXT*, 2015.

[28] H. Ghasemirahni, T. Barbette, G. Katsikas, A. Farshin, A. Girondi, Massimoand Roozbeh, M. Chiesa, G. Maguire, and D. Kostic. Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets. In *USENIX NSDI*, 2022.

[29] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues Don't Matter When You Can Jump Them! In *USENIX NSDI*, 2015.

[30] C. Guillemot, M. Renaud, P. Gambini, C. Janz, I. Andonovic, R. Bauknecht, B. Bostica, M. Burzio, F. Callegati, M. Casoni, et al. Transparent Optical Packet Switching: The European ACTS KEOPS Project Approach. In *Journal of lightwave technology*, 1998.

[31] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2013.

[32] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *ACM SIGCOMM*, 2017.

[33] S. Hu, W. Bai, G. Zeng, Z. Wang, B. Qiao, K. Chen, K. Tan, and Y. Wang. Aeolus: a Building Block for Proactive Transport in Datacenters. In *ACM SIGCOMM*, 2020.

[34] S. Hu, Y. Zhu, P. Cheng, C. Guo, K. Tan, J. Padhye, and K. Chen. Deadlocks in datacenter networks: Why do they form, and how to avoid them. In *ACM HotNets*, 2016.

[35] S. Hu, Y. Zhu, P. Cheng, C. Guo, K. Tan, J. Padhye, and K. Chen. Deadlocks in datacenter networks: Why do they form, and how to avoid them. In *ACM HotNets*, 2016.

[36] S. Hu, Y. Zhu, P. Cheng, C. Guo, K. Tan, J. Padhye, and K. Chen. Tagger: Practical pfc deadlock prevention in data center networks. In *ACM CoNext*, 2017.

[37] J. Hwang, M. Vuppalapati, S. Peter, and R. Agarwal. Rearchitecting Linux Storage Stack for μs Latency and High Throughput. In *USENIX OSDI*, 2021.

[38] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be General and Fast. In *USENIX NSDI*, 2019.

[39] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *ACM SoCC*, 2012.

[40] S. Keshav and S. Kesahv. *An Engineering Approach to Computer Networking: ATM networks, the Internet, and the Telephone Network*. Addison-Wesley Reading, 1997.

[41] G. Kumar, N. Dukkipati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, et al. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *ACM SIGCOMM*, 2020.

[42] S. Legtchenko, N. Chen, D. Cletheroe, A. Rowstron, H. Williams, and X. Zhao. XFabric: a Reconfigurable In-rack Network for Rack-scale Computers. In *USENIX NSDI*, 2016.

[43] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *ACM SOSP*, 2011.

[44] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, et al. HPCC: High Precision Congestion Control. In *SIGCOMM*. 2019.

[45] M. A. Marsan, E. Leonardi, M. Mellia, and F. Neri. On the stability of input-buffer cell switches with speed-up. In *IEEE INFOCOM*, 2000.

[46] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, et al. Snap: a Microkernel Approach to Host Networking. In *ACM SOSP*, 2019.

[47] N. McKeown, A. Mekkittikul, V. Anantharam, and J. Walrand. Achieving 100% throughput in an input-queued switch. *IEEE Transactions on Communications*, 1999.

[48] W. M. Mellette, R. McGuinness, A. Roy, A. Forencich, G. Papen, A. C. Snoeren, and G. Porter. Rotornet: A Scalable, Low-complexity, Optical Datacenter Network. In *ACM SIGCOMM*, 2017.

[49] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats, et al. TIMELY: RTT-based Congestion Control for the Datacenter. In *ACM SIGCOMM*, 2015.

[50] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker. Revisiting Network Support for RDMA. In *ACM SIGCOMM*, 2018.

[51] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *ACM SIGCOMM*, 2018.

[52] C. S. R. Murthy and M. Gurusamy. *WDM Optical Networks: Concepts, Design, and Algorithms*. Prentice Hall, 2002.

[53] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving High {CPU} Efficiency for Latency-sensitive Datacenter Workloads. In *USENIX NSDI*, 2019.

[54] C. Özveren, R. Simcoe, and G. Varghese. Reliable and Efficient Hop-by-hop Flow Control. In *ACM SIGCOMM*, 1994.

[55] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *ACM SIGCOMM*, 2014.

[56] S. Plotkin. Competitive Routing of Virtual Circuits in ATM Networks. In *IEEE JSAC*, 1995.

[57] G. Porter, R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat. Integrating Microsecond Circuit Switching into the Data Center. In *ACM SIGCOMM CCR*, 2013.

[58] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *IEEE/ACM ISCA*, 2014.

[59] M. Raab and A. Steger. "Balls into Bins"—A Simple and Tight Analysis. In *RANDOM*, 1998.

[60] J. Rexford. Tailoring Router Architectures to Performance Requirements in Cut-through Networks. 1999.

[61] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM*, 2015.

[62] D. Shah. Maximal matching scheduling is good enough. In *IEEE GLOBECOM*, 2003.

[63] V. Shrivastav, A. Valadarsky, H. Ballani, P. Costa, K. S. Lee, H. Wang, R. Agarwal, and H. Weatherspoon. Shoal: A Network Architecture for Disaggregated Racks. In *USENIX NSDI*, 2019.

[64] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, and et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *ACM SIGCOMM*, 2015.

[65] R. Sivaram. Some measured google flow sizes. Technical report, 2008.

[66] P. Stuedi, A. Trivedi, B. Metzler, and J. Pfefferle. DaRPC: Data Center RPC. In *ACM SoCC*, 2014.

[67] P. Tammana, R. Agarwal, and M. Lee. Cherrypick: Tracing Packet Trajectory in Software-defined Datacenter Networks. In *ACM SOSR*, 2015.

[68] P. Tammana, R. Agarwal, and M. Lee. Simplifying Network Debugging with PathDump. In *USENIX OSDI*, 2016.

[69] B. C. Vattikonda, G. Porter, A. Vahdat, and A. C. Snoeren. Practical TDMA for Datacenter Ethernet. In *ACM EuroSys*, 2012.

[70] L. Zhang, S. Berson, S. Herzog, S. Jamin, and R. Braden. RFC2205: Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification, 1997.

[71] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource Reservation Protocol. 1993.

[72] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion Control for Large-scale RDMA Deployments. In *ACM SIGCOMM*, 2015.

SwiftPaxos: Fast Geo-Replicated State Machines

Fedor Ryabinin
IMDEA Software Institute
Universidad Politécnica de Madrid

Alexey Gotsman
IMDEA Software Institute

Pierre Sutra
Télécom SudParis
INRIA

Abstract

Cloud services improve their availability by replicating data across sites in different geographical regions. A variety of state-machine replication protocols have been proposed for this setting that reduce the latency under workloads with low contention. However, when contention increases, these protocols may deliver lower performance than Paxos. This paper introduces SwiftPaxos—a protocol that lowers the best-case latency in comparison to Paxos without hurting the worst-case one. SwiftPaxos executes a command in 2 message delays if there is no contention, and in 3 message delays otherwise. To achieve this, the protocol allows replicas to vote on the order in which they receive state-machine commands. Differently from previous protocols, SwiftPaxos permits a replica to vote twice: first for its own ordering proposal, and then to follow the leader. This mechanism avoids restarting the voting process when a disagreement occurs among replicas, saving computation time and message delays. Our evaluation shows that the throughput of SwiftPaxos is up to 2.9× better than state-of-the-art alternatives.

1 Introduction

Context. Today's cloud services run in data centers scattered around the world. The critical part of these services is replicated at different geographical sites and maintained strongly consistent [9, 14, 40, 48]. To achieve this, cloud providers rely on *state-machine replication (SMR)* [45], where a service is defined by a deterministic state machine and each site maintains its own replica of the machine. An SMR protocol coordinates the execution of commands at the sites, ensuring that they stay in sync. The resulting system is *linearizable* [21], providing an illusion that each command applied to the service executes instantaneously at all sites.

Unfortunately, common SMR protocols, such as Paxos [29] and Raft [39], have a high latency in geo-replicated deployments. These protocols funnel all commands through a *leader* site, which orders commands and persists them at replicas. If the leader fails, a new one is elected; the period when a given site acts as a leader is called a *ballot*. In such protocols the client finds out the result of a command execution after 4 message delays: a round trip from the client to the leader plus a round trip from the leader to the replicas. This high latency is made worse by the fact that geo-replicated transaction processing systems (such as Spanner [14]) use Paxos multiple times when executing a single transaction, e.g., to implement a fault-tolerant version of two-phase commit.

Problem. There have been several proposals of SMR protocols aiming to lower latency. One approach, pioneered by Fast Paxos [31], is for clients to contact replicas directly, bypassing the leader, and let each replica order the command independently. If enough replicas (usually $> 3/4$ of the total) agree on the ordering of the command, it takes the *fast path*. Otherwise, it is processed via a *slow path*, which requires extra message exchanges to resolve the disagreement. We can increase the chances of spontaneous agreement on command ordering by observing that, to satisfy linearizability, it is enough that replicas only agree on the order of non-commuting (aka *conflicting*) commands [30, 42]. This allows taking the fast path when there is no contention, i.e., when replicas receive conflicting commands in the same order, as is often the case in application workloads [26, 38, 41]. Unfortunately, protocols using this approach, such as Generalized Paxos [30], become very expensive once even a single pair of commands contend: in this case the protocol changes the ballot, similarly to how it would handle a leader failure. This requires transferring a large amount of state between replicas and disrupts the processing of all commands, even those that do not conflict. For this reason, Generalized Paxos has not been used in practice.

An alternative approach eliminates the leader altogether, allowing the replicas to order commands in a peer-to-peer manner (EPaxos [38] and follow-ups [3, 6, 17]). To perform optimally, such leaderless protocols require clients to be colocated with a data replica. However, as replication is expensive, geo-distributed systems often do not maintain a data replica at each client site [9, 40]. In addition, the execution of commands in this class of SMR protocols is often delayed by a form of a convoy effect. As a consequence, these protocols have a high tail latency even at low conflict rates; once the conflict rate increases, the latency skyrockets [6, 19, 50].

Contributions. To address these limitations, in this paper we present SwiftPaxos—a new SMR protocol providing fast linearizable operations over geo-replicated data. It processes a command in 2 message delays when this command commutes with concurrently submitted ones (fast path). Otherwise, SwiftPaxos takes just one extra delay for the leader to resolve the conflict, requiring 3 message delays overall (slow

path). Unlike in Generalized Paxos [30], the slow path does not disrupt the system and does not require transferring significant amounts of state. Since in SwiftPaxos conflicts are resolved by the leader, it also exhibits low tail latency, unlike EPaxos [38]. Thus, SwiftPaxos lowers the best-case latency in comparison to Paxos without hurting the worst-case one.

To achieve such benefits, in SwiftPaxos a client sends its command c directly to the replicas, which compute the set of c's *dependencies*—commands conflicting with c that should be executed before it. At each replica, dependencies form a partial order on commands, dictating how they should be applied to the local copy of the state machine. To compute the dependencies of a command, replicas make *proposals* based on the order in which they receive commands, and then agree on one of these. If enough replicas make the same proposal, the command is processed in a single round-trip (fast path). Otherwise, the replicas adopt the leader's proposal, requiring one extra message delay (slow path).

The key novelty of this scheme is that on the slow path a replica votes for two different proposals: first for its own and then for the leader's, with latter superseding the former. Such double voting would usually be unsafe in Paxos-like SMR protocols: this is why Generalized Paxos [30] can only resolve a disagreement by changing the ballot. It is safe in SwiftPaxos because we include the leader into all fast quorums. Hence, if a fast quorum replica disagreed with the leader, the replica knows that the command could not have been committed on the fast path. The replica can then quickly correct its vote without compromising safety by sending a new message overruling the previous one. This enables processing a command on the slow path in just 3 message delays.

We have experimentally evaluated SwiftPaxos across 13 regions on Amazon EC2. Depending on contention, the protocol delivers 16–29% lower average latency than Paxos, and in mixed YCSB workloads [13] its throughput is up to 2.9× higher than EPaxos.

2 System Model

We consider a geo-distributed message-passing system where processes may fail by crashing, but do not behave maliciously. The processes are split into *replicas*, running a distributed service, and *clients* using it. We denote the set of replicas by $\mathcal{R}$ and assume that there are $N = 2f + 1$ of these, at most f of which may fail. Each process in $\mathcal{R}$ models a server located in a separate data center; clients can be co-located with servers or located elsewhere. The set of replicas can be changed using standard reconfiguration techniques [29, 46], and we omit details related to this.

State-machine replication (SMR) is a common approach to designing highly available distributed services in the above system [14, 45]. The service is defined by a deterministic state machine, which has a set of states $\mathcal{S}$ and accepts a set of *commands* $\mathcal{C}$. Given a command c and a state s, a function $\texttt{exec}(c,s)$ returns a pair (r,s') of the return value r and the new state s' obtained by executing c in s. Each replica maintains its own copy of the state machine, accessible via a variable `state`. An *SMR protocol* coordinates the execution of commands at the replicas, ensuring that their copies of the state machine stay in sync.

Clients are stateless, but they know how to contact the replicas. The SMR protocol allows a client to submit a command c for execution using an API call $\texttt{submit}(c)$. Each submitted command c is tagged with a unique identifier $\texttt{id}(c)$. A function `client` associates each identifier with the client that submitted the command. When the SMR protocol obtains the response value r of a command c, it upcalls into the client with a notification $\texttt{response}(\texttt{id}(c), r)$.

The protocol we propose in this paper satisfies *linearizability* [21]. Informally, this means that commands appear to clients as if executed sequentially on a single copy of the state machine in an order consistent with the *real-time order*, i.e., the order of non-overlapping command invocations. To satisfy linearizability, it is enough that replicas agree on the execution order of non-commuting commands [30, 42]. More precisely, two commands c and d *commute* if for every state s of the state machine: *(i)* executing c followed by d or d followed by c in s leads to the same state; and *(ii)* c returns the same response in s as in the state obtained by executing d from s, and vice versa. When the two commands do not commute, we say that they *conflict*, written $c \bowtie d$. Conflicts can be over-approximated using the service API: e.g., in a key-value store operations on different keys commute. In transactional systems with deferred update replication, such as Spanner [14], conflicts can be detected at commit time.

To give a specification of an SMR protocol, we first define a relation $\prec$ over $\mathcal{C}$ so that $c \prec d$ if: *(i)* $c \bowtie d$ and some replica p executes c before d; or *(ii)* c was executed by some replica before any client submitted d. An SMR protocol needs to satisfy the following properties:

Validity. If a replica executes a command c, then some client has submitted c before.

Integrity. A replica executes each command at most once.

Ordering. The relation $\prec$ is acyclic.

Nontriviality. The return value r obtained by the client for a command c is the result of c's execution at some replica.

Liveness. If a command c is submitted by a non-faulty client or executed by some replica, then c is eventually executed by all non-faulty replicas.

Implementing this specification is sufficient to ensure the linearizability and responsiveness of the service [30, 42]. In particular, Ordering guarantees that different replicas cannot execute conflicting commands in different orders. As is standard, to ensure liveness we assume that the system is *eventually synchronous* [16], so that message delays between non-failed processes are eventually bounded.

3 Core Concepts and Protocol Overview

We first provide an overview of SwiftPaxos, and then cover it in detail. We have rigorously proved SwiftPaxos correct, but due to space constraints we defer this proof to §A. Instead, in our explanations we state key protocol invariants and informally explain why they hold.

3.1 Ballots

As usual for Paxos-like protocols, SwiftPaxos's execution is divided into a sequence of *ballots*. A replica can be in a single ballot at a time, tracked in a variable bal. Each ballot b has a fixed *leader* replica $\mathsf{leader}(b) = p_{(b \bmod N)}$; all other participants of b are *followers*. If $\mathsf{leader}(b)$ is suspected of failure, a follower initiates a *recovery* procedure, which switches to a higher ballot with a new leader. A variable status at a replica records whether it is operating normally (NORMAL) or is recovering (RECOVERING).

We call a majority of replicas a *quorum*. Each ballot b is associated with a set of *fast quorums* $\mathcal{FQ}(b)$ and a set of *slow quorums* $\mathcal{SQ}(b)$; these are respectively used on the fast and slow paths of SwiftPaxos. For a replica p, we write $\mathit{fast}(p,b)$ and $\mathit{slow}(p,b)$ if p belongs to a fast and a slow quorum of b, respectively. We require that the leader of b belong to every fast and slow quorum of b. Apart from this constraint, slow quorums can be any majorities of replicas. Fast quorums must satisfy a stricter condition (as in Fast Paxos [31]), requiring that any two fast quorums intersect in a majority:

$$\forall Q_1, Q_2 \in \mathcal{FQ}(b).\, |Q_1 \cap Q_2| > N/2. \qquad \text{(FQI)}$$

This condition can be satisfied in different ways. In the rest of the paper we consider the following two:

(C1) A fast quorum is any set containing $> 3/4$ of all replicas, including the leader.

(C2) There is a unique fast quorum consisting of a fixed majority of replicas, including the leader.

These two quorum systems offer different optima in the trade-off defined by (FQI). On the one hand, (C1) supports up to $(N/4-1)$ follower failures without blocking the fast path. On the other, (C2) has a better complexity, only requiring a majority of replicas to order commands [32, 33, 36, 38], but may block the fast path when any of these replicas fails.

3.2 Dependencies and Key Invariants

As we noted earlier, to ensure linearizability it is enough for replicas to only agree on the order of non-commuting commands. SwiftPaxos represents such an order by associating with each command c the set of its *dependencies*—commands conflicting with c that must be executed before it. To compute this set, the replicas make their *proposals* and then agree on one of them. We say that the command c is *committed* when the replicas agree on all its transitive dependencies, i.e., the dependencies of c, the dependencies of these dependencies, and so on. The protocol ensures:

INVARIANT 1. Any two replicas commit a command with the same set of dependencies.

A command moves through different phases as it is processed, from the initial phase (START) to the final one, where the command is committed (COMMIT). Each replica tracks the progress of commands in an array phase, indexed by command identifiers. The name of the phase written in italics denotes the set of all commands in this phase, e.g., *Commit* stands for $\{id \mid \mathsf{phase}[id] = \text{COMMIT}\}$. An array dep maps a command identifier to its direct dependencies: if the command's entry in the phase array is COMMIT, then the dependencies are committed; otherwise, they record the replica's proposal. The dep array defines the edges of a *dependency graph* at a replica. Initially, all the entries are null ($\perp$).

A replica executes a command once it is committed and all its dependencies are executed. Since dependencies relate only conflicting commands, replicas are thus free to execute independent commands in any order. Then to satisfy the Ordering property of SMR (§2), the protocol needs to ensure:

INVARIANT 2. For any two conflicting commands c and c' committed at a replica, either c belongs to the dependencies of c', or the converse holds.

To illustrate, consider commands x, y and z such that $\mathtt{x} \bowtie \mathtt{y}$ and $\mathtt{y} \bowtie \mathtt{z}$ and their committed dependencies are $\mathtt{dep}[\mathtt{id}(\mathtt{x})] = \mathtt{dep}[\mathtt{id}(\mathtt{z})] = \emptyset$ and $\mathtt{dep}[\mathtt{id}(\mathtt{y})] = \{\mathtt{x}, \mathtt{z}\}$. Then a replica can execute x and z in any order, provided they execute before y.

Finally, as the protocol delays executing a committed command until its dependencies are executed, for the protocol to be live, committed dependencies should not form a cycle:

INVARIANT 3. The committed part of the dependency graph at each replica is acyclic.

3.3 Agreeing on Dependencies

Before describing SwiftPaxos in detail, we give its overview using the example in Figure 1. Here the system consists of 5 replicas, thus tolerating 2 faults. Replicas p_4 and p_5 host one client each. A third client is not co-located with any of the replicas. The fast quorum intersection requirement (FQI) is satisfied with configuration (C2), so that there is a single fast quorum $\{p_1, p_2, p_3\}$. To avoid clutter we omit most of the interactions with the two replicas outside this quorum.

Propagation. A client sends a submitted command c to the replicas in a $\mathtt{Propagate}(c)$ message. When a replica receives c, it computes its proposal for c's dependencies as the set of the previously received commands that conflict with c—to satisfy Invariant 2. In Figure 1, proposals are depicted using $\rightarrow$, so that $\{\mathtt{x}, \mathtt{y}\} \rightarrow \mathtt{z}$ means that z depends on x and y.

Fast path. Each replica in a fast quorum broadcasts its proposal in a FastAck message. Replicas wait until they receive

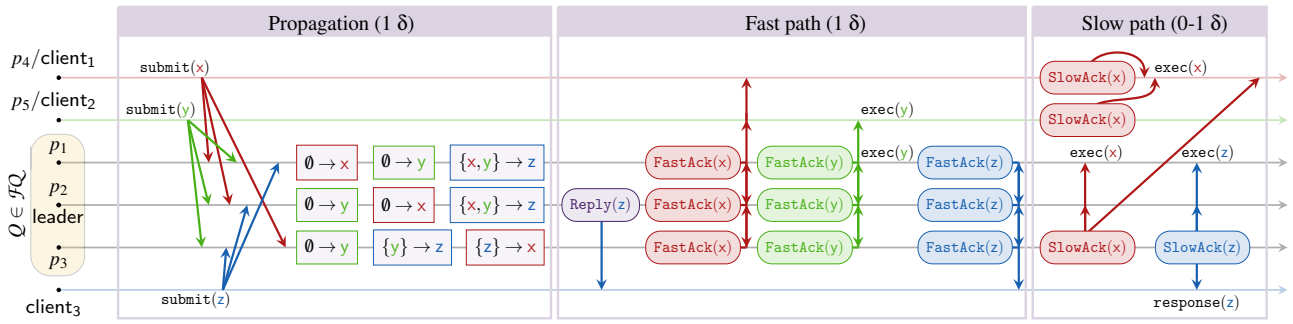


Figure 1: An execution of the protocol processing commands x, y and z such that x ⋈ z and y ⋈ z.

proposals from all members of some fast quorum. If all such proposals are the same, then there is a spontaneous agreement [31]. In this case, a replica commits the command once its dependencies are also committed. This represents the *fast path* of the protocol, which under favorable conditions allows a replica to execute a command within 2 message delays from the time it was submitted. For example, in Figure 1 replica p_1 commits and executes y immediately after receiving the same proposal $\emptyset \to$ y from all fast quorum members. Since each replica computes its proposal from the conflicting commands it has seen before, a command takes the fast path when there is no contention—something that occurs frequently in application workloads [26, 38, 41].

Slow path via double voting in a fast quorum. Fast quorum replicas compute their proposals based on the order in which they receive conflicting commands. As this depends on the ordering generated by the network, the replicas may disagree, in which case the command takes the *slow path*. To this end, each fast quorum replica that disagreed with the leader adopts its proposal and acknowledges this by broadcasting a `SlowAck` message. A replica can commit the command once it receives a set of matching `FastAck` and `SlowAck` messages from all the members of some fast quorum and it has committed the command's dependencies. This may require an extra message delay in comparison to the fast path. For example, in Figure 1 replicas p_1 and p_2 receive x before z, while p_3 receives z before x, which results in different proposals. Since p_3 disagrees with the leader p_2 on the dependencies of x and z, it adopts the leader's proposals $\emptyset \to$ x and $\{$x,y$\} \to$ z and broadcasts `SlowAck`s for the two commands. Consequently, replica p_1 commits and executes x and z in 3 message delays.

Note that on the slow path a fast quorum replica votes for two different proposals in the same ballot—once for its own (`FastAck`) and once for the leader's (`SlowAck`). Such double voting would usually be unsafe in Paxos-like protocols. In SwiftPaxos this is safe because the leader belongs to all fast quorums: if a fast quorum replica disagreed with the leader, the replica knows that the command could not have been committed on the fast path, and may safely correct its vote.

Slow path via a slow quorum. SwiftPaxos can also commit commands using a slow quorum, just like Paxos. A slow quorum replica acts similarly to a fast quorum replica, except that it does not make proposals of its own. Namely, when it receives a `FastAck` from the leader, a slow quorum replica adopts the leader's proposal and broadcasts a `SlowAck` message. Any replica can commit the command after receiving `FastAck` from the leader and `SlowAck`s from the followers of a slow quorum. In a wide-area network, where latencies between different pairs of replicas vary, the slow path may be faster than the double-voting mechanism in the fast path. Going back to Figure 1, assume that the latency between p_4 and p_3 is higher than between p_4 and p_5. Instead of waiting for a `SlowAck` message for x from p_3, replica p_4 can use the earlier `SlowAck` message from p_5 and the `FastAck` from the leader p_2 to commit x via the slow quorum $\{p_4, p_5, p_2\}$. Since a slow quorum can be any majority, this additional mechanism for committing commands ensures that SwiftPaxos is *at least as fast* as Paxos. In fact, on the slow path SwiftPaxos corresponds to a well-known faster variation of Paxos where followers broadcast 2B messages to all replicas.

Message complexity. SwiftPaxos has a quadratic message complexity, as opposed to linear complexity of classical Paxos. However, for typical geo-replicated SMR deployments ($N = 3$ or $N = 5$) [9, 14] the difference between these complexities is fairly small. Besides, the additional messages issued by SwiftPaxos are light, as they only carry metadata. We compare the bandwidth usage of SwiftPaxos and Paxos in §5.1.

3.4 Ensuring Low Tail Latency

Dependencies are also used to order commands in EPaxos [38] and its follow-ups [3, 17]. However, in EPaxos dependencies may form cycles, and thus command execution cannot simply follow their order. Instead, EPaxos waits until it forms strongly connected components of the dependency graph and then executes these components one at a time. Since such components can be arbitrary large, the protocol may delay command execution for an unbounded amount of time. This phenomenon is known as *convoy effect* and in practice it leads to a high tail latency [6, 19, 50].

In contrast, dependencies in SwiftPaxos remain acyclic (Invariant 3, §3.2), and command execution can simply follow

	Dependency graph	Dependencies of c_3	Paths to c_3
leader	$c_1 \to c_2 \to c_3$	$\{c_2\}$	$\{[c_3, c_2, c_1]\}$
p_1	$c_2 \to c_3$	$\{c_2\}$	$\{[c_3, c_2]\}$
p_2	$c_2 \to c_3$	$\{c_2\}$	$\{[c_3, c_2]\}$

Figure 2: To accept the result of optimistic execution, clients wait for matching dependency paths.

them. The only delay between committing a command and executing it is similar to one present in Multi-Paxos, where each consensus instance depends on the prior ones. This makes the tail latencies of SwiftPaxos and Paxos comparable, as we empirically show in §5.2. We compute the theoretical latency of SwiftPaxos and compare it against other protocols in §A.6.

3.5 Faster Responses at Non-Collocated Clients

Clients located in the same data center as a replica receive the result of a command directly from that replica. For instance, client$_2$ in Figure 1 is co-located with p_5 and receives the response to y immediately after p_5 executes y. Clients that are not co-located with a replica need to wait for an extra message delay to hear the response. To speed up delivering responses at such clients, SwiftPaxos optimistically executes commands at the leader (similarly to Zookeeper [23] and CURP [41]).

In more detail, when the leader receives a command c from a client that is not co-located with a replica, it computes the result of executing c and replies to the client with a `Reply` message. In Figure 1, client$_3$ gets this message for command z. Followers send their `FastAck` and `SlowAck` messages for c not only to other replicas—as described before—but also to the client that submitted c. The client accepts the response in `Reply` once it receives matching `FastAck` or `SlowAck` messages from a fast or slow quorum. This ensures that a client always gets a response within 2 or 3 message delays, regardless of whether it is co-located with a replica.

There is a subtlety, though: on the fast path a client cannot accept the result of optimistic execution using only the dependency sets, because these sets contain direct dependencies of the command but say nothing about their predecessors. Figure 2 illustrates this. It depicts the dependency graphs at some point in time where (FQI) from §3.1 is satisfied with a single quorum $\{p_0, p_1, p_2\}$, with p_0 as the leader. In Figure 2, clients submit 3 commands c_1, c_2 and c_3, with c_2 conflicting with both c_1 and c_3. The replica p_0 is the only one to receive c_1, yet all three replicas agree on the ordering of c_2 and c_3. Assume that p_0 now optimistically executes c_1, c_2 and c_3 in this order before it gets any messages from other replicas, and thus before this ordering is durable. It will then send the result of executing c_3 to the client. The leader's proposal for the dependency set of c_3 matches the proposals of p_1 and p_2, but the three replicas disagree on c_3's transitive dependencies. Hence, the client cannot accept the reply from the leader: if the leader crashes, its ordering of c_1 will be lost, invalidating the optimistic execution. Notice that this problem does not occur at replicas, as they execute each command only after it gets committed, implying that all its transitive dependencies are also committed, and thus durable.

To ensure that clients accept only valid results from optimistic execution, `FastAck` messages contain not only the direct dependencies of the command but also the set of all the paths leading to it in the dependency graph. If these sets match, the ordering of the command is durable and the client may accept the result via the fast path. For a command c, we represent the set of c's dependency paths as a set of ordered lists, where each list starts with c and continues with its transitive dependencies arranged according to the dependency graph (see Figure 2). In §4.1 we explain how dependency paths can be efficiently implemented in practice.

In Figure 1 when the fast quorum replicas receive z, their `FastAck` replies carry not just dependency sets, but also dependency paths: replicas p_1 and p_2 send $\{[z,x];[z,y]\}$, while p_3 sends $\{[z,y]\}$. Since these sets do not match, the result of z cannot be accepted by client$_3$ on the fast path.

4 SwiftPaxos in Detail

We define the protocol logic in Figures 3–5 using a set of handlers, each of which executes atomically once its preconditions are true (keyword **pre**).

4.1 Normal Operation

Propagation. Upon receiving a `Propagate`(c) message from a client (lines 2 and 6), a replica p saves c in an array cmd and moves c to the PREACCEPT phase, to record that it is now waiting for the leader's proposal. If the replica belongs to some fast quorum, it also computes its proposal dep[`id`(c)] for c's dependencies and broadcasts it in a `FastAck` message together with c's dependency paths `paths`$[id]$ (line 21).

If p is the leader, upon receiving a command p also executes it and sends its result in a `Reply` message (lines 16-19). The command is not committed at this point. The execution is optimistic and p does not modify its local copy of the state machine. Instead, p maintains a list pending_log of commands that are received but not committed yet. The command is added to the list and p determines its result using a function `opt_exec` (further discussed in §4.4).

Fast path. A replica p commits a command c via the fast path when it receives matching dependency sets from a fast quorum. In this case, p first processes the `FastAck` message from the leader (line 22), which advances c to the ACCEPT phase (line 24). Then p processes `FastAck`s from the other quorum members (line 30). Once the dependencies of c are committed (line 31), p commits c as well. A client receives a response on the fast path once it gets a `Reply` message from the leader and matching dependency paths from the followers of a fast quorum (line 3).

```
1  function submit(c):
2      send Propagate(c) to R
3  when received Reply(b,id,P,r) from leader(b) and
                 FastAck(b,id,_,P) or SlowAck(b,id)
                 from all followers in Q ∈ FQ(b), or
4  when received Reply(b,id,_,r) from leader(b) and
                 SlowAck(b,id) from all followers in
                 Q ∈ SQ(b)
5      response(id,r)
```

Figure 3: Client code.

Slow path. Upon the receipt of $\texttt{FastAck}(b,id,D,P)$ from the leader, a fast quorum replica checks whether its local value of $\mathsf{dep}[id]$ is equal to D (line 25). If this is not the case, the command takes the slow path: the replica overwrites $\mathsf{dep}[id]$ with D, advances the command's phase to ACCEPT and notifies the other replicas and the client with a `SlowAck` message. A slow quorum replica also sends the `SlowAck` to speed up agreement on dependencies, as described in §3.3. To commit the command, the replica then waits until the other fast (or slow) quorum members send either a `FastAck` matching the leader's proposal or a `SlowAck` (line 30).

The client acts similarly to compute the response to its command, except that it waits for matching dependency paths instead of dependency sets (line 3). This justifies line 29: when a fast quorum replica receives the leader's proposal that matches its own vote but does not match the dependency paths, it sends a `SlowAck` message to the client. In this way the client learns that the replica is now in sync with the leader and can thus accept the result of the optimistic execution.

To ensure that the replica is indeed in sync with the leader, we need the last conjunct of the precondition at line 23. This requires the replica to handle a $\texttt{FastAck}(_,_,D,_)$ message from the leader only if it has already handled such messages from all the commands in D. This condition is automatically satisfied when communication channels are FIFO. To illustrate its role, imagine that in Figure 1 replica p_3 updated the dependencies of z before those of x, violating the precondition in line 23. Then right after p_3 changed $\mathsf{dep}[\mathsf{z}]$ to $\{\mathsf{x},\mathsf{y}\}$, the three replicas of the fast quorum disagree on the set of dependency paths of z: at replicas p_1 and p_2 the set is $\{[\mathsf{z},\mathsf{x}];[\mathsf{z},\mathsf{y}]\}$ but according to p_3 there is a path $[\mathsf{z},\mathsf{x},\mathsf{z}]$. As seen in §3.5 accepting the result of the execution of z would be unsafe.

Command execution. A replica keeps track of commands executed on its copy of the state machine using a variable Exec. A replica executes a command once it is committed and all its dependencies are executed (line 33). If the replica is the leader, it then removes the command from pending_log.

Representing dependency paths. Sending full dependency paths is not an option in practice due to their fast-growing size. This issue is solved by trimming the paths of a command c up to the last committed or accepted transitive dependency. For example, if c's set of dependency paths is $\{[c,d,e]\}$ and

```
6  when received Propagate(c) from client q
7      pre: status = NORMAL ∧ id(c) ∈ Start
8      phase[id(c)] ← PREACCEPT
9      cmd[id(c)] ← c
10     if fast(p, bal) then
11         let D = {id | cmd[id] ⋈ c}
12         let P = pset(id(c))
13         dep[id(c)] ← D
14         paths[id(c)] ← P
15         if p = leader(bal) then
16             pending_log ← pending_log · c
17             let r = opt_exec(pending_log, state)
18             send Reply(bal, id(c), P, r) to q
19             send FastAck(bal, id(c), D, P) to R
20         else
21             send FastAck(bal, id(c), D, P) to R ∪ {q}
22 when received FastAck(b,id,D,P) from leader(b)
23     pre: status = NORMAL ∧ id ∈ Preaccept ∧ bal = b
           ∧ D ⊆ Accept ∪ Commit
24     phase[id] ← ACCEPT
25     if (fast(p,b) ∧ dep[id] ≠ D) ∨ slow(p,b) then
26         dep[id] ← D
27         send SlowAck(bal, id) to R ∪ {client(id)}
28     else if fast(p,b) ∧ paths[id] ≠ P then
29         send SlowAck(bal, id) to client(id)
30 when received FastAck(b,id,D,_) or SlowAck(b,id)
       from all followers in Q_f ∈ FQ(b), or
       SlowAck(b,id) from all followers in Q_s ∈ SQ(b)
31     pre: status = NORMAL ∧ id ∈ Accept ∧ bal = b
           ∧ D = dep[id] ⊆ Commit
32     phase[id] ← COMMIT
33 when there exists id ∈ Commit \ Exec with dep[id] ⊆ Exec
34     (_, state) ← exec(cmd[id], state)
35     Exec ← {id} ∪ Exec
36     if p = leader(bal) then remove(pending_log, id)
37 function pset(id):
38     if dep[id] = ∅ then return {[id]}
39     else return {id :: l | ∃id' ∈ dep[id]. l ∈ pset(id')}
```

Figure 4: Normal operation at a replica p.

d is in the COMMIT or ACCEPT phase, the set is reduced to $\{[c,d]\}$. Thanks to the preconditions at lines 23 and 31, this is safe: if a command is committed or accepted then so is any of its transitive dependencies. When the reduced sets match, the omitted parts match as well, as they correspond to the ordering at the leader. Additionally, instead of the paths themselves one may send hashes of the (reduced) paths [44]. This optimization is also applied in our implementation.

Garbage collection. Standard periodic checkpoints can be used to bound memory usage of the protocol [7,23]. In our implementation, we simply trim a command from the protocol's state once it is executed at all replicas. In §5.1 we discuss the memory consumption in more detail.

```
40 function recover():
41     let b > bal such that leader(b) = p
42     send NewLeader(b) to all

43 when received NewLeader(b) from q
44     pre: b > bal
45     status ← RECOVERING
46     bal ← b
47     send NewLeaderAck(b, cbal, phase, cmd, dep) to q

48 when received NewLeaderAck(b, cbal_q, phase_q, cmd_q, dep_q)
       from all q ∈ Q
49     pre: status = RECOVERING ∧ bal = b ∧ |Q| > n/2
50     reset phase, cmd and dep
51     let b_max = max{cbal_q | q ∈ Q}
52     let U = {q ∈ Q | cbal_q = b_max}
53     forall id such that
         ∃q ∈ U. phase_q[id] ∈ {ACCEPT, COMMIT} do
54         phase[id] ← phase_q[id]
55         cmd[id] ← cmd_q[id]
56         dep[id] ← dep_q[id]
57     forall id ∉ Accept ∪ Commit do
58         if ∃D ≠ ⊥. ∃Q_f ∈ FQ(b_max). Q ∩ Q_f ⊆ U
             ∧ ∀q ∈ Q ∩ Q_f. dep_q[id] = D then
59             phase[id] ← ACCEPT
60             cmd[id] ← cmd_q[id]
61             dep[id] ←
                 D ∪ {id' | cmd[id'] ⋈ cmd[id] ∧ id ∉ dep[id']}
62     forall (id, id') such that id' ∈ Start ∩ dep[id] do
63         dep[id] ← dep[id] \ {id'}
64     arbitrarily break cycles in dep
65     send Sync(b, phase, cmd, dep) to R

66 when received Sync(b, phase, cmd, dep)
67     pre: b ≥ bal
68     status ← NORMAL; bal ← b; cbal ← b
69     phase ← phase; cmd ← cmd; dep ← dep
70     clear(pending_log)
71     if slow(p, b) then
72         forall id in an order consistent with dep do
73             if p = leader(b) ∧ id ∉ Exec then
74                 pending_log ← pending_log · cmd[id]
75             else if id ∈ Accept then
76                 send SlowAck(b, id) to R
```

Figure 5: Recovery at a replica p.

4.2 Recovery from Leader Failures

Replicas continuously monitor the progress of the protocol. When the current leader is suspected of hindering progress, the protocol nominates a new one to take over. This nomination can be done in a standard way, e.g., using a failure detector [8], and we defer the details to §A.4. We now explain the algorithm followed by the potential leader.

Leadership change. The recovery procedure begins similarly to Paxos [29]. When a replica p is nominated to become the new leader, it calls `recover` (line 40). This function picks a new ballot b led by p and higher than any ballot p has previously joined. Replica p then sends b in a `NewLeader` message to all replicas, asking them to support its leadership. A replica acknowledges p as the new leader only if b is higher than any ballot it has previously joined (line 44). In this case the replica changes its status to RECOVERING, which stops normal message handling, and replies with a `NewLeaderAck` message carrying the commands it knows about.

Once the leadership of p is approved by a quorum Q (line 48), p's next goal is to bring the replicas into the same state from which they will resume processing commands.

Recovering commands. To maintain consensus on dependencies (Invariant 1), the new leader's state must include all commands committed in lower ballots. To ensure this, the leader computes the initial state of b based on the states and the values of a cbal variable reported by the replicas in their `NewLeaderAck` messages. This variable maintains the last ballot at which each replica successfully completed recovery.

Similarly to Paxos [29], p focuses on the set U of replicas that reported the maximal ballot $b_{\max}$: the state of these replicas supersedes that of replicas from lower ballots. The leader p then incorporates into its state all the commands that could have been committed up to $b_{\max}$. In lines 53–56 the leader does this for commands that could have been committed on the slow path. Any such command would have to be accepted by a slow quorum Q_s, which must intersect with the recovery quorum Q in at least one process. Hence, for each command c in the ACCEPT or COMMIT phase at a replica $q \in U$, the leader incorporates c and its dependencies as reported by q into its state. In lines 58–61 the leader collects all commands that could have been committed up to $b_{\max}$ on the fast path. Any such command would have to be preaccepted with the same dependencies by a fast quorum Q_f. Hence, p adds to its state all commands c such that $Q \cap Q_f \subseteq U$ and all replicas in $Q \cap Q_f$ report the same dependency set D for c. This condition is similar to Fast Paxos [31]; we later explain the rationale for the second term of the union at line 61 and how we ensure that we don't create cycles in the dependency graph. Note that the condition at line 58 defines D uniquely: there may not be two different fast quorums Q_f and Q'_f whose members in Q report different dependency sets. This is because (FQI) implies $(Q \cap Q_f) \cap (Q \cap Q'_f) = Q \cap Q_f \cap Q'_f \neq \emptyset$. But then the dependencies reported for the command c by $Q \cap Q_f$ and $Q \cap Q'_f$ cannot be distinct, since they were reported by some replica belonging to both sets.

Thanks to the conditions at lines 53 and 58, all commands committed at earlier ballots are included into the new state. Next, the leader removes from the dependency graph all commands that do not satisfy those conditions (lines 62-63). To see that the loop at line 62 does not violate safety properties, note that for a pair of commands c and c' whose identifiers id and id' satisfy the condition at line 62, neither command can be committed at a previous ballot: c' is not committed because it does not satisfy any of the two conditions at line 53 and line 58; c is not committed because it depends on an uncommitted command c'. Thus, it is safe to completely remove c' from the state and update the dependencies of c.

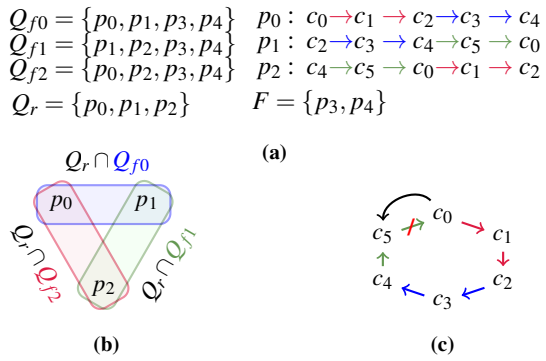


Figure 6: Avoiding dependency cycles during recovery.

After the above computation, the leader breaks cycles in the dependency graph; we explain this step shortly. The leader then broadcasts a `Sync` message, defining the state from which the new ballot starts. When another replica receives this message (line 66), it overwrites its state with the one provided, changes its status to NORMAL, and clears the log of pending commands. If the replica belongs to a slow quorum, it broadcasts a `SlowAck` message for each uncommitted command (line 76), to ensure that such commands are committed in the new ballot. Finally, if the replica is the new leader, it adds all unexecuted commands to `pending_log` in an order consistent with the dependencies (line 72).

Recovery can be easily optimized so that `Sync` messages only transfer the parts each replica is missing (although our prototype implementation omits this optimization).

Preserving the invariants. During the normal protocol operation, both on the fast and slow paths replicas commit commands with the dependencies proposed by the leader. Then, due to the way the leader computes its proposals, committed conflicting commands cannot be independent, and the dependencies of committed commands cannot form a cycle, as respectively required by Invariants 2 and 3. However, preserving these invariants during recovery requires care.

First, during recovery the new leader can introduce a cycle when it merges replica states. To see how this can happen, consider the example illustrated in Figure 6. The system consists of five replicas and recovers from $Q_r = \{p_0, p_1, p_2\}$ after the failure of the remaining replicas $F = \{p_3, p_4\}$. Assume that on recovery all replicas in Q_r report the same $\mathsf{cbal} = b$ and the orderings shown in Figure 6a. Assume also that the leader of b is p_4, and that this ballot is associated with three fast quorums Q_{f0}, Q_{f1} and Q_{f3}, defined in the figure. On recovery the replicas in $Q_r \cap Q_{f0}$, $Q_r \cap Q_{f1}$ and $Q_r \cap Q_{f2}$ agree on the orders $c_2 \to c_3 \to c_4$, $c_4 \to c_5 \to c_0$ and $c_0 \to c_1 \to c_2$, respectively (Figure 6b). According to line 58, the new leader has to incorporate all of them into its state. But when these orderings are combined, they yield a cycle over commands $c_0, \ldots, c_5$. To preserve Invariant 3 in such cases, the leader arbitrarily breaks cycles by inverting relations between any two commands in them (line 64). For example, the cycle in Figure 6 can be broken by removing c_5 from $\mathsf{dep}[\mathtt{id}(c_0)]$ and adding c_0 to $\mathsf{dep}[\mathtt{id}(c_5)]$ (Figure 6c). Note that this computation does not violate Invariant 1 because it changes only those dependencies that could not have been committed in previous ballots. Namely, if *some* command on the cycle is committed, then so are all its predecessors, i.e., *all* commands on the cycle (by line 31); but this is impossible by Invariant 3. We refer to §A for a detailed proof.

When computing the new state during recovery, the new leader can also end up with two conflicting commands none of which is a dependency of the other. To see how this can happen, imagine that in Figure 1 recovery occurs right after the moment p_3 received $\mathtt{FastAck}(b, \mathtt{x}, \emptyset, _)$ from the leader. At this moment the replica has x in the ACCEPT phase with an empty set of dependencies, and z in the PREACCEPT phase, with y as its only dependency. Incorporating both commands into the new state with the dependencies reported by p_3 would violate Invariant 2, because neither of the conflicting commands x and z is a dependency of the other. To avoid such situations, the new leader updates dependency sets of the commands that are not in the ACCEPT or COMMIT phase at some replica of the recovery quorum (line 61). The leader adds to the dependencies of c all conflicting commands, excluding those that already depend on c. In the above example, the leader includes x in the dependency set of z (line 61), which is safe because z is not committed at previous ballots. In §A we prove that this computation is also safe in general.

4.3 Recovery from Client and Follower Failures

Before voting on a command, a replica must know its payload (lines 6 and 23), which may not be available locally if the client has failed. To deal with this, if the replica receives an acknowledgment from the leader regarding this command, after some time it tries to fetch the payload from the leader. Additionally, followers re-propose commands if they are not committed after some timeout. This helps when the payload is available at followers but missing at the leader.

Follower failures do not impact the availability of SwiftPaxos. But the failure of a follower within the fast quorum can disable the fast path of the protocol and degrade its performance. SwiftPaxos can deal with this using the same procedure as for leader failures, changing the ballot to one with a different fast quorum.

4.4 Optimistic Execution

The execution performed by the leader to compute the result of a command (line 17 in Figure 4) is optimistic: at the time the leader invokes `opt_exec`, the command has not yet been accepted by the followers. However, the followers will always accept the command unless one of them suspects the leader of failure, which happens rarely. Hence, the work done during the optimistic execution is rarely lost.

The optimistic execution in SwiftPaxos can be implemented using existing mechanisms [25, 28, 41, 49]. For ex-

ample, the command's response can be computed by reading the local copy while taking into account prior uncommitted commands in pending_log, and its side effects can be buffered until commit. Such a mechanism is already at work in industrial-grade systems, such as Zookeeper [23].

SwiftPaxos can also be optimized by speculatively executing read-only commands not at the leader, but at any fast quorum replica. A client receives a reply from this replica and, as before, accepts it on the fast path when other fast quorum members report matching dependency paths. This optimization distributes the load across replicas, preventing the leader from saturating. We assess its benefits in §5.6.

5 Evaluation

Our evaluation compares SwiftPaxos against several other protocols, as detailed next. All protocols are written in Go, building on the codebase of EPaxos [38]. The source code is publicly available [1].

***Paxos* [29].** Commands get ordered and disseminated by the leader replica, which is also in charge of replying to clients.

***N²Paxos*.** A variation of Paxos that broadcasts 2B messages (corresponding to our `SlowAcks`) to all replicas. A client receives the response from the site closest to it. This cuts one message delay for clients close to a replica, allowing them to learn a response in 3 message delays instead of 4.

***FastPaxos+* [31].** Clients send their requests directly to all replicas, and a command is committed if enough replicas agree on its ordering. When replicas disagree, the protocol starts a new ballot of N²Paxos. We implement uncoordinated collision recovery [31] (hence the +), which reduces commit latency in exchange for a fixed collision-recovery quorum. In this version replicas locally compute a proposal for the next ballot, bypassing the coordinator and saving one round trip.

***GPaxos* [30].** Generalized Paxos improves Fast Paxos by increasing fast path rate using the commutativity of commands. However, it needs heavy metadata that requires frequent checkpointing [20, 47].

***CURP+*.** CURP [41] boosts leader-based SMR protocols by separating ordering from durability. Each command is optimistically executed at the leader while being stored at $> 3/4$ of replicas ("witnesses"). If there is no conflict, the result of the optimistic execution at the leader is usable right away, without waiting for the command to commit. CURP is a primary-backup replication protocol, i.e., it uses only $f+1$ replicas but requires accurate failure detection. We use its variant for $2f+1$ replicas [41, Appendix B.2] based on N²Paxos, to reduce latency (hence the +). See §B for the details.

***EPaxos* [38].** A leaderless protocol where clients connect to the closest replica, which coordinates access to the replicated service. In the conflict-free case, clients co-located with a replica know the response after 2 message delays. Far-away clients need an extra round trip to receive it. We deploy EPaxos in thrifty mode, which reduces the overall number of sent messages and the size of fast quorums.

***Mencius* [37].** Mencius rotates the role of the consensus leader among the replicas. This spreads the load, but makes the system run at the speed of the slowest replica.

We use two benchmarks: a no-op service and a key-value store with an API following the one of YCSB [13]. The data model is a set of records that are accessed using the commands `insert`, `get`, and `update`. Each record is stored under some key. Two commands conflict when they access the same record and one of them is a write. The no-op service executes commands accessing a random key and carrying a default payload of 1 KB (the standard YCSB payload size). Two commands conflict when they are on the same key. Following standard practice [38], to measure performance under a conflict rate ρ a client chooses key 0 with a probability ρ, and some unique key otherwise.

We deploy the services on Amazon EC2 over 5 replicas in different geographical regions, so that the system tolerates both a failure and a planned outage due to maintenance—a common deployment configuration [14]. Clients are spread over 10 regions all around the world, 2 of which also host replicas. Hence, our experiments use 13 EC2 regions in total (see §C for more details). Both clients and replicas execute in virtual machines running Amazon Linux 2 with 16 vCPUs and 32 GB of main memory. Unless stated otherwise, clients execute commands in a closed loop, waiting for the previous command to return before submitting a new one. Leader-based protocols execute with the leader placed at the site that minimizes the average (mean) latency across all clients. By default, our experiments use configuration (C2) to satisfy (FQI) for both SwiftPaxos and FastPaxos+, as it is slightly more favorable than (C1). We investigate the trade-offs between the two configurations in §5.5.

5.1 Impact of the Conflict Rate

Our first experiment varies the conflict rate from 0 to 100% in 10% increments. Each site hosts 100 clients (1000 clients in total). Figure 7*(a)* presents the speedup relative to Paxos. The latency experienced by clients under each protocol depends on where they are relative to the replicas. The top row of Figure 7*(a)* reports the average (mean) latency across all sites. The middle row reports the speedup observed at the best site for SwiftPaxos, and the bottom row reports the worst.

The latencies of Paxos, N²Paxos, and FastPaxos+ are independent of the conflict rate. On average the improvement of N²Paxos over Paxos is slightly below 1.05x: from 239 ms to 229 ms. This is due to clients co-located with replicas, which benefit from the broadcast of 2B messages. FastPaxos+ has an almost null fast path ratio because it is rare that the replicas receive two concurrent commands (conflicting or not) in the same order. Its slow path follows the message

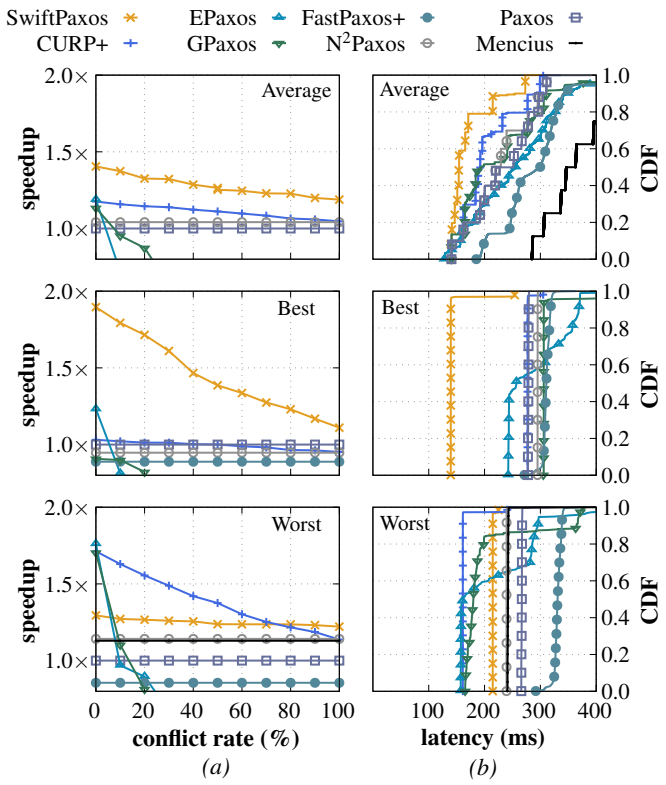


Figure 7: *(a)* Speedup over Paxos when varying the conflict rate and *(b)* latency distribution under 2% conflicts. From top to bottom, we report the average (mean), best, and worst sites for SwiftPaxos.

flow of N^{2}Paxos with one difference: to take the slow path, a replica must first detect a collision (with a majority of 2B messages). This explains why FastPaxos+ is slower than N^{2}Paxos: 300 ms against 229 ms. For this reason, the average latency of FastPaxos+ is omitted from Figure 7*(a)*. The same happens with Mencius, which displays a latency of 360 ms on average.

The average latency of SwiftPaxos is between 170 ms and 201 ms. With a 0% conflict rate this is 1.4× faster than Paxos. Each 10% increase in the conflict rate yields a 1–3% increase in latency. With 100% of conflicts, the speedup is 1.19×. At 7 out of 10 client sites SwiftPaxos is more than 20% faster than Paxos, achieving more than 40% improvement at 3 of them.

At a low conflict rate, EPaxos and GPaxos are faster than Paxos, yet both see their performance decrease abruptly as conflicts increase. This is mainly explained by the drop in the fast path ratio. Another reason is that at 8 out of 10 client sites there is no service replica. These clients need an extra round-trip to get a reply. In EPaxos, performance also deteriorates due to the convoy effect in command execution (§3.4). Thanks to its optimistic execution mechanism, CURP+ is the second best protocol, with a 1.18× speedup on average. Similarly to other protocols, its performance decreases with conflicts.

SwiftPaxos is faster than other protocols at all sites except one (last row in Figure 7*(a)*): this site is too far away from the fast quorum to leverage it. The improvement over CURP+ comes from a better fast path condition and smaller fast path quorums.

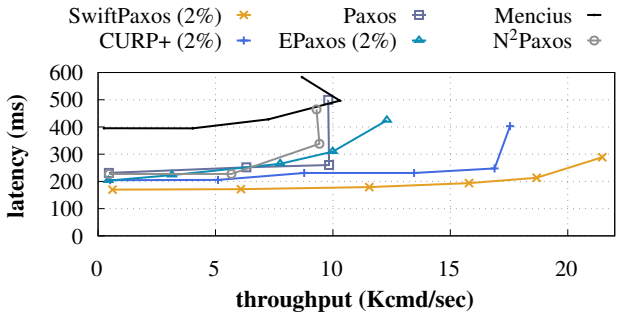


Figure 8: Saturation points for 5 replicas when the total number of clients increases from 100 up to 5000.

5.2 Tail Latency

Figure 7*(b)* reports the cumulative distribution function (CDF) of the latency at clients with 2% of conflicts. As before, we run 100 clients per site. EPaxos has a suboptimal tail latency distribution due to a convoy effect in its execution mechanism (§3.4). GPaxos has similar issues because it requires complex data structures and needs to regularly start new ballots to minimize metadata. Other leader-based protocols do not suffer from tail latency issues.

5.3 Metadata Usage

The leader in Paxos sends the payload of each command to the quorum of replicas. This is not the case with SwiftPaxos, because the protocol uses command identifiers for ordering. As a consequence, SwiftPaxos can be cheaper than Paxos when contention is rare. With 0% conflicts, metadata in SwiftPaxos consumes 2.83 GB in the experiment of Figure 7*(a)*. This is 1.16× better than Paxos (3.27 GB). The protocol message complexity increases with more conflicts. In Figure 7*(a)* with 100% conflicts data consumption is 3.36 GB, around 3% more than Paxos. At a replica, the dependency graph grows over time, but shrinks due to garbage collection (see §4.1). In Figure 7*(a)*, the average size of a command's dependency set is only 24 B. On average, the dependency graph at a replica is about 8% of the protocol's memory usage.

5.4 Scalability

To evaluate the scalability of SwiftPaxos, we run an experiment that progressively brings it to saturation. The results are reported in Figure 8. The total number of clients increases from 100 to 5000, and the payload of each command is set to 3 KB (this payload size is chosen so as to saturate the system with a reasonable number of client machines).

During this experiment, Paxos and N^{2}Paxos display similar behavior. The two protocols saturate when 2500 clients are deployed across the 10 sites. This is due to the fact that the leader is in charge of broadcasting the commands to the

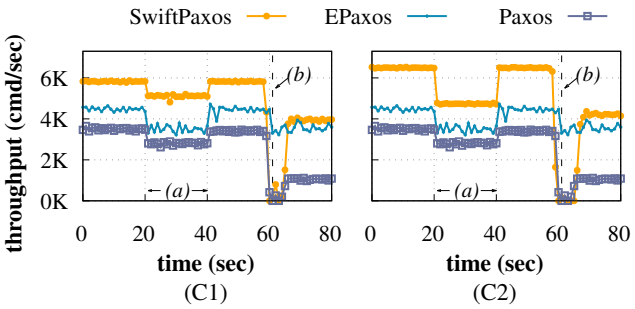


Figure 9: An execution for configurations (C1) and (C2) where *(a)* a replica slows down and *(b)* the leader fails.

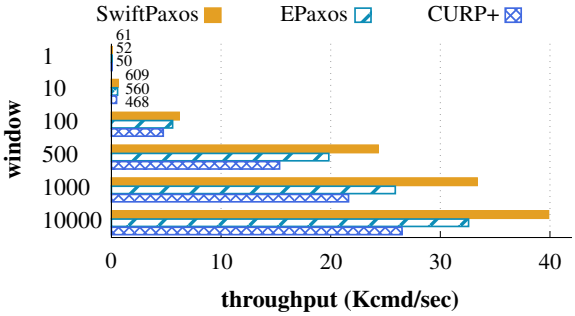


Figure 10: Throughput under a pipelined workload with different window sizes.

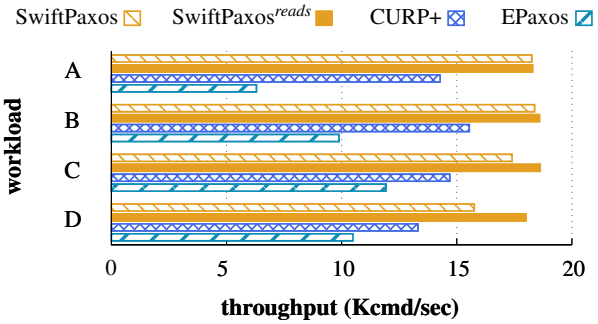


Figure 11: Throughput under different YCSB workloads.

replicas. In the absence of conflicts, the throughput of EPaxos is around 24% higher than that of Paxos (7.8K ops/s against 6.3K ops/s) at almost the same average latency (≈260 ms). When we further increase the system load, responsiveness is affected. Saturation occurs at 400 clients per site, where the system delivers 12K ops/s at an average latency of 424 ms.

In contrast to Paxos, leader-based algorithms with fast paths (CURP+ and SwiftPaxos) do not funnel commands through the leader. Instead, each client is responsible for sending its commands to the replicas. This increases the throughput of the protocol by (at least) 30% on average.

5.5 Performance under Asynchrony

Figure 9 compares how SwiftPaxos, EPaxos, and Paxos behave under asynchrony. We run this experiment for both configurations (C1) and (C2) with 100 clients per site. For (C2) the fast quorum of SwiftPaxos and the lowest-latency quorum of Paxos are the same. For both configurations after 20 s, the latency at one site increases by 200 ms, and it drops back to normal after another 20 s (interval *(a)*). Then the leader fails (event *(b)*).

Before and after the first slowdown, SwiftPaxos performs slightly better in configuration (C2), confirming that a single majority fast quorum is beneficial in our configuration. However, during the slowdown the throughput of (C2) decreases by 27%, whereas it decreases only by 12% for (C1). The degradation is smoother in the latter case because (C1) allows multiple fast quorums, and thus makes the fast path more robust. In Figure 9, SwiftPaxos always outperforms Paxos. This is because, in the worst case, the protocol can commit a command using a slow quorum (see §3.3). Upon event *(b)*, the two algorithms recover at similar speeds. The gap is the largest with configuration (C2) where SwiftPaxos and Paxos take respectively 7 s and 6 s. EPaxos is the most stable protocol because it is leaderless: during the two experiments, its throughput never drops by more than 20%.

5.6 Applications

We now evaluate the protocols under two representative application scenarios. In a first scenario, we consider applications that are pipelining state-machine commands. Such a pattern occurs in log replication as well as in pub/sub architectures. In the second scenario, we run the Yahoo! Cloud Serving Benchmark (YCSB) [13].

Pipelining (Figure 10). In this synthetic benchmark clients pipeline commands to the protocol. This pattern is common in distributed applications, for instance in pub/sub systems such as Apache Kafka [27]: in this case all the messages published by the producer are appended to a log before being read by the consumers. A similar pattern is used when updating an object o accessible through a reference r [23]. The client creates a new version o' of o via asynchronous changes, and then atomically updates r to store the address of o'.

In this experiment, we deploy a client at every datacenter. Each client pipelines commands accessing the same key. Each command carries a payload of 4 KB. The window parameter determines the maximum number of in-flight commands.

Even though commands are received in the same order at the replicas, CURP+ never takes the fast path. This is because witnesses in CURP+ cannot process more than one conflicting command at a time. Conversely, because the run is contention-free, replicas in EPaxos and SwiftPaxos compute the same dependencies, enabling the fast path. On average, both protocols improve over CURP+ by 24% and 49%, respectively.

YCSB (Figure 11). This benchmark consists of workloads with zipfian-like access patterns: workload A is update-heavy (20% `update`, 80% `get`), B is read-heavy (95% `get`, 5% `update`) and C is read-only. Workload D contains repeated

reads (95% of the calls) mixed with insertions of new records. The key-value store service holds 10^6 records. A record has 10 fields of 800 B each (8 KB in total). We run 4000 YCSB clients scattered around the world, 400 per site.

In Figure 11, SwiftPaxos^{reads} indicates that read requests are optimistically executed at the closest fast quorum replica, and thus not necessarily at the leader (§4.4). Paxos is not evaluated because, as shown in Figures 7 and 8, EPaxos performs better than Paxos under low conflict rates. This is the case in YCSB, as the benchmark is read-dominated.

EPaxos is most efficient with the read-only workload (C). In this case, all commands execute after a single round trip to the closest quorum, and only the replica that submitted a command executes it. In Workload C, the performance of CURP+ and SwiftPaxos without optimized reads is limited, as only the leader responds to clients. When reads are optimistically executed at any fast quorum replica, SwiftPaxos^{reads} improves over the base version of the protocol, from 17K ops/s to 19K ops/s. In Workload D, the protocols behave similarly to Workload C, as there are also no conflicts.

Workloads A and B contain respectively 20% and 5% of writes. The access distribution is zipfian, with 20% of chance to access the 12 most popular records. In Workload B, CURP+ and SwiftPaxos are around 1.8× faster than EPaxos. This difference is due to the high ratio of slow paths and the convoy effect: 5% of the slowest commands take 470 ms to execute. This performance gap further increases with workload A, where CURP+ and SwiftPaxos provide respectively 2.2× and 2.9× improvement over EPaxos.

6 Related Work

Standard SMR protocols [22, 29, 39] are leader-driven, requiring 4 message delays for a client to receive a response: a round-trip from the client to the leader plus a round-trip from the leader to the replicas. Replicas can compute the response earlier if they exchange Paxos 2B messages. This cuts one message delay for clients close to a replica, but faraway clients still need one more message delay to get notified. We evaluate this variation of Paxos in §5. SDPaxos [52] separates durability from ordering. As the leader still orchestrates ordering, clients not co-located with a replica wait for 4 message delays for a response.

Detecting conflicts to boost parallelism has a long history in distributed systems [10, 34, 49, 51]. Fast Paxos [31] was the first SMR protocol that used fast paths to minimize latency, allowing clients to contact replicas directly. Later work leveraged commutativity to increase the chances of taking the fast path [30, 42, 47]. As explained in §1, leader-based protocols following this approach can only resolve a conflict via a ballot change, which may require extensive state transfer between replicas and disrupts the system.

EPaxos, a leaderless SMR protocol, reduces the average latency under low conflict rates. But as we noted in §3.4 and §5.2, its tail latency is high due to a convoy effect in command execution [6, 19, 50]. It also requires clients to route requests via a data replica, increasing latency. As shown in [18], follow-ups to EPaxos, such as Caesar [3] and Atlas [17], suffer from the same problems. Tempo [18] uses a decentralized timestamping mechanism to reduce the convoy effect in leaderless SMR, but does not fully eliminate it. Gryff [6] mixes EPaxos with ABD [4]. The protocol speeds up blind writes, but like in ABD, has expensive reads [15]. Mencius [37] distributes leader responsibilities round-robin. When conflicts are rare, this protocol is slower than EPaxos [17, 38].

Several protocols rely on optimistic execution to boost performance. Speculative Paxos [44] enforces spontaneous ordering in the network to obtain identical optimistic execution at all replicas. Eve [25] executes state-machine commands optimistically in parallel, failing back to sequential execution if the results do not match. In practice, clients might not be located near a service replica [2,5]. For such clients, CURP [41] speeds up the response by executing optimistically commands at the leader. CURP does not use dependencies but instead computes a total order. In particular, a client accepts the result of optimistic execution only if it is conflict-free. SwiftPaxos is more permissive, allowing the result to be used as long as the dependency paths match.

Some protocols leverage access locality to boost SMR performance [11,12,43]. These protocols use Paxos as a black box, invoking it one or more times per command. Spanner [14] and CockroachDB [48] also use it to implement the more complex abstraction of strongly consistent transactions. All these systems may benefit from the performance improvement brought by SwiftPaxos.

Some recent works [24, 35] use dedicated network components to implement fast SMR protocols. But this approach has not yet been applied to geo-distributed systems.

7 Conclusion

Over the past decade a plethora of protocols was proposed to improve SMR in geo-distributed systems. Unfortunately, these protocols may deliver a lower performance than Paxos when contention on the replicated service increases. SwiftPaxos does not have this drawback. It executes a command in 2 message delays if there is no contention, and in 3 message delays otherwise. Our experimental evaluation demonstrates the benefits of this design. SwiftPaxos delivers 16–29% lower average latency than Paxos, and its throughput is up to 2.9× that of EPaxos.

Acknowledgements. This work was partially supported by the RACCOON project funded by the European Research Council, the PRODIGY and DECO projects funded by MCIN/AEI, and the BLOQUES project funded by the Madrid regional government. The authors thank the anonymous reviewers and their shepherd, Jay Lorch.

References

[1] SwiftPaxos codebase. `https://github.com/imdea-software/swiftpaxos`.

[2] M. S. Ardekani and D. B. Terry. A Self-Configurable Geo-Replicated Cloud Storage System. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[3] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran. Speeding up Consensus by Chasing Fast Decisions. In *Conference on Dependable Systems and Networks (DSN)*, 2017.

[4] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM*, 42(1), 1995.

[5] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Conference on Innovative Data Systems Research (CIDR)*, 2011.

[6] M. Burke, A. Cheng, and W. Lloyd. Gryff: Unifying Consensus and Shared Registers. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[7] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[8] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2), 1996.

[9] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips. Giza: Erasure Coding Objects across Global Data Centers. In *USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[10] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. *ACM Transactions on Computer Systems*, 32(4), 2015.

[11] P. R. Coelho and F. Pedone. Geographic State Machine Replication. In *Symposium on Reliable Distributed Systems (SRDS)*, 2018.

[12] P. R. Coelho and F. Pedone. GeoPaxos+: Practical Geographical State Machine Replication. In *Symposium on Reliable Distributed Systems (SRDS)*, 2021.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Symposium on Cloud Computing (SoCC)*, 2010.

[14] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-Distributed Database. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Symposium on Operating Systems Principles (SOSP)*, 2007.

[16] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2), 1988.

[17] V. Enes, C. Baquero, T. França Rezende, A. Gotsman, M. Perrin, and P. Sutra. State-Machine Replication for Planet-Scale Systems. In *European Conference on Computer Systems (EuroSys)*, 2020.

[18] V. Enes, C. Baquero, A. Gotsman, and P. Sutra. Efficient Replication via Timestamp Stability. In *European Conference on Computer Systems (EuroSys)*, 2021.

[19] T. França Rezende and P. Sutra. Leaderless State-Machine Replication: Specification, Properties, Limits. In *International Symposium on Distributed Computing (DISC)*, 2020.

[20] T. França Rezende, P. Sutra, R. Q. Saramago, and L. J. Camargos. On Making Generalized Paxos Practical. In *International Conference on Advanced Information Networking and Applications (AINA)*, 2017.

[21] M. P. Herlihy and J. M. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3), 1990.

[22] H. Howard, D. Malkhi, and A. Spiegelman. Flexible Paxos: Quorum Intersection Revisited. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2016.

[23] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference (USENIX ATC)*, 2010.

[24] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. Netchain: Scale-Free Sub-RTT Coordination. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[25] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[26] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-Data Center Consistency. In *European Conference on Computer Systems (EuroSys)*, 2013.

[27] J. Kreps, N. Narkhede, and J. Rao. Kafka: a Distributed Messaging System for Log Processing. *Workshop on Networking Meets Databases (NetDB)*, 2011.

[28] R. Ladin, B. Liskov, and L. Shrira. Lazy Replication: Exploiting the Semantics of Distributed Services. In *Symposium on Principles of Distributed Computing (PODC)*, 1990.

[29] L. Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2), 1998.

[30] L. Lamport. Generalized Consensus and Paxos. Technical report, Microsoft Research, 2005.

[31] L. Lamport. Fast Paxos. *Distributed Computing*, 19, 2006.

[32] L. Lamport, D. Malkhi, and L. Zhou. Vertical Paxos and Primary-Backup Replication. In *Symposium on Principles of Distributed Computing (PODC)*, 2009.

[33] L. Lamport and M. Massa. Cheap Paxos. In *Conference on Dependable Systems and Networks (DSN)*, 2004.

[34] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making Geo-Replicated Systems Fast as Possible, Consistent When Necessary. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[35] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[36] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *Symposium on Operating Systems Principles (SOSP)*, 1991.

[37] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machine for WANs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[38] I. Moraru, D. G. Andersen, and M. Kaminsky. There Is More Consensus in Egalitarian Parliaments. In *Symposium on Operating Systems Principles (SOSP)*, 2013.

[39] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[40] R. Pang, R. Cáceres, M. Burrows, Z. Chen, P. Dave, N. Germer, A. Golynski, K. Graney, N. Kang, L. Kissner, J. L. Korn, A. Parmar, C. D. Richards, and M. Wang. Zanzibar: Google's Consistent, Global Authorization System. In *USENIX Annual Technical Conference (USENIX ATC)*, 2019.

[41] S. J. Park and J. Ousterhout. Exploiting Commutativity For Practical Fast Replication. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[42] F. Pedone and A. Schiper. Generic Broadcast. In *International Symposium on Distributed Computing (DISC)*, 1999.

[43] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran. Making Fast Consensus Generally Faster. In *Conference on Dependable Systems and Networks (DSN)*, 2016.

[44] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Conference on Networked Systems Design and Implementation (NSDI)*, 2015.

[45] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22, 1990.

[46] A. Shraer, B. Reed, D. Malkhi, and F. P. Junqueira. Dynamic reconfiguration of primary/backup clusters. In *USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[47] P. Sutra and M. Shapiro. Fast Genuine Generalized Consensus. In *Symposium on Reliable Distributed Systems (SRDS)*, 2011.

[48] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *International Conference on Management of Data (SIGMOD)*, 2020.

[49] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Symposium on Operating Systems Principles (SOSP)*, 1995.

[50] S. Tollman, S. J. Park, and J. K. Ousterhout. EPaxos Revisited. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[51] W. E. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Transactions on Computers*, 37(12), 1988.

[52] H. Zhao, Q. Zhang, Z. Yang, M. Wu, and Y. Dai. SDPaxos: Building Efficient Semi-Decentralized Geo-replicated State Machines. In *Symposium on Cloud Computing (SoCC)*, 2018.

A Correctness

4. If at a replica $\mathsf{cmd}[id] = c \neq \bot$ then $\mathtt{id}(c) = id$ and $\mathtt{Propagate}(c)$ is a sent message.
5. At a replica, whenever a command c is in the START phase, it does not belong to the dependencies of any command.
6. When a replica p sends an $\mathtt{Ack}$ message, $\mathsf{cbal} = \mathsf{bal}$ at p.
7. At a ballot b
   - if $\mathsf{leader}(b)$ sends $\mathtt{Ack}(b, id, D, _)$ and $\mathtt{Ack}(b, id, D', _)$ then $D = D'$;
   - if $\mathsf{leader}(b)$ sends $\mathtt{Ack}(b, id, D, _)$ and $\mathtt{Sync}(b, _, _, dep)$ then $dep[id] = D$;
   - if $\mathsf{leader}(b)$ sends $\mathtt{Sync}(b, phase, cmd, dep)$ and $\mathtt{Sync}(b, phase', cmd', dep')$ then $phase = phase'$, $cmd = cmd'$ and $dep = dep'$.
8. If a replica sends an $\mathtt{Ack}$ message at a ballot b and a $\mathtt{NewLeaderAck}(b', _, _, _, _)$ message with $b' > b$ then it sends $\mathtt{Ack}$ before sending $\mathtt{NewLeaderAck}$.
9. If the leader of a ballot b sends $\mathtt{Ack}(b, _, _, P)$ after sending $\mathtt{Ack}(b, id, D, _)$ for $id \in Ids(P)$ then $D = Dep(id, P)$.
10. If a replica has $\mathsf{cbal} = b$, $\mathsf{dep}[id] = D$ and $id \in Accept \cup Commit$ then there is a moment when $\mathsf{leader}(b)$ has $\mathsf{cbal} = b$ and $\mathsf{status} = \text{NORMAL}$, and after which whenever it has $\mathsf{cbal} = b$ and $\mathsf{status} = \text{NORMAL}$ it also has $\mathsf{dep}[id] = D$.
11. If $\Diamond acc(b, id, c, P)$ and $\Diamond acc(b, id, c', P')$ hold then $c = c'$ and $Dep(id, P) = Dep(id, P')$.
12. If $\Diamond acc(b, _, _, P)$ then for all $id \in Ids(P)$, whenever a replicas has $\mathsf{cbal} = b$ and $id \in Accept \cup Commit$, it also has $\mathsf{dep}[id] = Dep(id, P)$.
13. Whenever at a replica a command c is in the ACCEPT or the COMMIT phase, any transitive dependency of c is in the ACCEPT or the COMMIT phase as well.
14. If $\Diamond acc(b, _, _, P)$ holds then for all $id \in Ids(P)$ whenever at a replica we have $\mathsf{cbal} > b$, we also have $\mathsf{dep}[id] = Dep(id, P)$ and $id \in Accept \cup Commit$.

Figure 12: Additional invariants of SwiftPaxos.

In this section we prove Invariants 1–3 (defined in §3.2) together with additional low-level invariants listed in Figure 12. We then show how these invariants imply that SwiftPaxos satisfies the SMR protocol specification (§2).

Definition 1 (*acc*)**.** We say that a command c with an identifier id is *accepted* with dependency paths P at a ballot b—and we write $acc(b, id, c, P)$—if at b there is a fast or slow quorum such that each replica of this quorum acknowledges P as the dependency paths of c with $\mathtt{FastAck}$ or $\mathtt{SlowAck}$ messages.

In the following we say that a replica sends $\mathtt{Ack}(b, id, D, P)$ if it sends $\mathtt{FastAck}(b, id, D, P)$ or $\mathtt{SlowAck}(b, id)$ and by that time it has $\mathsf{dep}[id] = D$ and $\mathtt{pset}(id) = P$.

We write $\Diamond acc(b, id, c, P)$ to denote that at some point of protocol execution $acc(b, id, c, P)$ holds.

Definition 2 (*Ids*)**.** We say that *id appears* in a set of dependency paths P (or simply *id in P*) if there exists at least one path in P that contains id. We define $Ids(P)$ as the set of all id in P.

Definition 3 (*Dep*)**.** For a set of dependency paths P and $id \in Ids(P)$, we define $Dep(id, P)$ as the dependency set of id inferred from P.

Note that by definitions of *acc* and dependency paths, if $\Diamond acc(b, id_0, c, P)$ holds for some quorum Q then at the moment when a replica $q \in Q$ sends $\mathtt{Ack}(b, id_0, _, P)$ message it has $\mathsf{dep}[id] = Dep(id, P)$ for all $id \in Ids(P)$.

Invariants 4–10 easily follow from the structure of the protocol. We now prove the rest of the invariants.

Proof of Invariant 11. Each time a command c is accepted at a ballot the leader of this ballot must acknowledge c's dependencies either with an Ack or a Sync message. By Invariant 7 the leader cannot acknowledge two distinct dependency sets for the same command at a single ballot. □

Proof of Invariant 12. Assume $\Diamond acc(b, id_0, _, P)$ and let id be any identifier in $Ids(P)$. Take a replica p with $\mathsf{cbal} = b$ and $id \in Accept \cup Commit$. Two cases are possible: either p receives $\texttt{Ack}(b, id, D, _)$ from $\mathsf{leader}(b)$, or it receives Sync carrying id in the ACCEPT or COMMIT phase.

First assume that p receives $\texttt{Ack}(b, id, D, _)$ from the leader of b. Since $\Diamond acc(b, id_0, _, P)$ holds, the leader also sends $\texttt{Ack}(b, id_0, _, P)$. Then by Invariant 9, we have $D = Dep(id, P)$. Hence, when p receives $\texttt{Ack}(b, id, D, _)$ from $\mathsf{leader}(b)$ it sets $\mathsf{dep}[id]$ to $D = Dep(id, P)$ and after that—according to Invariant 7—$\mathsf{dep}[id]$ remains unchanged at b, as required.

Suppose now, that p receives Sync carrying id in the ACCEPT or COMMIT phase with the dependency set D. Then any replica with $\mathsf{cbal} = b$ receives that message, sets $\mathsf{dep}[id]$ to D and by Invariant 7 never changes it at b. When $\mathsf{leader}(b)$ sends $\texttt{Ack}(b, id_0, _, P)$ message it has $\mathsf{dep}[id] = Dep(id, P) = D$, as required. □

Proof of Invariant 13. We prove the invariant by induction on the length of the protocol execution. Initially at any replica each command is in the START phase and hence, the invariant is satisfied. We now consider transitions at lines 22, 30, and 48, as they are the only transitions affecting the validity of the invariant in a nontrivial way.

- Transition at line 22. According to line 23 after the execution of the handler at line 22 for a command c, any dependency of c is in the ACCEPT or COMMIT phase and hence, by induction hypothesis any transitive dependency of c satisfies the invariant.
- Transition at line 30. The handler simply moves an ACCEPTed command to the COMMIT phase, which allows us to conclude with the induction hypothesis.
- Transition at line 48. By the structure of the protocol after a replica p executes this handler all the commands are in the START, ACCEPT or COMMIT phase at p. From Invariant 5 the commands that are in the START phase do not belong to the dependency set of any command, hence our property.

□

Proof of Invariant 3. By the induction on the length of the protocol execution: at the beginning of the execution dep is empty at each replica, which makes the dependency graphs acyclic; for the induction step we consider transitions at lines 6, 22, and 48—the only transitions affecting the validity of the invariant in a nontrivial way.

- Transition at line 6. In order to introduce a cycle upon receipt of a $\texttt{Propagate}(c)$ message at a replica p, $\texttt{id}(c)$ must belong to the dependencies of some other command before the transition is triggered. However, according to the precondition at line 7 at this moment c is in the START phase and, thus, from Invariant 5 no command depends on it, as required.
- Transition at line 22. We prove this case by contradiction. Assume that right after a replica p executed the handler at line 22 for a command c, some command transitively depends on itself at p. According to Invariants 10 and 13, and the precondition at line 23, all commands of this cycle are in sync with the leader at the time when the leader sends Ack for c. However, by induction hypothesis, by that time the dependency graph at the leader is acyclic, which yields a contradiction.
- Transition at line 48. According to line 64, after a replica executes the handler at line 48 its dependency graph is acyclic, as required.

□

The proof of Invariant 2 relies on the following proposition:

Proposition 1. Whenever at a replica p we have $id \notin p.\mathsf{dep}[id']$ and $id' \notin p.\mathsf{dep}[id]$ for two identifiers id and id' of two conflicting commands:

1. p is not the leader of its ballot (i.e., $p \neq \mathsf{leader}(p.\mathsf{cbal})$);
2. at p one of two identifiers is in the PREACCEPT phase while the other is in the ACCEPT or the COMMIT phase.

Proof. We prove the proposition by induction on the length of the protocol execution. The property holds at the beginning of the execution because no commands are recorded at any of replicas. For the induction step we consider transitions at lines 6, 22, and 48, as they are the only transitions affecting the validity of the proposition in a nontrivial way.

- Transition at line 6. Consider a moment when a replica p receives a $\texttt{Propagate}(c')$ message and take any command c that conflicts with c' and which is already recorded in $p.\mathsf{cmd}$. According to line 11, after the execution of handler at line 6 we have $\texttt{id}(c) \in p.\mathsf{dep}[\texttt{id}(c')]$.

- Transition at line 22. Assume that p receives a $\texttt{FastAck}(b, id, D, _)$ message for a command c from the leader. Take any command conflicting with c with an identifier id' that is already recorded at p at the moment of the execution of transition. Let D' be $p.\mathsf{dep}[id']$ at this moment.

  If p is the leader of b then it receives its own proposal, and thus the handler is not affecting dep at p. By induction hypothesis, before executing this transition process p has $id \in p.\mathsf{dep}[id']$ or $id' \in p.\mathsf{dep}[id]$, as required.

  Suppose that $p \neq \mathsf{leader}(b)$. If by the moment p executes line 22 it has $\mathsf{phase}[id'] = \text{PREACCEPT}$ then the transition does not affect the validity of the proposition because in this case p moves id to the ACCEPT phase without changing $\mathsf{phase}[id']$. Thus, p has $\mathsf{phase}[id] = \text{ACCEPT}$ and $\mathsf{phase}[id'] = \text{PREACCEPT}$ after executing the transition, as required. If $\mathsf{phase}[id'] \in \{\text{ACCEPT}, \text{COMMIT}\}$ then after the execution of the transition both identifiers are in the ACCEPT or the COMMIT phase. Therefore, from Invariant 10 there is a moment when $\mathsf{leader}(b)$ has $\mathsf{dep}[id] = D$ and $\mathsf{dep}[id'] = D'$. By induction hypothesis at this moment $\mathsf{leader}(b)$ has $id \in \mathsf{dep}[id']$ or $id' \in \mathsf{dep}[id]$, as required.
- Transition at line 48. Let id and id' be two identifiers of conflicting commands such that process p has $\mathsf{dep}[id] = D \neq \bot$ and $\mathsf{dep}[id'] = D' \neq \bot$ after executing transition at line 48. By the structure of the algorithm $id \in Accept \cup Commit$ and $id' \in Accept \cup Commit$ (lines 54 and 59). We prove that either $id \in \mathsf{dep}[id']$ or $id' \in \mathsf{dep}[id]$. We know that both identifiers satisfy condition at line 53 or condition at line 58. If both identifiers satisfy condition at line 53 then at some point $\mathsf{leader}(b_{\max})$ has $\mathsf{dep}[id] = D$ and $\mathsf{dep}[id'] = D'$ (Invariant 10). By induction hypothesis by this moment either $id \in D'$ or $id' \in D$, as required. Assume now that at least one command satisfies condition at line 58. According to line 61 either $id \in \mathsf{dep}[id']$ or $id' \in \mathsf{dep}[id]$.

□

Proof of Invariant 2. Whenever two conflicting commands are committed at a replica p both of them are in the COMMIT phase, which by Proposition 1 means that one of the two is a dependency of the other. □

Proposition 2. If $\Diamond acc(b, _, _, P)$ then whenever there is a quorum Q such that each replica $q \in Q$ sends the $\texttt{NewLeaderAck}(b', cbal_q, phase_q, _, dep_q)$ message with $b' > b$ and $cbal_q \leq b$, the following property holds for any $id \in Ids(P)$:

> There is a quorum $Q' \in \mathcal{FQ}(b) \cup \mathcal{SQ}(b)$ such that for any replica $q \in Q \cap Q'$ we have $dep_q[id] = Dep(id, P)$, $cbal_q = b$ and if $Q' \in \mathcal{SQ}(b)$ then $\forall q \in Q \cap Q'.\, phase_q[id] \in \{\text{ACCEPT}, \text{COMMIT}\}$.

Proof. Assume $\Diamond acc(b, id_0, _, P)$ and let Q be a quorum such that each replica $q \in Q$ sends a $\texttt{NewLeaderAck}(b', cbal_q, phase_q, _, dep_q)$ message with $b' > b$ and $cbal_q \leq b$. Let $id \in Ids(P)$. Since $\Diamond acc(b, id_0, _, P)$, there exists a quorum $Q' \in \mathcal{FQ}(b) \cup \mathcal{SQ}(b)$ such that each replica in Q' sends $\texttt{Ack}(b, id_0, _, P)$ message, and by that time it has $\mathsf{dep}[id] = Dep(id, P)$. After sending Acks, replicas in Q' can change $\mathsf{dep}[id]$ at b only if they receive an Ack message from $\mathsf{leader}(b)$ with a proposal different from D. However, according to Invariant 7, once $\mathsf{leader}(b)$ has voted for D it never sends a new proposal at b. Thus, after sending Ack, replicas in Q' never change $\mathsf{dep}[id]$ at b. Moreover, from Invariant 8 we know that at the moment a replica in Q' sends a NewLeaderAck message it has $\mathsf{cbal} \geq b$ and since we assume that for all $q \in Q$ we have $cbal_q \leq b$ we also have $\forall q \in Q \cap Q'$, $cbal_q = b$. Hence, when a replica $q \in Q \cap Q'$ sends the NewLeaderAck message, it has $dep_q[id] = Dep(id, P)$, as required. If now we assume that $Q' \in \mathcal{SQ}(b)$ then all replicas in Q' send $\texttt{SlowAck}(b, id)$ messages, implying that for all $q \in Q \cap Q'$, $phase_q[id] \in \{\text{ACCEPT}, \text{COMMIT}\}$ (lines 24 and 75). □

Proof Invariant 14. We prove the invariant by induction on the value of cbal. Assume $\Diamond acc(b, id_0, _, P)$ and consider a ballot $b' > b$ such that

> For all $id \in Ids(P)$, whenever a replica has $b < \mathsf{cbal} < b'$, it also has $\mathsf{dep}[id] = Dep(id, P)$ and $id \in Accept \cup Commit$. (H)

We prove by induction on the length of the protocol execution that

> For all $id \in Ids(P)$, whenever a replica has $\mathsf{cbal} = b'$, it also has $\mathsf{dep}[id] = Dep(id, P)$ and $id \in Accept \cup Commit$. (1)

At the beginning of the execution at each replica we have $\mathsf{cbal} = 0 \leq b < b'$, hence (1) trivially holds. For the induction step, only handler at line 48 affects validity of the invariant in a nontrivial way. Let p be a replica that receives a $\texttt{NewLeaderAck}(b', cbal_q, phase_q, _, dep_q)$ message from each $q \in Q$. Let $b_{\max} = \texttt{max}\{cbal_q \mid q \in Q\}$ and $U = \{q \in Q \mid cbal_q = b_{\max}\}$. We prove that after executing transition at line 48, p has $\mathsf{dep}[id] = Dep(id, P)$ and $id \in Accept \cup Commit$ for all $id \in Ids(P)$.

We first note that $b_{\max} \geq b$. Indeed, from Invariant 8 replicas in Q that have voted for $Dep(id, P)$ at b send an Ack message before NewLeaderAck, hence $b_{\max} \geq b$.

As the first step, we prove that after executing transition at line 48 up to line 61, p has $\mathsf{dep}[id] = Dep(id,P)$ and $id \in Accept \cup Commit$ for any $id \in Ids(P)$. We then prove that after executing lines 62–64 the dependency sets of all identifiers in $Ids(P)$ remain unchanged.

Step 1. Take any $id \in Ids(P)$ and c such that $\mathtt{id}(c) = id$.

- Suppose that $b_{\max} > b$. By (H), id satisfies condition at line 53. Process p sets $\mathsf{dep}[id]$ to $Dep(id,P)$ and $\mathsf{phase}[id]$ to ACCEPT or COMMIT, as required.
- Suppose now that $b_{\max} = b$ (i.e., $\forall q \in Q.\, cbal_q \leq b$). From Proposition 2, either the condition at line 53 or the one at line 58 is satisfied for id. Assume that the condition at line 53 holds: there exists a replica $q \in Q$ such that $cbal_q = b$ and $phase_q[id] \in \{\text{ACCEPT}, \text{COMMIT}\}$. Then at line 56 process p sets $\mathsf{dep}[id]$ to $dep_q[id] = Dep(id,P)$ (Invariant 12) and $\mathsf{phase}[id]$ to $phase_q[id]$, as required. Assume now, that the condition at line 58 holds. Then for all $q \in U$ we have $phase_q[id] \notin \{\text{ACCEPT}, \text{COMMIT}\}$ and there exists a quorum $Q' \in \mathcal{FQ}(b_{\max})$ such that $Q \cap Q' \subseteq U$ and all replicas in $Q \cap Q'$ report the same dependency set D for id. From Proposition 2, there exists a quorum Q_0 such that $Q \cap Q_0 \subseteq U$ and that for any replica in $q \in Q \cap Q_0$, $dep_q[id] = Dep(id,P)$. Since $Q \cap Q' \cap Q_0 \neq \emptyset$, we have $D = Dep(id,P)$. We now prove by contradiction that $\mathsf{dep}[id]$ computed at line 61 equals D. Assume the converse. Then there exists a command conflicting with c with an identifier id' such that $id' \notin D$ and by the time p executes line 61 for id it has $id \notin \mathsf{dep}[id']$ and $id' \notin Start$. Let D' be $\mathsf{dep}[id']$ at that moment. Since $id' \notin Start$ either condition at line 53 or the one at line 58 is satisfied for id' with D'.
  (a) Assume that condition at line 53 is satisfied. There exists a replica $q \in U$ such that $phase_q[id'] \in \{\text{ACCEPT}, \text{COMMIT}\}$ and $dep_q[id'] = D'$, with $id \notin D'$. From Invariant 10 there is a moment t after which whenever $\mathsf{leader}(b)$ has $\mathsf{cbal} = b$ it also has $\mathsf{dep}[id'] = D'$. Moreover, there is also a moment t' when $\mathsf{leader}(b)$ sends the $\mathtt{Ack}(b, id_0, _, P)$ message, and by this moment it has $\mathsf{dep}[id] = Dep(id,P) = D$. Therefore, at $\max(t,t')$, the leader of b has $\mathsf{dep}[id] = D$ and $\mathsf{dep}[id'] = D'$, with $id \notin D'$ and $id' \notin D$, contradicting Proposition 1.
  (b) Suppose that the condition at line 58 is satisfied for id': there exists a fast quorum Q'' such that $Q \cap Q'' \subseteq U$ and all replicas in $Q \cap Q''$ report D' as the dependency set of id'. Since $Q \cap Q'' \cap Q_0 \neq \emptyset$, there is a replica in $q \in Q \cap Q''$ with $dep_q[id] = Dep(id,P)$ and $dep_q[id'] = D'$. Recall that we are under the assumption that both identifiers, id and id', satisfy condition at line 58. Therefore, they also satisfy loop condition at line 57. As a result we have $\forall q' \in U.\, phase_{q'}[id] \notin \{\text{ACCEPT}, \text{COMMIT}\} \wedge phase_{q'}[id'] \notin \{\text{ACCEPT}, \text{COMMIT}\}$. Thus, $phase_q[id] = \text{PREACCEPT}$ and $phase_q[id'] = \text{PREACCEPT}$, contradicting Proposition 1.

  We thus have proved that $\mathsf{dep}[id]$ computed at line 61 equals $D = Dep(id,P)$, as required.

Step 2. Now that we have proved that after executing transition at line 48 up to line 61, p has $\mathsf{dep}[id] = Dep(id,P)$ for all $id \in Ids(P)$, we prove that the loop at line 62 does not affect these dependencies. Let $id \in Ids(P)$ and let id' be any identifier in $\mathsf{dep}[id]$ at p at the moment when p executes lines 62–63. Since at this moment $\mathsf{dep}[id] = Dep(id,P)$, we have $id' \in Ids(P)$ (by definition of Ids). As we saw earlier, either $b_{\max} = b$ or $b_{\max} > b$ and according to Proposition 2 and (H), id' either satisfies condition at line 53 or condition at line 58. Thus, at the moment p executes lines 62–63 it has $id' \notin Start$ and hence condition at line 62 does not hold for (id, id'), as required.

Step 3. Finally we prove by contradiction that after executing line 64, p has $\mathsf{dep}[id] = Dep(id,P)$ for all $id \in Ids(P)$. Assume that at p, just before the execution of line 64 there is a cycle $id_1 \to \cdots \to id_n \to id_1$ in the dependency graph. We prove by contradiction that none of the identifiers of this cycle belongs to $Ids(P)$. Assume the converse, i.e., there exists $m \in [1;n]$ such that $id_m \in Ids(P)$. We have already proved that prior to executing line 64 process p has $\mathsf{dep}[id] = Dep(id,P)$ for all $id \in Ids(P)$. Thus $id_{m-1} \in Dep(id_m,P)$ also belongs to $Ids(P)$. Moreover, all identifiers of the cycle are in $Ids(P)$ because each of them is a transitive dependency of each other. Hence, when $\mathsf{leader}(b)$ sends an $\mathtt{Ack}(b, id_0, _, P)$ message it has this cycle in its dependency graph, contradicting Invariant 3.

□

Proof of Invariant 1. Take two replicas that commit a command c with an identifier id. We have $\Diamond acc(b, id, c, P)$ and $\Diamond acc(b', id, c, P')$ for some b and b'. Let D and D' be $Dep(id,P)$ and $Dep(id,P')$, respectively. We prove that $D = D'$. Assuming that b equals b' we conclude with Invariant 11. Suppose now that $b \neq b'$ and (without loss of generality) that $b < b'$. Consider the moment when a replica p sends $\mathtt{Ack}(b', id, D', P')$. From Invariant 6 at this moment $\mathsf{cbal} = b' > b$ at p. Hence, from Invariant 14 we have $D = D'$, as required. □

A.1 Validity

Proof of Validity. Consider a replica that executes a command c. By Invariant 4, $\mathtt{Propagate}(c)$ is a sent message, hence, some client submitted c. □

A.2 Integrity

Proof of Integrity. At a replica, each executed command is tracked with the set Exec: a command can be executed only if its identifier is not included in the set, and once it is executed, the set is updated to include the identifier. Furthermore, during the protocol execution no elements are removed from Exec and only one identifier can be associated with a single command. Hence, at each replica line 34 is executed at most once per command. □

A.3 Ordering

To prove Ordering we first introduce the following notations: for two commands c and d we write (i) $c \lhd_p d$ if $c \bowtie d$ and some replica p executes c before d, and (ii) $c \blacktriangleleft d$ if c was executed by some replica before any process submitted d. Using these notations the relation $\prec$ from §2 can be expressed as $\prec = (\bigcup_p \lhd_p) \cup \blacktriangleleft$.

Proposition 3. If for two conflicting commands c and d we have $c \lhd_p d$ for some replica p then at the moment when p executes c, we have $\texttt{id}(d) \notin \texttt{dep}[\texttt{id}(c)]$.

Proof. Consider the moment when p executes c. The Integrity property and $c \lhd_p d$ imply that p has not executed d at this moment, and thus $\texttt{id}(d) \notin$ Exec at p. From the precondition at line 33, $\texttt{dep}[\texttt{id}(c)] \subseteq$ Exec, so that $\texttt{id}(d) \notin \texttt{dep}[\texttt{id}(c)]$. □

Proposition 4. The relation $\bigcup_p \lhd_p$ is asymmetric.

Proof. Assume the contrary: $c \lhd_p d$ and $d \lhd_q c$ for two conflicting commands c and d. From Proposition 3, when p executes c it has $\texttt{id}(d) \notin \texttt{dep}[\texttt{id}(c)]$, and when q executes d it has $\texttt{id}(c) \notin \texttt{dep}[\texttt{id}(d)]$. But this contradicts Invariant 2. □

Proposition 5. Assume $c_1 \prec c_2$. Whenever a replica p executes c_2, some replica has already executed c_1.

Proof. Assume $c_1 \prec c_2$ and that replica p executes c_2. Then we have either $c_1 \blacktriangleleft c_2$ or $c_1 \lhd_q c_2$ for some replica q. Consider first the case when $c_1 \blacktriangleleft c_2$. Then c_1 is executed at some replica before c_2 is submitted and, hence, before c_2 is executed at p, as required. Consider now the case when $c_1 \lhd_q c_2$ for some replica q. We must have either $c_1 \lhd_p c_2$ or $c_2 \lhd_p c_1$. The latter case would contradict Proposition 4, so that $c_1 \lhd_p c_2$, as required. □

Proposition 6. Assume $c_1 \prec \cdots \prec c_n$ for $n \geq 2$. Whenever a replica p executes c_n, some replica has already executed c_1.

Proof. Follows from Proposition 5 by induction on n. □

Proof of Ordering. By contradiction, assume that $c_1 \prec \cdots \prec c_n = c_1$ for $n \geq 2$. Consider the first time when some replica executes c_1. By Proposition 6, at this time c_1 has already executed at some replica, which yields a contradiction. □

A.4 Liveness

For simplicity, in this section we assume that the set of slow quorums consists of all majorities, i.e., $\forall b.\, \mathcal{SQ}(b) = \{Q \in \mathcal{P}(\mathcal{R}) \mid |Q| \geq \lceil N/2 \rceil\}$. Since we assume that the system is eventually synchronous (§2), it is possible to implement an *eventually perfect failure detector*, denoted $\Diamond P$ [8]. At each replica p, the failure detector $\Diamond P$ outputs the set of replicas that p believes are correct. $\Diamond P$ guarantees that eventually all crashed replicas are detected by p and no correct replica remains indefinitely suspected.

```
77 Alive ← R ⊆ R
78 trusted ← leader(0) ∈ R
79 when ◇P ≠ Alive
80     Alive ← ◇P
81     trusted ← select(Alive)
82 when leader(bal) ≠ trusted
83     if p = trusted then recover()
84     else send Recover(bal) to trusted
85 when received Recover(b)
86     if p = trusted then recover()
```

Figure 13: Recovery policy at a replica p.

In Figure 13 we present an algorithm according to which a replica p invokes the recovery mechanism. $\Diamond P$ constantly outputs a set of correct replicas into a variable Alive (line 80), which is then used to select the trusted replica (line 81), i.e., the replica that (according to p) should be the leader. To this end, p passes Alive to a function `select` that deterministically picks one replica among the set of replicas. A possible implementation of `select` returns the replica with the lowest identifier. A replica constantly checks if its leader is the trusted one (line 82). If this is not the case p checks whether it trusts itself—in which case p initiates recovery (line 83)—or some other replica—which causes p to send a `Recover` message to trusted (line 84). With this message p asks trusted to impose its leadership: upon the receipt of `Recover`(b), a replica verifies whether it should be a leader and then starts the recovery (line 86). The `Recover` message carries the value of bal, which helps trusted to choose at line 41 a ballot high enough to convince p to follow it: when p receives `Recover`(b) it selects a ballot b' owned by p such that $b' > b$. Additionally, if possible, trusted tries to choose b' such that all replicas of at least one fast quorum associated with b' belong to Alive, i.e., $\exists Q \in \mathcal{FQ}(b').\, Q \subseteq$ Alive.

We now describe the measures to be taken on the client-side of the protocol. First of all, a submitted command may be lost during the recovery if it has not been accepted before. For example, suppose that at some ballot b a replica p receives a command c submitted by a correct client. Then p starts a new ballot b' and becomes its leader. If c is not accepted at b before the recovery is started, p might not include c in the new state (conditions at lines 53 and 58 do not hold for c). Since p already received c, it will never make a new proposal for c at b' or any subsequent ballots. To avoid such scenarios, we allow clients to resubmit their requests. Note that this does not affect the proof of Integrity. In the example above, the leader receives c at b' or any higher ballot and eventually makes its proposal.

Finally, as discussed in §4.3 an additional mechanism is needed to deal with client failures. A client may crash at any time and thus may fail to send a command to a subset of the correct replicas. If for each fast and slow quorum only some replicas receive request c then further progress is impossible not only for c but also for any command c' that depends on c at the leader. To circumvent such situations, a replica tracks the command identifiers that belong to the dependency sets received in leader's `FastAck` messages. If the replica detects that the payload of one of such commands is missing, it asks its leader to retransmit it. It is enough to contact the leader because no command is committed if it is not received by the leader. It is also possible that some followers receive a command c while the leader does not. These followers will take a dependency on c for any conflicting command, disabling a fast path for each of them. To deal with this, followers re-propose c if it is not committed after some timeout.

To prove that the above mechanisms ensure Liveness we rely on the following propositions.

Proposition 7. Either no correct replica ever receives a `NewLeader` message, or there exist a correct replica p and a ballot b owned by p such that

1. each correct replica eventually trusts only p;
2. b is the maximal ballot received by any correct replica in `NewLeader` messages.

Proof. If no correct replica ever receives a `NewLeader` message then the proposition trivially holds. Assume that at least one correct replica receives a `NewLeader` message. From the properties of $\diamond P$, there exist a replica p and a time t_0 such that after this time failures stop occurring and at each correct replica $\mathsf{trusted} = p$ and Alive is the set of all correct replicas.

We first prove by contradiction that p sends a finite number of `NewLeader` messages. Assume the converse. After t_0, p is the only correct replica that can start the recovery. Therefore, there is only a finite number of the `NewLeader` messages sent by the correct replicas other than p, because all such messages are sent before t_0. Moreover, the number of messages sent by a faulty replica is finite by definition. Thus, each correct replica joins only a finite number of ballots that are not owned by p. Since p increases its ballot with each issued `NewLeader` message and because the number of such messages is infinite, there is a time $t_1 \geq t_0$ after which each correct replica always follows p (i.e., $\mathsf{leader}(p.\mathsf{bal}) = p$). Therefore, at t_1, there are no replica at which the condition at line 82 is satisfied. Hence, no replica sends a `Recover` message after t_1 and thus the total number of such messages is finite. Let $t_2 \geq t_1$ be the time after which p no longer receives `Recover` messages. After t_2, p never executes handler at line 85. Therefore, after t_2, p never executes lines 83 or 86, and hence, it stops sending the `NewLeader` messages, which contradicts our assumption.

Now, that we have proved that there is only a finite number of `NewLeader` messages sent by p, let b be the maximal ballot for which p sends a `NewLeader` message. We prove by contradiction that for each $\texttt{NewLeader}(b')$ message received by a correct replica, $b' \leq b$. Assume the contrary. Let q be a correct replica receiving at time t' the $\texttt{NewLeader}(b')$ message from a replica $p' \neq p$ with $b' > b$. After t', q has $\mathsf{bal} \geq b'$. Since p never chooses a ballot greater than $b < b'$, after t' we also have $\mathsf{leader}(q.\mathsf{bal}) \neq p$. Eventually q trusts p and thus at some point it forces p to recover (line 82) with the ballot at least as high as b' (lines 83 and 84). As a result, p sends the $\texttt{NewLeader}(b'')$ message with $b'' > b' > b$, contradicting the fact that b is the maximal ballot chosen by p during the whole execution. □

Proposition 8. There exist a ballot b and a point in time after which every correct replica has $\mathsf{status} = \text{NORMAL}$ and $\mathsf{bal} = \mathsf{cbal} = b$.

Proof. If during the whole execution no correct replica ever receives a `NewLeader` message then all correct replicas always follow the same correct replica at the same ballot, as required. If at least one correct replica receives a `NewLeader` message then according to Proposition 7, there exists a correct replica p eventually trusted by all correct replicas and a ballot b such that b is the maximal ballot received by a correct replica in `NewLeader` messages. Eventually a correct replica q receives the $\texttt{NewLeader}(b)$ message and joins b. After that, q never changes its ballot, as it ignores any other `NewLeader` messages (line 44). All correct replicas eventually answer to p with a `NewLeaderAck` message. Thus, eventually p sends `Sync` to all correct replicas, as required. □

Proposition 9. If a replica p commits c with a dependency set D then any correct replica eventually commits c with D.

Proof. Suppose that a replica p commits c with D at a ballot b. From Proposition 8 we know that there exists a ballot b' after which the system stabilizes with the one correct leader.

We first prove by contradiction that $b \leq b'$. Suppose that $b > b'$. Since c is committed at b, there is at least one correct replica q that sends an `Ack` message for c at b. To this end, q has to join b (i.e., at some point we have $q.\mathsf{cbal} = b$), however, $b' < b$ is the maximal ballot joined by q (Proposition 8), hence the contradiction.

We thus have $b \leq b'$. Suppose that $b = b'$. This means that there is a quorum of correct replicas sending `Ack` messages that lead to acceptance of c. Eventually each correct replica receives all these messages and commits c with D.

Suppose now that $b < b'$. Eventually each correct replica joins b' (i.e., $\mathsf{cbal} = b'$). In order to join b', each correct replica receives the $\texttt{Sync}(b', _, _, _)$ message, sent by $\mathsf{leader}(b')$. Moreover, according to Invariant 14 this message carries a state that contains c with the dependencies D. □

Proof of Liveness. We first prove that if a command c is executed by a replica p then c is eventually executed by all correct replicas. Before executing c, p commits c together with c's transitive dependencies D. According to Proposition 9, the correct replicas eventually commit c as well as any command in D, which enables the condition at line 33 for c.

We now prove that if a command c is submitted by a non-faulty client then c is eventually executed by all correct replicas. Let b be the last stable ballot (Proposition 8). Note that the client submitting c is correct and that it resubmits c if it does not receive a response after some time. Hence, eventually c is accepted at some ballot. Let D be the set of all transitive dependencies of c. Similarly to c, those commands in D that are submitted by non-faulty clients are eventually accepted. Moreover, from Invariant 13 and Invariant 14, when c is accepted and the leader of b has $\mathsf{cbal} = b$, it also has payload of any command in D. Therefore, by contacting the leader and thus obtaining the missing payloads according to the mechanics described at the beginning of the section, the replicas eventually accept the commands in D submitted by the faulty clients. Regardless of the ballots at which the different commands in D are accepted, from Invariant 14, each correct replica eventually receives and commits c together with any command in D. We conclude with Invariant 3 according to which, once committed, c is never blocked in the execution. □

A.5 Nontriviality

The proof of Nontriviality relies on the following proposition:

Proposition 10. If $\Diamond acc(b, _, _, P)$ then for all $id \in Ids(P)$ all correct replicas eventually commit id with $Dep(id, P)$.

Proof. From Proposition 8 we know that there exist a ballot $b' \geq b$ and a time t after which recovery stops occurring and at least a majority of replicas are correct. If $b' = b$ then by the structure of the protocol all correct replicas eventually commit $id \in Ids(P)$ with $Dep(id, P)$—this is because communication between replicas cannot be disturbed by recovery or failures and because replicas communicate through reliable channels.

Assume that $b' > b$. Eventually each correct replica receives a $\texttt{Sync}(b', _, _, _)$ message carrying id with $Dep(id, P)$ and $\mathsf{phase}[id] = \text{COMMIT}$ or $\mathsf{phase}[id] = \text{ACCEPT}$ (Invariant 14). If $\mathsf{phase}[id] = \text{COMMIT}$, a correct replica commits id immediately after receiving `Sync`, as required. If $\mathsf{phase}[id] = \text{ACCEPT}$, correct replicas of some slow quorum eventually broadcast a $\texttt{SlowAck}(b', id)$ message (line 76). Eventually replicas at b' receive `SlowAcks` from each replica of this quorum and commit id with $Dep(id, P)$, as required. □

Proof of Nontriviality. A client accepts the result of execution of a command c only if there is a ballot b at which $\mathsf{leader}(b)$ optimistically executes c and replicas of some quorum of b agree on c's dependency paths P. Thus, if the client gets the result of c then we have $\Diamond acc(b, _, c, P)$. Note that $\mathsf{leader}(b)$ executes client requests in the order inferred form P, i.e., for each $id \in Ids(P)$, $\mathsf{cmd}[id]$ is executed with respect to dependency set $Dep(id, P)$. Moreover, according to Proposition 10, each correct replica eventually commits $id \in Ids(P)$ with $Dep(id, P)$, and hence, it eventually executes c following the same order. Thus, the result of execution of c at any correct replica equals to the result of optimistic execution at $\mathsf{leader}(b)$, as required. □

A.6 Latency

Asynchrony and failures may arbitrarily delay the execution of a command. For that reason, we measure latency only after the system stabilizes. We denote by δ the upper bound on the message delay after stabilization (see §2). As usual, we ignore the cost of local computation (in our setting, it is orders of magnitude lower than latency). The latency of a protocol is the maximum amount of time a client must wait before delivering a response once the system is stable.

Table 1 summarizes the latency of various state-machine replication protocols. We consider the following classes of runs commonly found in practice: *(sequential)* commands are never concurrent, i.e., upon the submission of a command, prior submitted commands are already committed everywhere; *(conflict-free)* concurrent commands are all commuting; and *(contention-free)* all

	sequential	*conflict-free*	*contention-free*	*general*
Paxos [29]	4δ			
N^{2}Paxos [29]	3δ+1			
Mencius [37]	2δ+1	4δ+1		
FastPaxos+ [31]	2δ+1	3δ+1		
Generalized Paxos [30]	2δ+1			6δ+1
Egalitarian Paxos [38]	2δ+1			$O(n\delta)$
CURP+N^{2}Paxos [41]	2δ		3δ+1	
SwiftPaxos	2δ			3δ

Table 1: Latency of state-machine replication protocols. We denote by δ the upper bound on message delay when the system is stable. For non-colocated clients +1 denotes the additional message delay.

concurrent conflicting commands are received in the same order at the replicas. The last class *(general)* is the worst-case latency when the system is stable. For protocols that have a fast path this corresponds to the slow path. Worst-case latency matters for hot items and when the fast path is no longer available (e.g., if a machine gets disconnected). Notice that these classes are ordered as follows: sequential $\subsetneq$ conflict-free $\subsetneq$ contention-free $\subsetneq$ general. In Table 1, the notation +1 refers to the additional message delay to reach non-colocated clients.

Below, we prove that the latency of SwiftPaxos matches the results in Table 1. Namely, we establish that SwiftPaxos executes state-machine commands in two message delays in the absence of contention, and three otherwise.

Lemma 1. Assume a contention-free run ρ of SwiftPaxos. At any time t in ρ, if p and q in some $\mathcal{FQ}(b)$ have both received c at t, then *(i)* $\mathsf{dep}[c]$ is the same at the two processes, and *(ii)* the evaluation of $\mathtt{pset}(id(c))$ on the two processes is identical.

Proof. We prove this result inductively. The property holds obviously at $t = 0$. Assume that it is true at $t-1$. At time t, if no command is newly received by either p or q, we are done. Otherwise, let c be such a command and wlog. consider that q receives c at time t. Process q executes line 11 for command c at that time. Consider a command d in $\mathsf{dep}_p[c]$ at t. As the run is contention-free, d is received before time t at q. Thus, it is added to $\mathsf{dep}_q[c]$ at t. This shows that $\mathsf{dep}_p[c] \subseteq \mathsf{dep}_q[c]$ at t. Using a symmetrical argument, we have $\mathsf{dep}_q[c] \subseteq \mathsf{dep}_p[c]$, as required. Now consider that q evaluates $\mathtt{pset}(id(c))$ to the set of paths P at t. By a short induction, because for any received command d, $\mathsf{dep}_p[d] = \mathsf{dep}_q[c]$ at t, the evaluation on p must also return P. □

Proposition 11. The latency of SwiftPaxos is 2δ when the run is contention-free, and 3δ otherwise.

Proof. Consider a run ρ of SwiftPaxos. Let t_0 be the point in time after which the system is stable in ρ. By Proposition 8, for some ballot b_0 at every correct replica, $\mathsf{bal} = \mathsf{cbal} = b_0$ and $\mathsf{status} = \text{NORMAL}$ hold forever after time t_0. Let p be the leader of ballot b_0. Pick a command c submitted after time t_0. According to SwiftPaxos (line 2), $\mathtt{client}(c)$ broadcasts a message $\mathtt{Propagate}(c)$ to all replicas. Any replica receives such a message after δ units When receiving it, p sends a message $\mathtt{Reply}(\mathsf{bal}, \mathtt{id}(c), P, r)$ to $\mathtt{client}(c)$, with r the result of the optimistic execution of c (line 17). There are two cases to consider:

- *(ρ is contention-free)* Assume a command d non-commuting with c was received at replica p before c. Because c is uncontended, it must be the case that d was also received before c at any other replica $q \in \mathcal{FQ}(b_0)$. By Lemma 1, p and q execute the exact same computation at lines 11 and 12. From which, we deduce that all the fast quorum replicas broadcast the same message $\mathtt{FastAck}(b_0, \mathtt{id}(c), D, P)$ to $\mathtt{client}(c)$ (lines 19 and 21). The precondition at line 3 on $\mathtt{client}(c)$ triggers after (at most) δ units of time, delivering the response of c. It follows that the message delay of c is 2δ in this case.
- *(Otherwise)* Leader p sends a message $\mathtt{FastAck}(b_0, \mathtt{id}(c), D, P)$ to all replicas. Choose some replica $q \neq p$ in $\mathcal{SQ}(b_0)$. Consider the point in time when q has already received c as well as the above message from p. This takes at most 2δ units of time. Because the system is stable, recovery never takes place after time t_0; hence $\mathtt{id}(c) \in \mathit{Preaccept}$. Furthermore, we know that $\mathsf{bal} = \mathsf{cbal} = b_0$ and $\mathsf{status} = \text{NORMAL}$ at q. Pick some command $d \in D$. From the fact that the system is stable, replica q has already received a message FastAck regarding command d from the leader. Hence, by a short induction, the precondition $D \subseteq \mathit{Accept} \cup \mathit{Commit}$ is eventually true at q. It follows that replica q eventually sends a message $\mathtt{SlowAck}(b_0, id(c))$ to $\mathtt{client}(c)$ (either at line 27, or at line 29). From what precedes, $\mathtt{client}(c)$ eventually receives a message $\mathtt{SlowAck}(b_0, id(c))$ from all such replicas. Consider the point in time at which this holds for some slow quorum of the ballot b_0. Eventually, the client also receives a Reply message from the leader. When these two happen, the precondition at line 4 is true. In this case, the response of command c is known at the client after 3δ units of time. □

B CURP for Geo-Replication

Although CURP is a primary-backup protocol, its authors also describe a variation for eventually synchronous systems [41, Appendix B.2]. This variation reuses a black-box leader-driven SMR protocol to enforce the agreement on the state-machine transitions at the replicas. In Figures 14 and 15 we use N^{2}Paxos for this purpose. For the sake of brevity, we omit recovery from the protocol's description. CURP+N^{2}Paxos slightly differs from the logic described in the CURP paper [41], as highlighted in blue in the figures.

CURP+N^{2}Paxos is a leader-based protocol. The non-leader replicas (aka followers) acknowledge each transition proposed by the leader. The protocol uses a set of witnesses, which are co-located with the followers and durably store each state-machine command (variable Unsynced in Figure 15). If a command is conflict-free (line 21), its client receives the result via the fast path after a single round-trip to the replicas (line 5). Otherwise, the command takes the slow path (line 7).

```
1  func submit(c):
2      send Record(c) to P

3  when received Reply(b, id, r) from leader(b)
     and RecordAck(b, id) from all q ∈ Q
4      pre: |Q| > 3N/4
5      response(id, r)

6  when received SyncReply(id, r)
7      response(id, r)
```

Figure 14: CURP+N^{2}Paxos: client.

CURP+N^{2}Paxos cuts one message delay in comparison to a combination of CURP with Paxos or Raft. This is because a replica sends an `AcceptAck` message to all the replicas and not simply to the leader (line 27). Each replica can now rapidly find out when a command is committed, and the replica closest to the client returns the response (lines 36-37). This change is helpful in geo-distributed systems, where reducing the latency is key for performance. Another difference with CURP [41, Appendix B.2] is the client's logic. To address the problem of zombie leaders, the authors of CURP propose to rely on a cached value of the term number (or ballot) at each client. Instead, we piggyback this value on the messages addressed to the client, which simplifies the protocol (see line 3 in Figure 14).

```
8  when received Record(c) from client q
9      pre: p = leader(b) ∧ cmd[id(c)] = ⊥
10     cmd[id(c)] ← c
11     next ← next + 1
12     log[next] ← id
13     pending_log ← pending_log · c
14     if ∀n < next. cmd[log[n]] ⋈ c ⟹
         log[n] ∈ Commit then
15         let r = opt_exec(pending_log, state)
16         send Reply(bal, id(c), r) to q
17     send Accept(bal, id(c), next) to all

18 when received Record(c) from client q
19     pre: p ≠ leader(b) ∧ cmd[id(c)] = ⊥
20     cmd[id(c)] ← c
21     if ∀id ∈ Unsynced. ¬(cmd[id] ⋈ c) then
22         Unsynced ← Unsynced ∪ {id(c)}
23         send RecordAck(bal, id(c)) to q

24 when received Accept(b, id, n) from q
25     pre: bal = b ∧ cmd[id] ≠ ⊥
26     log[n] ← id
27     send AcceptAck(b, id, n) to all

28 when received AcceptAck(b, id, n) from all q ∈ Q
29     pre: bal = b ∧ |Q| > N/2 ∧ cmd[id] ≠ ⊥ ∧
         log[n] = id ∧ ∀n' < n. log[n'] ∈ Commit
30     Commit ← Commit ∪ {id}
31     if p ≠ leader(b) then Unsynced ← Unsynced \ {id}

32 when there exists (n, id) such that id ∉ Exec ∧
   log[n] = id ∧ id ∈ Commit ∧ ∀n' < n. log[n'] ∈ Exec
33     Exec ← Exec ∪ {id}
34     let (r, state) = exec(cmd[id], state)
35     if p = leader(bal) then remove(pending_log, id)
36     if IsClosest(p, client(id)) then
37         send SyncReply(id, r)
```

Figure 15: CURP+N^{2}Paxos: replica p.

C Configuration of the Experiments from §5

	ap-south-1	ap-northeast-1	eu-west-3	us-west-1	af-south-1	ap-east-1	ap-southeast-2	ca-central-1	eu-west-1	sa-east-1	us-east-1	us-east-2
ap-northeast-1	128 ms											
eu-west-3	108 ms	217 ms										
us-west-1	231 ms	110 ms	143 ms									
af-south-1	164 ms	359 ms	152 ms	292 ms								
ap-east-1	91 ms	54 ms	201 ms	156 ms	254 ms							
ap-southeast-2	152 ms	111 ms	281 ms	140 ms	414 ms	130 ms						
ca-central-1	191 ms	146 ms	86 ms	81 ms	228 ms	192 ms	200 ms					
eu-west-1	125 ms	203 ms	20 ms	130 ms	163 ms	216 ms	258 ms	72 ms				
sa-east-1	301 ms	259 ms	197 ms	175 ms	344 ms	304 ms	314 ms	127 ms	180 ms			
us-east-1	190 ms	147 ms	84 ms	64 ms	232 ms	193 ms	201 ms	17 ms	71 ms	116 ms		
us-east-2	201 ms	136 ms	93 ms	55 ms	242 ms	180 ms	191 ms	27 ms	81 ms	125 ms	18 ms	
us-west-2	221 ms	99 ms	135 ms	24 ms	277 ms	146 ms	142 ms	61 ms	121 ms	176 ms	65 ms	54 ms

Figure 16: Latency table of AWS regions.

Replicas
ap-south-1
ap-northeast-1
eu-west-3
us-west-1
af-south-1

Clients
ap-east-1
ap-northeast-1
ap-southeast-2
eu-west-1
ca-central-1
sa-east-1
us-east-1
us-east-2
us-west-1
us-west-2

Figure 17: Regions used for replicas and clients.

The Bedrock of Byzantine Fault Tolerance: A Unified Platform for BFT Protocols Analysis, Implementation, and Experimentation

Mohammad Javad Amiri
Stony Brook University

Chenyuan Wu
University of Pennsylvania

Divyakant Agrawal
UC Santa Barbara

Amr El Abbadi
UC Santa Barbara

Boon Thau Loo
University of Pennsylvania

Mohammad Sadoghi
UC Davis

Abstract

Byzantine Fault-Tolerant (BFT) protocols cover a broad spectrum of design dimensions from infrastructure settings, such as the communication topology, to more technical features, such as commitment strategy and even fundamental social choice properties like order-fairness. The proliferation of different protocols has made it difficult to navigate the BFT landscape, let alone determine the protocol that best meets application needs. This paper presents *Bedrock*, a unified platform for BFT protocols analysis, implementation, and experimentation. Bedrock proposes a design space consisting of a set of dimensions and explores several design choices that capture the trade-offs between different design space dimensions. Within Bedrock, a wide range of BFT protocols can be implemented and uniformly evaluated under a unified deployment environment.

1 Introduction

Distributed systems rely on fault-tolerant protocols to provide robustness and high availability [43, 57, 63, 87, 102, 145, 197]. While cloud systems, e.g., Google's Spanner [87], Amazon's Dynamo [102], and Facebook's Tao [63], rely on crash fault-tolerant protocols, e.g., Paxos [164], to establish consensus, a Byzantine fault-tolerant (BFT) protocol is a key ingredient in distributed systems with non-trustworthy infrastructures, e.g., permissioned blockchains [1–3, 26, 29, 30, 32, 45, 67, 82, 127–129, 137, 162, 213, 220, 223], permissionless blockchains [64, 154, 156, 183, 253], distributed file systems [14, 75, 85], locking service [86], firewalls [55, 122, 123, 219, 230, 251], certificate authority systems [257], SCADA systems [41, 153, 205, 256], key-value datastores [53, 106, 126, 140, 219], and key management [187].

BFT protocols use the State Machine Replication (SMR) technique [163, 221] to ensure that non-faulty replicas execute client requests in the same order despite the concurrent failure of at most f Byzantine replicas. BFT SMR protocols are different along several dimensions, including the number of replicas, processing strategy (i.e., optimistic, pessimistic, or robust), supporting load balancing, etc. While dependencies and trade-offs among these dimensions lead to several design choices, there is currently no unifying tool that provides the foundations for studying and analyzing BFT protocols' design dimensions and their trade-offs. We envision that such a unifying foundation will provide an in-depth understanding of existing BFT protocols, highlight the trade-offs among dimensions, and will enable protocol designers to find the protocol that best fits their needs.

This paper presents *Bedrock*, a unified platform that enables us to analyze, implement, and experimentally evaluate partially asynchronous SMR BFT protocols within the design space of possible variants. Bedrock presents a design space to characterize BFT protocols based on different dimensions that capture the environmental settings, protocol structure, QoS features, and performance optimizations. Each protocol is a plausible point in the design space. Within the design space, Bedrock defines a set of design choices demonstrating trade-offs between different dimensions. For example, the communication complexity can be reduced by increasing the number of commitment phases or the number of phases can be reduced by adding more replicas. Each design choice expresses a *one-to-one function* to map plausible input points (i.e., a BFT protocol) to plausible output points (i.e., another BFT protocol) in the design space.

The Bedrock platform has three main practical uses:

- **BFT protocols analysis.** Bedrock can be used to analyze and navigate the evergrowing BFT landscape to principally compare and differentiate among BFT protocols. The Bedrock design space and its design choices organize protocols in an ordered fashion and provide new insights into the properties of existing BFT protocols.
- **BFT protocols implementation.** Within Bedrock, a wide range of BFT protocols, e.g., PBFT [73], SBFT [131], HotStuff [252], Kauri [202], Themis [149], Tendermint [66], Prime [24], PoE [135], CheapBFT [146], Q/U [5], FaB [190], and Zyzzyva [157], are implemented. The Bedrock implementation supports different stages of protocols, e.g., ordering, execution, view-change, and checkpointing. A domain-specific language (DSL) is

provided to rapidly prototype BFT protocols by specifying the protocol config, including the chosen value for each dimension in the design space, the list of roles, phases, states, and exchange messages of the protocol. Bedrock also includes a plugin manager to first, implement protocol-specific behaviors that can not be specified by the protocol config and second, enable users to add their own methods, dimensions, or values to support more protocols or to modify existing dimensions, e.g., add a new signature algorithm.

- **BFT protocols experimentation.** In addition to rapid prototyping, the unified deployment environment of Bedrock enables users to experimentally evaluate and compare different BFT protocols proposed in diverse settings and contexts under one unified platform. To our best knowledge, our paper presents the largest (and most varied) number of BFT protocols compared and experimented with within a single unified platform.

The paper makes the following contributions.

- A design space for BFT protocols, a set of design choices and possible design trade-offs are presented to help users analyze BFT protocols and understand how different protocols are related to each other.
- We present *Bedrock*, a platform that aims to unify BFT protocols. Bedrock derives valid protocols by combining different design choices in the design space.
- A wide range of BFT protocols can be implemented in Bedrock. The DSL specifications result in orders of magnitude reduction in code size compared to equivalent open-source implementations, greatly improving code readability and the ability to rapidly prototype protocols.
- The unified experimentation environment of Bedrock provides for the first time new opportunities to evaluate and compare different existing BFT protocols fairly and efficiently (e.g., identical programming language, used libraries, cryptographic tools, etc.).

2 Bedrock Overview

System model. A BFT protocol runs on a network consisting of a set of nodes that may exhibit arbitrary, potentially malicious, behavior. BFT protocols use the State Machine Replication (SMR) algorithm [163, 221] where the system provides a replicated service whose state is mirrored across different deterministic replicas. At a high level, the goal of a BFT SMR protocol is to assign each client request an order in the global service history and execute it in that order [226]. In a BFT SMR protocol, all non-faulty replicas execute the same requests in the same order (*safety*) and all correct requests are eventually executed (*liveness*). In an asynchronous system, where replicas can fail, no consensus solutions guarantee both safety and liveness (FLP result) [117]. As a result, asynchronous consensus protocols rely on techniques such as randomization [48, 70, 121, 214], failure detectors [80, 185], hybridization/wormholes [88, 204] and partial synchrony [108, 111] to circumvent the FLP impossibility.

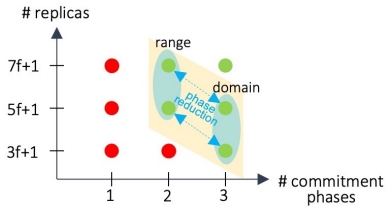


Figure 1: A simplified design space with two dimensions: number of replicas and number of commitment phases. Green dots (●) specify valid points (i.e., BFT protocols) while red dots (●) show invalid points (i.e., impossible protocols). A design choice, i.e., phase reduction, is a one-to-one transformation function that maps a protocol in its domain to another protocol in its range.

Bedrock assumes the partial synchrony model as it is used in most practical BFT protocols [73, 131, 157, 252]. In the partial synchrony model, there exists an unknown global stabilization time (GST), after which all messages between correct replicas are received within some known bound Δ. Bedrock further inherits the standard assumptions of existing BFT protocols. First, while there is no upper bound on the number of faulty clients, the maximum number of concurrent malicious replicas is assumed to be f. Second, replicas are connected via an unreliable network that might drop, corrupt, or delay messages. Third, the network uses point-to-point bi-directional communication channels to connect replicas. Fourth, the failure of replicas is independent of each other, where a single fault does not lead to the failure of multiple replicas. This can be achieved by either diversifying replica implementation (e.g., n-version programming) [40, 118] or placing replicas at different geographic locations (e.g., datacenters) [51, 112, 229, 241]. Finally, a strong adversary can coordinate malicious replicas and delay communication. However, the adversary cannot subvert cryptographic assumptions.

Usage model. Bedrock aims to help users analyze, implement, and evaluate BFT protocols within one unified platform and find the protocol that fits their needs. To achieve this goal, the Bedrock platform makes available the design dimensions of BFT protocols and different design choices, i.e., trade-offs between dimensions, to users to tune. Figure 1 illustrates an example highlighting the relation between design space, dimensions, design choices, and protocols in Bedrock. For the sake of simplicity, we present only two dimensions of the design space (among more than 10 dimensions, as described in Section 3), i.e., number of replicas and number of commitment phases. Each dimension, e.g., number of replicas, can take different values, e.g., $3f+1$, $5f+1$, or $7f+1$. A BFT protocol is then a point in this design space, e.g., ("3", "$3f+1$"). Note that each dimension not presented in this figure also takes a value, e.g., communication strategy is assumed to be pessimistic.

Moreover, only a subset of points is valid and represents BFT protocols. In Figure 1, green dots (●) specify valid points (i.e., BFT protocols) while red dots (●) show invalid points (i.e., impossible protocols). For example, there is no (pessimistic) BFT protocol with $3f+1$ nodes that commit re-

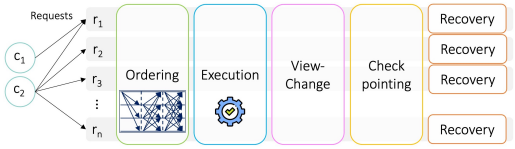


Figure 2: Different stages of replicas in a BFT protocol

quests in a single commitment phase. A design choice is then a one-to-one function that maps each BFT protocol in its domain to another protocol in its range. For example, phase reduction (through redundancy) maps each protocol with $3f+1$ nodes and 3 communication phases, e.g., PBFT [73], to a protocol with $5f+1$ nodes and 2 communication phases, e.g., FaB [190] (assuming both protocols are pessimistic with clique topology). The domain and range of each design choice are a subset of protocols in the design space.

BFT protocols structure. In a BFT protocol, as presented in Figure 2, clients communicate with a set of replicas that maintain a copy of the application state. A replica's lifecycle consists of ordering, execution, view-change, checkpointing, and recovery stages. The goal of the ordering stage is to establish agreement on a unique order among requests executing on the application state. In leader-based consensus protocols, a designated *leader* node proposes the order and, to ensure fault tolerance, needs to get agreement from a subset of the nodes, referred to as a *quorum*. In the execution stage, requests are applied to the replicated state machine. The view-change stage replaces the current leader. Checkpointing is used to garbage-collect data and enable trailing replicas to catch up, and finally, the recovery stage recovers replicas from faults by applying software rejuvenation techniques.

PBFT Protocol. To better illustrate the Bedrock design space, we give an overview of the PBFT protocol [73, 75] as a driving example. PBFT, as shown in Figure 3, is a leader-based protocol that operates in a succession of configurations called *views* [114, 115]. Each view is coordinated by a *stable* leader (primary) and the protocol *pessimistically* processes requests. In PBFT, the number of replicas, n, is assumed to be $3f+1$ and the ordering stage consists of pre-prepare, prepare, and commit phases. The pre-prepare phase assigns an order to the request, the prepare phase guarantees the uniqueness of the assigned order and the commit phase guarantees that the next leader can safely assign the order.

During a normal case execution of PBFT, clients send their signed request messages to the leader. In the pre-prepare phase, the leader assigns a sequence number to the request to determine the execution order of the request and multicasts a pre-prepare message to all *backups*. Upon receiving a valid pre-prepare message from the leader, each backup replica multicasts a prepare message to all replicas and waits for prepare messages from $2f$ different replicas (including the replica itself) that match the pre-prepare message. The goal of the prepare phase is to guarantee safety within the view, i.e., $2f$ replicas received matching pre-prepare messages from the

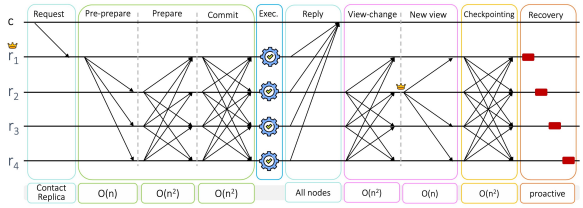


Figure 3: Different stages of PBFT protocol

leader replica and agree with the order of the request.

Each replica then multicasts a commit message to all replicas. Once a replica receives $2f+1$ valid commit messages from different replicas, including itself, that match the pre-prepare message, it commits the request. The goal of the commit phase is to ensure safety across views, i.e., the request has been replicated on a majority of non-faulty replicas and can be recovered after (leader) failures. The second and third phases of PBFT follow the *clique* topology, i.e., have $O(n^2)$ message complexity. If the replica has executed all requests with lower sequence numbers, it executes the request and sends a reply to the client. The client waits for $f+1$ matching results from different replicas.

In the view change stage, upon detecting the failure of the leader of view v using timeouts, backups exchange view-change messages including requests that have been received by the replicas. After receiving $2f+1$ view-change messages, the designated stable leader of view $v+1$ (the replica with `ID` = $v+1$ `mod` n) proposes a new view message, including the list of requests that should be processed in the new view.

In PBFT, replicas periodically generate checkpoint messages and send them to all replicas. If a replica receives $2f+1$ matching checkpoint messages, the checkpoint is stable. PBFT includes a *proactive* recovery mechanism that periodically rejuvenates replicas one by one. PBFT uses either signatures [73] or MACs [75] for authentication. Using MACs, replicas need to send view-change-ack messages to the leader after receiving view-change messages. Since new view messages are not signed, these view-change-ack messages enable replicas to verify the authenticity of new view messages.

3 Design Space

In Bedrock, each BFT protocol can be analyzed along several dimensions. These dimensions (and values associated with each dimension) collectively help to define the overall design space of BFT protocols supported by Bedrock. The dimensions are categorized into four main families: *protocol structure* and *environmental settings* that present the core dimensions of BFT protocols, two optional *QoS features* including order-fairness and load balancing that a BFT protocol might support, and a set of *performance optimizations*, such as request pipelining, parallel execution, and trusted hardware, for tuning BFT protocols. Due to space limitations, the performance optimizations are discussed in Appendix A. In the rest of this section, we describe these families of dimensions in greater detail. As we describe each dimension, we prefix label

them with "E" for environmental settings, "P" for protocol structure, etc. Hence, "E 1" refers to the first dimension in the environmental settings dimensions family.

This section is not meant to provide a fully exhaustive set of dimensions but rather to demonstrate the overall methodology used to define dimensions usable in Bedrock.

3.1 Protocol Structure

Our first family of dimensions concerns customization of the protocol structure by Bedrock, which will further define the class of protocols permitted.

P 1. Commitment strategy. Bedrock supports BFT protocols that process transactions in either an optimistic, pessimistic, or robust manner. *Optimistic* BFT protocols make optimistic assumptions on failures, synchrony, or data contention and might execute requests without necessarily establishing consensus. An optimistic BFT protocol might make a subset of the following assumptions:

- a_1. The leader is non-faulty, assigns a correct order to requests and sends it to all backups, e.g., Zyzzyva [157],
- a_2. The backups are non-faulty and *actively* and *honestly* participate in the protocol, e.g., CheapBFT [146],
- a_3. All non-leaf replicas in a tree topology are non-faulty, e.g., Kauri [202],
- a_4. The workload is conflict-free and concurrent requests update disjoint sets of data objects, e.g., Q/U [5],
- a_5. The clients are honest, e.g., Quorum [37], and
- a_6. The network is synchronous (in a time window), and messages are not lost or delayed, e.g., Tendermint [65].

Optimistic protocols are classified into *speculative* and *non-speculative* protocols. In non-speculative protocols, e.g., SBFT [131] and CheapBFT [146], replicas execute a transaction only if the optimistic assumption holds. Speculative protocols, e.g., Zyzzyva [157] and PoE [135], on the other hand, optimistically execute transactions. If the assumption is not fulfilled, replicas might have to rollback the executed transactions. Optimistic BFT protocols improve performance in fault-free situations. If the assumption does not hold, the replicas, e.g., SBFT [131], or clients, e.g., Zyzzyva [157], detect the failure and use the fallback protocol.

Pessimistic BFT protocols, on the other hand, do not make any optimistic assumptions about failures, synchrony, or data contention. In pessimistic BFT protocols, replicas communicate to agree on the order of requests. Finally, *robust* protocols, e.g., Prime [24], Aardvark [86], R-Aliph [37], Spinning [240] and RBFT [38], go one step further and consider scenarios where the system is under attack by a very strong adversary.

In summary, BFT protocols demonstrate different performances in failure-free, low-failure, and under-attack situations. Optimistic protocols deliver superior performance in failure-free situations. However, in the presence of failure, their performance is significantly reduced, especially when the system is under attack. On the other hand, pessimistic protocols provide high performance in failure-free situations and are able to handle low failures with acceptable overhead. However, they show poor performance when the system is under attack. Finally, robust protocols are designed for under-attack situations and demonstrate moderate performance in all three situations.

P 2. Number of commitment phases. The number of commitment (ordering) phases or *good-case latency* [12] of a BFT SMR protocol is the number of phases needed for all non-faulty replicas to commit when the leader is non-faulty, and the network is synchronous. We consider the number of commitment phases from the first time a replica (typically the leader) receives a request to the first time any participant (i.e., leader, backups, client) learns the commitment of the request, e.g., PBFT executes in 3 phases.

P 3. View-change. BFT protocols follow either the *stable leader* or the *rotating leader* mechanism to replace the current leader. The stable leader mechanism [73, 131, 157, 190] replaces the leader when the leader is suspected to be faulty by other replicas. In the rotating leader mechanism [20, 67, 76–78, 86, 125, 139, 155, 162, 240, 241, 252], the leader is replaced periodically, e.g., after a single attempt, insufficient performance, or an epoch (multiple requests).

Using the stable leader mechanism, the view-change stage becomes more complex. However, the routine is only executed when the leader is suspected to be faulty. On the other hand, the rotating leader mechanism requires ensuring view synchronization frequently (whenever the leader is rotated). Rotating the leader has several benefits, such as balancing load across replicas [46, 47, 240], improving resilience against slow replicas [86], and minimizing communication delays between clients and the leader [112, 189, 241].

P 4. Checkpointing. The checkpointing mechanism is used to first, garbage-collect data of completed consensus instances to save space and second, restore in-dark replicas (due to network unreliability or leader maliciousness) to ensure all non-faulty replicas are up-to-date [73, 103, 135]. Checkpointing is typically initiated after a fixed window in a decentralized manner without relying on a leader [73].

P 5. Recovery. When there are more than f failures, BFT protocols, apart from some exceptions [84, 178], completely fail and do not give any guarantees on their behavior [103]. BFT protocols perform recovery using *reactive* or *proactive* mechanisms (or a combination [230]). Reactive recovery mechanisms detect faulty replica behavior [138] and recover the replica by applying software rejuvenation techniques [95, 142] where the replica reboots, reestablishes its connection with other replicas and clients, and updates its state. On the other hand, proactive recovery mechanisms recover replicas in periodic time intervals. Proactive mechanisms do not require any fault detection techniques; however, they might unnecessarily recover non-faulty replicas [103]. During recovery, a replica is unavailable. A BFT protocol can rely on $3f+2k+1$ replicas to improve resilience and availability during recovery

where k is the maximum number of servers that rejuvenate concurrently [230]. To prevent attackers from disrupting the recovery process, each replica requires a trusted component, e.g., secure coprocessor [75], a synchronous wormhole [239] or a virtualization layer [105, 216], that remains operational even if the attacker controls the replica and a read-only memory that an attacker cannot manipulate. The memory content remains persistent (e.g., on disk) across machine reboots and includes all information needed for bootstrapping a correct replica after restart [103].

P 6. Types of clients. Bedrock supports three types of clients: requester, proposer, and repairer. *Requester* clients perform a basic functionality and communicate with replicas by sending requests and receiving replies. A requester client may need to verify the results by waiting for a number of matching replies, e.g., $f+1$ in PBFT [73], $2f+1$ in PoE [135] and PBFT (for read-only requests) [75], or $3f+1$ is Zyzzyva [157]. Using trusted components, e.g., Troxy [175], or threshold signatures, e.g., SBFT [131], the client does not even need to wait for and verify multiple results from replicas. Clients might also play the *proposer* role by proposing a sequence number (acting as the leader) for its request [5, 130, 186, 188]. *Repairer* clients, on the other hand, detect the failure of replicas, e.g., Zyzzyva [157], or even change the protocol configuration, e.g., Scrooge [222], Abstract [37], and Q/U [5].

3.2 Environmental Settings

Environmental settings, broadly speaking, encompass the deployment environment for a BFT protocol. These input parameters help scope the class of BFT protocols that can be supported to fit each deployment environment best.

E 1. Number of replicas. The first dimension concerns selecting BFT protocols based on the number of replicas (i.e., network and quorum size) used in a deployment. In the presence of f malicious failures, BFT protocols require at least $3f+1$ replicas to guarantee safety [59, 60, 91, 111, 170]. Using trusted hardware, the malicious behavior of replicas is restricted and safety can be guaranteed using $2f+1$ replicas [84, 90, 92, 216, 241, 241, 242]. Similarly, leveraging new hardware capabilities or using message-and-memory models the required number of replicas can be reduced to $2f+1$ [15–17]. On the other hand, the number of communication phases can be reduced by increasing the number of replicas to $5f+1$ [190] (its proven lower bound, $5f-1$ [12, 161]) or $7f+1$ [228]. A BFT protocol might also optimistically assume the existence of a quorum of $2f+1$ active non-faulty replicas (put f replicas as passive) to establish consensus [104, 146]. Using both trusted hardware and active/passive replication, the quorum size is further reduced to $f+1$ during failure-free situations [104, 105, 146].

E 2. Communication topology. Bedrock allows users to analyze BFT protocols based on communication topologies, including: (1) the star topology where communication is strictly from a designated replica, e.g., the leader, to all other replicas and vice-versa, resulting in linear message complexity [157, 252], (2) the clique topology where all (or a subset of) replicas communicate directly with each other resulting in quadratic message complexity [73], (3) the tree topology where the replicas are organized in a tree with the leader placed at the root, and at each phase, a replica communicates with either its child replicas or its parent replica, causing logarithmic message complexity [154, 155, 202], or (4) the chain topology where replicas construct a pipeline and each replica communicates with its neighbor replicas [37].

E 3. Authentication. Participants authenticate their messages to enable other replicas to verify a message's origin. Bedrock support both signatures, e.g., RSA [218], and authenticators [73], i.e., MACs [237]. Constant-sized threshold signatures [70, 224] have also been used to reduce the size of a set (quorum) of signatures. A protocol might even use different techniques (i.e., signatures, MACs) in different stages to authenticate messages sent by clients, sent by replicas in the ordering stage, and sent by replicas during view-change.

E 4. Responsiveness, synchronization, and timers. A BFT protocol is *responsive* if its normal case commit latency depends only on the actual network delay needed for replicas to process and exchange messages rather than any (usually much larger) predefined upper bound on message transmission delay [36, 208, 209, 225]. Responsiveness might be sacrificed in different ways. First, rotating the leader, the new leader might need to wait for a predefined time before initiating the next request to ensure that it receives the decided value from all non-faulty but slow replicas, e.g., Tendermint [162] and Casper [68]. Second, optimistically assuming all replicas are non-faulty, replicas (or clients) need to wait for a predefined upper bound to receive messages from all replicas, e.g., SBFT [131] and Zyzzyva [157].

BFT protocols need to guarantee that all non-faulty replicas will eventually be synchronized to the same view with a non-faulty leader enabling the leader to collect the decided values in previous views and making progress in the new view [62, 198, 199]. This is needed because a quorum of $2f+1$ replicas might include f Byzantine replicas and the remaining f "slow" non-faulty replicas might stay behind (i.e., in-dark) and not even advance views at all. *View synchronization* can be achieved by integrating the functionality with the core consensus protocol, e.g., PBFT [73], or assigning a distinct synchronizer component, e.g., Pacemaker in HotStuff [252], and hardware clocks [6].

Depending on the environment, network characteristics, and processing strategy, BFT protocols use different timers to ensure responsiveness and synchronization. Protocols can be configured with the following timers by Bedrock.

τ_1. Waiting for reply messages, e.g., Zyzzyva [157],
τ_2. Triggering (consecutive) view-change, e.g., PBFT [73],
τ_3. Detecting backup failures, e.g., SBFT [131],
τ_4. Quorum construction in an ordering phase, e.g., prevote

and precommit timeouts in Tendermint [65],
τ_5. View synchronization, e.g., Tendermint [65],
τ_6. Finishing a (preordering) round, e.g., Themis [149],
τ_7. Performance check (heartbeat), e.g., Aardvark [86], and
τ_8. Atomic recovery (watchdog timer) to periodically hand control to a recovery monitor [74], e.g., PBFT [75].

3.3 Quality of Service

There are some optional QoS features that Bedrock can support. We list two example dimensions.

Q 1. Order-fairness. Order-fairness deals with preventing adversarial manipulation of request ordering [42, 71, 149, 150, 159, 160, 255]. Order-fairness is defined as: "if a large number of replicas receives a request t_1 before another request t_2, then t_1 should be ordered before t_2" [150]. Order-fairness has been partially addressed using different techniques: (1) monitoring the leader to ensure it does not initiate two new requests from the same client before initiating an old request of another client, e.g., Aardvark [86], (2) adding a preordering phase, e.g., Prime [24], where replicas order the received requests locally and share their orderings with each other, (3) encrypting requests and revealing the contents only once their ordering is fixed [35, 69, 193, 232], (4) reputation-based systems [35, 99, 156, 173] to detect unfair censorship of specific client requests, and (5) providing opportunities for every replica to propose and commit its requests using fair election [9, 35, 125, 151, 173, 208, 248].

Q 2. Load balancing. The performance of fault-tolerant protocols is usually limited by the computing and bandwidth capacity of the leader [18, 21, 56, 81, 195, 196, 202, 245]. The leader coordinates the consensus protocol and multicasts/-collects messages to all other replicas in different protocol phases. Load balancing is defined as distributing the load among the replicas of the system to balance the number of messages any single replica has to process.

Load balancing can be partially achieved using the rotating leader mechanism, multi-layer, or multi-leader protocols. Using leader rotation, one replica (leader) is still highly loaded in each consensus instance. In multi-layer protocols [25, 137, 179, 201, 203], the load is distributed between the leaders of different clusters. However, the system still suffers from load imbalance between the leader and backups in each cluster. In multi-leader protocols [22, 33, 39, 136, 233, 243], all replicas can initiate consensus to partially order requests in parallel. However, slow replicas still affect the global ordering of requests. To resolve the bandwidth limit of small replicas decomposing consensus into data availability agreement and block retrieval is proposed [250].

4 Design Choices Landscape

Given a set of specified dimension values in Section 3, each protocol represents a point in the Bedrock design space. In this section, using PBFT as a driving example, as illustrated in Section 2, we demonstrate how different points in the design space lead to different trade-offs.

4.1 Expanding the Design Choices of PBFT

Using PBFT and our design dimensions as a baseline, we illustrate a series of design choices that expose different trade-offs BFT protocols need to make. Each design choice acts as a one-to-one function that maps each valid input point (i.e., a protocol) to another valid output point in the design space.

Design Choice 1. ***(Linearization).*** This function explores a trade-off between communication topology and communication phases. The function takes a quadratic phase, e.g., prepare or commit in PBFT, and splits it into two linear phases: one phase from all replicas to a collector (typically the leader) and one phase from the collector to all replicas, e.g., SBFT [131], HotStuff [252] and HotStuff-2 [184]. The output protocol requires (threshold) signatures for authentication. The collector collects a quorum of (typically $n-f$) *signatures* from replicas and broadcasts its message including the signatures, as a certificate of having received the required signatures. Using threshold signatures [69, 70, 215, 224] the collector message size becomes constant. Some BFT protocols [124, 143, 235] use linear communication during the ordering phase but follow the quadratic view-change routine of PBFT.

Design Choice 2. ***(Phase reduction through redundancy).*** This function explores a trade-off between the number of ordering phases and the number of replicas. The function transforms a protocol with $3f+1$ replicas and 3 ordering phases (i.e., one linear, two quadratic), e.g., PBFT, to a fast protocol with $5f+1$ replicas and 2 ordering phases (one linear, one quadratic), e.g., FaB [190]. In the second phase of the protocol, matching messages from a quorum of $4f+1$ replicas are required. Recently, $5f-1$ has been proven as the lower bound for two-step Byzantine consensus [12, 161]. The intuition behind the $5f-1$ lower bound is that in an authenticated model, when replicas detect leader equivocation and initiate view-change, they do not include view-change messages coming from the malicious leader, reducing the maximum number of faulty messages to $f-1$ [12, 161].

Design Choice 3. ***(Leader rotation).*** This function replaces the stable leader with the rotating leader mechanism, e.g., HotStuff [252], where the rotation happens after each request or epoch or due to low performance (as discussed in P **3**). This function eliminates the view-change stage and adds a quadratic phase or two linear phases (using the linearization function) to the ordering stage to ensure that the new leader is aware of the correct state of the system.

Design Choice 4. ***(Non-responsive leader rotation).*** This function replaces the stable leader mechanism with the rotating leader mechanism *without* adding a new ordering phase (in contrast to design choice **3**) while sacrificing responsiveness. The new leader assumes that the network is synchronous (after GST) and waits for a predefined known upper bound Δ (Timer τ_5) before initiating the next request. This is needed to ensure that the new leader is aware of the highest assigned order to the requests, e.g., Tendermint [66, 162] and Casper [68].

As an optimization, if the new leader is aware of the highest assigned order (the leader was part of the quorum), it can initiate the next request right after receiving $2f+1$ votes (without necessarily waiting for Δ [184]).

Design Choice 5. ***(Optimistic replica reduction).*** This function reduces the number of involved replicas in consensus from $3f+1$ to $2f+1$ while optimistically assuming all $2f+1$ replicas are non-faulty (assumption P **1**, a_2). In each phase of a BFT protocol, matching messages from a quorum of $2f+1$ replicas is needed. If a quorum of $2f+1$ non-faulty replicas is identified, they can order (and execute) requests without the participation of the remaining f replicas. Those f replicas remain passive and are needed if any of the active replicas become faulty [104, 146]. Note that n is still $3f+1$.

Design Choice 6. ***(Optimistic phase reduction).*** Given a *linear* BFT protocol, this function optimistically eliminates two linear phases (i.e., the equivalence of a single quadratic prepare pahse) assuming all replicas are non-faulty, e.g., SBFT [131]. The leader (collector) waits for signed messages from all $3f+1$ replicas in the second phase of ordering, combines signatures and sends a signed message to all replicas. Upon receiving the signed message from the leader, each replica ensures that all non-faulty replicas have received the request and agreed with the order. As a result, the third phase of communication can be omitted and replicas can directly commit the request. If the leader has not received $3f+1$ messages after a predefined time (timer τ_3), the protocol fallbacks to its slow path and runs the third phase of ordering.

Design Choice 7. ***(Speculative phase reduction).*** This function, similar to the previous one, optimistically eliminates two linear phases of the ordering stage assuming that non-faulty replicas construct the quorum of responses, e.g., PoE [135]. The main difference is that the leader waits for signed messages from only $2f+1$ replicas in the second phase of ordering and sends a signed message to all replicas. Upon receiving a message signed by $2f+1$ replicas from the leader, each replica speculatively executes the transaction, optimistically assuming that either (1) all $2f+1$ signatures are from non-faulty replicas or (2) at least $f+1$ non-faulty replicas received the signed message from the leader. If (1) does not hold, other replicas receive and execute transactions during the view-change. However, if (2) does not hold, the replica might have to rollback the executed transaction.

Design Choice 8. ***(Speculative execution).*** This function eliminates the prepare and commit phases while optimistically assuming that all replicas are non-faulty (assumptions P **1**, a_1 and a_2), e.g., Zyzzyva [157]. Replicas speculatively execute transactions upon receiving them from the leader. If the client does not receive $3f+1$ matching replies after a predefined time (timer τ_1) or it receives conflicting messages, the (repairer) client detects failure and communicates with replicas to receive $2f+1$ commit messages.

Design Choice 9. ***(Optimistic conflict-free).*** If requests of different clients are conflict-free (assumption P **1**, a_4), there is no need for a total order among all transactions. This function eliminates all ordering phases while optimistically assuming that requests are conflict-free and all replicas are non-faulty. The client becomes the *proposer* and sends its request to all (or a quorum of) replicas where replicas execute the requests without any communication [5, 97].

Design Choice 10. ***(Resilience).*** This function increases the number of replicas by $2f$ enabling the protocol to tolerate f more failure with the same safety guarantees. In particular, optimistic BFT protocols that assume all $3f+1$ replicas are non-faulty (quorum size is also $3f+1$) tolerate zero failures. By increasing the number of replicas to $5f+1$ replicas, such BFT protocols can provide the same safety guarantees with quorums of size $4f+1$ while tolerating f failures, e.g., Zyzzyva5 [157], Q/U [5]. Similarly, a protocol with the network size of $5f+1$ can tolerate f more faulty replicas by increasing the network size to $7f+1$ [228].

This function can also provide high availability during the (proactive) recovery stage by increasing the number of replicas by $2k$ (the quorum size by k) where k is the maximum number of servers that recover concurrently [230].

Design Choice 11. ***(Authentication).*** This function replaces MACs with signatures for a given stage. Signatures are typically more costly than MACs. However, in contrast to MACs, signatures provide non-repudiation and are not vulnerable to MAC-based attacks from malicious clients. If a protocol follows the star communication topology where a replica needs to include a quorum of signatures as a proof of its messages, e.g., HotStuff [252], k signatures can be replaced with a threshold signature. In such protocols MACs cannot be used since MACs do not provide non-repudiation.

Design Choice 12. ***(Robust).*** This function makes a pessimistic protocol robust by adding a preordering stage to the protocol, e.g., Prime [24]. In the preordering stage and, upon receiving a request, each replica locally orders and broadcasts the request to all other replicas. All replicas then acknowledge the receipt of the request in an all-to-all communication phase and add the request to their local request vector. Replicas periodically share their vectors with each other. The robust function provides (partial) fairness as well. Robustness has also been addressed in other ways, e.g., using the leader rotation and a blacklisting mechanism in Spinning [240] or isolating the incoming traffic of different replicas, and checking the performance of the leader in Aardvark [86].

Design Choice 13. ***(Fair).*** This function transforms an unfair protocol, e.g., PBFT, into a fair protocol by adding a preordering phase to the protocol. In the preordering phase, clients send requests to all replicas, and once a round ends (timer τ_6), each replica sends a batch of requests in the received order to the leader. The leader then initiates consensus on the requests following the order of requests in the received batches.

Table 1: Comparing selected BFT protocols based on different dimensions of Bedrock design space

Protocol	E1. Nodes	E2. Topo.	E3. Auth.	E4. Timers	P1. Strategy	P2. Phases	P3. V-change	P5. Rec.	P6. Client	Q1. Fair.	Q2. Load.	Design Choices
PBFT [73]	$3f+1$	clique	MAC ‖ Sign	τ_1, τ_2, τ_8	pessimistic	3	stable	pro.	Req.	□	□	(11)
Zyzzyva [157]	$3f+1$	star	MAC ‖ Sign	τ_1, τ_2	optimistic (spec): a_1, a_2	1 (3)	stable	-	Rep.	□	□	8, (11)
Zyzzyva5 [157]	$5f+1$	star	MAC ‖ Sign	τ_1, τ_2	optimistic (spec): a_1	1 (3)	stable	-	Rep.	□	□	8, 10, (11)
PoE [135]	$3f+1$	star	MAC ‖ T-Sign	τ_1, τ_2	optimistic (spec): a_2	3	stable	-	Req.	□	□	1, 7, 11
SBFT [131]	$3f+1$	star	T-Sign	τ_1, τ_2, τ_3	optimistic: a_2	3 (5)	stable	-	Req.	□	□	1, 6, 11
HotStuff [252]	$3f+1$	star	T-Sign	τ_1, τ_2	pessimistic	7	rotating	-	Req.	□	□	1, 3, 11
Tendermint [66]	$3f+1$	clique	Sign	$\tau_1, \tau_2, \tau_5, \tau_6$	optimistic: a_6	3	rotating	-	Req.	□	□	4, 11
Themis [149]	$4f+1$	star	T-Sign	τ_1, τ_2, τ_6	pessimistic	$1+7$	rotating	-	Req.	■	□	1, 3, 13, 11
Kauri [202]	$3f+1$	tree	T-Sign	τ_1, τ_2	optimistic: a_3	$7h$	stable*	-	Req.	□	■	(3), 14, 11
CheapBFT [146]	$2f+1$	clique	MAC	τ_1, τ_2	optimistic: a_2	3	stable	-	Req.	□	□	5
FaB [190]	$5f+1$	clique	(Sign)	τ_1, τ_2	pessimistic	2	stable	-	Req.	□	□	2
Prime [24]	$3f+1$	clique	Sign	$\tau_1, \tau_2, \tau_6, \tau_7$	robust	6	stable	-	Req.	◨	□	11, 12
Q/U [5]	$5f+1$	star	MAC	τ_1, τ_2	optimistic: a_4, a_5	1 (3)	stable	-	Rep.	□	□	9, 10
FLB	$5f-1$	clique	Sign	τ_1, τ_2	pessimistic	2	stable	-	Req.	□	□	1, 2, 11
FTB	$5f-1$	tree	T-Sign	τ_1, τ_2	optimistic: a_3	$3h$	stable	-	Req.	□	■	1, 2, 14, 11

Hint: "T-Sign": threshold signatures, "Req": requester client, "Rep": repairer client, "Pro": proactive recovery. The number of phases in the slow path of protocols is shown in parentheses. While Kauri is implemented on top of HotStuff, it does not use rotating leaders. Prime provides partial fairness.

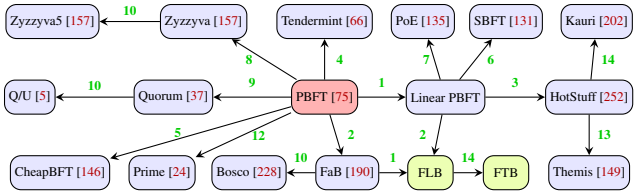


Figure 4: Derivation of protocols from PBFT using design choices

Depending on the order-fairness parameter γ ($0.5<\gamma\leq 1$) that defines the fraction of replicas receiving the requests in that specific order, at least $4f+1$ replicas ($n>\frac{4f}{2\gamma-1}$) replicas are needed to provide order-fairness [149, 150] [1].

Design Choice 14. ***(Tree-based LoadBalancer).*** This function explores a trade-off between the communication topology and load balancing where load balancing is supported by organizing replicas in a tree topology, with the leader at the root, e.g., Kauri [202]. This function splits a linear communication phase into h phases where h is the tree's height and each replica uniformly communicates with its child/parent replicas in the tree. The protocol optimistically assumes all non-leaf replicas are non-faulty (assumption P 1, a_3). Otherwise, the tree is reconfigured (i.e., view change).

4.2 Deriving and Evolving Protocols

Figure 4 demonstrates the derivation of a wide spectrum of BFT protocols from PBFT using design choices. Table 1 provides insights into how each BFT protocol maps into the Bedrock design space. The table also presents the design choices used by each BFT protocol. A detailed explanation of protocols is presented in Appendix B.

[1] With $3f+1$ replicas, as shown in [149], order-fairness requires a synchronized clock [255] or does not provide censorship resistance [159].

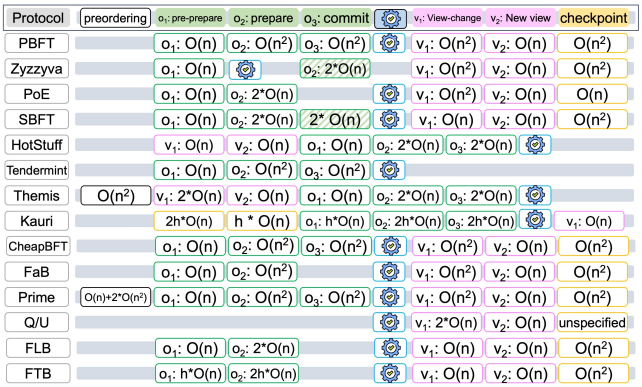


Figure 5: Overview of BFT protocols

Figure 5 focuses on different stages of replicas and demonstrates the communication complexity of each stage. The figure presents: (1) the preordering phases used in Themis and Prime, (2) the three ordering phases, e.g., pre-prepare, prepare or commit in PBFT (labeled by o_1, o_2, and o_3), (3) the execution stage, (4) the view-change stages consisting of view-change and new-view phases (labeled by v_1 and v_2), and (5) the checkpointing stage. As can be seen, some protocols do not have all three ordering phases, i.e., using different design choices, the number of ordering phases is reduced. The dashed boxes present the slow-path of protocols, e.g., the third ordering phase of SBFT is used only in its slow-path. Finally, the order of stages might be changed. For example, HotStuff runs view-change (leader rotation) for every single message and this phase takes place at the beginning of a consensus

instance to synchronize nodes within a view.

These case studies demonstrate the value of Bedrock in providing a unified platform for analyzing a range of existing BFT protocols. Note that the Bedrock platform enables users to implement new dimensions or design choices. For example, recently directed acyclic graph (DAG)-based BFT protocols [42, 100, 101, 119, 148, 231, 247] have emerged as an efficient way of establishing consensus. In DAG-based protocols and in each round, replicas independently send their own block of transactions as well as references to $2f+1$ received blocks (in the previous round) to other replicas in parallel. The references that blocks carry then become the backbone of a causally ordered DAG structure. DAG-based protocols provide higher throughput by separating transaction dissemination (by all replicas) from ordering. One can evolve PBFT to a DAG-based protocol in three steps (using three design choices); linearization, pipelining, and parallelization, with some minor modifications. Linearization makes PBFT linear (design choice **1**), pipelining enables a node to piggyback the messages of a new consensus instance on the second round messages of the previous instance (as it is used in Chained-HotStuff [4]), and parallelization enables multiple replicas to propose messages in parallel (as used in multi-leader protocols [136, 233], discussed in Appendix A, **3**).

Bedrock's utility can go beyond an analysis platform towards a *discovery* tool as well. Appendix C demonstrates two BFT protocols (FLB and FTB) uncovered using Bedrock.

5 Bedrock Implementation

Bedrock enables users, e.g., application developers, to implement and evaluate different BFT protocols. Bedrock is implemented in Java. The modular design of Bedrock enables a fair and efficient evaluation of BFT protocols using identical libraries, cryptographic functions, etc. The Bedrock platform consists of four main components: the core unit, the state manager, the plugin manager, and the coordination unit.

The core unit defines entities, e.g., clients and nodes, and maintains the application logic and application data. Client transactions are executed using the application logic resulting in updating the data. Entities track the execution of requests through various state variables, e.g., view and sequence number. Within the core unit, different workloads and benchmarks can be defined. Client requests can be initiated using a constant interval or a dynamic interval updated based on a moving average of response times. Different utility classes, such as `Timekeeper` to handle timers, and `BenchmarkManager` to measure and report results are also defined within the core unit.

The state manager enables the core unit to track the states and transitions of each entity according to the utilized BFT protocol, e.g., different stages of a replica or different phases of consensus. Bedrock defines a domain-specific language (DSL) to rapidly prototype BFT protocols. The DSL code written in the protocol config defines different dimensions and the chosen value for each dimension, the list of roles, phases, states, exchange messages, quorum conditions of the protocol,

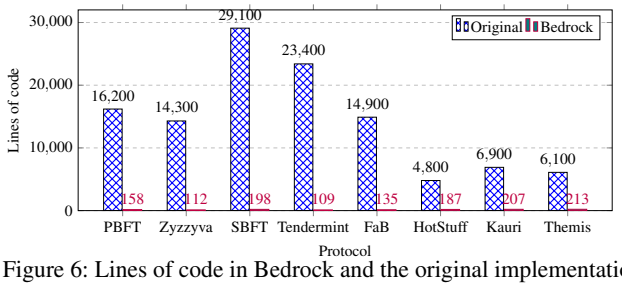


Figure 6: Lines of code in Bedrock and the original implementation

and also, the list of protocol-specific plugins required to run the protocol. The `EO-YAML` and `Apache Commons Lang` libraries are used for parsing, loading, and holding the protocol config data. Appendix E demonstrates the PBFT code using the DSL. The protocol config greatly reduces the effort needed to write a BFT protocol. Figure 6 compares the lines of code in the original open-source implementation of several known protocols and their implementation in Bedrock., e.g., the original Zyzzyva source code includes more than 14000 lines while its config in Bedrock is only 112 lines[2]. Overall, using Bedrock, the code size is reduced by orders of magnitude. Each protocol, in addition to the config file, uses a set of plugins defined in Bedrock, as explained in the next part. Chained-HotStuff, as a protocol that uses the most plugins (five), requires only 412 more lines of code to implement its five plugins, several of them are shared with multiple protocols.

The plugin manager serves two purposes. First, it enables the implementation of protocol-specific behaviors that cannot be handled by the protocol config defined in the state manager. For example, the speculative execution in Zyzzyva [157] or handling view-change without using a different process or states in Tendermint [162]. Second, it enables Bedrock users to define their own dimensions/values to support more protocols or to update existing dimensions without requiring changes to the platform code or rebuilding the platform binaries. For example, if a developer wants to use a new digest or signature algorithm for an existing or a new protocol, the algorithm can be implemented within a plugin.

Four types of plugins have been defined in the current version of Bedrock. *Role* plugins that define specific behavior for a certain role in a specific sequence number, view number, state, etc., e.g., message dissemination by the primary node in CheapBFT [146] where nodes are divided into active and passive nodes. *Message* plugins that define specific methods to process incoming or outgoing messages, e.g., perform digest validation. *Transition* plugins that specify an action to be performed during or after a state transition, e.g., how to process checkpoint messages. *Pipeline* plugins that enable manipulating the flow of messages, e.g., Chained-HotStuff [252] (as discussed in Appendix A, O **2**).

The coordination unit manages the run-time execution of Bedrock. The coordination unit consists of a coordinator and a set of executors. The coordinator manages the benchmark pro-

[2] We count only the lines of source code related to the core consensus protocol and not the applications or the utilized libraries.

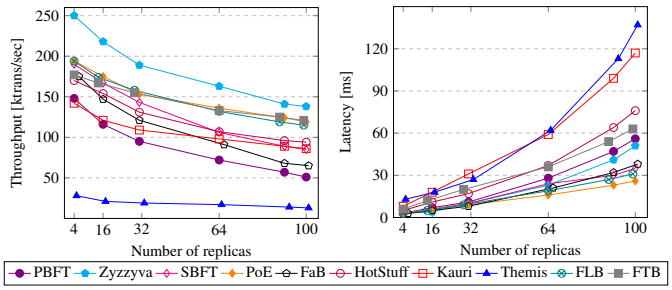


Figure 7: Performance with different number of replicas

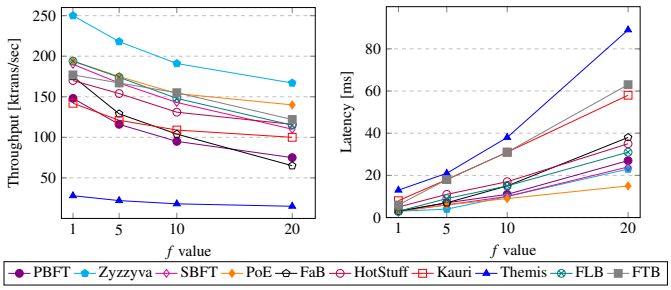


Figure 8: Performance with different f value

cess and setups all entities by initializing replicas and clients, sending config parameters to executors, enabling plugins to run additional initialization steps, starting and stopping execution threads, and reporting results. The executors, on the other hand, run the utilized BFT protocol.

The data (e.g., messages, requests, blocks) for the events and messages transmitted between nodes and clients is defined using the `Google Protocol Buffers` syntax and then compiled using the `protoc` tool.

6 Experimental Evaluation

Our evaluation studies the practical impact of the design dimensions and the exposed trade-offs presented as design choices on the performance of BFT protocols under one unified platform. We use typical experimental scenarios used for existing BFT protocols and permissioned blockchains, including (1) varying the number of replicas, (2) under a backup failure, (3) multiple request batch sizes, and (4) a geo-distributed setup (presented in Appendix D).

All protocols listed in Table 1 are implemented in Bedrock. Note that the original implementations of such BFT protocols utilize different (often old, inefficient) libraries, crypto algorithms, etc. Hence, it was unfair to experimentally compare such original implementations with their implementations in bedrock. Using the platform, we also experimented with many new protocols resulting from the combination of design choices. Due to space limitations, we present the performance evaluation of a subset of protocols. In particular, we evaluate PBFT, Zyzzyva, SBFT, FaB, PoE, (Chained-) HotStuff, Kauri, Themis, and two of the more interesting new variants (FLB and FTB). This set of protocols enables us to see the impact of design choices **1**, **2**, **3**, **6**, **7**, **8**, **10**, **11**, **13**, and **14** (discussed in Section 4). We also use the out-of-order processing technique for protocols with a stable leader and the request pipelining technique for protocols with a rotating leader. In our experiments, Kauri and FTB are deployed on trees of height 2 and the order-fairness parameter γ of Themis is considered to be 1 (i.e., $n = 4f + 1$). We use 4 as the base pipelining stretch for both Kauri and FTB and change it depending on the batch size and deployment setting (local vs. geo-distributed).

The experiments were conducted on the Amazon EC2 platform. Each VM is a c4.2xlarge instance with 8 vCPUs and 15GB RAM, Intel Xeon E5-2666 v3 processor clocked at 3.50 GHz. When reporting throughput, we use an increasing number of client requests until the end-to-end throughput is saturated and state the throughput and latency just below saturation. The results reflect end-to-end measurements from the clients. Clients execute in a closed loop. We use micro-benchmarks commonly used to evaluate BFT systems, e.g., BFT-SMART. The results are the average of five runs.

6.1 Fault Tolerance and Scalability

In the first set of experiments, we evaluate the performance of the protocols by increasing the number of replicas n (each runs on a separate VM) from 4 to 100 in a failure-free situation. For some protocols, the smallest network size might differ, e.g., FaB requires $5f + 1 = 6$ replicas. We use a batch size of 400 (we discuss this choice later) and a workload with client request/reply payload sizes of 128/128 byte. Figure 7 reports the results.

Zyzzyva shows the highest throughput among all protocols in small networks due to its optimistic ordering stage (design choice **8**). However, as n increases, its throughput significantly reduces as clients need to wait for reply from *all* replicas. Increasing the number of replicas also has a large impact on PBFT and FaB (65% and 63% reduction, respectively) due to their quadratic message complexity.

On the other hand, the throughput of Kauri and FTB is less affected (31% and 32% reduction, respectively) by increasing n because of their tree topology (design choice **14**) that reduced the bandwidth utilization of each replica. Similarly, PoE, SBFT and HotStuff incur less throughput reduction (39%, 55% and 45% respectively) compared to PBFT and FaB due to their linear message complexity (design choice **1**). In Bedrock, Chained-HotStuff has been implemented using the pipelining technique, resulting in lower average latency. In comparison to HotStuff, SBFT has slightly lower throughput in large networks (e.g., 8% lower when $n = 100$) because the leader waits for messages from all replicas. SBFT, on the other hand, shows higher throughput compared to HotStuff in smaller networks (e.g., 12% higher when $n = 4$) due to its fast ordering stage (design choice **6**). PoE demonstrates higher throughput compared to both SBFT and HotStuff, especially in larger networks (e.g., 39% higher than SBFT and 26% higher than HotStuff when $n = 100$). This is expected because, in PoE, the leader does not need to wait for messages from all replicas and optimistically combines signatures from $2f + 1$ replicas (design choice **7**). Compared to PBFT, while HotStuff shows better throughput (e.g., 48% higher when $n = 64$), the latency of PBFT is lower (e.g., 32% lower when

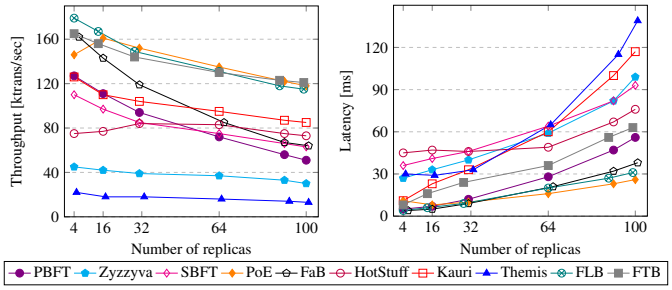


Figure 9: Performance with faulty backups

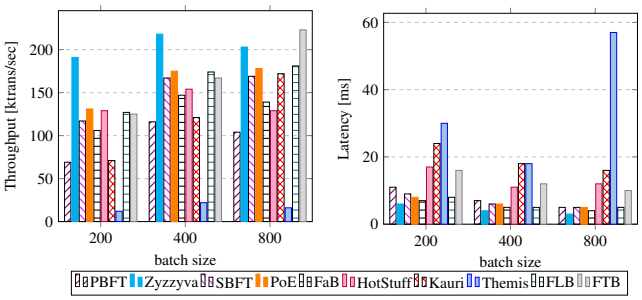


Figure 10: Impact of request batching

$n = 64$). One reason behind the high latency of HotStuff is its extra communication round (design choice **3**).

Supporting order-fairness (design choice **13**) leads to deficient performance of Themis compared to HotStuff (83% lower throughput when n=5). In Themis, replicas order transactions and send batches of transactions to the leader, and the leader needs to generate a fair order. As the number of replicas increases, Themis incurs higher latency (the latency increases from 9 to 137 ms as n increases to 101), mainly due to the overhead of generating the dependency graph and reaching a fair order by the leader. Using design choice **2** and reducing the number of communication phases results in 41% higher throughput and 46% lower latency of FTB compared to Kauri in a setting with 99 replicas (100 for Kauri).

Finally, using design choices **1** and **2**, FLB demonstrates better performance for large n (2.25x throughput and 0.55x latency compared to PBFT). This is because FLB reduces both message complexity and communication phases, and replicas do not need to wait for responses from all replicas.

Figure 7 depicts the results with different numbers of replicas. However, with the same number of replicas, different protocols tolerate different numbers of failures. For instance, PBFT requires $3f+1$ and when $n = 100$ tolerates 33 failures while FaB requires $5f+1$ and tolerates 19 failures with $n = 100$. To compare protocols based on the maximum number of tolerated failures, we represent the results of the first experiments in Figure 8. With $f = 20$, Themis incurs the highest latency because it requires 81 ($4f+1$) replicas and deals with the high cost of achieving order-fairness.

6.2 Performance with Faulty Backups

In this set of experiments, we force a backup replica to fail and repeat the first set of experiments. Figure 9 reports the results. Zyzzyva is mostly affected by failures (82% lower throughput) as clients need to collect responses from *all* replicas. A client waits for $\Delta = 5ms$ to receive reply from all replicas and then the protocol switches to its normal path.

We also run this experiment on Zyzzyva5 to validate design choice **10**, i.e., tolerating f faulty replicas by increasing the number of replicas. With a single faulty backup, Zyzzyva5 incurs only 8% lower throughput when $n = 6$.

Backup failure reduces the throughput of SBFT by 42%. In the fast path of SBFT, all replicas need to participate, and even when a single replica is faulty, the protocol falls back to its slow path, which requires two more phases. Interestingly, while the throughput of PoE is reduced by 26% in a small network (4 replicas), its throughput is not significantly affected in large networks. This is because the faulty replica (which participates in the quorum construction but does not send reply messages to the clients) has a higher chance of becoming a quorum member in small networks.

Faulty backups also affect the performance of HotStuff, especially in small networks. This is expected because HotStuff uses the rotating leader mechanism. When n is small, the faulty replica is the leader of more views during the experiments, resulting in reduced performance. HotStuff demonstrates its best performance when $n = 31$ (still, 36% lower throughput and 2.7x latency compared to the failure-free scenario). While Themis uses HotStuff as its ordering stage, a single faulty backup has less impact on its performance compared to HotStuff (25% reduction vs. 66% reduction in throughput). This is because Themis has a larger network size ($4f+1$ vs. $3f+1$) that reduces the impact of the faulty replica. In Kauri and FTB, we force a leaf replica to fail in order to avoid triggering a reconfiguration. As a result, the failure of a backup does not significantly affect their performance (e.g., 3% lower throughput with 31 replicas in Kauri). Finally, in small networks, FLB demonstrates the best performance as it incurs only 8% throughput reduction.

6.3 Impact of Request Batching

In the next set of experiments, we measure the impact of request batching. We consider three scenarios with batch sizes of 200, 400 and 800. The network includes 16 non-faulty replicas (17 replicas for Themis, 14 replicas for FLB and FTB). Figure 10 depicts the results. Increasing the batch size from 200 to 400 requests improves the performance of all protocols. This is because, with larger batch sizes, more transactions can be committed while the number of communication phases and exchanged messages is the same and the bandwidth and computing resources are not fully utilized yet. Different protocols behave differently when the batch size increases from 400 to 800. First, Kauri and FTB still process a higher number of transactions (42% and 34% higher throughput) as both protocols balance the load and utilize the bandwidth of all replicas. Second, SBFT and FaB demonstrate similar performance as before; a trade-off between smaller consensus quorums and a higher cost of signature verification and bandwidth utilization. Third, the performance of Themis decreases (24% lower

throughput and 3.16x latency) compared to a batch size of 400 due to two main reasons. First, the higher cost of signature verification and bandwidth utilization, and second, the higher complexity of generating fair order for a block of 800 transactions (CPU utilization).

6.4 Evaluation Summary

We summarize some of the evaluation results as follows. First, optimistic protocols that require all nodes to participate, e.g., Zyzzyva and SBFT, do not perform well in large networks, especially when nodes are far apart. In small networks also, a single faulty node significantly reduces the performance of optimistic protocols. Second, the performance of pessimistic protocols highly depends on the communication topology. While the performance of protocols with quadratic communication complexity, e.g., PBFT and FaB, is significantly reduced by increasing the network size, the performance of protocols with linear complexity, e.g., HotStuff, and especially logarithmic complexity, e.g., Kauri and FTB, is less affected. Interestingly in small networks, protocols that use the leader rotation mechanism show poor performance. This is because the chance of the faulty node becoming the leader is relatively high. Third, the load-balancing techniques, e.g., tree topology, enable a protocol to process larger batches. Finally, in a wide-area network, out-of-order processing of transactions significantly improves performance.

7 Related Work

SMR regulates the deterministic execution of requests on multiple replicas, such that every non-faulty replica executes every request in the same order [163,221]. Several approaches [164,207,221] generalize SMR to support crash failures. CFT protocols [19,61,79,81,107,141,141,144,165,166,168,169,171,177,181,200,206,207,210,238] utilize the design trade-offs between design dimensions, e.g., Fast Paxos [166] adds f replicas to reduce a communication phase.

Byzantine fault tolerance refers to nodes that behave arbitrarily after the seminal work by Lamport, et al. [170]. BFT protocols have been analyzed in several surveys and empirical studies [7,8,23,28,44,50,52,54,72,93,103,120,133,134,211,226,244,254]. We discuss some of the relevant studies.

Berger and Reiser [50] present a survey on BFT protocols used in blockchains where the focus is on scalability techniques. Similarly, a survey on BFT protocols consisting of classical protocols, e.g., PBFT, blockchain protocols, e.g., PoW, and hybrid protocols, e.g., OmniLedger [156], and their applications in permissionless blockchains, is conducted by Bano et al. [44]. Platania et al. [211] classify BFT protocols into client-side and server-side protocols depending on the client's role. The paper compares these two classes of protocols and analyzes their performance and correctness attacks. Three families of leader-based, leaderless, and robust BFT protocols with a focus on message and time complexities have been analyzed by Zhang et al. [254]. Finally, Distler [103] analyzes BFT protocols along several main dimensions: architecture, clients, agreement, execution, checkpoint, and recovery. The paper shares several dimensions with Bedrock.

A recent line of work [10–13] also study good-case latency of BFT protocols. Bedrock, in contrast to all these survey and analysis papers, provides a design space, systematically discusses design choices (trade-offs), and, more importantly, provides a tool to analyze BFT protocols experimentally.

BFTSim [226] is a simulation environment for BFT protocols that leverages a declarative networking system and compares a set of representative protocols using the simulator. Abstract [37] develops each protocol as a sequence of BFT instances, e.g., AZyzzyva, Aliph, and R-Aliph as three protocols where Each protocol itself is a composition of Abstract instances presented to handle different situations (e.g., fault-free, under attack). In contrast to such studies, Bedrock develops a design space for BFT protocols, enabling end-users to analyze, implement, and evaluate different protocols.

In addition to CFT and BFT protocols, consensus with multiple failure modes has also been studied for both synchronous [152,192,227,236], and partial synchronous [31,85,131,182,212,222] models. Finally, leaderless protocols [58,98,109,132,167,193,234] have been proposed to avoid the implications of relying on a leader.

8 Conclusion

Bedrock is a unified platform for BFT protocols analysis, implementation, and experimentation. Bedrock demonstrates how different BFT protocols relate to one another within a design space and along different design dimensions. Using a domain-specific language, the Bedrock facilitates rapid prototyping of BFT protocols. Finally, different BFT protocols proposed in diverse settings and contexts can be experimentally evaluated under one unified platform fairly and efficiently.

As future work, we plan to enable users to check the correctness of their written protocols by transforming the DSL code written in Bedrock to the specification language used by tools such as DistAlgo [180] or TLAPS [83,96]. Moreover, to ensure the independent failure of replicas, we plan to diversify replica implementation using n-version programming, where Bedrock provides different implementations of the same protocol config. We will further design a constraint checker to automatically find all plausible points (valid combinations of design choices) in the design space based on user queries. Incorporating automatic selection strategies in Bedrock based on the deployment environment and application requirements could be the next step. Machine learning techniques may be useful here in aiding the user in selecting the appropriate BFT protocol, or switch one protocol to another at runtime as system parameters are updated. Finally, we plan to extend the supported protocols to include synchronous and fully asynchronous protocols and expand the design space accordingly.

Acknowledgements

We thank the anonymous reviewers for their insightful feedback and suggestions. This work is funded by NSF grants CNS-2104882, CNS-2107147, and CNS-2245373.

References

[1] Chain. http://chain.com.

[2] Corda. https://github.com/corda/corda.

[3] Hyperledger iroha. https://github.com/hyperledger/iroha.

[4] libhotstuff: A general-purpose bft state machine replication library with modularity and simplicity. https://github.com/hot-stuff/libhotstuff, 2018.

[5] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. *Operating Systems Review (OSR)*, 39(5):59–74, 2005.

[6] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected o(1) rounds, expected $o(n^2)$ communication, and optimal resilience. In *Int. Conf. on Financial Cryptography and Data Security*, pages 320–334. Springer, 2019.

[7] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance. *arXiv preprint arXiv:1712.01367*, 2017.

[8] Ittai Abraham, Dahlia Malkhi, et al. The blockchain consensus layer and bft. *Bulletin of EATCS*, 3(123), 2017.

[9] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A blockchain protocol based on reconfigurable byzantine consensus. In *Int. Conf. on Principles of Distributed Systems (OPODIS)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[10] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *Symposium on Security and Privacy (SP)*, pages 106–118. IEEE, 2020.

[11] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Brief announcement: Byzantine agreement, broadcast and state machine replication with optimal good-case latency. In *Int. Symposium on Distributed Computing (DISC)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[12] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: a complete categorization. In *Symposium on Principles of Distributed Computing (PODC)*, pages 331–341. ACM, 2021.

[13] Ittai Abraham, Ling Ren, and Zhuolun Xiang. Good-case and bad-case latency of unauthenticated byzantine broadcast: A complete categorization. In *Int. Conf. on Principles of Distributed Systems (OPODIS)*, 2022.

[14] Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. *FARSITE*: Federated, available, and reliable storage for an incompletely trusted environment. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2002.

[15] Marcos K Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *Symposium on Principles of Distributed Computing (PODC)*, pages 51–60. ACM, 2018.

[16] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of rdma on agreement. In *Symposium on Principles of Distributed Computing (PODC)*, pages 409–418, 2019.

[17] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Dalia Papuc, Athanasios Xygkis, and Igor Zablotchi. Frugal byzantine computing. In *Int. Symposium on Distributed Computing*, 2021.

[18] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. Dissecting the performance of strongly-consistent replication protocols. In *SIGMOD Int. Conf. on Management of Data*, pages 1696–1710. ACM, 2019.

[19] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. Wpaxos: Wide area network flexible consensus. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):211–223, 2019.

[20] Amitanand S Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. Bar fault tolerance for cooperative services. In *Symposium on Operating systems principles (SOSP)*, pages 45–58. ACM, 2005.

[21] Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. Balanced byzantine reliable broadcast with near-optimal communication and improved computation. In *Symposium on Principles of Distributed Computing (PODC)*, pages 399–417. ACM, 2022.

[22] Salem Alqahtani and Murat Demirbas. Bigbft: A multileader byzantine fault tolerance protocol for high throughput. In *Int. Performance Computing and Communications Conf. (IPCCC)*, pages 1–10. IEEE, 2021.

[23] Salem Alqahtani and Murat Demirbas. Bottlenecks in blockchain consensus protocols. In *Int. Conf. on Omni-Layer Intelligent Systems (COINS)*, pages 1–8. IEEE, 2021.

[24] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *Transactions on Dependable and Secure Computing*, 8(4):564–577, 2011.

[25] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Steward: Scaling byzantine fault-tolerant replication to wide area networks. *IEEE Transactions on Dependable and Secure Computing*, 7(1):80–93, 2008.

[26] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. CAPER: a cross-application permissioned blockchain. *Proc. of the VLDB Endowment*, 12(11):1385–1398, 2019.

[27] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. ParBlockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 1337–1347. IEEE, 2019.

[28] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Modern large-scale data management systems after 40 years of consensus. In *Int. Conf. on Data Engineering (ICDE)*, pages 1794–1797. IEEE, 2020.

[29] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. SharPer: Sharding permissioned blockchains over network clusters. In *SIGMOD Int. Conf. on Management of Data*, pages 76–88. ACM, 2021.

[30] Mohammad Javad Amiri, Boon Thau Loo, Divyakant Agrawal, and Amr El Abbadi. Qanaat: A scalable multi-enterprise permissioned blockchain system with confidentiality guarantees. *Proc. of the VLDB Endowment*, 15(11):2839–2852, 2022.

[31] Mohammad Javad Amiri, Sujaya Maiyya, Divyakant Agrawal, and Amr El Abbadi. SeeMoRe: A fault-tolerant protocol for hybrid cloud environments. In *Int. Conf. on Data Engineering (ICDE)*, pages 1345–1356. IEEE, 2020.

[32] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, and Yacov Manevich. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *European Conf. on Computer Systems (EuroSys)*, pages 30:1–30:15. ACM, 2018.

[33] Balaji Arun, Sebastiano Peluso, and Binoy Ravindran. ezbft: Decentralizing byzantine fault-tolerant state machine replication. In *Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 565–577. IEEE, 2019.

[34] Balaji Arun and Binoy Ravindran. Scalable byzantine fault tolerance via partial decentralization. *Proc. of the VLDB Endowment*, 15(9):1739–1752, 2022.

[35] Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, Ronen Tamari, and David Yakira. A fair consensus protocol for transaction ordering. In *Int. Conf. on Network Protocols (ICNP)*, pages 55–65. IEEE, 2018.

[36] Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM (JACM)*, 41(1):122–152, 1994.

[37] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. *Transactions on Computer Systems (TOCS)*, 32(4):12, 2015.

[38] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. Rbft: Redundant byzantine fault tolerance. In *Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 297–306. IEEE, 2013.

[39] Zeta Avarikioti, Lioba Heimbach, Roland Schmid, Laurent Vanbever, Roger Wattenhofer, and Patrick Wintermeyer. Fnf-bft: A bft protocol with provable performance under attack. In *Int. Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 165–198. Springer, 2023.

[40] Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, (12):1491–1501, 1985.

[41] Amy Babay, John Schultz, Thomas Tantillo, Samuel Beckley, Eamon Jordan, Kevin Ruddell, Kevin Jordan, and Yair Amir. Deploying intrusion-tolerant scada for the power grid. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 328–335. IEEE, 2019.

[42] Leemon Baird. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep*, 2016.

[43] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Conf. on Innovative Data Systems Research (CIDR)*, 2011.

[44] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Sok: Consensus in the age of blockchains. In *Conf. on Advances in Financial Technologies (AFT)*, pages 183–198. ACM, 2019.

[45] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep*, 2019.

[46] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Consensus-oriented parallelization: How to earn your first million. In *Annual Middleware Conf. (Middleware)*, pages 173–184, 2015.

[47] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on steroids: Sgx-based high performance bft. In *European Conf. on Computer Systems (EuroSys)*, pages 222–237, 2017.

[48] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Symposium on Principles of Distributed Computing (PODC)*, pages 27–30. ACM, 1983.

[49] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *Annual international cryptology conference*, pages 701–732. Springer, 2019.

[50] Christian Berger and Hans P Reiser. Scaling byzantine consensus: A broad analysis. In *Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, pages 13–18, 2018.

[51] Christian Berger, Hans P Reiser, João Sousa, and Alysson Bessani. Resilient wide-area byzantine consensus using adaptive weighted replication. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 183–18309. IEEE, 2019.

[52] Christian Berger, Sadok Ben Toumia, and Hans P Reiser. Does my bft protocol implementation scale? In *Int. Workshop on Distributed Infrastructure for the Common Good*, pages 19–24, 2022.

[53] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *Transactions on Storage (TOS)*, 9(4):12, 2013.

[54] Alysson Bessani, Joao Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 355–362. IEEE, 2014.

[55] Alysson Neves Bessani, Paulo Sousa, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. The crutial way of critical infrastructure protection. *IEEE Security & Privacy*, 6(6):44–51, 2008.

[56] Martin Biely, Zarko Milosevic, Nuno Santos, and Andre Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 111–120. IEEE, 2012.

[57] Kenneth P Birman, Thomas A Joseph, Thomas Raeuchle, and Amr El Abbadi. Implementing fault-tolerant distributed objects. *Trans. on Software Engineering*, (6):502–508, 1985.

[58] Fatemeh Borran and André Schiper. A leader-free byzantine consensus algorithm. In *Int. Conf. on Distributed Computing and Networking (ICDCN)*, pages 67–78. Springer, 2010.

[59] Gabriel Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *Symposium on Principles of Distributed Computing (PODC)*, pages 154–162. ACM, 1984.

[60] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[61] F. Brasileiro, F. Greve, A. Mostéfaoui, and M. Raynal. Consensus in one communication step. In *Int. Conf. on Parallel Computing Technologies (PaCT)*, pages 42–50. Springer, 2001.

[62] Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making byzantine consensus live. In *Int. Symposium on Distributed Computing (DISC)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[63] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, and Harry Li. Tao: Facebook's distributed data store for the social graph. In *Annual Technical Conf. (ATC)*, pages 49–60. USENIX Association, 2013.

[64] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: an introduction. *R3 CEV, August*, 1(15):14, 2016.

[65] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, 2016.

[66] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus. *arXiv preprint arXiv:1807.04938*, 2018.

[67] Yehonatan Buchnik and Roy Friedman. Fireledger: a high throughput blockchain consensus protocol. *Proceedings of the VLDB Endowment*, 13(9):1525–1539, 2020.

[68] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[69] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual Int. Cryptology Conf.*, pages 524–541. Springer, 2001.

[70] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[71] Christian Cachin, Jovana Mićić, and Nathalie Steinhauer. Quick order fairness. In *Int. Conf. on Financial Cryptography and Data Security (FC)*, pages 1–18. Springer, 2022.

[72] Christian Cachin and Marko Vukolić. Blockchain consensus protocols in the wild. In *Int. Symposium on Distributed Computing (DISC)*, pages 1–16, 2017.

[73] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186. USENIX Association, 1999.

[74] Miguel Castro and Barbara Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2000.

[75] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[76] Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. In *Conf. on Advances in Financial Technologies (AFT)*, pages 1–11. ACM, 2020.

[77] TH Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous blockchain. *Cryptology ePrint Archive*, 2018.

[78] TH Hubert Chan, Rafael Pass, and Elaine Shi. Pili: An extremely simple synchronous blockchain. *Cryptology ePrint Archive*, 2018.

[79] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *symposium on Principles of Distributed Computing (PODC)*, pages 398–407. ACM, 2007.

[80] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[81] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. Pigpaxos: Devouring the communication bottlenecks in distributed consensus. In *SIGMOD Int. Conf. on Management of Data*, pages 235–247. ACM, 2021.

[82] JP Morgan Chase. Quorum white paper, 2016.

[83] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the tla+ proof system. In *Int. Conf., on Automated Reasoning (IJCAR)*, pages 142–148. Springer, 2010.

[84] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Operating Systems Review (OSR)*, volume 41-6, pages 189–204. ACM SIGOPS, 2007.

[85] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Symposium on Operating Systems Principles (SOSP)*, pages 277–290. ACM, 2009.

[86] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Symposium on Networked Systems Design and Implementation (NSDI)*, volume 9, pages 153–168. USENIX Association, 2009.

[87] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, and Peter Hochschild. Spanner: Google's globally distributed database. *Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[88] Miguel Correia, Nuno Ferreira Neves, Lau Cheuk Lung, and Paulo Veríssimo. Low complexity byzantine-resilient consensus. *Distributed Computing*, 17(3):237–249, 2005.

[89] Miguel Correia, Nuno Ferreira Neves, Lau Cheuk Lung, and Paulo Veríssimo. Worm-it–a wormhole-based intrusion-tolerant group communication system. *Journal of Systems and Software*, 80(2):178–197, 2007.

[90] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *Int. Symposium on Reliable Distributed Systems (SRDS)*, pages 174–183. IEEE, 2004.

[91] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82–96, 2006.

[92] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Bft-to: Intrusion tolerance with less replicas. *The Computer Journal*, 56(6):693–715, 2013.

[93] Miguel Correia, Giuliana Santos Veronese, Nuno Ferreira Neves, and Paulo Verissimo. Byzantine consensus in asynchronous message-passing systems: a survey. *Int. Journal of Critical Computer-Based Systems*, 2(2):141–161, 2011.

[94] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Security Symposium*, pages 857–874. USENIX Association, 2016.

[95] Domenico Cotroneo, Roberto Natella, Roberto Pietrantuono, and Stefano Russo. A survey of software aging and rejuvenation studies. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 10(1):1–34, 2014.

[96] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. Tla+ proofs. In *Int. Symposium on Formal Methods (FM)*, pages 147–154. Springer, 2012.

[97] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 177–190. USENIX Association, 2006.

[98] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. Dbft: Efficient leaderless byzantine consensus and its application to blockchains. In *Int. Symposium on Network Computing and Applications (NCA)*, pages 1–8. IEEE, 2018.

[99] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red belly: a secure, fair and scalable open blockchain. In *Symposium on Security and Privacy (SP)*. IEEE, 2021.

[100] George Danezis and David Hrycyszyn. Blockmania: from block dags to consensus. *arXiv preprint arXiv:1809.01620*, 2018.

[101] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *European Conf. on Computer Systems (EuroSys)*, pages 34–50, 2022.

[102] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Operating Systems Review (OSR)*, volume 41, pages 205–220. ACM, 2007.

[103] Tobias Distler. Byzantine fault-tolerant state-machine replication from a systems perspective. *ACM Computing Surveys (CSUR)*, 54(1):1–38, 2021.

[104] Tobias Distler, Christian Cachin, and Rüdiger Kapitza. Resource-efficient byzantine fault tolerance. *Transactions on Computers*, 65(9):2807–2819, 2016.

[105] Tobias Distler, Ivan Popov, Wolfgang Schröder-Preikschat, Hans P Reiser, and Rüdiger Kapitza. Spare: Replicas on hold. In *Network and Distributed System Security Symposium (NDSS)*, 2011.

[106] Dan Dobre, Ghassan Karame, Wenting Li, Matthias Majuntke, Neeraj Suri, and Marko Vukolić. Powerstore: Proofs of writing for efficient and robust storage. In *Conf. on Computer and communications security (CCS)*, pages 285–298. ACM, 2013.

[107] Dan Dobre, Matthias Majuntke, Marco Serafini, and Neeraj Suri. Hp: Hybrid paxos for wans. In *European Dependable Computing Conf. (EDCC)*, pages 117–126. IEEE, 2010.

[108] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)*, 34(1):77–97, 1987.

[109] Sisi Duan, Michael K Reiter, and Haibin Zhang. Beat: Asynchronous bft made practical. In *Conf. on Computer and Communications Security (CCS)*, pages 2028–2041. ACM, 2018.

[110] Sisi Duan and Haibin Zhang. Practical state machine replication with confidentiality. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 187–196. IEEE, 2016.

[111] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[112] Michael Eischer and Tobias Distler. Latency-aware leader selection for geo-replicated byzantine fault-tolerant systems. In *Int. Conf. on Dependable Systems and Networks Workshops (DSN-W)*, pages 140–145. IEEE, 2018.

[113] Michael Eischer and Tobias Distler. Scalable byzantine fault-tolerant state-machine replication on heterogeneous servers. *Computing*, 101(2):97–118, 2019.

[114] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. An efficient, fault-tolerant protocol for replicated data management. In *SIGACT-SIGMOD symposium on Principles of database systems*, pages 215–229. ACM, 1985.

[115] Amr El Abbadi and Sam Toueg. Availability in partitioned replicated databases. In *SIGACT-SIGMOD symposium on Principles of database systems*, pages 240–251. ACM, 1986.

[116] Ian Aragon Escobar, Eduardo Alchieri, Fernando Luís Dotti, and Fernando Pedone. Boosting concurrency in parallel state machine replication. In *Int. Middleware Conf.*, pages 228–240, 2019.

[117] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[118] Stephanie Forrest, Anil Somayaji, and David H Ackley. Building diverse computer systems. In *Workshop on Hot Topics in Operating Systems*, pages 67–72. IEEE, 1997.

[119] Adam Gągol, Damian Leśniak, Damian Straszak, and Michał Świętek. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. In *Conf. on Advances in Financial Technologies*, pages 214–228. ACM, 2019.

[120] Fangyu Gai, Ali Farahbakhsh, Jianyu Niu, Chen Feng, Ivan Beschastnikh, and Hao Duan. Dissecting the performance of chained-bft. In *Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 595–606. IEEE, 2021.

[121] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Efficient asynchronous byzantine agreement without private setups. In *Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 246–257. IEEE, 2022.

[122] Miguel Garcia, Nuno Neves, and Alysson Bessani. An intrusion-tolerant firewall design for protecting siem systems. In *Conf. on Dependable Systems and Networks Workshop (DSN-W)*, pages 1–7. IEEE, 2013.

[123] Miguel Garcia, Nuno Neves, and Alysson Bessani. Sieveq: A layered bft protection system for critical services. *IEEE Transactions on Dependable and Secure Computing*, 15(3):511–525, 2016.

[124] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *Int. Conf. on Financial Cryptography and Data Security*, pages 296–315. Springer, 2022.

[125] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Symposium on Operating Systems Principles (SOSP)*, pages 51–68. ACM, 2017.

[126] Garth R Goodson, Jay J Wylie, Gregory R Ganger, and Michael K Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 135–144. IEEE, 2004.

[127] Christian Gorenflo, Lukasz Golab, and Srinivasan Keshav. Xox fabric: A hybrid approach to transaction execution. In *Int. Conf. on Blockchain and Cryptocurrency (ICBC)*, pages 1–9. IEEE, 2020.

[128] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second. In *Int. Conf. on Blockchain and Cryptocurrency (ICBC)*, pages 455–463. IEEE, 2019.

[129] Gideon Greenspan. Multichain private blockchain-white paper. *URl: http://www. multichain. com/download/MultiChain-White-Paper. pdf*, 2015.

[130] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *European conf. on Computer systems (EuroSys)*, pages 363–376. ACM, 2010.

[131] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable decentralized trust infrastructure for blockchains. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 568–580. IEEE/IFIP, 2019.

[132] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous bft protocols. In *Conf. on Computer and Communications Security (CCS)*, pages 803–818. ACM, 2020.

[133] Divya Gupta, Lucas Perronne, and Sara Bouchenak. Bft-bench: Towards a practical evaluation of robustness and effectiveness of bft protocols. In *Int. Conf. on Distributed Applications and Interoperable Systems*, pages 115–128. Springer, 2016.

[134] Suyash Gupta, Mohammad Javad Amiri, and Mohammad Sadoghi. Chemistry behind agreement. In *Conf. on Innovative Data Systems Research (CIDR)*, 2023.

[135] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. Proof-of-execution: Reaching consensus through fault-tolerant speculation. In *Int. Conf. on Extending Database Technology (EDBT)*, pages 301–312, 2021.

[136] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. Rcc: Resilient concurrent consensus for high-throughput secure transaction processing. In *Int. Conf. on Data Engineering (ICDE)*, pages 1392–1403. IEEE, 2021.

[137] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. Resilientdb: Global scale resilient blockchain fabric. *Proceedings of the VLDB Endowment*, 13(6):868–883, 2020.

[138] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. The case for byzantine fault detection. In *HotDep*, 2006.

[139] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548*, 2018.

[140] James Hendricks, Gregory R Ganger, and Michael K Reiter. Low-overhead byzantine fault-tolerant storage. *ACM SIGOPS Operating Systems Review*, 41(6):73–86, 2007.

[141] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. In *Int. Conf. on Principles of Distributed Systems (OPODIS)*, 2017.

[142] Yennun Huang, Chandra Kintala, Nick Kolettis, and N Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Int. symposium on fault-tolerant computing. Digest of papers*, pages 381–390. IEEE, 1995.

[143] Mohammad M Jalalzai, Jianyu Niu, Chen Feng, and Fangyu Gai. Fast-hotstuff: A fast and resilient hotstuff protocol. *arXiv preprint arXiv:2010.11454*, 2020.

[144] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 245–256. IEEE, 2011.

[145] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, and Yang Zhang. H-store: a high-performance, distributed main memory transaction processing system. *Proc. of the VLDB Endowment*, 1(2):1496–1499, 2008.

[146] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: resource-efficient byzantine fault tolerance. In *European Conf. on Computer Systems (EuroSys)*, pages 295–308. ACM, 2012.

[147] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, Mike Dahlin, et al. All about eve: Execute-verify replication for multi-core servers. In *Symposium on Operating Systems Design and Implementation (OSDI)*, volume 12, pages 237–250. USENIX Association, 2012.

[148] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Symposium on Principles of Distributed Computing (PODC)*, pages 165–175. ACM, 2021.

[149] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. Themis: Fast, strong order-fairness in byzantine consensus. *The Science of Blockchain Conf. (SBC)*, 2022.

[150] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In *Annual Int. Cryptology Conf.*, pages 451–480. Springer, 2020.

[151] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual Int. Cryptology Conf.*, pages 357–388. Springer, 2017.

[152] Roger M. Kieckhafer and Mohammad H. Azadmanesh. Reaching approximate agreement with mixed-mode faults. *Transactions on Parallel and Distributed Systems*, 5(1):53–63, 1994.

[153] Jonathan Kirsch, Stuart Goose, Yair Amir, Dong Wei, and Paul Skare. Survivable scada via intrusion-tolerant replication. *IEEE Transactions on Smart Grid*, 5(1):60–70, 2013.

[154] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *Security Symposium*, pages 279–296. USENIX Association, 2016.

[155] Eleftherios Kokoris-Kogias. Robust and scalable consensus for sharded distributed ledgers.

[156] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[157] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. *Operating Systems Review (OSR)*, 41(6):45–58, 2007.

[158] Ramakrishna Kotla and Michael Dahlin. High throughput byzantine fault tolerance. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 575–584. IEEE, 2004.

[159] Klaus Kursawe. Wendy, the good little fairness widget: Achieving order fairness for blockchains. In *Conf. on Advances in Financial Technologies (AFT)*, pages 25–36. ACM, 2020.

[160] Klaus Kursawe. Wendy grows up: More order fairness. In *Int. Conf. on Financial Cryptography and Data Security (FC)*, pages 191–196. Springer, 2021.

[161] Petr Kuznetsov, Andrei Tonkikh, and Yan X Zhang. Revisiting optimal resilience of fast byzantine consensus. In *Symposium on Principles of Distributed Computing (PODC)*, pages 343–353. ACM, 2021.

[162] Jae Kwon. Tendermint: Consensus without mining. 2014.

[163] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[164] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[165] Leslie Lamport. Generalized consensus and paxos. 2005.

[166] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[167] Leslie Lamport. Brief announcement: Leaderless byzantine paxos. In *Int. Symposium on Distributed Computing (DISC)*, pages 141–142, 2011.

[168] Leslie Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. 2019.

[169] Leslie Lamport and Mike Massa. Cheap paxos. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 307–314. IEEE, 2004.

[170] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[171] Butler Lampson. The abcd's of paxos. In *Symposium on Principles of Distributed Computing (PODC)*, volume 1, page 13, 2001.

[172] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *European Conf. on Computer Systems (EuroSys)*, pages 1–16, 2020.

[173] Kfir Lev-Ari, Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Fairledger: A fair blockchain protocol for financial institutions. In *Int. Conf. on Principles of Distributed Systems (OPODIS)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[174] Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *Symposium on Networked Systems Design and Implementation (NSDI)*, volume 9, pages 1–14. USENIX Association, 2009.

[175] Bijun Li, Nico Weichbrodt, Johannes Behl, Pierre-Louis Aublin, Tobias Distler, and Rüdiger Kapitza. Troxy: Transparent access to byzantine fault-tolerant systems. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 59–70. IEEE, 2018.

[176] Bijun Li, Wenbo Xu, Muhammad Zeeshan Abid, Tobias Distler, and Rüdiger Kapitza. Sarek: Optimistic parallel ordering in byzantine fault tolerance. In *European Dependable Computing Conf. (EDCC)*, pages 77–88. IEEE, 2016.

[177] Harry C Li, Allen Clement, Amitanand S Aiyer, and Lorenzo Alvisi. The paxos register. In *IEEE Int. Symposium on Reliable Distributed Systems (SRDS)*, pages 114–126. IEEE, 2007.

[178] Jinyuan Li and David Maziéres. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2007.

[179] Wenyu Li, Chenglin Feng, Lei Zhang, Hao Xu, Bin Cao, and Muhammad Ali Imran. A scalable multi-layer pbft consensus for blockchain. *Transactions on Parallel and Distributed Systems*, 32(5):1146–1160, 2020.

[180] Bo Lin and Yanhong A Liu. Distalgo: A language for distributed algorithms, 2017.

[181] Barbara Liskov and James Cowling. Viewstamped replication revisited. 2012.

[182] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. Xft: Practical fault tolerance beyond crashes. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 485–500. USENIX Association, 2016.

[183] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 17–30. ACM, 2016.

[184] Dahlia Malkhi and Kartik Nayak. Hotstuff-2: Optimal two-phase responsive bft. *Cryptology ePrint Archive*, 2023.

[185] Dahlia Malkhi and Michael Reiter. Unreliable intrusion detection in distributed computations. In *Computer Security Foundations Workshop*, pages 116–124. IEEE, 1997.

[186] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed computing*, 11(4):203–213, 1998.

[187] Dahlia Malkhi and Michael K Reiter. Secure and scalable replication in phalanx. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 51–58. IEEE, 1998.

[188] Dahlia Malkhi and Michael K Reiter. Survivable consensus objects. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 271–279. IEEE, 1998.

[189] Yanhua Mao, Flavio P Junqueira, and Keith Marzullo. Towards low latency state machine replication for uncivil wide-area networks. In *Workshop on Hot Topics in System Dependability*. Citeseer, 2009.

[190] J-P Martin and Lorenzo Alvisi. Fast byzantine consensus. *Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

[191] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.

[192] Fred J. Meyer and Dhiraj K. Pradhan. Consensus with dual failure modes. *Transactions on Parallel and Distributed Systems*, (2):214–222, 1991.

[193] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Conf. on Computer and Communications Security (CCS)*, pages 31–42. ACM, 2016.

[194] Zarko Milosevic, Martin Biely, and André Schiper. Bounded delay in byzantine-tolerant state machine replication. In *Int. Symposium on Reliable Distributed Systems (SRDS)*, pages 61–70. IEEE, 2013.

[195] Iulian Moraru, David G Andersen, and Michael Kaminsky. Egalitarian paxos. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 2012.

[196] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Symposium on Operating Systems Principles (SOSP)*, pages 358–372. ACM, 2013.

[197] Louise E Moser, Peter M Melliar-Smith, Priya Narasimhan, Lauren A Tewksbury, and Vana Kalogeraki. The eternal system: An architecture for enterprise applications. In *Int. Enterprise Distributed Object Computing Conf. (EDOC)*, pages 214–222. IEEE, 1999.

[198] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. *arXiv preprint arXiv:1909.05204*, 2019.

[199] Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine smr. In *Int. Symposium on Distributed Computing (DISC)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[200] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Dpaxos: Managing data closer to users for low-latency and mobile applications. In *SIGMOD Int. Conf. on Management of Data*, pages 1221–1236. ACM, 2018.

[201] Faisal Nawab and Mohammad Sadoghi. Blockplane: A global-scale byzantizing middleware. In *Int. Conf. on Data Engineering (ICDE)*, pages 124–135. IEEE, 2019.

[202] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. Kauri: Scalable bft consensus with pipelined tree-based dissemination and aggregation. In *Symposium on Operating Systems Principles (SOSP)*, pages 35–48. ACM, 2021.

[203] Ray Neiheiser, Daniel Presser, Luciana Rech, Manuel Bravo, Luís Rodrigues, and Miguel Correia. Fireplug: Flexible and robust n-version geo-replication of graph databases. In *Int. Conf. on Information Networking (ICOIN)*, pages 110–115. IEEE, 2018.

[204] Nuno Ferreira Neves, Miguel Correia, and Paulo Verissimo. Solving vector consensus with a wormhole. *IEEE Transactions on Parallel and Distributed Systems*, 16(12):1120–1131, 2005.

[205] André Nogueira, Miguel Garcia, Alysson Bessani, and Nuno Neves. On the challenges of building a bft scada. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 163–170. IEEE, 2018.

[206] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Symposium on Principles of Distributed Computing (PODC)*, pages 8–17. ACM, 1988.

[207] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *Annual Technical Conf. (ATC)*, pages 305–319. USENIX Association, 2014.

[208] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *Int.Symposium on Distributed Computing (DISC)*, page 6, 2017.

[209] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Annual Int. Conf. on the Theory and Applications of Cryptographic Techniques*, pages 3–33. Springer, 2018.

[210] Fernando Pedone. Boosting system performance with optimistic distributed protocols. *Computer*, 34(12):80–86, 2001.

[211] Marco Platania, Daniel Obenshain, Thomas Tantillo, Yair Amir, and Neeraj Suri. On choosing server-or client-side solutions for bft. *ACM Computing Surveys (CSUR)*, 48(4):1–30, 2016.

[212] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. Visigoth fault tolerance. In *European Conf. on Computer Systems (EuroSys)*, page 8. ACM, 2015.

[213] Ji Qi, Xusheng Chen, Yunpeng Jiang, Jianyu Jiang, Tianxiang Shen, Shixiong Zhao, Sen Wang, Gong Zhang, Li Chen, Man Ho Au, et al. Bidl: A high-throughput, low-latency permissioned blockchain framework for datacenter networks. In *Symposium on Operating Systems Principles (SOSP)*, pages 18–34. ACM, 2021.

[214] Michael O Rabin. Randomized byzantine generals. In *Symposium on Foundations of Computer Science (SFCS)*, pages 403–409. IEEE, 1983.

[215] HariGovind V Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *Int. Conf. On Principles Of Distributed Systems (OPODIS)*, pages 88–102. Springer, 2005.

[216] Hans P Reiser and Rudiger Kapitza. Hypervisor-based efficient proactive recovery. In *Int. Symposium on Reliable Distributed Systems (SRDS)*, pages 83–92. IEEE, 2007.

[217] Robbert van Renesse, Chi Ho, and Nicolas Schiper. Byzantine chain replication. In *Int. Conf. On Principles Of Distributed Systems (OPODIS)*, pages 345–359. Springer, 2012.

[218] Ronald L Rivest, Adi Shamir, and Leonard M Adleman. *A method for obtaining digital signatures and public key cryptosystems*. Routledge, 2019.

[219] Tom Roeder and Fred B Schneider. Proactive obfuscation. *ACM Transactions on Computer Systems (TOCS)*, 28(2):1–54, 2010.

[220] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. A transactional perspective on execute-order-validate blockchains. In *SIGMOD Int. Conf. on Management of Data*, pages 543–557. ACM, 2020.

[221] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[222] Marco Serafini, Péter Bokor, Dan Dobre, Matthias Majuntke, and Neeraj Suri. Scrooge: Reducing the costs of fast byzantine replication in presence of unresponsive replicas. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 353–362. IEEE, 2010.

[223] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *SIGMOD Int. Conf. on Management of Data*, pages 105–122. ACM, 2019.

[224] Victor Shoup. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 207–220. Springer, 2000.

[225] Nibesh Shrestha, Ittai Abraham, Ling Ren, and Kartik Nayak. On the optimality of optimistic responsiveness. In *Conf. on Computer and Communications Security (CCS)*, pages 839–857. ACM, 2020.

[226] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. Bft protocols under fire. In *Symposium on Networked Systems Design and Implementation (NSDI)*, volume 8, pages 189–204. USENIX Association, 2008.

[227] Hin-Sing Siu, Yeh-Hao Chin, and Wei-Pang Yang. A note on consensus on dual failure modes. *Transactions on Parallel and Distributed Systems*, 7(3):225–230, 1996.

[228] Yee Jiun Song and Robbert van Renesse. Bosco: One-step byzantine asynchronous consensus. In *Int. Symposium on Distributed Computing (DISC)*, pages 438–450. Springer, 2008.

[229] João Sousa and Alysson Bessani. Separating the wheat from the chaff: An empirical design for geo-replicated state machines. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 146–155. IEEE, 2015.

[230] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, 2009.

[231] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 2705–2718, 2022.

[232] Chrysoula Stathakopoulou, Signe Rüsch, Marcus Brandenburger, and Marko Vukolić. Adding fairness to order: Preventing front-running attacks in bft protocols using tees. In *Int. Symp on Reliable Distributed Systems (SRDS)*, pages 34–45. IEEE, 2021.

[233] Chrysoula Stathakopoulou, David Tudor, Matej Pavlovic, and Marko Vukolić. [solution] mir-bft: Scalable and robust bft for decentralized networks. *Journal of Systems Research*, 2(1), 2022.

[234] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up bft with acid (transactions). In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 1–17. ACM, 2021.

[235] Diem Team. Diembft v4: State machine replication in the diem blockchain. Technical report, Technical Report. Diem. https://developers. diem. com/papers/diem-consensus . . . , 2021.

[236] Philip Thambidurai, You-Keun Park, et al. Interactive consistency with multiple failure modes. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 93–100. IEEE, 1988.

[237] Gene Tsudik. Message authentication with one-way hash functions. *ACM SIGCOMM Computer Communication Review*, 22(5):29–38, 1992.

[238] Robbert Van Renesse, Nicolas Schiper, and Fred B Schneider. Vive la différence: Paxos vs. viewstamped replication vs. zab. *IEEE Transactions on Dependable and Secure Computing*, 12(4):472–484, 2014.

[239] Paulo E Veríssimo. Travelling through wormholes: a new look at distributed systems models. *ACM SIGACT News*, 37(1):66–81, 2006.

[240] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *Int. Symposium on Reliable Distributed Systems (SRDS)*, pages 135–144. IEEE, 2009.

[241] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Ebawa: Efficient byzantine agreement for wide-area networks. In *Int. Symposium on High Assurance Systems Engineering (HASE)*, pages 10–19. IEEE, 2010.

[242] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *Transactions on Computers*, 62(1):16–30, 2013.

[243] Gauthier Voron and Vincent Gramoli. Dispel: Byzantine smr with distributed pipelining. *arXiv preprint arXiv:1912.10367*, 2019.

[244] Xin Wang, Sisi Duan, James Clavin, and Haibin Zhang. Bft in blockchains: From protocols to use cases. *ACM Computing Surveys (CSUR)*, 54(10s):1–37, 2022.

[245] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. Scaling replicated state machines with compartmentalization. *Proceedings of the VLDB Endowment*, 14(11):2203–2215, 2021.

[246] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. Zz and the art of practical bft execution. In *European Conf. on Computer Systems (EuroSys)*, pages 123–138. ACM, 2011.

[247] Jiang Xiao, Shijie Zhang, Zhiwei Zhang, Bo Li, Xiaohai Dai, and Hai Jin. Nezha: Exploiting concurrency for transaction processing in dag-based blockchains. In *Int.Conf. on Distributed Computing Systems (ICDCS)*, pages 269–279. IEEE, 2022.

[248] David Yakira, Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, and Ronen Tamari. Helix: A fair blockchain consensus protocol resistant to ordering manipulation. *IEEE Transactions on Network and Service Management*, 18(2):1584–1597, 2021.

[249] Sravya Yandamuri, Ittai Abraham, Kartik Nayak, and Michael K Reiter. Communication-efficient bft using small trusted hardware to tolerate minority corruption. In *Int. Conf. on Principles of Distributed Systems (OPODIS)*, 2022.

[250] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. DispersedLedger:high-throughput byzantine consensus on variable bandwidth networks. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 493–512. USENIX Association, 2022.

[251] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. *Operating Systems Review (OSR)*, 37(5):253–267, 2003.

[252] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Symposium on Principles of Distributed Computing (PODC)*, pages 347–356. ACM, 2019.

[253] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *SIGSAC Conf. on Computer and Communications Security*, pages 931–948. ACM, 2018.

[254] Gengrui Zhang, Fei Pan, Michael Dang'ana, Yunhao Mao, Shashank Motepalli, Shiquan Zhang, and Hans-Arno Jacobsen. Reaching consensus in the byzantine empire: A comprehensive review of bft consensus algorithms. *arXiv preprint arXiv:2204.03181*, 2022.

[255] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without byzantine oligarchy. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 633–649. USENIX Association, 2020.

[256] Lidong Zhou, Fred Schneider, Robbert VanRenesse, and Zygmunt Haas. Secure distributed on-line certification authority, August 22 2002. US Patent App. 10/001,588.

[257] Lidong Zhou, Fred B Schneider, and Robbert Van Renesse. Coca: A secure distributed online certification authority. *ACM Transactions on Computer Systems (TOCS)*, 20(4):329–368, 2002.

A Performance Optimization Dimensions

We present a set of optimization dimensions that target the performance of a BFT protocol.

O 1. Out-of-order processing. The out-of-order processing mechanism enables the leader to continuously propose new requests even when previous requests are still being processed by the backups [135]. Out-of-order processing of requests is possible if the leader does not need to include any certificate or hash of the previous request (block) in its next request.

O 2. Request pipelining. Using request pipelining, the messages of a new consensus instance are piggybacked on the second round messages of the previous instance [202, 252]. This technique is especially efficient when a protocol rotates the leader after every consensus instance.

O 3. Parallel ordering. Client requests can be ordered in parallel by relying on a set of independent ordering groups [46, 47, 176] where each group orders a subset of client requests and then all results are deterministically merged into the final order. Similarly, in multi-leader protocols [22, 33, 34, 39, 113, 136, 176, 194, 233, 243], different replicas are designated as the leader for different consensus instances in parallel and then a global order is determined.

O 4. Parallel execution. Transactions can be executed in parallel to improve the system's overall performance. One approach is to detect non-conflicting transactions and execute them in parallel [27, 116, 158]. This approach requires *a priori* knowledge of a transaction's read-set and write-set. Switching the order of agreement and execution stages and optimistically executing transactions in parallel is another approach [32, 147]. If the execution results are inconsistent (due to faulty replicas, conflicting transactions, or nondeterministic execution), replicas need to rollback their states and sequentially and deterministically re-execute the requests. switching the order of agreement and execution stages also enables replicas to detect any nondeterministic execution [32, 147].

O 5. Read-only requests processing. In pessimistic protocols, replicas can directly execute read-only requests without establishing consensus. However, since replicas may execute the read requests on different states, even non-faulty replicas might not return identical results. To resolve this, the number of required matching replies for both normal and read-only requests needs to be increased from $f+1$ to $2f+1$ in order to ensure consistency (i.e., quorum intersection requirement) [75]. This, however, results in a liveness challenge because f non-faulty replicas might be slow (or in-dark) and not receive the request. As a result, the client might not be able to collect $2f+1$ matching responses (since Byzantine replicas may not send a correct reply to the client).

O 6. Separating ordering and execution. The ordering and execution stages can be separated and implemented in different processes. This separation leads to several advantages [103] such as preventing malicious execution replicas from leaking confidential application state to clients [110, 251], enabling large requests to bypass the ordering stage [85], moving application logic to execution virtual machine [105, 216, 246] or simplifying the parallel ordering of requests [46, 49]. Moreover, while $3f+1$ replicas are needed for ordering, $2f+1$ replicas are sufficient to execute transactions [251].

O 7. Trusted hardware. Using trusted execution environments (TEEs) that prevent equivocation, e.g., Intel's SGX [191], Sanctum [94], and Keystone [172], the number of required replicas can be lowered to $2f+1$ because the trusted component prevents a faulty replica from sending conflicting messages to different replicas without being detected. A trusted component may include an entire virtualization layer [105, 216, 241], a multicast ordering service executed on a hardened Linux kernel [89, 90], a centralized configuration service [217], a trusted log [84], an append-only log [249], a trusted platform module, e.g., counter [241, 242], a smart card TrInc [174], or an FPGA [104, 146]. The current version of Bedrock does not support trusted hardware.

O 8. Request/reply dissemination. A client can either multicast its request to *all replicas* [54, 97, 240] where each replica relays the request to the leader or optimistically send its request to a *contact replica*, typically the leader. The contact replica is known to the client through a reply to an earlier request [73, 157]. If the client timer for the request (τ_1) expires, the client multicasts its request to all replicas. This optimistic mechanism requires fewer messages to be sent from clients to the replicas. However, this comes at the cost of increased network traffic between replicas, because the leader needs to disseminate the full request to other replicas to enable them to eventually execute it.

On the other hand, all replicas can send the results to clients in their reply messages. This, however, leads to significant network overhead for large results. A protocol can optimistically rely on a designated responder replica (chosen by the client or servers) to send the full results. Other replicas then either send the hash of the results to the client or send a signed message to the responder enabling the responder to generate a proof for the results, e.g., SBFT [131]. While this technique reduces network overhead, the client might not receive the results if the responder replica is faulty, the network is unreliable, or the responder replica was in-dark and skipped the execution and applied a checkpoint to catch up [103].

B Case Studies on Protocol Evolution

In this section, we provide insights into how each BFT protocol, mentioned in Figure 4 and Figure 5, maps into the Bedrock design space and relates to one another through using design choices. For illustrative purposes, we describe each protocol relative to PBFT, along with one or more design choices.

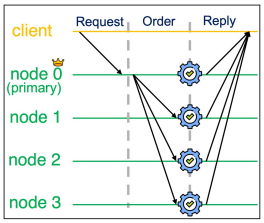


Figure 11: Zyzzyva

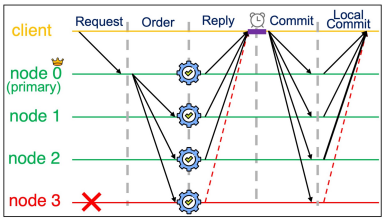


Figure 12: Zyzzyva (slow)

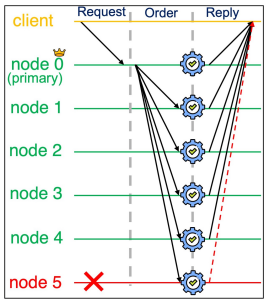


Figure 13: Zyzzyva5

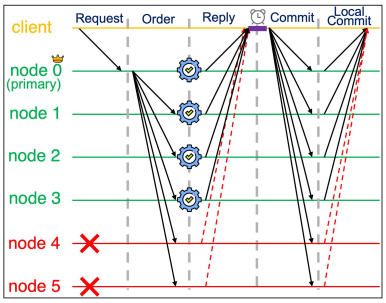


Figure 14: Zyzzyva5 (slow)

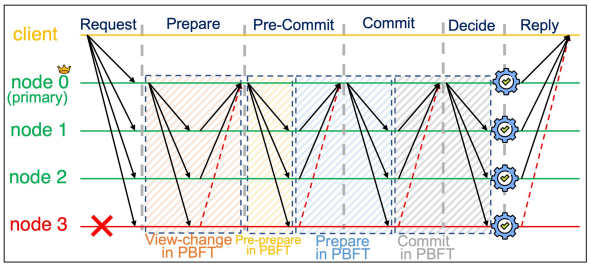


Figure 15: HotStuff

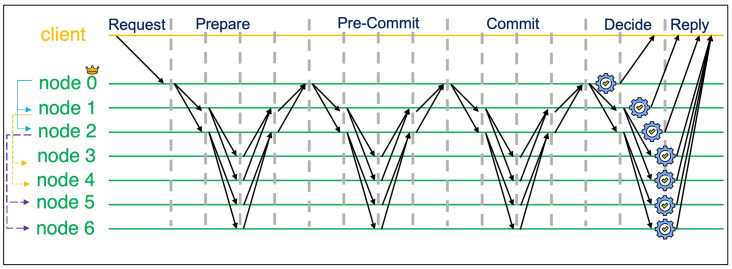


Figure 16: Kauri

Zyzzyva [157]. Zyzzyva[3] (Figure 11) can be derived from PBFT using the speculative execution function (design choice **8**) of Bedrock where assuming the leader and all backups are non-faulty, replicas speculatively execute requests without running any agreement and send reply messages to the client. The client waits for $3f+1$ matching replies to accept the results. If the timer τ_1 is expired and the client received matching replies from between $2f+1$ and $3f$ replicas, as presented in Figure 12, two more linear rounds of communication are needed to ensure that at least $2f+1$ replicas have committed the request. Finally, Zyzzyva5 is derived from Zyzzyva by using the resilience function (design choice **10**) where the number of replicas is increased to $5f+1$ and the protocol is able to tolerate f and $2f$ failures during its fast and slow path respectively (presented in (Figures 13 and 14) AZyzzyva [37, 130] also uses the fast path of Zyzzyva (called ZLight) in its fault-free situations.

PoE [135]. PoE Figure 17 uses the linearization and speculative phase reduction functions (design choices **1** and **7**). PoE does not assume that all replicas are non-faulty and constructs a quorum of $2f+1$ replicas possibly including Byzantine replicas. However, since a client waits for $2f+1$ matching reply messages, all $2f+1$ replicas constructing the quorum need to be well-behaving to guarantee client liveness in the fast path.

SBFT [131]. Bedrock derives SBFT[4] from PBFT using the sign choices **1** and **6**). SBFT presents an optimistic fast path (Figure 18), assuming all replicas are non-faulty. If the leader does not receive messages from *all* backups (in the prepare phase) and its timer is expired (i.e., non-responsiveness timer τ_3), SBFT switches to its slow path (Figure 19) and requires two more linear rounds of communication (commit phase). The Twin-path nature of SBFT requires replicas to sign each message with two schemes (i.e., $2f+1$ and $3f+1$). To send replies to the client, a single (collector) replica receives replies from all replicas and sends a single (threshold) signed reply message.

HotStuff [252]. HotStuff (Figure 15) can be derived from PBFT using the linearization and leader rotation functions (design choices **1** and **3**) of Bedrock. Chained-HotStuff (performance optimization **2**) benefits from pipelining to reduce the latency of request processing.

Tendermint [65, 66, 162]. Tendermint[5] leverages the non-responsive leader rotation function (design choice **4**) to rotate leaders without adding any new phases. The new leader, however, needs to wait for a predefined time (timer τ_4), i.e., the worst-case time it takes to propagate messages over a wide-area peer-to-peer gossip network, before proposing a new block. Tendermint also uses timers in all phases where a replica discards the request if it does not receive $2f+1$ messages before the timeout (timer τ_6). Note that the original Tendermint uses a gossip all-to-all mechanism and has $O(n \log n)$ message complexity.

[3] The view-change stage of the Zyzzyva protocol has a safety violation as described in [7]

[4] SBFT tolerates both crash and Byzantine failure ($n = 3f+2c+1$ where c is the number of crashed replicas). Since the focus of this paper is on linearization and optimistic phase reduction functions (design choices **1** and **6**). SBFT presents an optimistic fast path (Figure 18), assuming all replicas are non-faulty. If the leader does not receive messages from *all* backups (in the prepare phase) and its timer is expired (i.e., non-responsiveness timer τ_3), SBFT switches to its slow path (Figure 19) and requires two more linear rounds of communication (commit phase). The Twin-path nature of SBFT requires replicas to sign each message with two schemes (i.e., $2f+1$ and $3f+1$). To send replies to the client, a single (collector) replica receives replies from all replicas and sends a single (threshold) signed reply message.

Byzantine failures, we consider a variation of SBFT where $c = 0$.

[5] Tendermint uses a Proof-of-Stake variation of PBFT where each replica has a voting power equal to its stake (i.e., locked coins).

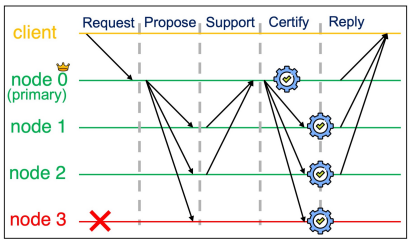


Figure 17: PoE

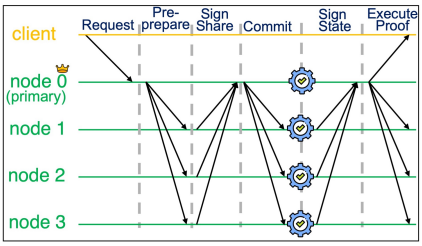


Figure 18: SBFT

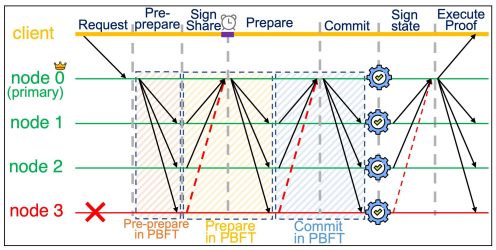


Figure 19: SBFT slow path (Linear PBFT)

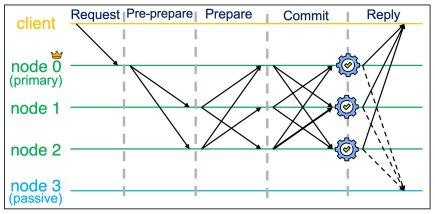


Figure 20: CheapBFT

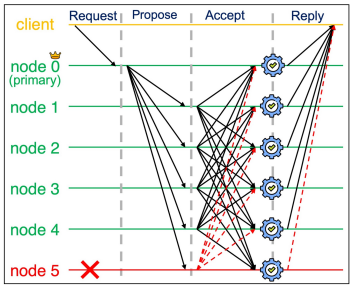


Figure 21: FaB

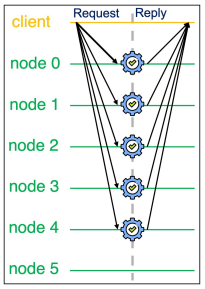


Figure 22: Q/U

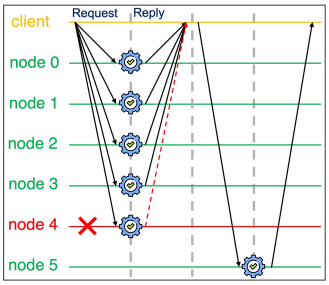


Figure 23: Q/U slow path

Themis [149]. Themis is derived from HotStuff using the fair function (design choice **13**). Themis add a new all-to-all preordering phase where replicas send a batch of requests in the order they received to the leader replica and the leader proposes requests in the order received (depending on the order-fairness parameter γ) [150]. Themis requires at least $4f+1$ replicas (if $\gamma = 1$) to provide order fairness.

Kauri [202]. Kauri (Figure 16) can be derived from HotStuff using the loadbalancer function (design choice **14**) that maps the star topology to the tree topology. The height of the tree is $h = \log_d n$ where d is the fanout of each replica.

CheapBFT [146]. CheapBFT (Figure 20) and its revised version, REBFT [104] is derived from PBFT using the optimistic replica reduction function (design choice **5**). Using trusted hardware (performance optimization O**7**), a variation of REBFT, called RWMINBFT, processes requests with $f+1$ active and f passive replicas in its normal case (optimistic) execution.

FaB [190]. FaB[6] (Figure 21) uses the phase reduction function (design choice **2**) to reduce one phase of communication while requiring $5f+1$ replicas. Fab does not use authentication in its ordering stage, however, requires signatures for the view-change stage (design choice **11**). Note that using authentication, $5f-1$ replicas is sufficient to reduce one phase of communication [12, 161].

Prime [24]. Prime is derived from PBFT using the robust functions (design choice **12**). In prime, a preordering stage is added where replicas exchange the requests they receive from clients and periodically share a vector of all received requests, which they expect the leader to order requests following those vectors. In this way, replicas can also monitor the leader to order requests in a fair manner.

Q/U [5]. Q/U (Figure 22) utilizes optimistic conflict-free and resilience functions (design choices **9** and **10**). Clients play the proposer role and replicas immediately execute an update request if the object has not been modified since the client's last query. Since Q/U is able to tolerate f faulty replicas, a client can optionally communicate with a subset $(4f+1)$ of replicas (preferred quorum). The client communicates with additional replicas only if it does not receive reply from all replicas of the preferred quorum (Figure 23). Both signatures (for large n) and MACs (for small n) can be used for authentication in Q/U. Quorum [37] uses a similar technique with $3f+1$ replicas, i.e., only the conflict-free function (design choices **9**) has been used.

[6] FaB, similar to the family of Paxos-like protocols, separates proposers from acceptors. In our implementation of FaB, however, replicas act as both proposers and acceptors.

C Discovering New Protocol Using Bedrock

Bedrock provides a systematic way to explore new valid points in the design space and help BFT researchers uncover *novel* BFT protocols. We uncover several such new protocols, although not all are necessarily practical or interesting. For example, simply making a protocol fair by adding the preordering phase of fairness results in a new protocol. While this is an interesting insight, the resulting protocol may have limited practical impact. We select as highlights two new BFT protocols (FLB and FTB) that are new and have practical value that we have uncovered using Bedrock.

Fast Linear BFT (FLB). FLB (Figure 24) is a fast linear BFT protocol that commits transactions in two phases of communication with linear message complexity. To achieve this, FLB uses the linearization and phase reduction through redundancy functions (design choices **1** and **2**). FLB requires

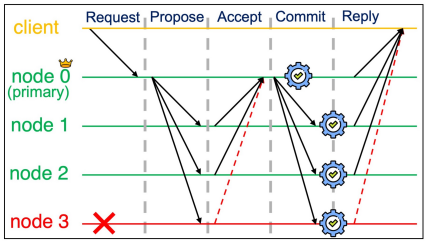


Figure 24: FLB

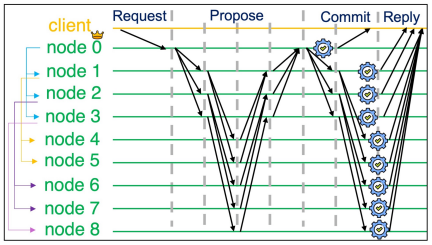


Figure 25: FTB

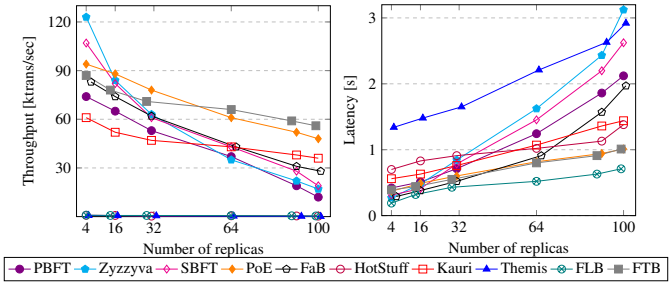


Figure 26: Performance with a geo-distributed setup

$5f-1$ replicas (following the lower bound results on fast Byzantine agreement [12, 161]). The ordering stage of FLB is similar to the fast path of SBFT in terms of the linearity of communication and the number of phases. However, FLB expands the network size to tolerate f failures (in contrast to SBFT, which optimistically assumes all replicas are non-faulty).

Fast Tree-based balanced BFT (FTB). A performance bottleneck of consensus protocols is the computing and bandwidth capacity of the leader. While Kauri [202] leverages a tree communication topology (design choice **14**) to distribute the load among all replicas, Kauri requires $7h$ phases of communication to commit each request, where h is the height of the communication tree.

FTB (Figure 25) reduces the latency of Kauri based on two observations. First, we noticed that while Kauri is implemented on top of HotStuff, it does not use the leader rotation mechanism. As a result, it does not need the two linear phases of HotStuff ($2h$ phases in Kauri) that are added for the purpose of leader rotation (design choice **3**). Second, similar to FLB, we can use the phase reduction through redundancy function (design choice **2**) to further reduce $2h$ more phases of communication. FTB establishes agreement with $5f-1$ replicas in $3h$ phases. FTB also uses the pipelining stretch mechanism of Kauri, where the leader continuously initiates consensus instances before receiving a response from its child nodes for the first instance (similar to the out-of-order processing used by many BFT protocols).

D Impact of a Geo-distributed Setup

In this part, we measure the performance of protocols in a wide-area network. Replicas are deployed in 4 different AWS regions, i.e., Tokyo (*TY*), Seoul (*SU*), Virginia (*VA*), and California (*CA*) with an average Round-Trip Time (RTT) of $TY \rightleftharpoons SU$: 33 ms, $TY \rightleftharpoons VA$: 148 ms, $TY \rightleftharpoons CA$: 107 ms, $SU \rightleftharpoons VA$: 175 ms, $SU \rightleftharpoons CA$: 135 ms, and $VA \rightleftharpoons CA$: 62 ms. The clients are also placed in Oregon (*OR*) with an average RTT of 97, 126, 68 and 22 ms from *TY*, *SU*, *VA* and *CA* respectively. We use a batch size of 400 and perform experiments in a failure-free situation. In this experiment, the pipelining stretch of Kauri and FTB is increased to 6. Figure 26 depicts the results.

Zyzzyva demonstrates the best performance when n is small. However, when n increases, its performance is significantly reduced (87% throughput reduction and 115x latency when n increases from 4 to 100). This is because, in Zyzzyva, clients need to receive reply messages from *all* replicas. Similarly, SBFT incurs a significant reduction in its performance due to its optimistic assumption that all replicas participate in a timely manner. In both protocols, replicas (client or leader) wait for $\Delta = 500$ ms to receive responses from all replicas before switching to the normal path. This reduction can be seen in PBFT as well (84% throughput reduction when n increases to 100) due to its quadratic communication complexity. PoE incurs a smaller throughput reduction (51%) in comparison to Zyzzyva, SBFT, and PBFT because it does not need to wait for all replicas and it has a linear communication complexity. Increasing the number of replicas does not significantly affect the throughput of FTB compared to other protocols (36% throughput reduction when n increases to 99) due to its logarithmic message complexity and pipelining.

Interestingly, HotStuff shows very low throughput. In HotStuff, the leader of the following view must wait for the previous view's decision before initiating its value. Even though Chained-HotStuff is implemented in Bedrock, the leader still needs to wait for one communication round (an RTT). As a result, in contrast to the single datacenter setting where each round takes $\sim$1 ms, request batches are proposed on average every $\sim$190 ms. Similarly, in Themis and FLB, the leader must wait for certificates from $n-f$ replicas before initiating consensus on the next request batch. In Themis, network latency also affects achieving order-fairness as replicas might propose different orders for client requests. This result demonstrates the significant impact of the out-of-order processing of requests on the performance of the protocol, especially in a wide area network.

E PBFT DSL Specification in Bedrock

```
plugins:
  role: primary
  message:
    - digest
    - mac
    - checkpoint
  transition:
    - checkpoint
  pipeline: direct

protocol:
  general:
    leader: stable
    requestTarget: primary

  roles:
    - primary
    - nodes
    - client

  phases:
    - name: normal
      states:
        - idle
        - wait_prepare
        - wait_commit
        - executed
      messages:
        - name: request
          requestBlock: true
        - name: reply
          requestBlock: true
        - name: preprepare
          requestBlock: true
        - prepare
        - commit
    - name: view_change
      states:
        - wait_view_change
        - wait_new_view
      messages:
        - view_change
        - new_view
    - name: checkpoint
      messages:
        - checkpoint

  transitions:
    from:
      - role: client
        state: idle
        to:
          - state: executed
            update: sequence
            condition:
              type: msg
              message: reply
              quorum: 2f + 1

      - role: primary
        state: idle
        to:
          - state: wait_prepare
            condition:
              type: msg
              message: request
              quorum: 1
            response:
              - target: nodes
                message: preprepare
            extra_tally:
              - role: primary
                message: prepare

      - role: nodes
        state: idle
        to:
          - state: wait_prepare
            condition:
              type: msg
              message: preprepare
              quorum: 1
            response:
              - target: nodes
                message: prepare
            extraTally:
              - role: primary
                message: prepare

      - role: nodes
        state: wait_prepare
        to:
          - state: wait_commit
            condition:
              type: msg
              message: prepare
              quorum: 2f + 1
            response:
              - target: nodes
                message: commit
          - state: wait_view_change
            update: view
            condition:
              type: timeout
              mode: sequence
            response:
              - target: nodes
                message: view_change

      - role: nodes
        state: wait_commit
        to:
          - state: executed
            update: sequence
            condition:
              type: msg
              message: commit
              quorum: 2f + 1
            response:
              - target: client
                message: reply

// view-change
      - role: nodes
        state: wait_view_change
        to:
          - state: wait_new_view
            condition:
              type: msg
              message: view_change
              quorum: 2f + 1
      - role: primary
        state: wait_new_view
        to:
          - state: executed
            update: sequence
            response:
              - target: nodes
                message: new_view
              - target: client
                message: reply
      - role: nodes
        state: wait_new_view
        to:
          - state: executed
            update: sequence
            condition:
              type: msg
              message: new_view
              quorum: 1
            response:
              - target: client
                message: reply
          - state: wait_new_view
            update: view
            condition:
              type: timeout
              mode: stat
```

Listing 1: The DSL specification of PBFT Protcol

This section shows the implementation of PBFT using the DSL defined by Bedrock. Listing 1 demonstrates the code. The specification has two main parts: the protocol and the plugins used by the protocol. The plugins, as discussed earlier, are categorized into four groups: role, message, transition, and pipeline. For each category, several plugins have been implemented in Bedrock that can be used by different protocols.

Users can also define their plugins or update existing ones. For example, the pipeline of messages could be `direct`, as it is used in most protocols including PBFT, or `chained` as it is used in Chained-HotStuff [4] or Kauri [202] where messages of consecutive requests are pipelined. The implementation of Digests, MACs, and checkpointing is presented as plugins to enable developers to update them quickly and reuse them in multiple protocols.

The protocol code defines roles, phases, transitions, and the view-change routine. Each phase itself consists of different states, e.g., `idle`, `wait-prepare`, `wait-commit`, and `executed`, and messages, e.g., `request`, `reply`, `preprepare`, `prepare`, and `commit`. The transitions between different states and the condition for each transition are specified in transitions. For example, `node` goes from `idle` state to `wait-prepare` by receiving a `single preprepare` message and in response to this event, `node` sends a `prepare` message to all other `nodes` (as shown in listing 1, lines 75-85). The state manager enables the core unit to track the states and possible transitions of each entity according to the protocol.

DINT: Fast In-Kernel Distributed Transactions with eBPF

Yang Zhou* Xingyu Xiang†* Matthew Kiley Sowmya Dharanipragada‡ Minlan Yu
Harvard University †*Peking University* ‡*Cornell University*

Abstract

Serializable distributed in-memory transactions are important building blocks for data center applications. To achieve high throughput and low latency, existing distributed transaction systems eschew the kernel networking stack and rely heavily on kernel-bypass networking techniques such as RDMA and DPDK. However, kernel-bypass networking techniques generally suffer from security, isolation, protection, maintainability, and debuggability issues, while the kernel networking stack supports these properties well, but performs poorly.

We present DINT, a kernel networking stack-based distributed transaction system that achieves kernel-bypass-like throughput and latency. To gain the performance back under the kernel stack, DINT offloads frequent-path transaction operations directly into the kernel via eBPF techniques without kernel modifications or customized kernel modules, avoiding most of the kernel stack overheads. DINT does not lose the good properties of the kernel stack, as eBPF is a kernel-native technique on modern OSes. On typical transaction workloads, DINT even achieves up to 2.6× higher throughput than using a DPDK-based kernel-bypass stack, while only adding at most 10%/16% average/99th-tail unloaded latency.

1 Introduction

Serializable distributed transactions are important programming abstractions and building blocks for distributed data center applications, such as object store and online transaction processing (OLTP) systems. With the advance of battery-backed DRAM [14] and fast NVRAM [10], the bottleneck of distributed in-memory transactions shifts from the storage to the networking. This has spurred extensive research on how to implement fast distributed in-memory transactions using kernel-bypass networking techniques, such as RDMA [14, 29, 82] and DPDK [6, 28]. One of the key assumptions for these works is that kernel-bypass is the key to realizing fast distributed in-memory transactions that match the underlying hardware speed.

However, kernel-bypass is not a panacea—it essentially trades security [69], isolation [38,39], protection [3,63], maintainability [53,79], and debuggability [70,79] for performance. In addition to these issues, kernel-bypass techniques such as DPDK usually burn one or more CPU cores for busy-polling even at low loads; this is usually non-acceptable in public cloud deployments due to per-core pricing [80]. These issues collectively have led to the well-known Open vSwitch giving up DPDK-based dataplane designs recently [79].

Instead, we choose to embrace the kernel networking stack with interrupt-driven packet processing. The kernel networking stack provides nice properties of good security, isolation, protection, maintainability, debuggability, and load-aware CPU scaling—but not performance. Its poor performance mainly comes from three sources: heavy-weight networking stack traversing [19, 89], user-kernel context switching [89], and interrupt handling.

This paper therefore asks: *can we remove such kernel stack overheads while keeping all of its nice properties for distributed in-memory transactions?* To this end, we follow a decade-old methodology called *extensible kernels* [4], and realize it in modern OS kernels without any kernel code modifications or customized kernel modules. The key enabler is the eBPF technique that allows users to run customized functions easily, safely, and efficiently inside the kernel networking stack at run time. With eBPF, we can run transaction processing logic at the early stage of the kernel networking datapath without going to the user space, avoiding most of the kernel networking stack functions and user-kernel context switching. For the overhead of interrupt handling, it could be amortized by adaptive batching [3] that the kernel networking stack NAPI [34] already did. Besides the potential performance benefit, eBPF is a kernel-native technique shipped with and well-maintained by each release of modern Linux kernels. Due to its safety and kernel-native nature, it has been rapidly adopted by applications and cloud vendors [2, 16, 55]. For example, Meta runs over 40 eBPF programs on every server with ∼100 loaded on demand [75].

We introduce DINT[1], which accelerates distributed transaction systems using eBPF. DINT handles as many transactions as possible in the kernel to improve their critical path performance. In distributed transaction systems, a transaction usually involves three components in its critical path: it first acquires various locks from a lock manager, then reads relevant key-values from a key-value store and does local updates, next logs key-value updates to a log manager, and finally com-

*Equal contribution

[1] DINT as a noun is an archaic word, meaning force and power.

mits key-value updates to the key-value store. Offloading the three components to eBPF is challenging because *eBPF has a constrained programming model (for kernel safety)*.

To address this challenge, our key idea is to redesign transaction-related data structures following the principle of *kernel-offloading* for frequent critical paths to guarantee high performance, and use *user space programs as backups* for rare paths to support full functionalities.

First, the lock manager normally maintains many locks with efficient indexing and complex locking operations. However, it is hard for eBPF to handle hash collision during indexing, because eBPF only allows statically-bounded loops. Further, it is also hard to maintain shared lock states because eBPF does not support common synchronization primitives like Mutex. To address these issues, the DINT lock manager embraces lock sharing to avoid the slow and complex hash collision handling, and directly leverages low-level eBPF atomics to implement transaction locking.

Second, the key-value store normally stores a large number of key-values with different sizes, and requires frequent lookups and updates. However, eBPF does not support dynamic memory allocations, causing low memory efficiency for the key-values. To address these issues, the DINT key-value store directly stores small key-values, which dominate in transaction workloads [12, 47, 77], in kernel memory using a set-associative cache, while leaving large key-values to the user space, avoiding dynamic memory allocations in eBPF. DINT further designs a write-back mechanism with Bloom filters [5] to efficiently handle most key-value lookups and updates in the kernel, while guaranteeing the key-value consistency across the user and kernel.

Third, for the log manager, DINT designs efficient per-CPU log buffers to record logs directly in eBPF, while supporting log replaying from the user space during failure recovery.

We evaluate DINT on two OLTP workloads: a read-intensive TATP workload [47] and a write-intensive SmallBank workload [77]. DINT achieves up to 2.6× higher throughput than using a recent well-engineered kernel-bypass stack based on DPDK (i.e., Caladan [17]), while only adding at most 10% and 16% unloaded latency for the average and 99th-tail respectively. We achieve even higher throughput mainly because the kernel-bypass baseline builds a high-level abstraction for packets that incurs packet copy overhead between network buffers and application buffers, while DINT directly modifies incoming packets and forwards back. DINT's designs are also generic to transaction protocols to some extent—it easily supports an OCC (opportunistic concurrency control) protocol for the read-intensive workload and a 2PL (two-phase locking) protocol for the write-intensive workload.

In summary, this paper makes three contributions:

- We design and implement a high-performance distributed transaction system under the widely-deployed kernel networking stack and the widely-available common commodity NICs, with the key idea of kernel offloading via eBPF.

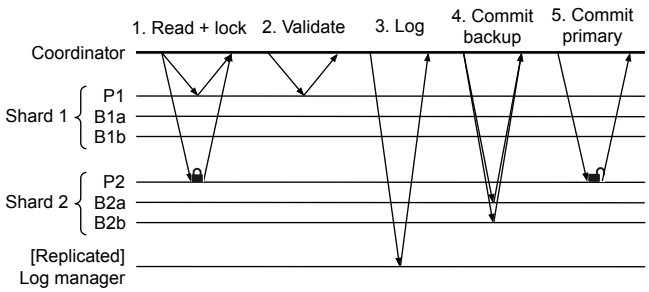


Figure 1: The FaSST [29] transaction protocol with two data shards and three-way replication. P = primary and B = backup. This example transaction reads from the shard 1 and writes to the shard 2.

- We design a state synchronization mechanism for the key-value cache across the kernel and user space that efficiently handles consistency and write-backs.
- We are the first to experimentally show that a distributed transaction system under the kernel networking stack can achieve kernel-bypass-like performance and latency.

DINT has some limitation: it currently targets UDP unreliable transport protocol to simplify packet processing in eBPF. Some research work [76] on designing offload-friendly reliable transports might help address our limitation.

2 Background

2.1 Distributed Transactions

We focus on serializable distributed transactions over a replicated sharded in-memory key-value store with replicated logging to handle failures. Along with recent works [14, 29, 72, 82] in this space, we assume logging into fast persistent storage like battery-backed DRAM or NVRAM (instead of disks) to match in-memory transaction speed, and having a separate fault-tolerance configuration manager to handle machine failures off the critical path of transaction processing. These works usually employ transaction protocols consisting of optimistic concurrency control (OCC) and two-phase commit for distributed atomic commit, and primary-backup replication to support high availability. Below, we briefly go through the critical path of one of such protocols from FaSST [29].

In the FaSST transaction protocol, each transaction has a set of keys to read (i.e., read-set) and a set of key-values to write (i.e., write-set), and a transaction coordinator issues transaction requests to finish each transaction. As shown in Figure 1, the primary in each shard runs a *lock manager*; both the primary and backups run a replicated *key-value store*; a set of servers run a replicated *log manager* (could just be on the primary and backups). To finish a transaction, the transaction coordinator executes the following phases:

1) **Read+lock:** the coordinator reads all values + locks + versions for the read-set and locks all key-values for the write-set. If any key-value in the two sets is already locked, the transaction aborts. The coordinator buffers key-value writes/updates locally.
2) **Validate:** the coordinator reads again all locks + versions

in the read-set, and checks if any read-set value has been changed or locked since the first phase. If so, the transaction aborts.

3) **Log:** the coordinator writes a transaction record containing the write-set's key-values and their versions into the replicated log manager.
4) **Commit backup:** the coordinator updates the write-set values to corresponding backup replicas.
5) **Commit primary:** the coordinator updates the write-set values to corresponding primary replicas, increments key-value versions, and unlocks key-values.

Besides the OCC, there are many more concurrency control protocols for serializable distributed transactions. Another well-known one is two-phase locking (2PL) used in Spanner [9]; it locks before each read and write, and is suitable for write-intensive workloads. More advanced protocols include MDCC [40], Tapir [87], Janus [58], ROCOCO [57], which reduces the number of transaction phases by co-designing concurrency control and replication, and allows more concurrency by tracking fine-grained transaction dependencies.

Distributed transactions inside a data center typically have bottlenecks on the networking stack. For example, when we run the above transaction protocol using a typical OLTP workload under the kernel UDP stack (see §5.2 for a detailed setup), we observe 64% of CPU time is spent on traversing the kernel networking stack, 16% is on the user-kernel context switching, and 12% is on the interrupt handling. This further motivates the huge performance benefits of kernel offloading by avoiding kernel stack overheads.

2.2 eBPF in Kernel Networking Stack

eBPF basics: eBPF (extended Berkeley Packet Filter) is a kernel-native mechanism to let users write *safe, customized* programs that run inside the OS kernel without kernel code modifications or customized kernel modules. Users typically write a high-level C-like eBPF program that gets compiled into low-level eBPF bytecode by Clang/LLVM. Users can then load the eBPF bytecode to predefined attachment points or the so-called *eBPF hooks* in the kernel. Upon loading, the kernel will first verify if the eBPF bytecode meets the safety (e.g., no out-of-bounds memory accesses) and liveness (i.e., it will always terminate in finite steps) requirements. If so, the kernel will compile the eBPF bytecode to native machine code, and run it in a kernel-embedded virtual machine in an event-driven manner; otherwise, the kernel will reject it.

The Linux kernel networking stack has two main eBPF hooks: XDP (eXpress Data Path) [21, 67] and TC (Traffic Control) [50]. The XDP hook only works for ingress packets, and triggers the eBPF program immediately after the NIC driver receives the packet upon NIC interrupts, before `sk_buff` [35] creation. The TC hook works for both ingress and egress packets, and triggers the eBPF program between the NIC driver layer and UDP/TCP layers. For ethernet packet forwarding, TC has lower performance than XDP, as it has

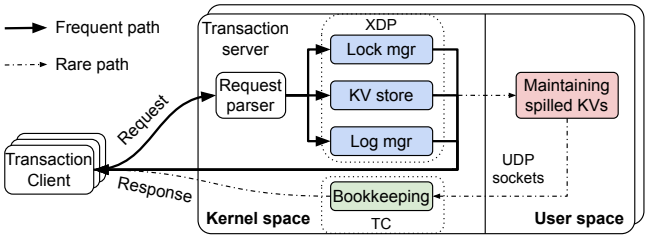


Figure 2: DINT's high-level architecture.

run more kernel networking stack functions.

eBPF maps: eBPF programs are event-driven, therefore program states that cross different invocations must be stored in a global heap-like memory region—eBPF maps are exactly for this purpose. eBPF maps are a variety of built-in data structures in the kernel to maintain eBPF program states with various eBPF helper functions. An eBPF map could contain up to $2^{32}-1$ elements each with maximum $2^{32}-1$ bytes, with total size bounded by the server memory; it must be declared and created statically with a fixed size. Typical eBPF maps include arrays, per-CPU arrays, stacks, and queues [49], with lookup and update functions [32]. The power of eBPF maps is that they can be shared among different eBPF programs and user-space processes. For example, the eBPF program attached to XDP can share an eBPF map with another program on TC and even with a user-space process.

eBPF programming constraints: Due to the safety and liveness verification by the kernel, eBPF programming has some constraints. Perhaps the most important one is not supporting dynamic memory allocations, as correctly handling memory allocation failure and verifying no memory leaks are challenging for eBPF. The second constraint is that eBPF only supports statically-determined bounded loops to ensure liveness. Finally, eBPF lacks high-level thread synchronization primitives such as Mutex. This is because eBPF code runs inside the kernel, and arbitrary/unexpected kernel sleeping by Mutex is dangerous. Instead, eBPF only supports spinlock (i.e., `bpf_spin_lock` [48]) with deep constraints that make it less useful: one cannot call any functions (including built-in eBPF helper functions) while holding the lock, and must release the lock before forwarding/dropping the packet.

3 DINT Design

Figure 2 shows the high-level architecture of DINT. DINT assumes an asymmetric transaction model or the so-called client-side transaction model, similar to [42, 56, 60, 87]. In this model, each transaction client, as the transaction coordinator, sends transaction requests to transaction servers to finish locking, key-value, and logging operations, and then receives responses. As described in Section 6, DINT could also support the symmetric model used in [14, 29, 82]. Like prior works, DINT shards transactions states (i.e., locks, key-values, and logs) among servers, and uses three-way replication and logging for high availability. DINT is generic to a variety

of transaction protocols, and currently supports two different ones: a 2PL-based protocol and an OCC-based protocol similar to FaSST [29].

Offloading request frequent path to kernel: To achieve high-performance transaction processing, DINT offloads frequent-path states and operations into the kernel, avoiding kernel stack overheads. Each DINT transaction server maintains *most of* its transaction states in the kernel memory via eBPF maps, and serves *most of* its transaction requests directly in the kernel via an eBPF program attached to the XDP hook. Since eBPF programs cannot generate new packets by themselves, DINT reuses the request packet by modifying its payload to carry the response message, and forwards it back to clients as the response.

Userspace as backups: To support the full functionalities of transaction processing, DINT handles rare-path states and operations in the user space. Each DINT transaction server runs a user-space process listening on UDP sockets to receive and handle *a small portion* of transaction requests that cannot be served directly in the kernel. Transaction responses returned from the user-space process will go through a bookkeeping eBPF program attached to the TC hook, which helps maintain transaction states in eBPF maps, e.g., releasing some internal locks (not transaction locks).

DINT uses UDP protocol between transaction clients and servers to allow easy parsing of transaction requests and responses in eBPF programs. While UDP protocol is lossy, packet losses happen rarely in modern data centers as shown by prior works [28, 29, 65]. When packet losses happen during severe network hardware failures, DINT would detect such losses using coarse-grained client-side timeouts and handle them by the transaction protocols, similar to FaSST [29]. DINT targets accelerating the handling of transaction requests/responses that can fit into one ethernet packet, i.e., up to 9KB for jumbo frames. This works well for transactions with mostly small key-values, which are quite common in many transaction processing workloads [12, 29, 47, 77, 82]. For large key-values, DINT could just pass them to the user-space process to handle, at the cost of lower throughput.

3.1 DINT Lock Manager

The DINT lock manager is responsible for the transaction concurrency control, i.e., controlling how multiple transaction clients concurrently access individual key-values. Such concurrency control mainly involves quickly indexing lock states by lock IDs and maintaining the shared lock states. These two operations are challenging for the constrained programming model in eBPF that lacks dynamic memory allocations, only supports bounded loops, and has nearly no high-level thread synchronization primitives like Mutex (§2.2). For example, lock state indexing usually requires implementing a hash table in eBPF; however, handling hash collisions is nearly impossible or very inefficient in eBPF for either open hashing that requires dynamically allocating a new hash table entry or closed hashing that may require unbounded loops.

To support efficient lock state indexing and shared lock state maintenance in eBPF, DINT leverages two techniques:

- lock sharing to avoid handling hash collisions. Lock sharing means two lock IDs may get mapped to and use the same lock state. DINT further designs a mechanism to avoid possible deadlocks during lock sharing.
- leveraging low-level eBPF atomics [23] to carefully synchronize shared states operations.

Lock sharing: DINT leverages eBPF array maps (i.e., `BPF_MAP_TYPE_ARRAY` [32]) to implement static tables of lock states in the kernel space. Typical lock states include lock status bits, sharer counters (for read-write locks), etc. Each lock ID gets mapped to one shared lock state in the table via a hash function, and later lock acquiring/releasing operations just work on this lock state. Lock sharing avoids handling tricky hash collisions, at the cost of slightly increasing the failure probability when acquiring locks.

However, deadlocks could happen if a transaction client tries to acquire two locks that get mapped to the same lock state (assuming exclusive locking). This is because: the first acquiring operation succeeds, while the second acquiring fails/blocks; however, the first acquiring will not release the lock until the transaction finishes, while the second acquiring always blocks the transaction progress. To resolve such possible deadlocks, DINT lets the lock manager check if any two exclusive lock acquiring operations on the same lock state come from the same transaction client, by maintaining a holder client ID (e.g., IP and port pair) for each exclusive lock; if so, the lock manager directly returns a locking success message.

By leveraging low-level eBPF atomics, DINT supports a variety of locking mechanisms for concurrency control protocols, including the basic read-write locking for 2PL and version-based locking for OCC, in a fail-and-retry manner [8, 18, 83]. Supporting more advanced concurrency control protocols [40, 57, 58, 87] is also possible in DINT, as they are essentially underpinned by the two basic locking mechanisms; we discuss further in Section 6.

Read-write locking: This locking mechanism includes two types of locks: exclusive locks and shared locks. Transaction clients send lock acquiring/releasing requests with lock IDs to the lock manager, and the manager responds with either success or failure. Lock acquiring requests may receive failure responses, while lock releasing requests always receive success responses. If a client receives a failure response, it will re-send the lock acquiring request again after an optional period of time, until receiving the success response (i.e., fail-and-retry).

To implement the read-write locking, the DINT log manager maintains a per-lock spinlock bit, a per-lock counter that counts how many sharers hold the lock, and a per-lock status bit indicating if this lock is held exclusively. Upon receiving an exclusive lock acquiring request, the lock manager looks up the corresponding spinlock bit and executes eBPF

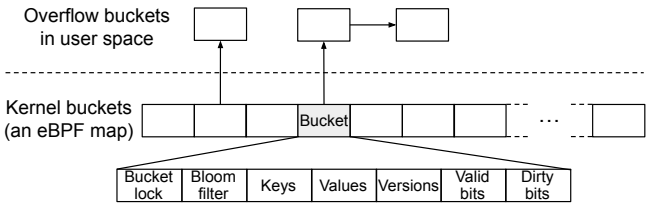


Figure 3: The layout of the key-value store in DINT (assume using the version-based locking).

atomics to check if it can acquire the spinlock. It runs the `__sync_val_compare_and_swap()` function inside eBPF to atomically test-and-set the spinlock bit. This function gets compiled into corresponding ISA-specific operations and is equally efficient as in the user space. If the lock test-and-set succeeds, which means the lock state is not being modified by other transaction clients, the load manager will check the exclusiveness status bit—if the bit is clear, it will set the bit and return a success response; in any other cases, a failure response is returned to let the client retry. Handling the exclusive lock releasing and shared lock operations involves similar atomic operations.

Version-based locking: Version-based locking is widely used in recent high-performance distributed transactions systems [14, 29, 82], together with the OCC protocol to avoid locking operations for key-value reads. It involves version checking to make sure the read key-values used in a transaction are not stale (see §2.1).

To implement version-based locking, DINT maintains a table for the lock status bit indexed by the lock ID, and maintains a per-key-value version counter in a key-value store that we discuss in the next Section. Every read operation directly reads the key-value and corresponding version from the key-value store. Every write operation tries to test-and-set the lock status bit (i.e., exclusive lock); if test-and-set fails, the transaction aborts. After acquiring all write locks and then finishing all transaction writes locally, the transaction coordinator reads the key-value versions again and compares them with the old versions. If the two version vectors do not change, the coordinator can successfully log and commit the transaction, and increment the versions; otherwise, the transaction aborts.

3.2 DINT Key-Value Store

The DINT key-value store maintains the mapping between keys and values, and supports various operations like GET, INSERT, UPDATE, and DELETE. Conventional user-space key-value store [54, 71] would normally maintain a hash index that maps keys to dynamically-allocated values. Unfortunately, this design does not work for eBPF that lacks dynamic memory allocations.

Figure 3 illustrates how DINT addresses this challenge by storing key-values into an in-kernel set-associative cache backed by a fixed-size eBPF map, while spilling overflowed key-values (includes corresponding versions) into the user space. The eBPF map contains many kernel buckets indexed by the key via a hash function, and each bucket stores multiple key-values and valid bits (denoting whether a key-value field stores object data)[2]. By default, DINT stores 4 key-values per kernel bucket. Inside each kernel bucket, DINT stores keys contiguously for better cache locality during lookups; DINT provisions each value field with a fixed size that can cover most of the transaction objects (e.g., dozens of bytes in TPC-C and SmallBank workloads [82, Table 3]). Any kernel bucket that gets too many key-values will spill some key-values into the user space (putting into the overflow buckets); any key-value that cannot fit into the fixed value field in the kernel bucket will also spill into the user space.

A kernel bucket contains a bucket-level lock implemented using eBPF atomics to synchronize concurrent key-value operations on the same bucket. We note that this lock is different from the transaction locks in Section 3.1. Each key-value operation will first try acquiring the bucket lock before touching the bucket data, in a fail-and-retry manner. Most of the time, the key-value operation finishes directly in eBPF and returns the response to clients, before which it releases the bucket lock. In rare cases where its interested key-value is in the user space, the operation needs to pass the operation request/packet to the user-space process via the UDP sockets. Under such cases, the operation still holds the bucket lock when going to the user space, and only releases the lock when it returns to eBPF. By "returns to eBPF", we mean that the response packet sent back by the user-space process will trigger an eBPF program attached to the TC egress hook, which releases the bucket lock.

However, to support high-performance key-value operations in this kernel-user-hybrid key-value store, we must address two additional challenges:

- How to efficiently perform INSERT and UPDATE operations while maintaining *read-all-write* consistency? Prior eBPF-offloaded key-value store BMC [19] adopts a simple write-through cache design and performs well when all operations are GETs. However, in workloads like TATP [47] where 20% of transactions involve INSERTs/UPDATEs, BMC would perform poorly because every such operation will go to the user space.
- How to minimize the chance of going to the user space, especially when clients issue many GET requests for non-existing keys? Non-existing key lookups would require enumerating all keys mapped to the kernel bucket including those spilled into the user space, incurring high kernel stack overheads. Such lookups are common in transaction workloads; e.g., 68.75% of GETs for TATP's largest table target non-existing keys.

To this end, DINT designs a write-back mechanism that lazily and efficiently maintains the read-after-write consistency, and leverages a per-kernel-bucket Bloom filter to avoid frequently going to the user space for non-existing key lookups.

[2] Maintaining the valid bit for each key-value should be straightforward; for conciseness, we do not explicitly describe it unless necessary.

3.2.1 Write-Back Key-Value Store in eBPF

As shown in Figure 3, a kernel bucket contains a dirty bit for each stored key-value, indicating whether the value is different from the user space; a key-value that only exists in eBPF will always have the dirty bit set. Below, we go through how DINT efficiently realizes each of the key-value operations across eBPF and the user space. A recurring theme in the design of each operation is that: DINT tries to support the majority of key-value operations directly in eBPF by leveraging the dirty bit, while maintaining consistency.

GET (Figure 4a): For simplicity, we assume the looked-up key exists in the key-value store; we describe the non-existing case in the next Section. In the frequent path (a) where the GET operation finds the key in the kernel bucket, DINT directly returns the requested value to the client. In the rare path (b) where the GET operation does not find the key: if the kernel bucket is full, DINT chooses one existing key-value to evict following a certain policy (described later) to make a space for the looked-up key-value; otherwise, DINT chooses one dirty key-value (if any) for lazily writing back to the user space. DINT then optionally piggybacks the chosen key-value on the packet and forwards it to the user-space process; DINT uses the `bpf_xdp_adjust_tail()` function [48] to increase the packet size for piggybacking. Once the process receives the request packet, it will look up the key-value in the overflow buckets, and send back the response packet to the client via the UDP sockets. For the piggybacked key-value, the user-space process will update it into the overflow buckets. Finally, the response packet goes through the TC egress eBPF program, which clears the dirty bit of the piggybacked key-value (if any), and fills the requested key-value into an empty or non-dirty key-value field in the kernel bucket.

INSERT (Figure 4b): For simplicity, we assume the to-be-inserted or the incoming key-value does not exist in the key-value store; if it already exists, we could just return a insertion failure message to the client. In the frequent path (a) where the INSERT operation finds an empty slot in the kernel bucket, DINT directly writes the incoming key-value there, sets an initial version and the dirty bit, and returns to the client. In the rare path (b) where there is no empty slot in the kernel bucket, DINT chooses a key-value to evict. Then there will be two cases:

- If the to-be-evicted key-value is not dirty, DINT will directly replace it by the incoming key-value with a *set dirty bit* and an initial version, and return to the client. DINT can directly return as the to-be-evicted key-value has the same copy in the user space, so there is no need to write it back. Since the incoming key-value is marked dirty, a later eviction will lazily write it back to the user space.
- If the to-be-evicted key-value is dirty, DINT will piggyback it on the request packet, replace the bucket's to-be-evicted key-value by the incoming key-value with a *clear dirty bit* and an initial version, then pass the request packet to the user space. The user-space process will then update both the evicted key-value and the incoming key-value into the overflow buckets, and send back a response packet to the client via the UDP sockets.

UPDATE (Figure 4c): For simplicity, we assume the to-be-updated or the incoming key-value exists in the key-value store; if it does not exist, we could just return an update failure message to the client. In the frequent path (a) where the key-value is found in the kernel bucket, DINT directly updates the key-value there with a set dirty bit, increments the version counter, and returns to the client[3]. In the rare path (b) where the key-value is not found, DINT chooses a key-value to evict. No matter whether the to-be-evicted key-value is dirty or not, DINT always needs to go to the user space, in order to fetch (and increment) the version counter corresponding to the incoming key-value. Therefore, DINT will piggyback the to-be-evicted key-value on the request packet, replace the bucket's to-be-evicted key-value by the incoming key-value with a clear dirty bit and an *undefined version*, then pass the request packet to the user space. The user-space process will then update both the evicted key-value and the incoming key-value into the overflow buckets, increment the version counter of the incoming key-value, and send back a response packet to the client via the UDP sockets. More importantly, this response packet will piggyback the updated version of the incoming key-value, so that the TC egress eBPF program can update the bucket's undefined version to the updated one.

Eviction policy: DINT currently uses a simple eviction policy: it tries to evict the first non-dirty key-value when enumerating the bucket; if all the key-values are dirty, it then evicts a random key-value. Prioritizing evicting non-dirty key-value avoids going to the user space as much as possible (especially under INSERT operations). Randomly choosing a key-value if all is dirty minimizes the compute for selecting a victim key-value. Implementing a more complex eviction policy, e.g., based on key-value accessing frequency, might help further reduce the chance of going to the space. But such policy should be compute-light; otherwise, it may incur performance drops [19]. Recent fast cache eviction algorithms such as QD-LP-FIFO [84] may shed light on this space, and we leave such exploration as future work.

Remark: So far, DINT carefully leverages the dirty bits to run the majority of key-value operations directly in eBPF, while maintaining read-all-write consistency and correct key-value versions. For the DELETE operation, we deliberately leave it for the next Section to describe, as it is highly related to how we efficiently handle non-existing key lookups.

3.2.2 Handling GETs for Non-Existing Keys

So far, in the DINT key-value store, GET requests for non-existing keys would enumerate all keys mapped to the indexed kernel bucket including those spilled into the user space, incurring high kernel stack overheads. Conventional key-value

[3] If the new value is too large for the kernel bucket, we will evict the key-value to the user space.

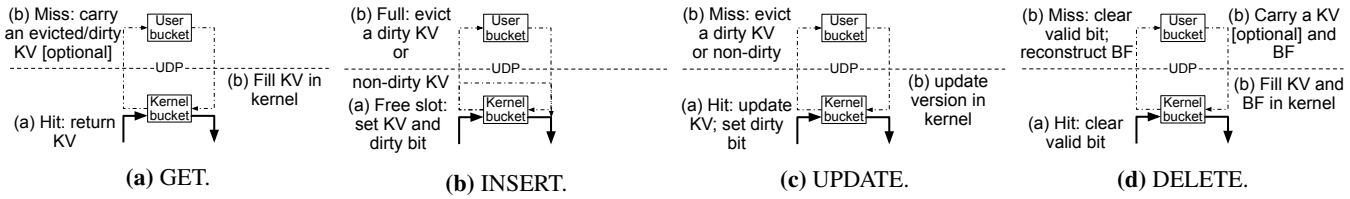


Figure 4: DINT key-value store operations. Solid thick lines indicate frequent paths, while dotted thin lines mean rare paths. BF = Bloom Filter.

stores implemented in the user space do not have such a problem because the enumeration overhead for them is only a few more memory accesses; however, for the key-value store in eBPF, the overhead escalates into expensive kernel networking stack traversing and user-kernel context switching [19,89].

To handle non-existing key GETs efficiently, DINT maintains a small Bloom filter [5] in each kernel bucket, representing the membership of key-values spilled into the user space (Figure 3). The Bloom filter is updated whenever a key-value gets spilled into the user space. When a GET operation does not find the looked-up key in its kernel bucket, it looks up the Bloom filter to check if the key possibly exists in the user space. If the Bloom filter answers no, then the GET operation can guarantee that the key does not exist in the key-value store, and directly return none to the client; otherwise, the operation must go to the user space to check the overflow buckets (see §3.2.1). Since the Bloom filter never reports an existing key as non-existing (i.e., no false negative errors), the above "early returning" in the GET operation is always correct. DINT currently provisions 64 bits for the Bloom filter in each kernel bucket, sufficient to handle dozens of spilled key-values. To reduce the hash calculation overhead for the Bloom filter, DINT reuses the highest six bits of the raw hash value from the key-value store. We choose to implement our custom Bloom filter instead of using the built-in Bloom filter eBPF map [33], in order to avoid extra eBPF map lookup overhead.

However, the Bloom filter design creates a challenge for the key-value DELETE operation. This is because: when the to-be-deleted key-value is in the user space, the DELETE operation will need to remove the key-value from the Bloom filter; however, the Bloom filter does not support membership removal in order to guarantee no false negative errors. To address this challenge, DINT lets the user-space process reconstruct a new Bloom filter for the remaining key-values whenever it deletes one, and then updates the new Bloom filter to the kernel. Reconstructing the Bloom filter is doable, as the user space records all the spilled key-values in its overflow buckets. Formally, the DELETE operation works as follows.

DELETE (Figure 4d): For simplicity, we assume the to-be-deleted key-value exists in the key-value store. In the frequent path (a) where the INSERT operation finds the key-value in the kernel bucket, it clears the valid bit and directly returns to the client. In the rare path (b) where the key-value is not found in the kernel bucket and the Bloom filter reports its existence in the user space, the DELETE operation must forward the request packet to the user space. The user-space process will look up the key-value in the overflow buckets, clear its valid bit, reconstruct a new Bloom filter based on the remaining spilled key-values, and send back a response packet to the client. The response packet will piggyback the new Bloom filter and an optional spilled key-value (if existing, and this key-value should not be covered in the new Bloom filter), and trigger the TC egress eBPF program, which fills the Bloom filter and key-value into the kernel bucket.

3.3 DINT Log Manager

High-performance distributed transaction systems store transaction logs in memory for failure recovery (assuming battery-backed DRAM or fast NVRAM [14, 29]). The transactions logs grow up as the transaction systems run: if they exceed the log space (e.g., memory capacity of the machine), the transaction systems usually truncate the oldest logs [14] or dump them into disks [78]; DINT follows the truncating manner. Since the logging operation is on the transaction critical path, DINT aims to provide a fast logging mechanism entirely inside the eBPF in failure-free cases, while supporting complex offline recovery in failure cases.

To this end, DINT leverages the eBPF maps to implement a circular log buffer abstraction entirely in the kernel. A circular log buffer allows pushing log entries to the tail to support logging operations in transaction systems; it also allows popping log entries from the head (from the user space) to support log replaying during failure recovery. DINT implements such a circular log buffer using a large-size eBPF array map to store log entries, and another eBPF array map to maintain the head and tail, both inside the kernel. These two eBPF maps are also accessible to the user space for log replaying. To avoid thread contentions during logging operations, DINT provisions a circular log buffer on each CPU core. This is achieved by using the eBPF per-CPU array map (`BPF_MAP_TYPE_PERCPU_ARRAY` [32]). When the log manager looks up a per-CPU array map, it will automatically get the map entry corresponding to its local CPU core.

4 DINT Implementation

Our DINT prototype consists of 2.1K lines of eBPF (for kernel code) and 4.3K lines of C++ (for user-space code). DINT uses Clang/LLVM-16 to compile the eBPF program into eBPF bytecode. The eBPF bytecode gets attached to and runs inside the XDP and TC hooks of the standard kernel networking stack, atop unmodified Linux OSes. The user-space process

uses the standard POSIX kernel-visible threads (i.e., pthreads) and the Linux UDP socket to receive rare-path request packets and send response packets. Our prototype currently supports two different transaction protocols, i.e., a 2PL-based protocol and an OCC-based protocol, demonstrating the genericity of DINT's designs to some extent. Our DINT prototype currently does not implement failure recovery to handle machine failures; as described in Section 2.1, we assume a separate configuration manager would handle them off the critical path, thus not impacting the critical-path performance we focus on.

To reduce the performance impact of user-kernel context switching when passing request packets to the user space, DINT runs the user-space process (that handles rare-path requests) on CPU cores that do not receive NIC interrupts or run eBPF programs, similar to prior work [89]. This is achieved by configuring the IRQ affinity of the NIC device to exclude the rare-path handling core. Note that the rare-path handling core does not do any busy polling and can be shared with other applications.

To better reason about the performance of DINT, we build two baseline transaction processing systems that run in the user space. One baseline uses the standard kernel UDP socket with `SO_REUSEPORT` enabled to reduce thread contentions [19], and pthreads. Another baseline uses the UDP stack from the kernel-bypass runtime Caladan [17] that supports DPDK-style packet busy-polling and user-space threading for fast context switching. Both baselines leverage DINT's performance optimizations (e.g., lock sharing) if helpful, but without eBPF programming constraints—so that they can handle hash collisions efficiently using state-of-the-art solutions [44]. The two baselines consist of 6.1K lines of C++.

5 Evaluation

This section aims to answer the following questions:

1. What is the throughput and latency of DINT compared to kernel-bypass approaches (§5.1 and §5.2)?
2. Can DINT support different transaction protocols on transaction workloads efficiently (§5.1 and §5.2)?
3. Can DINT provide load-aware CPU scaling (§5.3)?
4. What are the effects of the write-back mechanism, Bloom filter, and rare paths on DINT's performance (§5.6)?

Testbed: We use 13 `r650` physical machines from CloudLab [15]. Each machine has two 36-core (72 logic-core) Intel Xeon Platinum 8360Y CPUs at 2.4GHz, 256GB memory, and a dual-port Mellanox ConnectX-6 100Gb NIC via PCIe 4.0×16. All machines are connected via a Dell Z9432F switch under the same rack. For all experiments, we use a single CPU in the same NUMA domain as the NIC to enforce NUMA locality; we also use a single 100Gb NIC port, as CloudLab currently only wires one such port of `r650` to the switch.

For all experiments, each machine runs an unmodified Ubuntu 20.04 OS. For eBPF and UDP-related experiments, we use kernel v6.1.0 which has full support for eBPF atomics. We use the built-in Mellanox NIC driver on Linux kernel v6.1.0 that has a default NAPI poll budget/batch size of 64 upon each interrupt. We disable Mellanox NIC's interrupt coalescing feature [11], as we find it hurts latency while not increasing throughput, similar to prior work [89]. For Caladan-related experiments, we are not able to run the Caladan runtime on kernel v6.1.0, as it requires a customized kernel module that relies on specific kernels; instead, we manage to run it on kernel v5.8.0. Since Caladan uses the kernel-bypass networking stack and threading, different OS kernels should not have a significant impact on its performance.

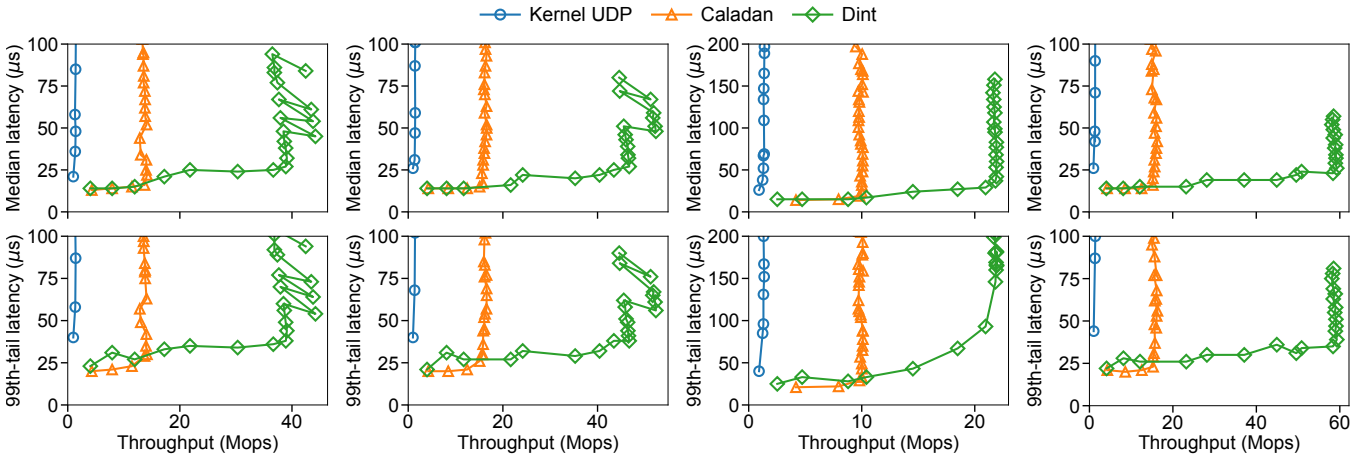
Measurement methodology: For transaction benchmarking, we use 3 machines to run transaction servers with three-way replication and sharding; that is, each machine is the primary for one shard and a replica for the other two. For microbenchmarks that benchmark individual lock manager, key-value store, and log manager, we use 1 machine to run the microbenchmark server without replication or sharding to understand their standalone performance. We use the rest machines to run multiple transaction/microbenchmark clients that issue requests in a closed-loop manner. To avoid the client machines becoming the bottleneck, we provision 8 cores on each transaction/microbenchmark server; the client machines further use Caladan's kernel-bypass UDP stack and user-space threading to generate requests. We then vary the number of clients, and measure the achieved throughput and client-perceived median/average and 99th-tail latency, similar to prior transaction works [6, 42, 56, 87].

Comparison baselines: As mentioned in Section 4, we compare DINT to two baseline transaction processing systems: one is based on the Linux kernel UDP socket, another is based on the UDP stack from the kernel-bypass runtime Caladan [17]. For simplicity, we just use kernel UDP and Caladan to refer to these two baselines respectively. The Caladan baseline is a challenging baseline that features DPDK-style packet busy-polling, NIC RSS to evenly spread packets among available cores, and well-implemented and efficient user-space UDP stack and threading.

We provision the memory sizes of the user-space key-value store (for the two baselines), eBPF key-value store (for DINT), and lock table (for all three) to be 1.5× of the key-values/locks in corresponding workloads, similar to FaSST [29]. By default, kernel UDP and Caladan use all provisioned cores to handle requests, while Caladan uses one extra core to run its scheduler. DINT devotes one core out of the provisioned cores to handle rare-path requests (§4), while the rest cores handle frequent-path requests.

5.1 Microbenchmarks

To understand how each DINT component compares to baselines, we implement a series of microbenchmarks, including a 2PL-based and an OCC-based lock manager with skewed locking requests (80% shared locking requests or version reads), a key-value store with 40B skewed reads, and a log manager with 56B writes. These microbenchmark parameters



(a) 2PL lock manager. **(b)** OCC lock manager. **(c)** Key-value store. **(d)** Log manager.

Figure 5: Microbenchmark load-latency curves (both median and 99th-tail).

(e.g., skewness, value size) are derived from the TATP workload [47]. The two lock managers and the key-value store are provisioned with 36 million lock/key slots, while their requests target 24 million locks/keys.

Lock manager: Figure 5a and 5b show how the latencies (both median and 99th-tail) of the 2PL and OCC lock manager vary with different achieved throughput for different systems, respectively. Each system performs similarly across the two lock managers with the OCC lock manager being slightly faster, as version reads in OCC do not run atomic operations. Overall, DINT achieves 3.1×-3.2× higher throughput than Caladan, with 0%-8%/5%-55% higher unloaded median/99th-tail latency, while kernel UDP performs much worse than others. We notice that DINT has throughput fluctuations at high loads; we think this is because the achieved batch size during interrupt handling gets changed unstably.

It might be supersizing that DINT achieves even higher throughput than the kernel-bypass Caladan system. However, this is achievable, as Caladan wraps raw UDP packets into a high-level connection-oriented abstraction (i.e., `rt::UdpConn`) for applications, which incurs packet copy overhead between network buffers and application buffers, thus losing some performance, while DINT directly works on low-level UDP/ethernet packets. Additionally, each Caladan transaction server creates a `rt::UdpConn` for each transaction client and spawns a user-space thread to handle corresponding transaction requests. Although `rt::UdpConn` only maintains simple connection states with small packet copy and user-space threading is efficient (e.g., 50ns per context switch [61]), they still consume extra CPU time, compared to DINT that directly modifies incoming ethernet packets and forwards back.

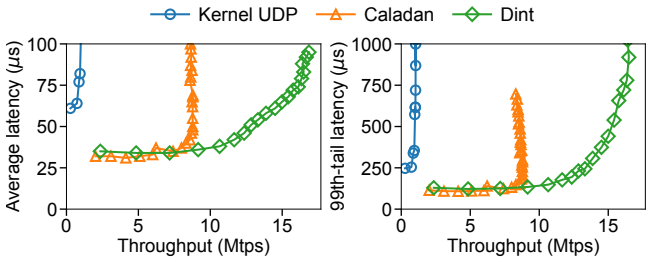
In terms of latency, kernel-bypass Caladan achieves lower minimum latency than kernel-stack DINT, e.g., 13μs vs. 14μs of the median and 20μs vs. 23μs of the 99th-tail for the 2PL lock server. The latency gap, especially for the 99th-tail, is mainly caused by the interrupt-driven nature of DINT, which includes the overheads of NIC interrupt delivery and running interrupt handler. We note that such overheads can be effectively amortized under high loads, thus not impacting throughput. We think the small increased latency is acceptable, as the current data center network usually has one or a few tens of microseconds RTT [20, 51].

Key-value store: Figure 5c shows the load-latency curves for the key-value store. Both Caladan and DINT's performance gets dropped compared to the lock managers, due to more compute in key-value operations. DINT achieves 2.17× higher throughput than Caladan, while having 0%-7%/27%-57% higher unloaded median/99th-tail latency. The minimum latency for Caladan and DINT is 14μs vs. 15μs for the median, and 21μs vs. 25μs for the 99th-tail, demonstrating DINT only incurs small interrupt handling overheads.

Log manager: Figure 5d shows the load-latency curves for the log manager. Similarly, DINT outperforms Caladan on throughput (by 3.6×) but sacrifices latency (by 0%-7% for unloaded median and 5%-40% for unloaded 99th-tail). Regarding the absolute performance number, DINT achieves up to 7.4 Mops per core. This translates into as low as 0.14μs per operation/packet, demonstrating the efficiency of offloading frequent-path operations into the kernel. Both DINT and Caladan achieve higher throughput on the log manager than the lock managers, as the serial log appending operations have better cache locality.

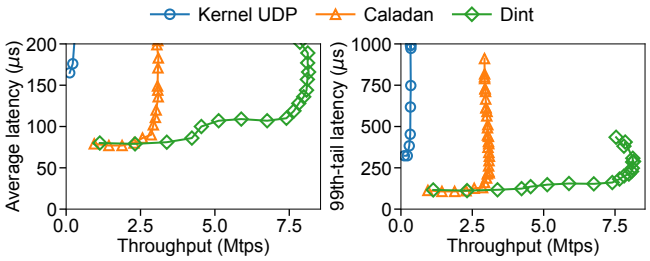
5.2 Transaction Benchmarks

We now evaluate DINT and other baselines on typical OLTP workloads, including TATP [47] and SmallBank [77]. TATP is a read-intensive OLTP benchmark modeling database behaviors of telecommunication providers. It features small key-values (8B keys and 40B values), 80% read-only transactions that read one or more keys, and 20% transactions that modify key-values. We provision 7 million TATP subscribers sharded across the three transaction servers. Similar to prior works [14, 29], we use the OCC-based transaction protocol



(a) Average latency vs. tput. **(b)** 99th-tail latency vs. tput.

Figure 6: OCC on TATP workload. Mtps = Million transactions per second.



(a) Average latency vs. tput. **(b)** 99th-tail latency vs. tput.

Figure 7: 2PL on SmallBank workload.

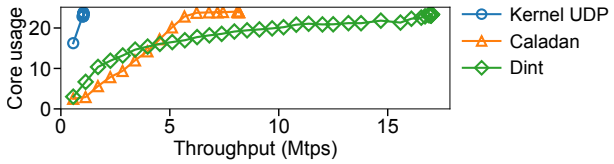


Figure 8: Core usage vs. throughput (on TATP).

(see §2.1) for the read-intensive TATP workload.

SmallBank is a write-intensive OLTP benchmark modeling bank account transactions, with 8B keys and values, and 85% write transactions. We provision 24 million bank accounts sharded across the three transaction servers. We use a 2PL-based transaction protocol suitable for write-intensive workloads. Compared to OCC, the 2PL-based protocol uses read-write locks in the read+lock phase without the validate phase; it has similar log and commit phases (§2.1).

DINT can easily support both transaction protocols by leveraging different lock managers and slightly changing client behaviors, demonstrating the genericity of its designs.

TATP: Figure 6a and 6b show how the average[4] and 99th-tail transaction latencies of different systems change when varying the throughput, respectively. DINT achieves 1.9× higher transaction throughput than Caladan with 6%-10%/12%-16% higher unloaded average/99th-tail latency. As described in Section 5.1, the higher throughput of DINT benefits from directly manipulating and forwarding raw ethernet/UDP packets immediately after the NIC driver receives the packets, in contrast to Caladan that works on a high-level connection-oriented abstraction. Meanwhile, batching effectively amortizes interrupt handling overheads in DINT, leading to a high sustained load on transaction servers. On the other hand, such batching inevitably causes higher latency for DINT when compared to the kernel-bypass polling-based Caladan, i.e., 3μs/14μs higher minimum average/99th-tail latency.

Although not an apple-to-apple comparison, we cite published performance numbers of RDMA-based transaction systems to demonstrate the throughput achieved by DINT is within the same order of magnitude as RDMA-based ones. For example, FaSST reports 8.7 Mtps/machine with 14 cores and 1 million TATP subscribers per machine [29, §6.2], while DINT achieves 5.62 Mtps/machine with 8 cores and 2.3 million subscribers per machine.

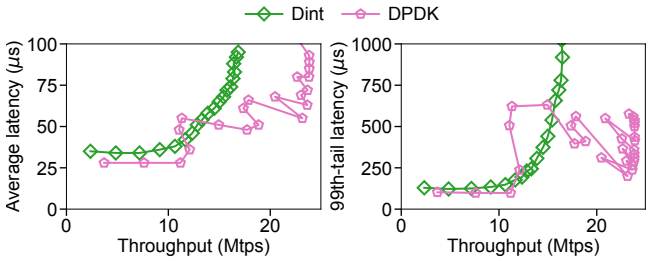
SmallBank: Figure 7a and 7b show the average and 99th-tail transaction latencies of different systems when varying transaction throughput under the SmallBank workload. DINT achieves 2.6× higher throughput than Caladan, while only adding 1%-5%/3%-9% unloaded average/99th-tail latency; the added minimum average/99th-tail latency is 2μs/5μs. Each SmallBank transaction consists of ∼10 transaction requests on average, including locking and key-value operations; therefore, DINT could sustain ∼82 million/sec request rate on 24 cores across three machines. Therefore, DINT's per-core request rate, i.e., ∼3.4 mops, is also within the same order of magnitude as RDMA two-sided operations, i.e., 3.6 mops reported by [81, Figure 3] on a ConnectX-6 NIC.

[4]We show the average rather than the median, as transaction workloads contain many small transactions that dominate the median latency.

5.3 CPU Utilization

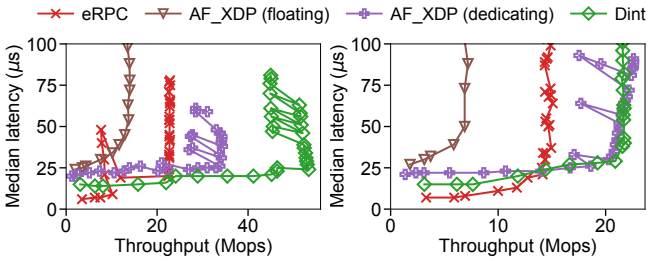
We now examine whether DINT can scale CPU usage as load changes, avoiding burning CPU cores. We use the same TATP workload as in Section 5.2, but provision enough number of clients, specify different transaction rates (by adjusting the sleeping time interval between two consecutive transaction requests in each client), and measure the CPU core usage of transaction servers. For kernel UDP and DINT, they rely on NIC interrupt to wake up any sleeping kernel-visible thread (i.e., pthread) when packets arrive. For Caladan, it supports a CPU-efficient non-spinning mode where the dedicated scheduler busy polls the NIC, and wakes up sleeping user-space threads when needed via IPIs (Inter-Process Interrupt); the Caladan scheduler also reallocates CPU cores every 5μs for the application process based on various load signals (e.g., packet and thread queueing delay [17]), to provision just-enough CPU cores for the current load.

Figure 8 shows how the CPU core usage varies with different throughput for different systems. Until 5 Mtps load, Caladan achieves the lowest core usage and can additively allocate more cores as the throughput increases, due to its fast core reallocation. After 5 Mtps load, DINT achieves lower CPU usage than Caladan and can additively scale its CPU usage to 17 Mtps, because of packet batching during interrupt handling. Kernel UDP has the worst CPU scaling curve, caused by the high overheads of frequent kernel networking stack traversing and user-kernel context switching. Neverthe-



(a) Average latency vs. tput. **(b)** 99th-tail latency vs. tput.

Figure 9: Comparing raw DPDK with DINT (on TATP).



(a) OCC lock manager. **(b)** Key-value store.

Figure 10: Comparing eRPC and AF_XDP with DINT.

less, to enable more efficient CPU scaling for DINT under low loads, one way could be consolidating multiple NIC interrupts onto fewer cores to leverage batching to reduce per-packet processing overheads. We discuss more in Section 6.

5.4 Comparison to Raw DPDK

Figure 9 compares raw DPDK performance with DINT on the TATP workload. Here, "raw DPDK" means busy polling a batch of transaction packets (up to 64, similar to kernel NAPI) from the NIC, processing transaction requests, and then directly modifying and forwarding packets back in a batch as responses. Therefore, it is more efficient than the Caladan baseline, but requires busy polling all cores. Overall, DINT achieves 71% of the raw DPDK performance with 21%-25%/24%-28% higher unloaded average/99th-tail latency. The lower performance of DINT is mainly caused by two factors: 1) DINT devotes one core (out of eight) to the user-space process, and 2) DINT is interrupt-driven, trading some performance for better CPU efficiency by not busy polling any core (see §5.3). We note that the curves of the raw DPDK experience latency spikes in the middle due to insufficient packet batching.

5.5 Comparison to More Baselines

We now compare the performance of DINT with more baselines that leverage other networking stacks. In particular, we compare to eRPC [28] and AF_XDP socket [31]. eRPC is a kernel-bypass event-driven RPC library that builds on top of raw ethernet packets with its own efficient reliable transport protocol. It supports both DPDK and RDMA in busy-polling manners; our testing uses DPDK. AF_XDP is a new kernel socket family that leverages eBPF/XDP to directly DMA packet payload to a pre-registered user-space memory region, so that user-space applications can efficiently receive and send packets in a zero-copy manner. AF_XDP appears to applications as a set of socket APIs, so the application's packet processing logic can be written in a normal programming language (e.g., C/C++, Go) without the strict kernel verification as in eBPF. We run AF_XDP with two modes: floating where all provisioned cores handle NIC interrupts and run transaction servers, and dedicating where half of the cores handle NIC interrupts and another half run transaction servers.

Figure 10a and 10b shows the load-latency curves of eRPC, AF_XDP, and DINT for the OCC lock manager and key-value store respectively. For both applications, eRPC achieves the lowest minimum latency—8μs lower than DINT on both applications. For the lock manager, DINT achieves the highest throughput, and outperforms AF_XDP by 1.6× and eRPC by 2.3×. eRPC suffers from latency spikes at low loads because of insufficient RPC batching. For the key-value store that has more compute per operation, DINT has similar throughput as AF_XDP while achieving 29% lower minimum latency, because of directly handling requests in the kernel without going into the user space; DINT achieves 1.4× higher throughput than eRPC. The throughput results for eRPC must be taken with a grain of salt: eRPC builds a generic loss-tolerant RPC abstraction with session management, while DINT relies on transaction semantics to handle packet losses and works on raw ethernet/UDP packets.

KV workload [Throughput (Mops)]	Write-through (BMC [19])	Write-back	Write-back+BF (DINT)
All GETs, all exists	21.6	21.7	21.7
80% GETs, all exists	1.0	21.1	20.9
80% GETs, 31% exists	0.4	0.5	25.0

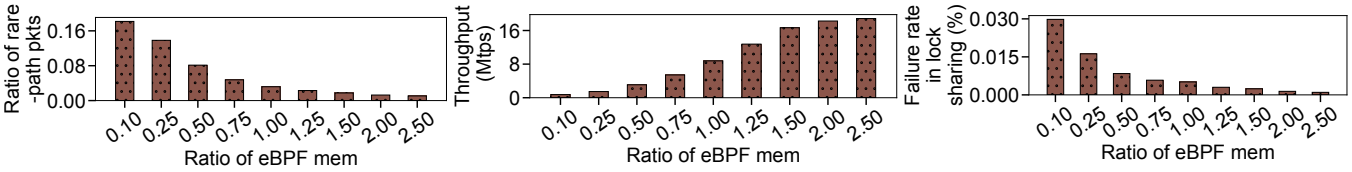
Table 1: Impact of write-back and Bloom filter. "80% GETs" and "31% exists" are based on the TATP workload and its largest table.

One interesting observation is that AF_XDP in the floating model performs much worse than the dedicating mode; similar results occur for DINT on the CPU placement of rare-path request handling process as described in Section 4. This is caused by the high user-kernel context switching overheads when co-locating interrupt handling and the application process on the same cores. We discuss further in Section 6.

5.6 Design Drill-Down

5.6.1 Impact of Write-Back and Bloom Filter

Table 1 shows how the write-back and Bloom filter designs impact DINT performance on different key-value store workloads. With all GETs and all keys existing, the write-through, write-back, and write-back + Bloom filter achieve similar throughput. Once with 20% PUTs, the write-through throughput drops to 1.0 Mops because of handling PUTs in the user space, while the other two keep similarly high throughput. Furthermore, adding 68.75% key-value operations for non-



(a) Rare-path ratio. **(b)** Throughput. **(c)** Failure rate in lock sharing.

Figure 11: Impact of varying the eBPF memory size under the TATP workload. "Ratio of eBPF memory" is against the workload dataset size including both locks and key-values.

Interrupt collocation	Lowest unloaded average/99p latency	Maximum throughput
Collocating with app	35/139 μs	7.0 Mtps
Not collocating (DINT)	34/122 μs	16.9 Mtps

Table 2: Impact of collocating interrupt processing and the application on the same cores (on TATP).

existing keys, only the write-back + Bloom filter can achieve high throughput, as it handles most key-value operations in the kernel, for both existing and non-existing keys.

5.6.2 Impact of Rare-Path Ratio

Figure 11a and 11b shows how different rare-path ratios (by changing the eBPF memory size) impact DINT's transaction throughput. The rare-path ratio significantly impacts DINT performance. For example, with 10% of the workload dataset size in the eBPF memory, which gives 18% of rare-path packet ratio, DINT only achieves 740 Kops. Once we provision the eBPF memory to be 1.5× of the workload dataset size, similar to how FaSST [29] provisions its hash table, there will be only 1.7% of rare-path packet ratio, and DINT reaches 16.7 Mops. This supports the DINT's design principle of offloading frequent-path operations as much as possible into the kernel.

5.6.3 Impact of Lock Sharing

Figure 11c shows how the failure rate caused by lock sharing varies with different sizes of the eBPF memory (i.e., different sizes of the lock table). Overall, the failure rate is under 0.03%; when we provision the eBPF memory to be 1.5× of the workload dataset size, the failure rate is only around 0.002%. This confirms that lock sharing works well on typical OLTP workloads.

5.6.4 Impact of Interrupt Collocation with Applications

Table 2 shows how collocating interrupt processing with the application impacts transaction latency and throughput. Interrupt collocation slightly increases the lowest unloaded average/99th-tail latency by 3%/14% compared to no collocation, because interrupt processing contends CPU cores with application threads; it significantly reduces the maximum throughput by 59%, as high interrupting rate easily starves the application threads, bottlenecking the system performance.

6 Discussion and Future Work

Symmetric vs. asymmetric models: DINT adopts an asymmetric client-side transaction model [42, 56, 60, 87], where each transaction server "passively" handles incoming transaction requests. DINT then leverages eBPF/XDP to offload transaction server operations into the kernel. In contrast, a symmetric model [14, 29, 82] requires the transaction server to also act as a client to issue transaction requests. This creates challenges to DINT, as eBPF/XDP itself cannot generate new packets. Fortunately, by leveraging the AF_XDP technique (see §5.5) that provides fast packet sending functionality, DINT could support symmetric models efficiently. We leave the integration of DINT with AF_XDP as future work.

Implications to networking stack research: DINT shows that the kernel networking stack can achieve kernel-bypass-like throughput and latency, but has worse CPU efficiency under low loads than well-engineered kernel-bypass stacks (§5.3). Therefore, we call for more research on optimizing the CPU efficiency of the kernel networking stack that offloads application operations. One idea may be smartly consolidating NIC interrupts to just-enough CPU cores by manipulating the NIC IRQ affinity, which leverages batching during interrupt handling to reduce per-packet processing overheads. This shares the same goal as Shenango [61] and Caladan [17], but targets the interrupt-driven kernel networking stack.

Another research problem is how to isolate the kernel stack-offloaded operations and user-space operations, as naively co-locating both on the same cores would cause severe performance drop due to frequent user-kernel context switching (see §4 and §5.5). DINT currently uses a simple static partitioning policy, but a more advanced dynamic partitioning policy could possibly provide better performance.

Implications to transaction protocol research: Co-designing transaction protocols with eBPF allows for both high performance and good CPU efficiency. In this work, we co-design an OCC/2PL-based transaction protocol in DINT. DINT should also be able to support more advanced transaction protocols like MDCC [40] and Tapir [87] that essentially rely on read-write and version-based locking. To support advanced protocols like ROCOCO [57] and Janus [58] that maintain transaction dependency DAGs in the lock manager, DINT would need to maintain complex graph data structures in eBPF, which calls for more co-designs to address the challenge of eBPF programming constraints. In an attempt

to reduce CPU utilization, many transaction systems have pushed to incorporate network offload devices like RDMA and smartNICs [14, 73]. However, these devices are much more expensive than commodity NICs, and come with customized network stacks that have high maintenance overheads in terms of engineering. DINT provides the opportunity for accomplishing similar goals without the need for expensive customized hardware, and provides a new point in the design space for transaction protocol developers to explore.

Wish list for eBPF: DINT suffers from the fixed-size eBPF maps, and no dynamic memory allocations for handling large key-values. Therefore, the most helpful eBPF feature would be supporting dynamic memory allocations so that offloaded states could be more memory-efficient. Another helpful feature would be the egress XDP hook. When developing DINT, we were thinking of using the AF_XDP socket to process rare-path request packets (instead of the slower UDP socket); however, AF_XDP faces troubles with the egress bookkeeping of in-kernel states (§3.2), as it relies on the ingress-only XDP while bypassing the egress TC hook. Currently, the only way for AF_XDP to work is by calling eBPF functions in the user space, but this suffers from high syscall overheads. If the kernel supports the egress XDP hook, DINT could instead leverage the faster AF_XDP socket to handle rare paths.

7 Related Work

Distributed in-memory transactions: By leveraging battery-backed DRAM or NVRAM, distributed transactions are no longer bottlenecked by disk IOs, but the networking IOs. This has spurred a series of research that leverages RDMA to implement distributed in-memory transactions, e.g., FaRM [14], FaSST [29], `DrTM` [83], `DrTM+R` [8], `DrTM+H` [82], and Prism [6]. Rather than using RDMA that bypasses kernels, DINT sticks to the most common commodity NICs with the kernel networking stack for better security, isolation, protection, maintainability, and debuggability, without losing performance.

High-performance networking stacks: The inefficiency of traditional kernel networking stack has motivated the designs of many kernel-bypass networking stacks, e.g., mTCP [24], eRPC [28], Snap [51], Demikernel [86] and more [17, 30, 37, 61, 63, 74]. These stacks generally require DPDK-style busy polling, and trades security, protection, maintainability, and more for high performance. Instead, DINT provides comparable high performance without busy polling for distributed transaction applications, while guaranteeing kernel-based security, protection, maintainability, etc.

Perhaps the most relevant work to DINT in this space is IX [3] which implements a protected kernel networking stack and achieves kernel-bypass performance. To achieve high networking performance, IX leverages adaptive batching to amortize user-kernel transition overheads, while DINT relies on the built-in batching of the existing kernel networking stack to amortize interrupt handling overheads. One advantage of DINT over IX is that DINT directly works for existing widely-deployed Linux kernels without any kernel modifications or customized kernel modules.

Hardware offloading for applications: Offloading network-intensive operations to specialized hardware such as FPGA [1, 22, 41, 45], SmartNICs [36, 43, 46, 64, 72, 81], and programmable switches [13, 25, 26, 85] significantly improves application performance. However, they are generally hard to deploy in today's cloud environments [28, 89], as these advanced hardware are not widely available in the public cloud. In contrast, DINT aims to be generic and readily-deployable without relying on any specialized hardware by leveraging the kernel-native eBPF techniques on widely-deployed modern Linux kernels and CPU platforms.

eBPF applications: eBPF is mostly used for packet filtering [52], infrastructure monitoring [2, 66], and L4 load balancing [16] in industry. Recent research has proposed more applications including: accelerating key-value stores [19], sidecar proxies [68], Paxos [89], DBMS proxies [7], gathering congestion control signals [59], guiding request scheduling [27], offloading storage functions [88], and optimizing locks [62]. DINT is a new eBPF application targeting distributed transactions.

8 Conclusion

DINT is a distributed in-memory transaction system under the kernel networking stack, yet achieving kernel-bypass-like throughput and latency. DINT achieves this by offloading transaction data structures and operations into the kernel via eBPF techniques, significantly reducing kernel stack overheads. Compared to a transaction system implemented using Caladan, a well-engineered kernel-bypass networking stack, DINT even achieves 2.6× higher throughput and only adds 10%/16% unloaded average/99th-tail latency.

More importantly, DINT challenges the conventional belief that the kernel networking stack is not suitable for distributed in-memory transactions, or generally, μs-scale networked applications; DINT shows that, with proper application-kernel co-design enabled by eBPF, one important class of such applications under the kernel networking stack can achieve kernel-bypass-like performance. DINT code is available at `https://github.com/DINT-NSDI24/DINT`.

Acknowledgments

We thank our shepherd Tom Barbette and the anonymous reviewers for their insightful comments. We thank Cloudlab [15] for providing us with the development and evaluation infrastructure. We also thank Zhiying Xu and Junzhi Gong for their helpful feedback. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Yang Zhou is also supported by the Google PhD Fellowship.

References

[1] Mohammadreza Alimadadi, Hieu Mai, Shenghsun Cho, Michael Ferdman, Peter Milder, and Shuai Mu. Waverunner: An Elegant Approach to Hardware Acceleration of State Machine Replication. In *Proceedings of USENIX NSDI*, pages 357–374, 2023.

[2] The Cilium Authors. Cilium: eBPF-Based Networking, Observability, Security. https://cilium.io/.

[3] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of USENIX OSDI*, pages 49–65, 2014.

[4] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of ACM SOSP*, pages 267–283, 1995.

[5] Burton H Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan RK Ports. PRISM: Rethinking the RDMA Interface for Distributed Systems. In *Proceedings of ACM SOSP*, pages 228–242, 2021.

[7] Matthew Butrovich, Karthik Ramanathan, John Rollinson, Wan Shen Lim, William Zhang, Justine Sherry, and Andrew Pavlo. Tigger: A Database Proxy That Bounces with User-Bypass. *Proceedings of the VLDB Endowment*, 16(11):3335–3348, 2023.

[8] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and General Distributed Transactions Using RDMA and HTM. In *Proceedings of ACM EuroSys*, pages 1–17, 2016.

[9] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[10] Intel Corporation. Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[11] NVIDIA Corporation. Understanding interrupt moderation. https://enterprise-support.nvidia.com/s/article/understanding-interrupt-moderation.

[12] The Transaction Processing Council. TPC-C: On-Line Transaction Processing Benchmark. https://www.tpc.org/tpcc/.

[13] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at Network Speed. In *Proceedings of ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, pages 1–7, 2015.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of ACM SOSP*, pages 54–70, 2015.

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The Design and Operation of CloudLab. In *Proceedings of USENIX ATC*, pages 1–14, 2019.

[16] Facebook. Katran: A High-Performance Layer 4 Load Balancer. https://github.com/facebookincubator/katran.

[17] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of USENIX OSDI*, pages 281–297, 2020.

[18] Jian Gao, Youyou Lu, Minhui Xie, Qing Wang, and Jiwu Shu. Citron: Distributed Range Lock Management with One-sided RDMA. In *Proceedings of USENIX FAST*, pages 297–314, 2023.

[19] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *Proceedings of USENIX NSDI*, pages 487–501, 2021.

[20] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan MG Wassel, Zhehua Wu, Sunghwan Yoo, et al. Aquila: A unified, low-latency fabric for datacenter networks. In *Proceedings of USENIX NSDI*, pages 1249–1266, 2022.

[21] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of ACM CoNEXT*, pages 54–66, 2018.

[22] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proceedings of USENIX NSDI*, pages 425–438, 2016.

[23] Brendan Jackman. Atomics for eBPF. https://lwn.net/Articles/840224/.

[24] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of USENIX NSDI*, pages 489–502, 2014.

[25] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *Proceedings of USENIX NSDI*, pages 35–49, 2018.

[26] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of ACM SOSP*, pages 121–136, 2017.

[27] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of ACM SOSP*, pages 605–620, 2021.

[28] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of USENIX NSDI*, pages 1–16, 2019.

[29] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of USENIX OSDI*, pages 185–201, 2016.

[30] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of EuroSys*, pages 1–16, 2019.

[31] The Linux kernel development community. AF_XDP. https://docs.kernel.org/networking/af_xdp.html.

[32] The Linux kernel development community. BPF_MAP_TYPE_ARRAY and BPF_MAP_TYPE_PERCPU_ARRAY. https://docs.kernel.org/bpf/map_array.html.

[33] The Linux kernel development community. BPF_MAP_TYPE_BLOOM_FILTER . https://docs.kernel.org/bpf/map_bloom_filter.html.

[34] The Linux kernel development community. NAPI. https://docs.kernel.org/networking/napi.html.

[35] The Linux kernel development community. struct sk_buff. https://docs.kernel.org/networking/skbuff.html.

[36] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *Proceedings of ACM SOSP*, pages 756–771, 2021.

[37] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs First-Class Datacenter Citizens. In *Proceedings of USENIX ATC*, pages 863–880, 2019.

[38] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R Lebeck, and Danyang Zhuo. Understanding RDMA Microarchitecture Resources for Performance Isolation. In *Proceedings of USENIX NSDI*, pages 31–48, 2023.

[39] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding Performance Anomalies in RDMA Subsystems. In *Proceedings of USENIX NSDI*, pages 287–305, 2022.

[40] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-Data Center Consistency. In *Proceedings of EuroSys*, pages 113–126, 2013.

[41] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of ACM SOSP*, pages 137–152, 2017.

[42] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of ACM SOSP*, pages 104–120, 2017.

[43] Junru Li, Youyou Lu, Qing Wang, Jiazhen Lin, Zhe Yang, and Jiwu Shu. AlNiCo: SmartNIC-accelerated Contention-aware Request Scheduling for Transaction Processing. In *Proceedings of USENIX ATC*, pages 951–966, 2022.

[44] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of USENIX NSDI*, pages 429–444, 2014.

[45] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E Stephens, Hassan Wassel, and Aditya Akella.

RingLeader: Efficiently Offloading Intra-Server Orchestration to NICs. In *Proceedings of USENIX NSDI*, pages 1293–1308, 2023.

[46] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading Distributed Applications onto SmartNICs Using iPipe. In *Proceedings of ACM SIGCOMM*, pages 318–333. 2019.

[47] IBM Software Group Information Management. Telecom Application Transaction Processing Benchmark. https://tatpbenchmark.sourceforge.net/.

[48] Linux Programmer's Manual. bpf-helpers(7). https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[49] Linux Programmer's Manual. bpf(2). https://man7.org/linux/man-pages/man2/bpf.2.html.

[50] Linux Programmer's Manual. tc-bpf(8). https://man7.org/linux/man-pages/man8/tc-bpf.8.html.

[51] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A Microkernel Approach to Host Networking. In *Proceedings of ACM SOSP*, pages 399–413, 2019.

[52] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter*, volume 46, 1993.

[53] John McNamara. API/ABI Stability and LTS: Current state and Future. https://www.dpdk.org/wp-content/uploads/sites/35/2017/09/DPDK-Userspace2017-Day2-2-ABI-Stability-and-LTS-Current-state-and-Future.pdf.

[54] The memcached contributors. Memcached - a Distributed Memory Object Caching System. https://memcached.org/.

[55] Microsoft. eBPF implementation that runs on top of Windows. https://github.com/microsoft/ebpf-for-windows.

[56] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a feather flock together: Scaling RDMA RPCs with Flock. In *Proceedings of ACM SOSP*, pages 212–227, 2021.

[57] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting More Concurrency from Distributed Transactions. In *Proceedings of USENIX OSDI*, pages 479–494, 2014.

[58] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Proceedings of USENIX OSDI*, pages 517–532, 2016.

[59] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring Endpoint Congestion Control. In *Proceedings of ACM SIGCOMM*, pages 30–43, 2018.

[60] The University of Texas at Austin. Natacha Crooks. A client-centric approach to transactional datastores. https://repositories.lib.utexas.edu/handle/2152/81352.

[61] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of USENIX NSDI*, pages 361–378, 2019.

[62] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Application-Informed Kernel Synchronization Primitives. In *Proceedings of USENIX OSDI*, pages 667–682, 2022.

[63] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):1–30, 2015.

[64] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *Proceedings of USENIX OSDI*, pages 663–679, 2018.

[65] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of USENIX NSDI*, pages 43–57, 2015.

[66] The IO Visor Project. BPF Compiler Collection (BCC). https://github.com/iovisor/bcc.

[67] The IO Visor Project. eXpress Data Path (XDP). https://www.iovisor.org/technology/xdp.

[68] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. SPRIGHT: Extracting the Server From Serverless Computing! High-Performance eBPF-Based Event-Driven, Shared-Memory Processing. In *Proceedings of ACM SIGCOMM*, pages 780–794, 2022.

[69] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMArk: Bypassing RDMA Security Mechanisms. In *Proceedings of USENIX Security*, pages 4277–4292, 2021.

[70] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S Berger, James C Hoe, Aurojit Panda, and Justine Sherry. We Need Kernel Interposition over the Network Dataplane. In *Proceedings of ACM HotOS*, pages 152–158, 2021.

[71] Salvatore Sanfilippo. Redis: An In-Memory Database That Persists on Disk. https://github.com/redis/redis.

[72] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of ACM SOSP*, pages 740–755, 2021.

[73] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 740–755, New York, NY, USA, 2021. Association for Computing Machinery.

[74] ScyllaDB. SeaStar High Performance Server-Side Application Framework. https://github.com/scylladb/seastar.

[75] Alexei Starovoitov. BPF at Facebook. https://kernel-recipes.org/en/2019/talks/bpf-at-facebook/.

[76] Brent E Stephens, Darius Grassi, Hamidreza Almasi, Tao Ji, Balajee Vamanan, and Aditya Akella. TCP is Harmful to In-Network Computing: Designing a Message Transport Protocol (MTP). In *Proceedings of ACM HotNets*, pages 61–68, 2021.

[77] The H-Store Team. SmallBank Benchmark. https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

[78] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of ACM SOSP*, pages 18–32, 2013.

[79] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the Open vSwitch Dataplane Ten Years Later. In *Proceedings of ACM SIGCOMM*, pages 245–257, 2021.

[80] VMware. Update to VMware's per-CPU Pricing Model | VMware. http://web.archive.org/web/20211023072913/https://news.vmware.com/company/cpu-pricing-model-update-feb-2020.

[81] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing Off-Path SmartNIC for Accelerating Distributed Systems. In *Proceedings of USENIX OSDI*, pages 987–1004, 2023.

[82] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-Enabled Distributed Transactions: Hybrid is Better! In *Proceedings of USENIX OSDI*, pages 233–251, 2018.

[83] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of ACM SOSP*, pages 87–104, 2015.

[84] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and KV Rashmi. FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion. In *Proceedings of ACM HotOS*, pages 70–79, 2023.

[85] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *Proceedings of ACM SIGCOMM*, pages 126–138, 2020.

[86] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of ACM SOSP*, pages 195–211, 2021.

[87] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building Consistent Transactions with Inconsistent Replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–37, 2018.

[88] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. XRP: In-Kernel Storage Functions with eBPF. In *Proceedings of USENIX OSDI*, pages 375–393, 2022.

[89] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating Distributed Protocols with eBPF. In *Proceedings of USENIX NSDI*, pages 1391–1407, 2023.

Brain-on-Switch: Towards Advanced Intelligent Network Data Plane via NN-Driven Traffic Analysis at Line-Speed

Jinzhu Yan[1,*] Haotian Xu[1,*] Zhuotao Liu[1,2,✉] Qi Li[1,2] Ke Xu[1,2]
Mingwei Xu[1,2] Jianping Wu[1,2]
[1] *Tsinghua University* [2] *Zhongguancun Laboratory*

Abstract

The emerging programmable networks sparked significant research on Intelligent Network Data Plane (INDP), which achieves learning-based traffic analysis at line-speed. Prior art in INDP focus on deploying tree/forest models on the data plane. We observe a fundamental limitation in tree-based INDP approaches: although it is possible to represent even larger tree/forest tables on the data plane, the flow features that are computable on the data plane are fundamentally limited by hardware constraints. In this paper, we present BoS to push the boundaries of INDP by enabling Neural Network (NN) driven traffic analysis at line-speed. Many types of NNs (such as Recurrent Neural Network (RNN), and transformers) that are designed to work with sequential data have advantages over tree-based models, because they can take raw network data as input without complex feature computations on the fly. However, the challenge is significant: the recurrent computation scheme used in RNN inference is fundamentally different from the match-action paradigm used on the network data plane. BoS addresses this challenge by (i) designing a novel data plane friendly RNN architecture that can execute unlimited RNN time steps with limited data plane stages, effectively achieving line-speed RNN inference; and (ii) complementing the on-switch RNN model with an off-switch transformer-based traffic analysis module to further boost the overall performance. We implement a prototype of BoS using a P4 programmable switch as our data plane, and extensively evaluate it over multiple traffic analysis tasks. The results show that BoS outperforms state-of-the-art in both analysis accuracy and scalability.

1 Introduction

The emerging programmable network hardware (*e.g.,* P4 switch [5], NetFPGA [36] and SmartNIC [13,37,54]) sparked significant research on Intelligent Network Data Plane (INDP). Compared with other *AI-assisted networking designs* which deploy learning models on either end-hosts (*e.g.,* congestion control [1,63]) or network control plane (including auxiliary servers) (*e.g.,* routing control [29,72]), INDP is *forwarding-native* since it deploys learning models directly on network data plane. Thus, the key merit of INDP, as first summarized in [71], is that it enables intelligent network traffic analysis at line-speed based on data-driven learning models rather than empirical rules/protocols.

The initial exploration of INDP begins with extracting fine-grained flow information from the programmable data plane to support a variety of overarching applications, such as covert channel detection [61], RTT measurement [45], traffic classification [4], and DDoS mitigation [30]. Yet, the subtle distinction between these early approaches and the native INDP paradigm is that they fail to directly deploy learning models on the data plane due to various hardware constraints. For example, the lack of support for floating-point arithmetic on P4 switches makes it significantly more difficult to execute model inference on the data plane than on general-purpose processors like CPUs and GPUs.

The community since then make substantial progress on realizing tree-based INDP [7,24,57,58,68,69,71], based on the insight that the decision making process in tree-models can be implemented using match-action tables on the programmable data plane. State-of-the-art (SOTA) in this regard is NetBeacon [71] which designs a novel ternary table encoding mechanism to efficiently deploy fairly large tree/forest models on programmable switches. Further, the recent art [43,50–52] embrace neural networks by deploying binarized Multi-Layer Perceptron (MLP) on SmartNIC. Yet, the capacity of SmartNIC (*e.g.,* 2×40 GbE for Netronome Agilio CX [37]), which co-locates with an end-host, is several orders of magnitude smaller than the in-network programmable switches (*e.g.,* 6.4 Tbps for Barefoot Tofino 1 switch).

In this paper, we propose Brain-on-Switch (BoS) to advance state-of-the-art of INDP in two fundamental ways. First, BoS enables the use of Neural Network (NN) in INDP. NNs have several advantages over tree-based models for traffic analysis. For instance, Recurrent Neural Network (RNN), a

*Equal contribution. ✉ Corresponding author.

type of NN designed to work with sequential data, outperforms tree-models in both efficiency (*e.g.,* not requiring complicated feature computations on the fly, consuming fewer hardware resources to maintain per-flow state, etc.) and accuracy (especially when handling more complex tasks, such as *multi-class* traffic classification). Second, BoS is architecturally complete in the sense that it can accommodate full-precision and advanced models in INDP. Hardware limitations (*e.g.,* lack of floating-point number support) on switches force model binarization [2, 51, 58], which, unfortunately, reduces performance. Although prior art (*e.g.,* IIsy [68], [49]) mentioned the hybrid analysis concept of forwarding certain flows to large tree-based models deployed at the endpoint for reevaluation, they lack the fundamental design to precisely control the amount of such *escalated* flows processed off-switch. In contrast, BoS proposes a novel approach to accommodating advanced off-switch models (*e.g.,* transformer-based models) into INDP to improve the overall analysis performance, while ensuring that the vast majority of traffic (*e.g.,* over 95% of flows) is still analyzed at line-speed on the data plane.

Concretely, BoS has the following innovative designs:

(i) A novel binary RNN architecture that retains full-precision model weights during on-switch inference (*i.e.,* only activation functions are binarized), realized by encoding the complex layer forward propagation functions as match-action tables. Compared to the fully-binarized MLP model [51], our binary RNN exhibits substantial performance advantages.

(ii) A sliding-window based computation scheme to execute unlimited RNN time steps using limited forwarding stages on switches. BoS overcomes various switch hardware limitations in realizing the critical operations essential to this computation scheme, such as the read/write of a ring buffer like data structure, and an argmax like operation to make comprehensive inference decisions by aggregating the intermediate analysis results as a flow proceeds.

(iii) An analysis-escalation module to accommodate full-precision transformer-based models in BoS. The key design is two-fold: accurately identifying the flows for which on-switch analysis confidence is insufficient, and designing an Integrated Model Inference System (IMIS) to enable fast off-switch inference for escalated flows.

Contributions. The major contribution of this paper is the design, implementation and evaluation of BoS, the first INDP design that enables NN-driven traffic analysis at line-speed. We implement a prototype of BoS and evaluate it extensively using several use cases. The experimental results show that BoS outperforms SOTA in analysis accuracies by non-trivial margins, achieving up to ∼19% higher F1-scores than tree-based NetBeacon [71] and up to ∼40% higher than binary MLP based N3IC [51]. We further perform thorough system-level evaluations, demonstrating that BoS is scalable to handle high network loads (flow concurrency), attributing to the co-design of the on-switch binary RNN and off-switch IMIS. Finally, we evaluate hardware resource utilization by BoS.

2 Background and Motivation

Programmable Network Data Plane. The emerging of Protocol-Independent Switch Architecture (PISA) enables flexible data plane programmability, empowering fast innovations of networking designs. In PISA, the switching pipeline can be programmed via P4 [5], a domain-specific programming language. A PISA pipeline consists of a parser for header parsing, multiple match-action stages for header fields and metadata manipulating, and a deparser for header reassembling. In general, the actual packet processing logic is implemented using these match-action stages. PISA also supports components for stateful storage, such as registers.

Despite the programmability mentioned above, PISA has the following limitations. First, only simple operations like add, subtract, shift and bit-wise operations are supported, excluding floating numbers, multiplication, division and complex comparisons. Second, the resources (such as the number of stages, SRAM, TCAM) are limited. For instance, on Barefoot Tofino 1, each pipeline has 12 stages, 120 Mbit SRAM and 6.2 Mbit TCAM [71]. Finally, each register can only be accessed once through an atomic operation for each packet.

RNNs and Transformers. RNN [31] is designed to process sequence data of varying lengths by maintaining an internal state (*i.e.,* the hidden state). Specifically, given the input $x_t \in \mathbb{R}^m$ at time step t, the hidden state $h_t \in \mathbb{R}^n$ is calculated as $h_t = \tanh(W[x_t, h_{t-1}] + b)$, where W and b are trainable parameters. As h_t encodes the current input x_t and the historical information h_{t-1} at the same time, RNN can capture the relationships between the data points in a sequence. The algorithm used to calculate the hidden states is called a recurrent unit, and the two most popular recurrent units are LSTM [17] and Gated Recurrent Unit (GRU) [8].

Transformers [56] excel at modeling sequential data. Recently, several traffic analysis approaches [11, 26, 28, 59, 66] treat the bytes of packets as words or images, and introduce a variety of transformer-based models to achieve impressive traffic classification performance. In addition, the self-supervised pre-training paradigm used in transformer-based traffic analysis requires a small amount of labeled data.

Motivation. Our community make substantial progress [7, 24, 57, 58, 68, 69, 71] on realizing Intelligent Network Data Plane (INDP) by embedding decision tree models in the forwarding pipeline of programmable switches. Their key insight is that the decision making process in tree/forest models (*i.e.,* comparing a value to a threshold and then moving on to the next tree node until a leaf node is reached) is very similar to the match-action table paradigm used on the data plane. For instance, state-of-the-art NetBeacon [71] designs a novel coding algorithm to effectively represent multiple tree models using ternary matching tables.

We forecast a performance ceiling in further innovating tree-based INDP designs. Specifically, tree models often rely on advanced feature engineering (*i.e.,* extracting various types

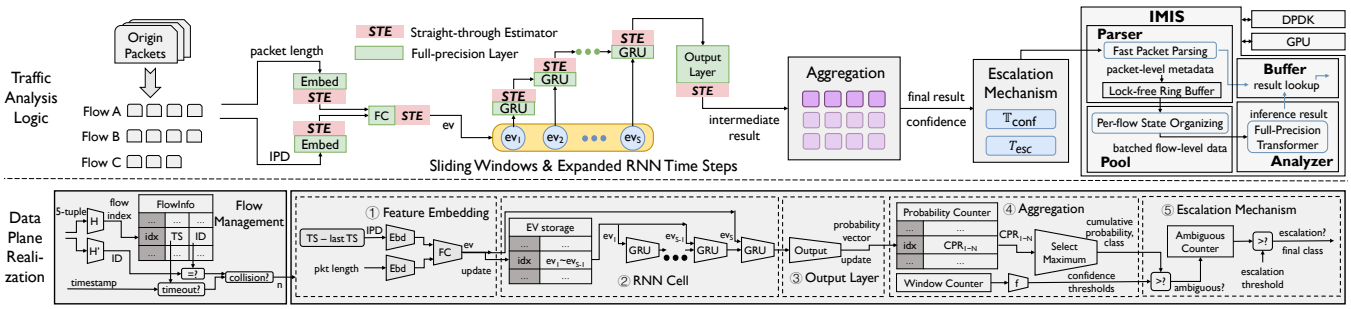


Figure 1: The BoS architecture enables NN-driven traffic analysis in INDP.

of statistics/properties/attributes from raw data) to boost accuracy. However, the features that are computable on the data plane at line-speed are fundamentally limited due to hardware constraints. For instance, flow features such as the s.t.d., frequency, and percentile of packet lengths are critical to tree models [40, 71]. Yet, computing these features is either impossible or difficult, often requiring ad-hoc tricks to estimate these statistics. For instance, prior art [71] estimates s.t.d. of packet lengths upon receiving certain packets (*i.e.,* the 2^k-th packet in each flow), indicating it can only execute inference at these locations. The limitation is obvious: an inference error obtained on the 2^k-th packet cannot be corrected until the arrival of the 2^{k+1}-th packet.

Design Goals. Therefore, philosophically, it is worth asking: can we expand the boundaries of INDP to a new type of learning models that is not limited by the availability of flow features on the data plane. In this paper, we address this research question concretely by enabling NN-driven traffic analysis in INDP. Unlike tree/forest models, many types of NNs (such as RNNs and transformers) that are designed to process sequential data can directly take raw network traffic data as input, eliminating the requirements of computing complex features on the data plane on the fly. However, the incorporation of NNs into INDP presents significant challenges. For instance, the recurrent computation scheme in RNN is fundamentally different from the match-action paradigm on the data plane, making it more difficult to realize on-switch RNN inference. Additionally, existing transformer-based traffic analysis approaches [11,26,28,59,66] simply treat network traffic as another form of sequential data, without constructing appropriate systems to analyze the network flows online while they are traversing the data plane.

To address these challenges, we architect BoS, the first NN-driven INDP system. A recent art N3IC [51] explores to deploy binary MLP models on SmartNIC, which is more computationally flexible, yet with much lower throughput than programmable switches. We focus on programmable switch based INDP in this paper (although we also compare the traffic analysis accuracy of BoS with N3IC in our evaluations). Concurrent with BoS, Broadcom unveils the early-stage development of their novel NN inference switching chip [6], underlining the significance of INDP.

3 Design Overview

We plot the architecture of BoS in Figure 1. The overarching traffic analysis logic in BoS centers around (i) a data plane friendly RNN inference architecture and (ii) a co-design with an Integrated Model Inference System (IMIS) to accommodate full-precision transformer-based traffic analysis models. The key designs toward hardware friendliness are two-fold: (i) realizing the online forward propagation of RNN layers via offline-trained input-output-mapping tables, and (ii) executing unlimited recurrence of RNN time steps via a sliding window mechanism that recurrently processes fixed-length packets segments. The key to co-design with IMIS is accurately identifying the flows that do not receive sufficient confidence from the on-switch analysis, and to only escalate these flows to the off-switch IMIS, so that BoS still processes the vast majority of traffic on-switch (*e.g.,* over 95% flows). Nevertheless, we optimize the system design of IMIS so that a single instance of IMIS can process ten million packets per second while maintaining low inference latencies (see § 7.3). This ensures the off-switch IMIS is unlikely to be the bottleneck of BoS.

4 Data Plane Friendly RNN Architecture

4.1 Raw Packet Sequences as Input Features

The on-switch RNN uses raw flow sequences as the input features, *i.e.,* the packet length sequence and the inter-packet delay (IPD) sequence. When a packet in the flow arrives at the switch, we extract its length and get IPD based on the subtraction of timestamps. Through feature embedding, these metadata are mapped into an embedding vector, which is stored in a sequence for subsequent model inference.

Using raw sequences as input features has several key advantages over using statistical flow features (such as the mean and s.t.d. of packet lengths). First, the availability of critical flow features is greatly limited on switch (explained in § 2). Second, storing per-flow statistical features on switch is expensive: NetBeacon [71] consumes 2.5x storage to store features as evaluated in § 7.2. Finally, feature engineering, without careful designs, could result in overfitting problem. For instance, we notice that some features (*e.g.,* the num-

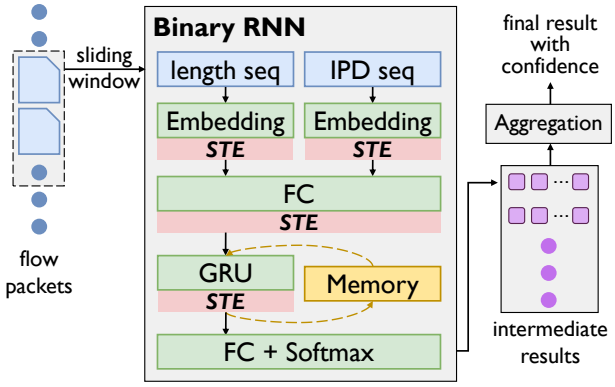


Figure 2: Data-plane-friendly binary RNN architecture.

ber of packets with packet size in $[48, 64)$) are heavily task-specific [3, 71], and features like port number can lead to overfitting on host configurations.

4.2 Binary RNN Architecture

As shown in Figure 2, architecturally, the binary RNN model in BoS consists of three building blocks: feature embedding, RNN cell and output layer. In the feature embedding, the length and IPD of each packet are passed through two different embedding layers, respectively, and then fed into a fully-connected layer to obtain an embedding vector. Taking the embedding vector sequence of per-flow packets as input, the RNN cell performs sequence analysis based on GRU [8]. In each time step of GRU calculation, the current embedding vector and the previous hidden state (*i.e.,* output activations of GRU) are used as input, and the output is used to update the hidden state. Finally, the hidden state of the last step is passed through the output layer (*i.e.,* a fully-connected layer with softmax) to obtain the analysis result.

Each GRU calculation contains 3 Hadamard product operations and 3 multiplications with nonlinear functions, which cannot be natively implemented on programmable switches due to hardware constraints. To cope with this challenge, we perform binarization on neural network activation functions to enable hardware-friendly model deployment. Specifically, we set all activation functions in the feature embedding and the RNN cell to Straight-Through Estimator (STE) [64]. STE performs a sign function in forward propagation, which makes all neural network activations +1 or -1. And in backward propagation, STE estimates the incoming gradient to be equal to the clipped outgoing gradient.

Prior art N3IC [51] performs binarization on both weights and activations of an MLP model, and then implements fully-connected layer forward propagation on the SmartNIC using XOR and customized population count (popcnt) operations. The popcnt operation, unfortunately, is not friendly to the switch pipeline: realizing a single popcnt operation for a 128-bit string takes 14 switch stages. Yet, one 128bit-to-64bit fully-connected layer in an MLP model requires 64 popcnt

Table 1: Binary RNN v.s. Binary MLP

Prior Work	Binary Activations	Full Precision Weights	Stage Consumption⋆	Model Accuracy†
Binary MLP (N3IC [51])	✔	✗	High	Low
Binary RNN	✔	✔	Low	High

⋆ Estimated if we were to implement the binary MLP on a programmable switch.
† See § 7.2 for quantitative results.

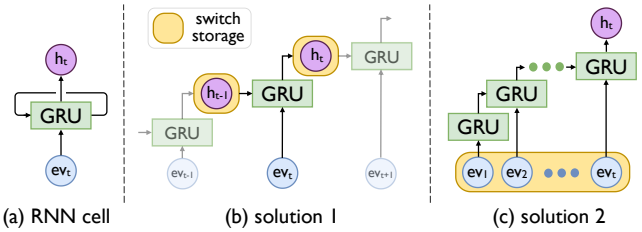


Figure 3: The design choices for RNN time steps.

operations. More crucially, as evaluated in § 7.2, full model binarization results in significant performance degradation. In Table 1, we summarize the key differences between our binary RNN and the binary MLP in N3IC [51].

4.3 Data Plane Native Model Inference

We now present the data plane native RNN inference.

Forward Propagation. The key to retain full precision model weights in our RNN models is to avoid direct computations of the layer forward propagation on the data plane. To this end, BoS realizes forward propagation based on match-action table lookup. Specifically, since all activations are binarized to +1 or -1, the input and output vectors of any neural network layer (*e.g.,* the embedding layer, FC layer and GRU layer in Figure 2) are essentially bit strings. Therefore, regardless of what computations are executed in a neural network layer, we can realize *equivalent input-output-relationship* by recording an enumerative mapping from input bit strings to output bit strings as a match-action table. Thus, in the online forward propagation through any layer, BoS uses input bit string as the key to match the output bit string stored in the corresponding table on switch. The caveat of this design is that the number of required entries N in each table is determined by the number of input bits, *i.e.,* $N = 2^{\text{input bit-length}}$. We recognize this constraint and demonstrate, via experiments (§ 7.2), that BoS can deploy efficient RNN models under this constraint.

RNN Time Steps. As shown in Figure 3(a), a straightforward way to implement RNN time steps is to store the RNN hidden state for each flow sequence. Upon packet arrival, we read the previous RNN hidden state, perform layer forward propagation, and then update the hidden state. Unfortunately, due to hardware constraint, each register can only be accessed once when a packet traverses the switching pipeline. Therefore, we need to expand RNN time steps in serial stages, as shown in Figure 3(b), where the read/write of hidden states spread across multiple stages. Alternatively, we can store a sequence

of embedding vectors corresponding to packet sequence of a flow, as shown in Figure 3(c). In online forward propagation, BoS calculates the embedding vector for each packet, updates the sequence in storage and executes all RNN time steps for this flow sequence in serial stages. We adopt this solution in the final prototype as it consumes fewer hardware resources.

Sliding Window Mechanism. Since the number of switch stages is limited, expanding the RNN time steps into serial stages would also limit the total number of time steps executable in our model. To address this problem, BoS designs a novel sliding window mechanism that can recurrently apply RNN inference on fixed-length flow segments. Therefore, the number of RNN time steps executable on a flow is no longer limited by switch stages. Specifically, in online traffic analysis, BoS uses a window with fixed-size S to extract a segment of S packets from the flow, executes S RNN time steps on this segment to obtain an intermediate inference result r_i, and swifts the window by one packet to obtain a new segment, and repeats the process. Therefore, BoS can continuously execute RNN time steps as the flow proceeds.

The key to fulfill the sliding window design is to properly aggregate these intermediate results. Specifically, upon receiving the jth packet, suppose that the binary RNN has processed g full segments. Then the inference result for the jth packet shall consider all the g intermediate inference results. In the case of multi-class traffic classification, one simple strategy is to select the majority class from these intermediate results. More crucially, we can co-design the aggregation algorithm with an off-switch module to improve the overall traffic analysis accuracy, as described below.

4.4 Analysis Escalation

Although BoS primarily relies on binary RNN to ensure line-speed traffic analysis, we still want to embrace full-precision and more advanced models (*e.g.,* transformers) to handle corner cases. For instance, in multi-class traffic classification, it is possible that none of the classes dominates (*e.g.,* the numbers of flow segments for different classes are close to each other). In this case, adopting the majority voting policy may reduce classification confidence.

To this end, BoS adopts an off-switch traffic analysis module co-located with the programmable data plane. We recognize two challenges in accommodating this analysis module. First, the aggregation algorithm must be carefully designed to ensure that it can accurately capture ambiguity, while avoiding consistently escalating flows (*i.e.,* only a small fraction of flows should be escalated to the co-located analysis module in order to preserve line-speed analysis for the vast majority of traffic). Second, the analysis module adopts a transformer-based model to improve accuracy. However, due to the complex computations required for inference, it is non-trivial to scale the online analysis throughput to maintain the high speed of network forwarding.

The Escalation Mechanism. To address the first challenge, we measure the *classification confidence* in the aggregation algorithm. Specifically, for each extracted packet segment in a flow, the binary RNN predicts an intermediate inference result, which is a vector of probabilities, one for each class. Suppose that upon receiving the jth packet of the flow, the binary RNN has processed g packet segments for the flow (*i.e.,* the arrival of the jth packet will form the $\{g+1\}^{\text{th}}$ segment). Then, our algorithm aggregates all $g+1$ intermediate inference results by accumulating the per-class prediction probabilities. The class with the largest cumulative probability is selected as the inference result for the jth packet.

Whether a flow should be escalated is determined by the number of *ambiguous packets* in the flow. Upon receiving the jth packet, suppose the largest cumulative probability among all classes is CPR_m and the total number of intermediate results is wincnt, then the classification confidence for the jth packet is quantified as $\mathsf{CPR}_m/\mathsf{wincnt}$. The packet is considered ambiguous if its confidence is below a predefined *confidence threshold*. We use $\mathbb{T}_{\mathsf{conf}}$ to represent the vector of confidence thresholds, one for each class. The flow is escalated when the number of ambiguous packets in the flow exceeds a predefined *escalation threshold* T_{esc}.

$\mathbb{T}_{\mathsf{conf}}$ and T_{esc} are learned based on the distributions of the classification confidences of the training samples. Consider the example in Figure 4. For the VoIP class in the ISCXVPN2016 dataset (see detailed descriptions in § 7.1), we plot the CDF of the confidence scores for both correctly classified packets and misclassified packets. The confidence scores are quantized because they are eventually computed on the data plane (see § 5.2). An appropriate $\mathbb{T}_{\mathsf{conf}}$ should escalate as many misclassified packets as possible without affecting correctly classified packets. To this end, we design the following loss function to train our binary RNN.

Suppose p_i is the RNN's prediction probability for class i and y is the ground-truth class. The classic cross entropy (CE) loss is $\text{CE} = -\log(p_y)$. The CE loss solely considers to improve the model's ability to predict the correct class. Our loss is defined as $\mathcal{L}_1 = -(1-p_y)^\gamma \log(p_y) - \lambda \sum_{i \neq y} p_i^\gamma \log(1-p_i)$, which adds another item to explicitly negate the model's prediction on the non-ground-truth classes. The factor λ balances the two items, while the modulating factors $(1-p_y)^\gamma$ and p_i^γ down-weight easy samples and focus on hard samples, as proposed in the Focal Loss [27]. Intuitively, this loss enhances the confidence differences between misclassified and correctly classified packets by reducing $p_i (i \neq y)$ while retaining high p_y. Since our aggregation algorithm chooses the class with the largest accumulative probability, a simplified version of the above loss function is to only reduce the maximum prediction probability among all the non-ground-truth classes, *i.e.,* $\mathcal{L}_2 = -(1-p_y)^\gamma \log(p_y) - \lambda p_{\mathsf{false}}^\gamma \log(1-p_{\mathsf{false}})$, where p_{false} is the largest p_i among all the non-ground-truth classes. We thoroughly evaluate these loss functions in § 7.3.

Once $\mathbb{T}_{\mathsf{conf}}$ is determined, we set T_{esc} to control the amount

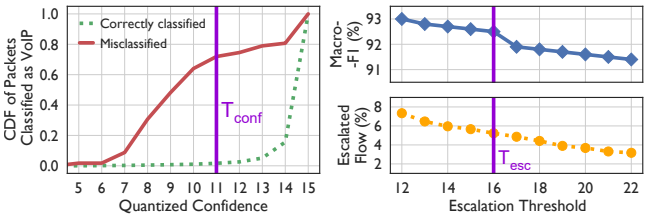


Figure 4: The selection of T_{conf} and T_{esc}.

of escalated flows. As shown in Figure 4, we select a T_{esc} to ensure that no more than 5% flows are escalated to the co-located analysis module for further analysis.

Integrated Model Inference System. To address the second challenge, we design an IMIS that enables fast online inference for a full-precision transformer-based model. As illustrated by Figure 1, IMIS orchestrates four types of stateful and single-threaded tasks (called *engines*) to realize a *non-blocking* traffic processing pipeline. We describe the transformer model training and the architecture of IMIS in § 6.

4.5 Integrated Analysis Logic

Algorithm 1 summarizes the complete logic of our online traffic analysis in BoS. Because our binary RNN model leverages flow-level data for inference, BoS designs a dedicated flow manager to store per-flow state. When a packet $\mathcal{P}$ is received, the flow manager first checks if per-flow state for $\mathcal{P}$ has already been allocated. If not, the flow manager allocates new per-flow storage for the packet. The packet then enters the normal flow-aware inference pipeline based on the retrieved flow state. Due to the limited capacity of the switch, when the flow manager cannot allocate storage for a new flow, BoS falls back to analyzing the packets of that flow using a tree model trained only using per-packet features (*e.g.,* packet length, TTL, Type of Service, TCP offset). This tree model is deployed on the data plane alongside our binary RNN model. The detailed design of the flow manager is deferred to § A.1.4.

We elaborate on one key design that has not been thoroughly discussed yet. In line 24, we periodically reset the window counter and per-class results every $\mathcal{K}$ packets. This effectively clears the contributions of very ancient flow segments (*i.e.,* more than $\mathcal{K}+1$ packets apart) when aggregating the intermediate inference results. This design rationale is that if we obtain a sub-flow f_{sub} by extracting a continuous and sufficiently long sequences of packets (starting from any position) from a flow f, it is very likely that f_{sub} and f are classified as the same class. Thus, clearing the results of very remote segments will not affect traffic analysis results. On the contrary, without periodical reset, the per-class results CPR would be consistently accumulated. To prevent buffer overflow, we need to allocate more bits to store CPR, which, unfortunately, results in significant hardware resource consumption (see § 5.2). Note that the periodical reset does not clear the embedding vectors for the previous $S-1$ packets.

Algorithm 1: Integrated Traffic Analysis Logic

Define : WIN$[0\ldots S-1]$ sliding window; N No. of classes; CPR$[0\ldots N-1]$ per-class results; $\mathbb{T}_{conf}[0\ldots N-1]$ the per-class confidence threshold; T_{esc} the escalation threshold

1 **if** FlowManager(packet $\mathcal{P}$) *fails* **then**
2 Fall back to use the per-packet model, and exit
3 Retrieve the flow state for $\mathcal{P}$
4 **if** $\mathcal{P}$ *is matched by the* EscTable **then**
5 Forward $\mathcal{P}$ to IMIS, and exit ▷ Escalated flows
6 pktcnt $\leftarrow$ pktcnt $+1$ ▷ Count packets
7 $ev \leftarrow$ **FeatureEmbedding**($\mathcal{P}$.length, $\mathcal{P}$.IPD)
8 WIN[pktcnt % S] $\leftarrow ev$ ▷ Slide the window
9 **if** pktcnt $< S$ **then** ▷ The first $S-1$ packets
10 Pre-analysis packet handling ▷ See § A.1.6
11 **else**
12 $h \leftarrow \vec{0}$
13 **for** $i \leftarrow 1$ **to** S **do** ▷ RNN time steps
14 $ev_i \leftarrow$ WIN[(pktcnt $+i$) % S]
15 $h \leftarrow$ **RNNCell**(ev_i, h)
16 $PR \leftarrow$ **OutputLayer**(h) ▷ Intermediate result: a probability vector
17 CPR $\leftarrow$ CPR $+ PR$
18 Class $\leftarrow$ argmax(CPR) ▷ Measure confidence
19 wincnt $\leftarrow$ wincnt $+1$ ▷ No. of windows
20 **if** CPR[Class] $< \mathbb{T}_{conf}$[Class] $*$ wincnt **then**
21 esccnt $\leftarrow$ esccnt $+1$ ▷ Ambiguous packets
22 **if** esccnt $\geq T_{esc}$ **then**
23 Initiate escalated analysis for subsequent packets
24 **if** pktcnt % $\mathcal{K} = 0$ **then** Reset(wincnt, CPR)

5 Model Realization on the Data Plane

5.1 Embedding Vector Storage and Retrieval

As shown in Figure 3(c), the RNN cell takes the embedding vectors of the packets in each sliding window (with length S) as input. Consider the flow segment starting with the kth packet and ending with the $\{k+S-1\}^{\text{th}}$ packet. Upon arrival of the $\{k+S-1\}^{\text{th}}$ packet, the current flow segment is full and is ready to execute all S RNN steps simultaneously, *i.e.,* using the embedding vector of each packet as the key to obtain the matched output in each GRU table (S tables in total). Thus, before a segment is full, we need to temporally hold the embedding vectors for prior $S-1$ packets. We design a ring buffer with $S-1$ bins (registers) to store embedding vectors. In particular, the kth packet in the segment is stored in the $\{k \,\%\, (S-1)+1\}^{\text{th}}$ bin (indexed from 1) of the ring. The embedding vector of the $\{k+S-1\}^{\text{th}}$ packet eventually takes the bin occupied by the kth packet, which will be out-of-scope upon the arrival of the $\{k+S\}^{\text{th}}$ packet. The second benefit of the ring buffer design is that all bins are mutually independent, so that they can be accessed in parallel.

The efficient storage of embedding vectors leads to a chal-

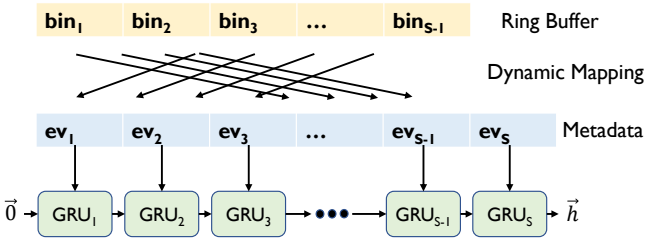


Figure 5: Storage and retrieval of embedding vectors.

lenge in reading these vectors. Specifically, given a flow segment, the embedding vector of the first packet in this segment is not always stored in the first bin of the ring buffer. Therefore, it is incorrect to statically align the ring buffer with GRU tables, *i.e.,* using the value stored in the kth bin as the input key of the kth GRU table. Instead, the input of the ith GRU (for $i \in [1, S-1]$) should be read from the $\{(k-S+i)\%(S-1)+1\}^{\text{th}}$ bin. However, when declaring a lookup table for GRU on switch, the storage locations of its keys must be static and predetermined. Thus, to realize the above dynamic mapping, we need to first read values from the ring buffer to several intermediate variables (called metadata), and then dispatch the metadata to the proper GRU tables, as shown in Figure 5.

5.2 Intermediate Results Aggregation

As described in § 4.4, the key operation in our RNN inference is to select the largest cumulative probability from all intermediate inference results, *i.e.,* executing an argmax operation.

Ternary-Matching Based Design. Argmax is not a primitive available on the switch. We realize argmax based on an efficient data plane design of number comparison. Intuitively, number comparison can be accomplished by either conditional statements or exact-matching table matching. Neither of them, however, is scalable (see § A.1.1).

In BoS, we propose a scalable ternary-matching based design. Suppose argmax compares n numbers each with m bits. The key of a table entry consists n segments, each with m ternary bits (*i.e.,* 0, 1, or $*$). The value represents the winner (*i.e.,* the largest number). Starting from the most significant bit (MSB), to generate the lth bit for each key segment, there are 2^n possible cases. Consider a case $\mathcal{C}_{(l,k)}$ where the lth bits of the first k ($k \in [1, n\text{-}1]$) segments are 1 and the lth bits of the remaining segments are 0. Clearly, the segments whose l^{th} bits are 0 will not be the winner, so that we can stop further enumerating the lower bits (*i.e.,* the $\{l+1\}^{\text{th}}, \{l+2\}^{\text{th}}, \ldots, m^{\text{th}}$ bits) for these segments. Thus, among all $2^{(m-l)\cdot n}$ sub-cases of the case $\mathcal{C}_{(l,k)}$, we do not need further enumerations for $2^{(m-l)\cdot k}$ of them, achieving a $2^{(m-l)\cdot(n-k)}$ reduction ratio. Take $\mathcal{C}_{(1,1)}$ as an example: it represents the case where the first (*i.e.,* the most significant bit, MSB) of the first segment is 1 and the MSBs for other remaining segments are 0. Thus, all $2^{(m-1)n}$ sub-cases for $\mathcal{C}_{(1,1)}$ have clear winners and are collapsed into one key (*e.g.,* $1***, 0***, 0***$ if n=3, m=4).

```
Inputs:
    n: number of keys; m: bit width of each key
1:  Procedure Generate(n, m):
2:      T = {1...n}  ▷ initial input, all the numbers can be the winner
3:      ▷ For explanation purpose, entry is represented as a 2-D array
4:      entry = array[1..n][1..m]  ▷ array of ternary bits (0, 1, *)
5:      Work(T, 1)
6:  Procedure Work(S, L):  ▷ S: possible winners in this iteration
7:      for num ∈ T\S ⇒ entry[num][L] = '*'  ▷ cases cannot win
8:      if L = m ⇒ Output(S), return  ▷ base case: last bit
9:      for φ ⊂ S' ⊂ S:  ▷ cases similar to C_(L,|S'|)
10:         for num ∈ S\S' ⇒ entry[num][L] = '0'
11:         for num ∈ S' ⇒ entry[num][L] = '1'
12:         Work(S', L+1)  ▷ Recursive resolve
13:     for num ∈ S ⇒ entry[num][L] = '*'  ▷ case C_(L,0) & C_(L,|S|)
14:     Work(S, L+1)  ▷ Recursive resolve
15: Procedure Output(S):  ▷ S: possible winners in this iteration
16:     a = list(S) in INCREASING order  ▷ indexed from 1
17:     for i = len(a) downto 2:  ▷ winning case for a[i≥2]
18:         for k ∈ [1..i-1] ⇒ entry[a[k]][m] = '0'
19:         entry[a[i]][m] = '1'
20:         for k ∈ [i+1..len(a)] ⇒ entry[a[k]][m] = '*'
21:         install(entry, winner=a[i])
22:     for num ∈ S ⇒ entry[num][m] = '*'  ▷ winning case for a[1]
23:     install(entry, winner=a[1])
```

Figure 6: The procedure to generate a ternary-matching table to realize argmax on the data plane.

Based on the above protocol, we derive the number of required table entries $F(n,m)$ as (see details in § A.1.2).

$$F(n,m) = 2 * F(n,m-1) + \sum_{i=1}^{n-1} \binom{n}{i} F(i,m-1) \quad \text{for} \quad n,m \geq 2 \qquad (1)$$
$$F(n,1) = 2^n \text{ for } n \geq 2; \quad F(1,m) = 1 \text{ for } m \geq 1.$$

Further Optimizations. We make two subsequent optimizations to further reduce $F(n,m)$. First, the two special cases $\mathcal{C}_{(l,n)}$ (*i.e.,* the lth bits in all n segments are 1) and $\mathcal{C}_{(l,0)}$ (*i.e.,* the lth bits in all n segments are 0) can be further merged. Specifically, for all 2^{m-l} sub-cases of $\mathcal{C}_{(l,n)}$, their winners remain the same if we modify the lth bits of all n segments to 0; and similarly for 2^{m-l} sub-cases of $\mathcal{C}_{(l,0)}$, their winners remain the same if we modify the lth bits of all the n segments to 1. Thus, $\mathcal{C}_{(l,0)}$ and $\mathcal{C}_{(l,n)}$ can be merged by modifying the lth bit as an wildcard asterisk in each segment. We handle this merged case lastly in the current enumeration of the lth bit (see lines 13 and 14 in Figure 6), so that these wildcard asterisks will not interfere with previous cases with higher priority (see lines 9 to 12 in Figure 6). With this optimization, $F(n,m)$ is reduced as $F(n,m) = F(n,m-1) + \sum_{i=1}^{n-1} \binom{n}{i} F(i,m-1)$.

The second optimization is reducing the base case $F(n,1)$. By reversely encoding the one-bit number comparison (see Figure 7), $F(n,1)$ is reduced to n from 2^n. Combining both optimizations, we obtain $F(n,m) = nm^{n-1}$.

segment$_1$	segment$_2$	···	segment$_{n-1}$	segment$_n$	Winner
x0	x0	···	x0	x1	n^{th} number
x0	x0	···	x1	x*	{n-1}th number
...	...	...	...	...	...
x0	x1	···	x*	x*	2nd number
x*	x*	···	x*	x*	1st number

Figure 7: The reverse encoding for $F(n,1)$.

6 Implementation

Our BoS prototype[1] includes: 1500 lines of Python code for model training, 1900 lines of P4 code for on-switch RNN, and 3300 lines of C++ code for IMIS. To evaluate the prototype, an additional 1600 lines of code are developed.

Model Training. We train a binary RNN to analyze flow segments extracted by the sliding window. Given the window size S and a flow sample $(\mathcal{P}_1, \mathcal{P}_2, \dots)$ in the training dataset, we slice this flow into all possible packets segments (*e.g.,* consecutive S packets like $(\mathcal{P}_1, \dots, \mathcal{P}_S)$ and $(\mathcal{P}_2, \dots, \mathcal{P}_{S+1})$) where the label of each segment is the flow label. For each segment, we use its packet length sequence and IPD sequence as inputs, and train the binary RNN to predict its label correctly. Recall that our binary RNN outputs vector of probabilities, one for each class. The training process is to maximize the prediction probability on the ground-truth class.

We use YaTC [66], a recent masked autoencoder (MAE) [16] based traffic transformer with multi-level flow representation, in IMIS to analyze escalated flows. YaTC only uses the first 5 packets of a flow for analysis. For each packet, it extracts the first 80 header bytes and 240 payload bytes as inputs. We first determine the two thresholds in § 4.4 to collect the escalated flows in the training set, and then fine-tune the pre-trained YaTC model [66] to obtain our final model.

On-Switch RNN Implementation. We implement a prototype on our Tofino 1 programmable switch. The top part of Figure 8 shows the workflow of all the components in our prototype. The left-bottom table in Figure 8 lists the hyper-parameters of our prototype, and the right-bottom table lists the detailed per-stage arrangement of our components. Due to space constraints, we defer the detailed description of our prototype to § A.2.1. Although the hardware resources on the Tofino 1 are very limited (*e.g.,* only 12 stages), we manage to implement a prototype that supports all four traffic analysis tasks evaluated in § 7.1. The on-switch RNN is programmable in runtime via the control plane (see § A.3).

IMIS Implementation. The core design of IMIS is a non-blocking traffic processing pipeline. As illustrated in Figure 1, architecturally, IMIS is designed around stateful, single-threaded tasks, which we call *engines*. The parser engine uses DPDK [18] (version 20.11) APIs to consistently collect the packet bytes from the escalated traffic; the pool engine takes the stream data as input and organizes it into per-flow state; the analyzer engine calls the pool engine to collect a batch of fresh per-flow data, and uses CUDA (version 11.7) [38] to interact with an auxiliary GPU to accelerate model inference; and the buffer engine stops packets without inference results to wait in memory, and sends those who have inference results to NIC. The pool engine is the key to dynamically coordinate the speeds of the parser engine and analyzer engine, thus achieving a non-blocking packet processing pipeline. The detailed architecture of the IMIS system is deferred to § A.2.2.

[1] Available at https://github.com/InspiringGroup-Lab/Brain-on-Switch

7 Evaluation

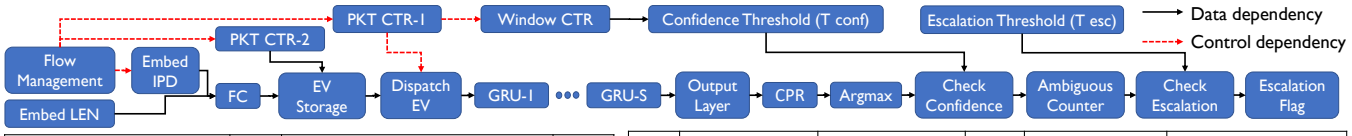
7.1 Experiment Setup

Testbed Setup. We deploy our binary RNN model using one pipe of a Barefoot Tofino 1 programmable switch. One server generates network traffic to an inbound port of the switch based on the pcap files we created for various traffic analysis tasks and traffic loads. Each flow is either analyzed by the on-switch RNN or redirected from one specific switch port to an off-switch server that deploys IMIS. For the flows analyzed on-switch, we develop a dedicated on-switch module to collect their analysis statistics online. Scaling the on-switch analysis of BoS beyond a single pipe of the switch is feasible given proper flow management. We discuss this in § A.3.

Tasks. We evaluate BoS using the following four tasks. (i) Encrypted traffic classification on VPN: this task classifies network traffic encrypted by VPNs. We use the ISCXVPN2016 [12] dataset, a six-class classification task (Email, Chat, Streaming, FTP, VoIP, P2P). (ii) Botnet traffic classification: this task classifies botnet traffic collected from the IoT systems. We use the BOTIOT [22] dataset, a four-class classification task (Data Exfiltration, Key Logging, OS Scan, Service Scan). (iii) Behavioral analysis of IoT devices: this task classifies traffic generated by IoT devices in different working states. We use the CICIOT2022 [10] dataset, a three-class classification task (Power, Idle, Interact). (iv) P2P application fingerprinting: this task classifies network traffic generated by P2P applications. We use the PeerRush dataset [40], a three-class classification task (eMule, uTorrent, and Vuze). We supplement additional details regarding the processing of these datasets in § A.4.

Network Load. We would like to evaluate BoS under different network loads. Similar to prior art [51, 61, 71], we use the number of new flows arrived in each second to represent the network load. Specifically, for each testing flow in a task, we extract its raw packets from the pcap file while preserving the inter-packet delays. Given the total number of flows in this task, and a desired network load, we calculate the total time period required to replay these flows, and then uniformly release these flows within this period. If the period is too short, we replay these flows multiple times in a loop to create consistent loads throughout our test.

The actual network load varies in different deployment. We make load estimations based on prior measurements. In 2015, Meta [42] reported that its external-facing web server generates 500 new flows per second (median). Meanwhile,



hyper-parameter	value	hyper-parameter	value
Window Size (S)	8	Reset Period of Window Counter (K)	128
No. of Classes (N)	6	Bit Width of Intermediate Probability	4
Bit Width of Embedded LEN	10	Bit Width of Cumulative Probability	11
Bit Width of Embedded IPD	8	Bit Width of ID	32
Bit Width of Embedding Vector	6	Bit Width of TS	32
Bit Width of Hidden State	9	Flow Capacity	65536

stage	ingress	egress	stage	ingress	egress
0	calculate ID, idx embed pkt length	GRU-5 window_counter	5	FC escalation_flag	CPR-4,5,6 u ← argmax(CPR-1,2,3)
1	FlowInfo	GRU-6	6	bin-4,5,6,7	v ← argmax(CPR-4,5,6)
2	last_TS pkt_counter-1,2	GRU-7 calculate threshold	7	bin-1,2,3	argmax(u,v)
			8	dispatch ev	ambiguous_counter
3	calculate IPD	Output ◦ GRU-8	9	GRU-2 ◦ GRU-1	set mirror
4	embed IPD	CPR-1,2,3	10~11	GRU-3, GRU-4	

Figure 8: The breakdown of our on-switch RNN implementation on a Tofino programmable switch.

Table 2: Experimental settings.

Datasets (Tasks)	ISCXVPN 2016	BOT IOT	CICIOT 2022	Peer Rush
Training Flows	7801	7835	5332	30770
Testing Flows	1951	1961	1335	7694
Classes	6	4	3	3
Class Ratio⋆	2:6:1:5:9:3	1:1:4:19	1:4:1	2:1:1
Best Loss	$\mathcal{L}_1$	$\mathcal{L}_1$	$\mathcal{L}_2$	$\mathcal{L}_1$
λ, γ	0.8, 0	0.5, 0.5	3, 1	1, 0
Optimizer	AdamW	AdamW	AdamW	AdamW
Learning Rate	0.01	0.005	0.005	0.005
RNN Hidden States†	9 bits	8 bits	6 bits	5 bits
Per-packet Model Acc.	0.596	0.327	0.759	0.684
Network Load	Low	Normal	High	Scaling
No. of flows / s	1000	2000	4000	up to 7.8M

⋆ See § A.4 for the accurate numbers of flows in each class.
† We evaluate BoS under different binary RNN model sizes in § A.6.

CISCO [9] measures that Internet traffic grows 3-fold from 2016 to 2021. Combining these measurements, we estimate that 2000 new flows per second are a reasonable network load that BoS may face in practice. In our scaling test (see § 7.3), we stress test BoS with up to 450,000 new flows per second on our testbed (a 225x increase from the normal load, and 30-300x over NetBeacon [71]).

Metrics. We use packet-level macro-F1 (the average of F1-score for different classes) as the accuracy metric, and further report the breakdown of the Precision / Recall of each class.

7.2 End-to-end Performance

In this section, we report the end-to-end performance of BoS for different tasks. The main experimental settings are summarized in Table 2. In § 7.3, we evaluate BoS under a variety of settings. We also compare BoS with two recent art NetBeacon [71] and N3IC [51]. N3IC deploys the binary MLP on a SmartNIC. For fair comparison, we simulate the switch-side traffic management logic and the binary MLP inference in software to obtain the traffic analysis results for N3IC. The detailed descriptions about the reproduced versions of the two art are given in § A.5.

Accuracy. We summarize the analysis accuracy results in Table 3. Across all evaluated tasks, BoS achieves significantly better performance than NetBeacon and N3IC, with an average F1-score improvement of 0.13 and 0.31, respectively. On more challenging tasks with more classification classes, the improvement is even greater, up to 0.19 and 0.42, respectively. We observe the binary MLP performs the worst because the accuracy loss caused by binarizing all model weights is significant. In fact, on the ISCXVPN2016 and CICIOT2022 datasets, the F1-scores of N3IC are even lower than these of our fallback tree-based model (0.596/0.759). Constrained by the availability of flow features, NetBeacon can only execute model inference at discrete locations. Thus, an inference error affects all its subsequent packets until it is corrected by the next inference point. This fundamentally limits its F1-scores, especially for more difficult tasks. In contrast, BoS retains full-precision model weights in the on-switch RNN model and continuously produces fresh inference results as a flow proceeds. Together with the co-located IMIS, BoS produces more accurate analysis results than existing arts, achieving over 0.920 F1-score in all tasks. We observe very minor declines of F1-scores in BoS as the network load increases, demonstrating the effectiveness of our flow management (note that we use the same flow management module for other two systems as well). The minor accuracy loss is because a small fraction of flows (*e.g.,* 2.77%/1.43%/2.05%/5.22% in the normal network load case) fall back to using the per-packet model.

Hardware Resource Utilization. We report the stateful SRAM and stateless SRAM/TCAM usage by BoS on the programmable switch in Table 4. The stateful SRAMs are consumed to maintain per-flow states, which mainly consist of the flow management information (*e.g.,* TrueID and timestamp, see § A.1.4), the embedding vectors (EV) for binary RNN inference, and the cumulative probability counter (CPR) for each class. In our prototype, the hardware consumption for the first two parts is task-irrelevant, and one task uses roughly 8.85% of SRAM. The consumption for the last part depends on the number of classification classes in a task, and the four tasks use roughly 5.63%/3.75%/2.81%/2.81% of SRAM, respectively. The embedding vectors stored for each flow take $8 \times (S-1) + 8$ bits (64 bits in our prototype). Compared with existing approaches [51, 71] that require online feature com-

Table 3: Analysis accuracy for BoS and other two closely related art.

Methods	BoS			NetBeacon [71] (Tree-based Models)			N3IC [51] (Binary MLP)		
Network Load	Low	Normal	High	Low	Normal	High	Low	Normal	High
Encrypted Traffic Classification on VPN (ISCXVPN2016)									
Email	0.935 / 0.933	0.936 / 0.925	0.933 / 0.923	0.309 / 0.514	0.315 / 0.524	0.320 / 0.525	0.347 / 0.326	0.354 / 0.339	0.367 / 0.350
Chat	0.903 / 0.818	0.902 / 0.818	0.901 / 0.814	0.739 / 0.935	0.739 / 0.933	0.742 / 0.925	0.336 / 0.655	0.336 / 0.654	0.342 / 0.656
Streaming	0.926 / 0.941	0.926 / 0.939	0.926 / 0.910	0.963 / 0.919	0.962 / 0.904	0.962 / 0.874	0.741 / 0.608	0.742 / 0.603	0.743 / 0.581
FTP	0.973 / 0.928	0.973 / 0.926	0.973 / 0.922	0.946 / 0.659	0.946 / 0.655	0.947 / 0.654	0.563 / 0.396	0.567 / 0.396	0.575 / 0.397
VoIP	0.968 / 0.958	0.968 / 0.958	0.968 / 0.957	0.938 / 0.882	0.939 / 0.881	0.939 / 0.882	0.883 / 0.783	0.884 / 0.782	0.886 / 0.787
P2P	0.905 / 0.927	0.903 / 0.928	0.876 / 0.930	0.810 / 0.959	0.798 / 0.959	0.778 / 0.960	0.578 / 0.739	0.577 / 0.742	0.565 / 0.748
Macro-F1	0.926	0.925	0.919	0.786	0.784	0.780	0.565	0.567	0.568
Botnet Traffic Classification on IoT (BOTIOT)									
Data Exfiltration	0.964 / 0.974	0.951 / 0.973	0.899 / 0.971	0.691 / 0.845	0.684 / 0.847	0.658 / 0.848	0.514 / 0.879	0.508 / 0.881	0.506 / 0.879
Key Logging	0.960 / 0.946	0.961 / 0.962	0.959 / 0.902	0.921 / 0.425	0.921 / 0.419	0.918 / 0.399	0.055 / 0.033	0.058 / 0.033	0.052 / 0.031
OS Scan	0.996 / 0.996	0.995 / 0.989	0.995 / 0.966	0.838 / 0.963	0.841 / 0.963	0.844 / 0.945	0.831 / 0.693	0.830 / 0.677	0.831 / 0.672
Service Scan	0.993 / 0.992	0.986 / 0.973	0.979 / 0.978	0.928 / 0.876	0.927 / 0.870	0.917 / 0.858	0.845 / 0.663	0.830 / 0.664	0.840 / 0.663
Macro-F1	0.978	0.974	0.955	0.785	0.782	0.769	0.547	0.542	0.541
Behavioral Analysis of IoT Devices (CICIOT2022)									
Power	0.926 / 0.887	0.924 / 0.882	0.921 / 0.882	0.819 / 0.726	0.820 / 0.724	0.817 / 0.724	0.639 / 0.750	0.640 / 0.750	0.640 / 0.748
Idle	0.922 / 0.943	0.921 / 0.942	0.918 / 0.941	0.810 / 0.938	0.808 / 0.938	0.806 / 0.936	0.618 / 0.640	0.620 / 0.642	0.622 / 0.646
Interact	0.934 / 0.946	0.934 / 0.948	0.934 / 0.943	0.871 / 0.786	0.873 / 0.786	0.872 / 0.784	0.651 / 0.504	0.655 / 0.506	0.661 / 0.510
Macro-F1	0.926	0.925	0.923	0.822	0.821	0.820	0.629	0.631	0.633
P2P Application Fingerprinting (PeerRush)									
eMule	0.943 / 0.949	0.918 / 0.949	0.898 / 0.950	0.846 / 0.954	0.821 / 0.955	0.805 / 0.954	0.734 / 0.866	0.730 / 0.867	0.723 / 0.875
uTorrent	0.949 / 0.924	0.950 / 0.912	0.941 / 0.894	0.882 / 0.870	0.885 / 0.858	0.885 / 0.831	0.734 / 0.789	0.735 / 0.790	0.738 / 0.783
Vuze	0.946 / 0.962	0.945 / 0.947	0.941 / 0.930	0.910 / 0.810	0.907 / 0.790	0.904 / 0.793	0.821 / 0.626	0.826 / 0.622	0.826 / 0.616
Macro-F1	0.945	0.937	0.925	0.877	0.866	0.858	0.755	0.755	0.752

Table 4: Hardware resource utilization.

	Datasets (Tasks)	ISCXVPN 2016	BOT IOT	CICIOT 2022	Peer Rush
SRAM	Flow Info. (stateful)	5.21%	5.21%	5.21%	5.21%
	EV (stateful)	3.65%	3.65%	3.65%	3.65%
	CPR (stateful)	5.63%	3.75%	2.81%	2.81%
	FE (stateless)	2.19%	2.19%	2.19%	2.19%
	GRU (stateless)	3.02%	1.56%	0.73%	0.73%
	Total⋆	23.44%	20.10%	18.33%	18.33%
TCAM	Argmax (Total)	1.74%	1.04%	0.69%	0.69%

⋆ Including other components not listed, *e.g.,* packet counters for each flow.

putation, their per-flow storage consumption depends on the used flow features. For instance, NetBeacon [71] engineers 7 important features for the P2P application fingerprinting task, which consumes roughly 150 bits.

The stateless SRAM is used to implement the lookup tables for feature embedding (FE) and GRU layers in our binary RNN. Specifically, the SRAM consumption of GRU layers depends on the number of bits used for storing RNN hidden states. Using the default bitwidth in Table 2, the four tasks use roughly 3.02%/1.56%/0.73%/0.73% of SRAM, respectively, for GRU layers. Additionally, each task uses 2.19% of SRAM for feature embedding. In total, the four tasks use 23.44%/20.10%/18.33%/18.33% of SRAM, respectively.

BoS uses TCAM to implement the argmax operation. Compared with NetBeacon [71], BoS consumes SRAM of similar size and 20x less TCAM (note that the ternary matching in TCAM is 6x more expensive than exact matching with SRAM, in terms of required silicon resources [51]).

7.3 BoS Deep Dive

Analysis Escalation. In this segment, we study the trade-off between the amount of escalated flows and the overall macro-F1, demonstrating that our loss functions defined in § 4.4 achieve a better trade-off than the classic cross entropy loss. As described in § 4.4, the escalation threshold T_{esc} controls the amount of escalated flows. Using the setting in Table 2, we train the binary RNN with our losses and cross entropy loss, respectively, and measure the overall macro-F1 with different amount of escalated flows under the normal network load (2000 flows/s). The results are plotted in Figure 9, and the best parameters (λ, γ) of our losses in each task are presented. We make the following key observations. (i) Regardless of the used loss functions, the overall macro-F1 scores for all tasks improve as the percentage of escalated flows increases from 0% to 5%. This demonstrates the necessity for accommodating the off-switch analysis model to compensate for on-switch analysis. (ii) For the same amount of escalated flows, our losses outperform the cross entropy loss by significant margins across all tasks. This shows that our loss designs can more effectively identify the ambiguous packets that require additional reevaluation. This is crucial to improve the overall system performance without consistently escalating flows. (iii) The performance of our two losses $\mathcal{L}_1$ and $\mathcal{L}_2$ is task-dependent. In general, $\mathcal{L}_1$ outperforms $\mathcal{L}_2$ in three tasks, yet $\mathcal{L}_2$ requires less training epochs to converge.

System Performance of IMIS. In this segment, we stress test the performance of off-switch IMIS upon a burst of concurrent

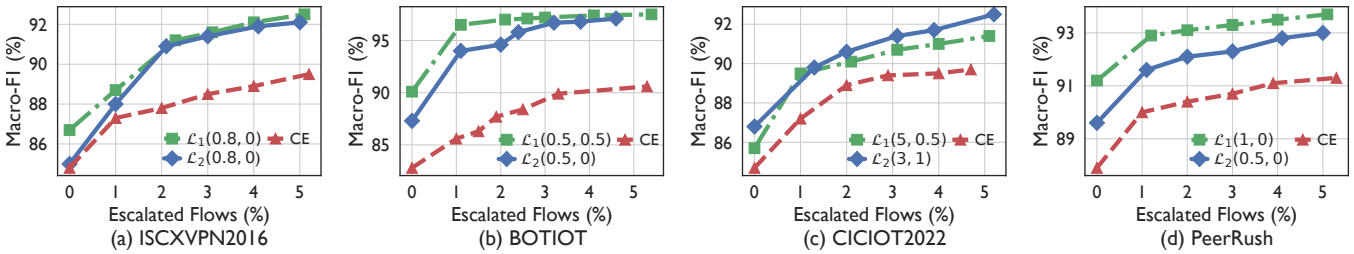


Figure 9: [Testbed] The trade-off between percentage of escalated flows and the overall accuracy.

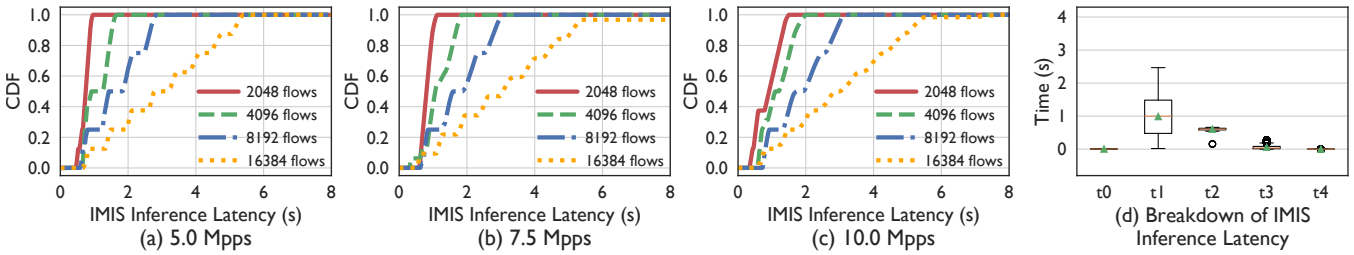


Figure 10: [Testbed] The inference throughput and latency of the off-switch IMIS.

flows. We run the IMIS with 8 parallel analysis modules. We evaluate four different levels of flow concurrency (2048, 4096, 8192, and 16384 flows) with three different aggregate inbound rates (5.0, 7.5 and 10.0 million packets per second). The complete inference pipeline for a packet $\mathcal{P}$ in IMIS has six phases: (1) $\mathcal{P}$ is fetched from the NIC by the parser engine; (2) its metadata is organized by the pool engine; (3) its metadata is sent to the analyzer engine; (4) the analyzer engine generates the inference result; (5) the result is collected by the buffer engine; (6) $\mathcal{P}$ is dispatched to NIC by the buffer engine.

The transformer model in IMIS performs inference on the first five packets of each flow. Given that the average length of the escalated flows in each tasks is 801, 255, 167, and 138 packets, respectively, the vast majority of packets in these escalated flows are directly forwarded to the buffer engine after being collected from the NIC, experiencing very minor latency (less than 1ms). In the following, we only consider the latency for the packets that traverse the entire inference pipeline. The CDFs of the end-to-end latencies are plotted in Figures 10(a) to (c). When the number of concurrent flows is below 4096, the maximum end-to-end latency imposed by IMIS is less than 2 seconds even for 10.0 Mpps inbound rate (equivalently 41 Gbps as the packet sizes we send are 512 B). Considering that BoS typically escalates less than 5% of flows, the flow concurrency levels experienced by the IMIS are expected to be low in most deployments. In Figure 10(d), we further report the breakdown of the end-to-end latency (*i.e.,* the time intervals between two consecutive phases in the inference pipeline) under 8192 concurrent flows and an inbound rate of 5.0 Mpps. We observe that the major latency occurs between the second and third phase, when the packets are waiting to be collected by the analyzer engine. The net inference time spent in the analyzer engine is about 0.6 s.

Scaling Test. We stress test BoS in high-throughput scenarios with high flow concurrency and high flow throughput. Specifically, because all the original network traces are collected in low bandwidth networks (*e.g.,* tens of Mbps), we create high-throughput network traces by concurrently packaging a large number of flows (while ensuring each flow has a unique identifier) and accelerating the packet replay speeds (by reducing the inter-packet delays). Then we replay these pcap files to generate traffic on our testbed. Figure 11 presents the scaling test results, where we progressively increase the flow concurrency to saturate the physical capacity of the NIC on our traffic generator. The results demonstrate that BoS can comfortably handle this level of scale, as the macro-F1 scores remain nearly identical compared to the results in Table 3.

To evaluate BoS at even larger scales, we build a simulator to emulate the entire workflow of BoS. The accuracy of the simulator is validated by replicating the experimental settings of Table 3 and Figure 11. The accuracy results obtained through the simulation are almost the same as those collected from our testbed. We subsequently employ the simulator to explore significantly larger scales, progressively increasing flow concurrency to up to 7.8 million flows per second, and the aggregate throughput to over 1.6 Tbps. The results depicted in Figure 12 reveal a sublinear decline in the macro-F1 scores of BoS, culminating in a $\sim$11.6% reduction at the largest scale.

Fallback Alternative. The default handling of the flows without dedicated per-flow storage is to analyze their packets using a tree-model trained trained only on per-packet features (see § A.1.5). Alternatively, a subset of the flows without dedicated per-flow storage can be forwarded to a new instance of off-switch IMIS dedicated to handling these flows. In Figure 11 and 12, we report the macro-F1 when forwarding a certain percentage of flows without per-flow storage to a dedicated

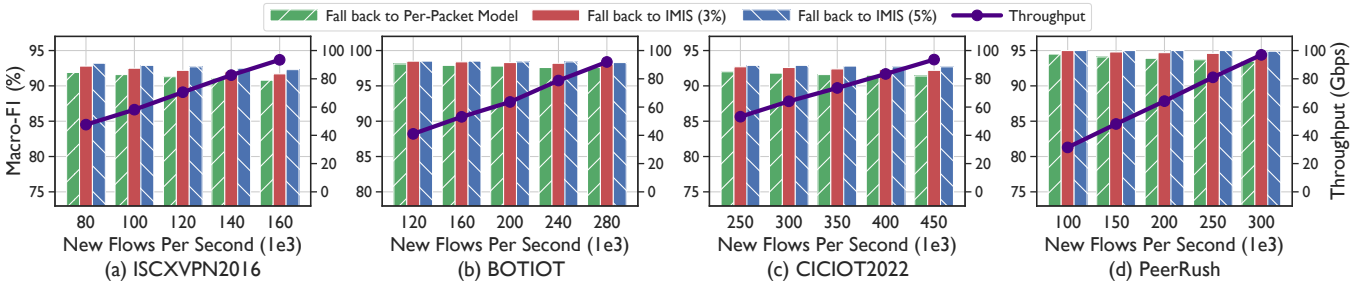


Figure 11: [Testbed] Scaling test of BoS when we progressively increase the aggregate throughput to 100 Gbps.

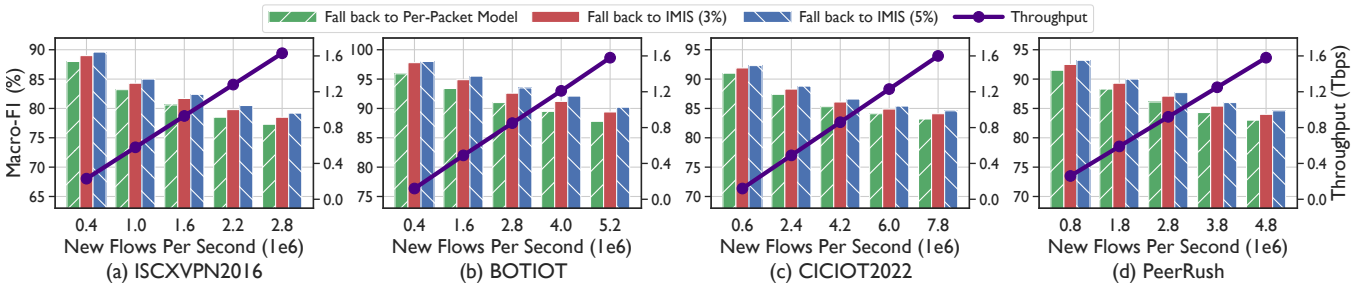


Figure 12: [Simulation] Scaling test of BoS when we progressively increase the aggregate throughput to 1.6 Tbps.

IMIS. When the flow concurrency is high (*i.e.,* Figure 12), this method exhibits reasonable accuracy advantages over falling back to use the per-packet model.

8 Discussion and Related Work

Hardware Dependency. BoS is generic in the sense that all its core designs (*e.g.,* retaining full-precision RNN model weights, using sliding windows to compute unlimited RNN step times) are all realizable using match tables. Since table matching is the universal primitive for any data plane, we expect our designs, with lightweight adaptation, are also deployable on other types of programmable data plane devices.

ML-driven Traffic Analysis. Our community has proposed various ML-powered traffic analysis designs, such as intrusion detection [15, 19, 33], website fingerprinting [11, 41, 46, 65], and encrypted traffic classification [39, 47, 48, 55]. However, it is difficult to directly apply their models in INDP due to the hardware constraints on the data plane.

Advances in the Programmable Data Plane. The flexibility of programmable switches encourages a number of customized applications on the data plane, including network telemetry and monitoring [34, 35, 45], network security [60, 62, 67, 71], and network functions [20]. Additionally, [23, 44] use programmable switches to accelerate ML training, and [25, 53, 70] design auxiliary modules within a switch or leverage off-switch FPGA. Our work focuses on enabling NN-driven INDP using only commodity hardware.

Deployment. BoS is an application-specific system designed for high-throughput and low-latency NN-driven traffic analysis. Therefore, we have not discussed co-deploying other networking functions with BoS on the same programmable switch. Although BoS consumes multiple stages, the SRAM/TCAM consumption per stage is small (see Table 4). Thus, networking functions orthogonal to BoS (*e.g.,* the ECMP in [14]) can be co-deployed with BoS in parallel. Additionally, the latest Tofino chips have almost doubled the number of stages and TCAM/SRAM resources compared to the Tofino 1 chip we use. Thus, we envision that networking functions that may depend on BoS's analysis results (*e.g.,* the traffic policing in [14]) may also be co-deployed with BoS.

9 Conclusion

In this paper, we present BoS, the first INDP design that enables NN-driven traffic analysis at line-speed. The key novelty of BoS is to realize complex RNN computations using a set of novel data plane native operations, and meanwhile to accommodate a transformer-based traffic analysis module via a carefully designed flow escalation mechanism. We implement a prototype of BoS and evaluate it thoroughly on four traffic analysis tasks. The results demonstrate that BoS advances SOTA in both traffic analysis accuracy and scalability.

Acknowledgement

We thank our shepherd Costin Raiciu and the anonymous NSDI reviewers for their insightful feedback. The research is supported in part by the National Key R&D Program of China under Grant 2022YFB2403900, NSFC under Grant 62132011 and Grant 61825204, and Beijing Outstanding Young Scientist Program under Grant BJJWZYJH012019100003011. The corresponding author of this paper is Zhuotao Liu.

References

[1] Soheil Abbasloo, Chen-Yu Yen, and H. Jonathan Chao. Classic Meets Modern: A Pragmatic Learning-Based Congestion Control for the Internet. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2020.

[2] Milad Alizadeh, Javier Fernández-Marqués, Nicholas D. Lane, and Yarin Gal. An Empirical Study of Binary Neural Networks' Optimisation. In *International Conference on Learning Representations (ICLR)*, 2019.

[3] Diogo Barradas, Nuno Santos, and Luís Rodrigues. Effective Detection of Multimedia Protocol Tunneling using Machine Learning. In *USENIX Security Symposium (USENIX Security)*, 2018.

[4] Diogo Barradas, Nuno Santos, Luís Rodrigues, Salvatore Signorello, Fernando MV Ramos, and André Madeira. FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 2014.

[6] Broadcom. Trident 5 Programmable Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78800, accessed on Feb. 2024.

[7] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. pForest: In-Network Inference with Random Forests. *arXiv preprint arXiv:1909.05680*, 2019.

[8] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-decoder for Statistical Machine Translation. *arXiv preprint arXiv:1406.1078*, 2014.

[9] CISCO. Global - 2021 Forecast Highlights. https://www.cisco.com/c/dam/m/en_us/solutions/service-provider/vni-forecast-highlights/pdf/Global_2021_Forecast_Highlights.pdf, accessed on Sep. 2023.

[10] Sajjad Dadkhah, Hassan Mahdikhani, Priscilla Kyei Danso, Alireza Zohourian, Kevin Anh Truong, and Ali A. Ghorbani. Towards the Development of a Realistic Multidimensional IoT Profiling Dataset. In *the 19th Annual International Conference on Privacy, Security & Trust (PST)*, 2022.

[11] Xinhao Deng, Qilei Yin, Zhuotao Liu, Xiyuan Zhao, Qi Li, Mingwei Xu, Ke Xu, and Jianping Wu. Robust Multi-tab Website Fingerprinting Attacks in the Wild. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[12] Gerard Draper-Gil, Arash Habibi Lashkari, Mohammad Saiful Islam Mamun, and Ali A. Ghorbani. Characterization of Encrypted and VPN Traffic using Time-related Features. In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP)*, 2016.

[13] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[14] Open Networking Foundation. Consolidated switch repo (API, SAI and Nettlink). https://github.com/p4lang/switch/tree/master, accessed on Sep. 2023.

[15] Chuanpu Fu, Qi Li, Meng Shen, and Ke Xu. Realtime Robust Malicious Traffic Detection via Frequency Domain Analysis. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[16] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked Autoencoders are Scalable Vision Learners. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[17] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 1997.

[18] Intel. Data Plane Development Kit. http://www.dpdk.org, accessed on Sep. 2023.

[19] Steve TK Jan, Qingying Hao, Tianrui Hu, Jiameng Pu, Sonal Oswal, Gang Wang, and Bimal Viswanath. Throwing Darts in the Dark? Detecting Bots with Limited Data using Neural Data Augmentation. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[20] Changhun Jung, Sian Kim, Rhongho Jang, David Mohaisen, and DaeHun Nyang. A Scalable and Dynamic ACL System for In-Network Defense. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[21] Piotr Jurkiewicz, Grzegorz Rzym, and Piotr Boryło. Flow Length and Size Distributions in Campus Internet Traffic. *arXiv preprint arXiv:1809.03486*, 2018.

[22] Nickolaos Koroniotis, Nour Moustafa, Elena Sitnikova, and Benjamin Turnbull. Towards the development of realistic botnet dataset in the Internet of Things for network forensic analytics: Bot-IoT dataset. *Future Generation Computer Systems*, 2019.

[23] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network Aggregation for Multi-tenant Learning. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[24] Jong-Hyouk Lee and Kamal Singh. SwitchTree: In-Network Computing and Traffic Analyses with Random Forests. *Neural Computing and Applications*, 2020.

[25] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. Accelerating Distributed Reinforcement Learning with In-Switch Computing. In *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[26] Peng Lin, Kejiang Ye, Yishen Hu, Yanying Lin, and Cheng-Zhong Xu. A Novel Multimodal Deep Learning Framework for Encrypted Traffic Classification. *IEEE/ACM Transactions on Networking (TON)*, 2023.

[27] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollar. Focal Loss for Dense Object Detection. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017.

[28] Xinjie Lin, Gang Xiong, Gaopeng Gou, Zhen Li, Junzheng Shi, and Jing Yu. ET-BERT: A Contextualized Datagram Representation with Pre-training Transformers for Encrypted Traffic Classification. In *Proceedings of the ACM Web Conference (WWW)*, 2022.

[29] Chenyi Liu, Mingwei Xu, Yuan Yang, and Nan Geng. DRL-OR: Deep Reinforcement Learning-based Online Routing for Multi-type Service Requirements. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2021.

[30] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches. In *USENIX Security Symposium (USENIX Security)*, 2021.

[31] Danilo Mandic and Jonathon Chambers. *Recurrent Neural Networks for Prediction: Learning Algorithms, Architectures and Stability*. John Wiley & Sons, Inc., 2001.

[32] Microsoft. Introduction to Receive Side Scaling. https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling, accessed on Sep. 2023.

[33] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[34] Edgar Costa Molero, Stefano Vissicchio, and Laurent Vanbever. FAst In-Network GraY Failure Detection for ISPs. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2022.

[35] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. Sketchlib: Enabling Efficient Sketch-based Monitoring on Programmable Switches. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[36] NetFPGA. NetFPGA. https://netfpga.org/, accessed on Sep. 2023.

[37] Netronome. Netronome AgilioTM CX 2x40GbE intelligent server adapter. https://www.netronome.com/media/redactor_files/PB_Agilio_CX_2x40GbE.pdf, accessed on Sep. 2023.

[38] Nvidia. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit, accessed on Sep. 2023.

[39] Yuqi Qing, Qilei Yin, Xinhao Deng, Yihao Chen, Zhuotao Liu, Kun Sun, Ke Xu, Jia Zhang, and Qi Li. Low-Quality Training Data Only? A Robust Framework for Detecting Encrypted Malicious Network Traffic. In *Network and Distributed System Security Symposium (NDSS)*, 2024.

[40] Babak Rahbarinia, Roberto Perdisci, Andrea Lanzi, and Kang Li. Peerrush: Mining for Unwanted P2P Traffic. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2013.

[41] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. Automated Website Fingerprinting through Deep Learning. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[42] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[43] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. Can the Network be the AI Accelerator? In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, 2018.

[44] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling Distributed Machine Learning with In-Network Aggregation. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[45] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. Continuous In-Network Round-Trip Time Monitoring. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2022.

[46] Meng Shen, Yiting Liu, Liehuang Zhu, Xiaojiang Du, and Jiankun Hu. Fine-grained Webpage Fingerprinting using Only Packet Length Information of Encrypted Traffic. *IEEE Transactions on Information Forensics and Security (TIFS)*, 2020.

[47] Meng Shen, Jinpeng Zhang, Liehuang Zhu, Ke Xu, and Xiaojiang Du. Accurate Decentralized Application Identification via Encrypted Traffic Analysis using Graph Neural Networks. *IEEE Transactions on Information Forensics and Security (TIFS)*, 2021.

[48] Sandra Siby, Marc Juarez, Claudia Diaz, Narseo Vallina-Rodriguez, and Carmela Troncoso. Encrypted DNS–> Privacy? A Traffic Analysis Perspective. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[49] Carlos N. Silla and Alex A. Freitas. A Survey of Hierarchical Classification across Different Application Domains. *Data Mining and Knowledge Discovery*, 2011.

[50] Giuseppe Siracusano and Roberto Bifulco. In-network Neural Networks. *arXiv preprint arXiv:1801.05731*, 2018.

[51] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. Re-architecting Traffic Analysis with Neural Network Interface Cards. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[52] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Hamed Haddadi, Gianni Antichi, and Roberto Bifulco. Running Neural Networks on the NIC. *arXiv preprint arXiv:2009.02353*, 2020.

[53] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. Taurus: A Data Plane Architecture for Per-Packet ML. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[54] Mellanox Technologies. BlueField SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf, accessed on Sep. 2023.

[55] Thijs Van Ede, Riccardo Bortolameotti, Andrea Continella, Jingjing Ren, Daniel J. Dubois, Martina Lindorfer, David Choffnes, Maarten van Steen, and Andreas Peter. Flowprint: Semi-supervised Mobile-app Fingerprinting on Encrypted Network Traffic. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[57] Bruno Missi Xavier, Rafael Silva Guimarães, Giovanni Comarela, and Magnos Martinello. Programmable Switches for In-networking Classification. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2021.

[58] Guorui Xie, Qing Li, Yutao Dong, Guanglin Duan, Yong Jiang, and Jingpu Duan. Mousika: Enable General In-network Intelligence in Programmable Switches by Knowledge Distillation. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2022.

[59] Renjie Xie, Yixiao Wang, Jiahao Cao, Enhuan Dong, Mingwei Xu, Kun Sun, Qi Li, Licheng Shen, and Menghao Zhang. Rosetta: Enabling Robust TLS Encrypted Traffic Classification in Diverse Network Environments with TCP-Aware Traffic Augmentation. In *USENIX Security Symposium (USENIX Security)*, 2023.

[60] Jiarong Xing, Kuo-Feng Hsu, Yiming Qiu, Ziyang Yang, Hongyi Liu, and Ang Chen. Bedrock: Programmable Network Support for Secure RDMA Systems. In *USENIX Security Symposium (USENIX Security)*, 2022.

[61] Jiarong Xing, Qiao Kang, and Ang Chen. NetWarden: Mitigating Network Covert Channels while Preserving Performance. In *USENIX Security Symposium (USENIX Security)*, 2020.

[62] Jiarong Xing, Wenqing Wu, and Ang Chen. Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries. In *USENIX Security Symposium (USENIX Security)*, 2021.

[63] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. ACC: Automatic ECN Tuning for High-Speed Datacenter Networks. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2021.

[64] Penghang Yin, Jiancheng Lyu, Shuai Zhang, Stanley J. Osher, Yingyong Qi, and Jack Xin. Understanding Straight-through Estimator in Training Activation Quantized Neural Nets. In *International Conference on Learning Representations (ICLR)*, 2019.

[65] Qilei Yin, Zhuotao Liu, Qi Li, Tao Wang, Qian Wang, Chao Shen, and Yixiao Xu. An Automated Multi-Tab Website Fingerprinting Attack. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2021.

[66] Ruijie Zhao, Mingwei Zhan, Xianwen Deng, Yanhao Wang, Yijun Wang, Guan Gui, and Zhi Xue. Yet another Traffic Classifier: A Masked Autoencoder based Traffic Transformer with Multi-Level Flow Representation. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2023.

[67] Ziming Zhao, Zhuotao Liu, Huan Chen, Fan Zhang, Zhuoxue Song, and Zhaoxuan Li. Effective DDoS Mitigation via ML-Driven In-network Traffic Shaping. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2024.

[68] Changgang Zheng, Zhaoqi Xiong, Thanh T. Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. IIsy: Practical In-Network Classification. *arXiv preprint arXiv:2205.08243*, 2022.

[69] Changgang Zheng and Noa Zilberman. Planter: Seeding Trees within Switches. In *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*, 2021.

[70] Zhizhen Zhong, Weiyang Wang, Manya Ghobadi, Alexander Sludds, Ryan Hamerly, Liane Bernstein, and Dirk Englund. IOI: In-network Optical Inference. In *Proceedings of the ACM SIGCOMM 2021 Workshop on Optical Systems*, 2021.

[71] Guangmeng Zhou, Zhuotao Liu, Chuanpu Fu, Qi Li, and Ke Xu. An Efficient Design of Intelligent Network Data Plane. In *USENIX Security Symposium (USENIX Security)*, 2023.

[72] Guihua Zhou, Guo Chen, Fusheng Lin, Tingting Xu, Dehui Wei, Jianbing Wu, Li Chen, Yuanwei Lu, Andrew Qu, Hua Shao, and Hongbo Jiang. Primus: Fast and Robust Centralized Routing for Large-scale Data Center Networks. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2021.

A Appendix

A.1 Model Realization on the Data Plane

In this segment, we supplement additional design details regarding deploying our binary RNN on the programmable data plane.

A.1.1 Intuitive Number Comparison

The foundation to realize argmax is number comparison. There are two types of approaches to compare numbers on the programmable data plane. The first is based on conditional statements. Specifically, a simple statement to compare num_A and num_B is

```
if (num_A > num_B) act_A; else act_B;
```

which, unfortunately, is not compiled because the condition is too complex. We could avoid complex condition by the following statement

```
tmp = num_A - num_B;
if (tmp > 0) act_A; else act_B;
```

which takes at least two switch stages. More crucially, it is difficult to scale statement-based approaches to compare n numbers since it would take $n \times (n-1)/2$ differences.

The alternative approach is designing a match-action table which takes the num_A and num_B as keys (*e.g.,* the concatenation of their bits) and performs act_A or act_B according to the lookup result. Meanwhile, we can easily extend width of keys to realize the comparison of multiple numbers, which is exactly the argmax operation. Yet, the drawback is the explosion of required table entries: to obtain the maximum number among n values, each with m-bits, it takes 2^{nm} entries to enumerate all possible key-value combinations.

A.1.2 Ternary-Matching Based Design

In § 5.2, we discuss the implementation of argmax operation with ternary matching. We define the problem as following conditions and restrictions.

1. There are n numbers of m bit(s) each, where $n, m \geq 1$.
2. There is a predefined order to determine which number to select when there is a tie for maximum value.
3. Only one ternary matching is allowed, i.e., the calculation before matching is prohibited.

We denote the number of table entries as $F(n,m)$. Based on the basic optimization in § 5.2, we get the following recurrence relationship of $F(n,m)$.

$$F(n,m) = 2 \times F(n,m-1) + \sum_{i=1}^{n-1} \binom{n}{i} F(i,m-1), n \geq 2, m \geq 2 \tag{2}$$

We explain the meaning of the above equation as follows. The entries should cover all possible combinations of the n numbers. We consider all the combinations in different categories classified according to the combination of most significant bits (MSBs). Among all 2^n categories, there are $\binom{n}{i}$ categories where i numbers are with $\mathsf{MSB} = 1$ and $n-i$ numbers are with $\mathsf{MSB} = 0$ ($i \in [1, n-1]$). In these categories, we do not further consider those $n-i$ numbers with $\mathsf{MSB} = 0$, and only focus on the possible combinations of the i numbers with $\mathsf{MSB} = 1$, which are sub-problems with $n' = i, m' = m-1$ and require $\binom{n}{i} F(i, m-1)$ entries in total. In the other two categories (all $\mathsf{MSB} = 0$ or all $\mathsf{MSB} = 1$), we continue to focus on the possible combinations of all the n numbers, which are both sub-problems with $n' = n, m' = m-1$ and require $2 \times F(n, m-1)$ entries in total.

After the two optimizations described in § 5.2, the recurrence relation of $F(n,m)$ is given as follows.

$$F(n,m) = F(n,m-1) + \sum_{i=1}^{n-1} \binom{n}{i} F(i,m-1), n \geq 2, m \geq 2 \tag{3}$$

$$F(n,1) = n, n \geq 1 \tag{4}$$

$$F(1,m) = 1, m \geq 1 \tag{5}$$

By solving this iterative formula, we obtain $F(n,m) = nm^{n-1}$. We provide the derivation process from Equation (3) as follows. First, let $F(0,m) = 0, m \geq 1$, which is consistent with the Equation (4). Then the Equation (3) can be written as:

$$F(n,m) = \sum_{i=0}^{n} \binom{n}{i} F(i,m-1), n \geq 0, m \geq 2 \tag{6}$$

Then, we transform the formula and get:

$$F(n,m) = \sum_{i=0}^{n} \frac{n!}{i!(n-i)!} F(i,m-1), n \geq 0, m \geq 2 \tag{7}$$

$$\frac{F(n,m)}{n!} = \sum_{i=0}^{n} \frac{F(i,m-1)}{i!} \frac{1}{(n-i)!}, n \geq 0, m \geq 2 \tag{8}$$

We denote $\frac{F(n,m)}{n!}$ as ${}^m g_n$, and $\frac{1}{n!}$ as h_n. Then we construct the generating function of ${}^m g$ and h.

$${}^m G(x) = \sum_{n=0} {}^m g_n x^n = \sum_{n=0} \frac{F(n,m)}{n!} x^n, m \geq 1 \tag{9}$$

$$H(x) = \sum_{n=0} h_n x^n = \sum_{n=0} \frac{x^n}{n!} = e^x \tag{10}$$

Table 5: The no. of entries required for different m, n.

No. of Entries	Opt 1 & 2	Opt 2 only	Opt 1 only	Base Design	2^{mn}
n=3,m=16	768	2949123	863	4587523	2.81e14
n=4,m=8	2048	44028	2788	76028	4.29e9
n=5,m=5	3125	10245	5472	21077	3.36e7
n=6,m=4	6144	10890	13438	26978	1.68e7

We can obtain the recursive relations between ${}^{m}G$ and ${}^{m-1}G$ from Equation (8).

$$ {}^{m}G =^{m-1} G \times H, m \geq 2 \tag{11} $$

And for $m = 1$, we have

$$ \begin{aligned} {}^{1}G(x) &= \sum_{n=0} {}^{1}g_n x^n = \sum_{n=0} \frac{F(n,1)}{n!} x^n \\ &= \sum_{n=0} \frac{n}{n!} x^n = \sum_{n=1} \frac{1}{(n-1)!} x^n \\ &= x \sum_{n=0} \frac{1}{n!} x^n = x e^x \end{aligned} \tag{12} $$

With the mathematical induction, we can get

$$ {}^{m}G(x) = x e^{mx}, m \geq 1 \tag{13} $$

Compared with the Equation (9) and we can get

$$ \begin{aligned} F(n,m) &=^{m} G^{(n)}(0) = (x e^{mx})^{(n)} \Big|_{x=0} \\ &= (nm^{n-1} + m^n x) e^{mx} \Big|_{x=0} \\ &= nm^{n-1}, n \geq 1, m \geq 1 \end{aligned} \tag{14} $$

We can verify the result using mathematical induction.

$$ \begin{aligned} & F(n,m-1) + \sum_{i=1}^{n-1} \binom{n}{i} F(i,m-1) \\ &= \sum_{i=1}^{n} \binom{n}{i} i(m-1)^{i-1} \\ &= n \sum_{i=1}^{n} \binom{n-1}{i-1} (m-1)^{i-1} \\ &= n \sum_{i=0}^{n-1} \binom{n-1}{i} (m-1)^{i} \\ &= nm^{n-1} = F(n,m) \end{aligned} \tag{15} $$

In Table 5, we list the number of entries required for different combinations of m and n. The results demonstrate that our design, augmented by two optimizations, significantly reduces table consumption for achieving the argmax operation.

A.1.3 Packet Counters

Because the number of packets in a flow is unknown in advance, statically allocating a fixed width of bits for packet counters may result in buffer overflow. Meanwhile, as described in § 5.1, we need to perform modulo operations on packet count (*i.e.,* pktcnt % $(S-1)$) when storing embedding vectors. Thus, packet counting in BoS is designed based on two parallel counters: the first counter increases from 1, and stops at S (the sliding window size). For the ith packet, it returns i if $i < S$, otherwise it returns S. The second counter increases from 0 and cycles back to 0 after $S-2$, simulating the modulo operation. Thus, when the number of arrived packets in the flow exceeds S, the first counter essentially becomes a flag indicating that index for the ring buffer (storing embedding vectors) can be read from the second counter.

A.1.4 Flow Management

BoS relies on stateful storage to maintain per-flow state. Prior art [4, 61] relies on the control plane to allocate non-conflicting storage indices for different flows. To achieve line-speed traffic analysis, BoS relies on the readily available hardware hashing to allocate flow storage indices. In particular, the storage index for flow f is computed as $\mathcal{H}(f(\text{5-tuple}) \% \text{N})$, where $\mathcal{H}$ is the hash function, and N represents the total number of continuous per-flow storage blocks allocated for maintaining per-flow state.

Both hash and modulo operations may result in flow index collisions, *i.e.,* two different flows (with different 5-tuples) may receive the same storage index. To avoid confusions, BoS stores a tuple {TrueID, timestamp} alongside the storage index, where TrueID represents the actual flow identifier[2] and timestamp represents the latest packet arrival time for the flow. When storage indices collide, BoS allows the new flow to take the occupied storage only if the existing flow is timed out (*i.e.,* the stored timestamp is earlier than a predefined threshold). Otherwise, the new arrived flow falls back to use the per-packet tree model trained using only per-packet features, or falls back to IMIS; see discussions in § 7.3.

When developing the prototype of BoS, we observe a possible corner case for flow management. Specifically, the switch has multiple forwarding pipes, each of which has several processing stages. To support more complex RNN models, we can simultaneously use the stages in both the ingress and egress pipes. However, if multiple ingress pipes could be mapped to the same egress pipe (*e.g.,* traffic entered from both pipe A and pipe B may exit from pipe A), we would need to deploy a flow management module in both the ingress and egress pipe, because flows that do not collide in their ingress pipes may collide in the egress pipe. We have not encountered this corner case even in our scaling experiments (see § 7.3), and therefore we only deploy the flow management module in the ingress pipe.

[2] To avoid resubmitting or recirculating packets, the read and write of the tuple need to be finished in an atomic operation. This restricts the length of TrueID so that we cannot directly use 5 tuple as the TrueID. Thus, we leverage a different hash function H' to calculate the TrueID as $H'(f(\text{5-tuple}))$.

A.1.5 Per-packet Fallback Model

When the flow manager cannot allocate storage for a new flow, BoS falls back to analyzing the packets of that flow using a tree model trained only using per-packet features. Specifically, we use a 2×9 Random Forest model (2 trees with max depth 9), and use the same per-packet features as in [71] (*e.g.,* packet length, TTL, Type of Service, TCP offset). We apply the coding mechanism from NetBeacon [71] to deploy this tree model on the data plane alongside our binary RNN model.

A.1.6 The Pre-analysis Issue

As discussed in § 4.3, we employ a sliding window mechanism in our binary RNN inference where the model continuously processes packet segments. The length of the segment is a hyper-parameter S (set to 8 in our prototype). As a result, the very first $S-1$ packets of a flow cannot form a complete segment. This results in the pre-analysis issue: the inference results on these packets may be inaccurate because the model simply has not observed enough information. Any model-driven (or data-driven) traffic analysis approach has this limitation.

To avoid premature inference results caused by the pre-analysis problem, BoS regards the first $S-1$ packets of a flow as *pre-analysis packets*, and only starts to produce inference results for the Sth and subsequent packets (*i.e.,* any inference result output by BoS is based on at least one full segment). The protocol for forwarding these pre-analysis packets should be application-specific. For instance, in security-oriented task, BoS can forward pre-analysis packets via a dedicated low priority queue so that a strategic adversary cannot overwhelm the network by sending very short flows (less than S packets). In other tasks (*e.g.,* an inbound gateway on a campus/enterprise that loads balance different types of traffic received from the Internet), simply forwarding these pre-analysis packets may be sufficient, considering the average length of campus Internet flows ($\sim$120) [21] is much larger than S (8 in our prototype). Finally, it is possible to employ another learning model trained only on per-packet features (such as [58]) to process these pre-analysis packets.

A.2 Prototype Implementation

In this segment, we supplement additional details about our implementation.

A.2.1 On-Switch RNN Inference

Component Overview. We plot the hardware implementation of the binary RNN in BoS in Figure 8. The top part shows the simplified dependency graph of all components. We plots two types of dependency: if a has data dependency on b, then the input data of a is (partially) provided by b; if a has control dependency on b, then the execution of a is determined by b. For instance, the GRU tables have data dependency on embedding vector storage; the window CTR (counting the number of windows/segments) has control dependency on the PKT CTR-1 (indicating whether the number of received packet is no less than S).

Per-Stage Breakdown. The bottom-right part shows the breakdown of stage usage for deploying a BoS model with the hyper-parameters shown in the bottom-left part. We use the stages in both ingress and egress pipeline. The kth ingress stage and k^{th} egress stage share the same underlying hardware resource.

We first introduce the stage usage in ingress. In the stage 0, besides calculating the flow index and TrueID for flow management, it also executes embedding of packet length, since it has no other dependency. Then, the FlowInfo tuple (*i.e.,* $\{\text{TrueID}, \text{timestamp}\}$) is stored in stage 1 for flow collision avoidance. Flow management is only necessary in the ingress pipeline (see § A.1.4). The inter-packet delay (IPD) embedding is implemented using the following three stages: stage 2 stores the last packet timestamp, stage 3 obtains IPD by subtracting the current packet arrival time with the last timestamp, and stage 4 computes the embedding of IPD. In stage 5, an FC layer takes in the packet length embedding and IPD embedding to output an embedding vector, which is stored using 7 bins and dispatched to corresponding GRU tables in stages 6 to 8. All seven bins cannot be allocated into one stage because only 4 registers (register arrays) are allowed in one stage. In the last three stages of ingress, the first four GRU tables (*i.e.,* $\text{GRU}_{1,2,3,4}$) are placed sequentially. The first two GRU tables (*i.e.,* $\text{GRU}_{1,2}$) are implemented with one match-action table, *i.e.,* $h{=}\text{GRU}_2(\text{GRU}_1(0,ev_1),ev_2)$ is merged as $h{=}(\text{GRU}_2\circ\text{GRU}_1)(0,ev_1,ev_2)$.

In the egress pipeline, the remaining four GRU tables (*i.e.,* $\text{GRU}_{5,6,7,8}$) are placed from stage 0 to stage 3. The output layer is merged with GRU_8, *i.e.,* $C{=}\text{Output}(\text{GRU}_8(h,ev_8))$ is merged as $C{=}(\text{Output}\circ\text{GRU}_8)(h,ev_8)$. The counters to accumulate per-class probabilities ($\text{CPR}_{1..6}$) are spread in stage 4 and 5. To accumulate the probability vectors (*i.e.,* the intermediate results) on the data plane, we quantize the probability for a class to an integer from 0 to 15. Considering the reset period of 128 packets, the width of cumulative probability is $\lceil log_2(16\times128)\rceil{=}11$. To implement argmax for $n{=}6,m{=}11$, we split it into three sequential argmax operations, two for $n{=}3,m{=}11$ and one for $n{=}2,m{=}11$, *i.e.,* comparing the first three numbers in stage 5, then comparing the rest three numbers in stage 6, then comparing the two winners in stage 7. Finally, the escalation logic is implemented in stage 8 to 9. To obtain the classification confidence for a packet, we do not actually divide the largest accumulative probability with wincnt (the total number of intermediate results for the flow), as division is not supported on the data plane. Instead, we compare the probability with $\mathbb{T}_{\text{conf}}\times\text{wincnt}$, which is divided into a subtraction and a comparison with 0. The subtraction

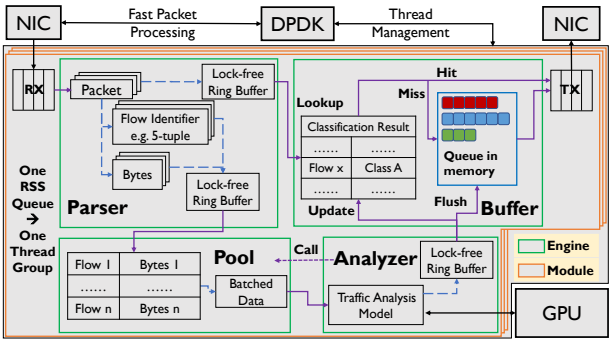


Figure 13: The architecture of IMIS.

is performed in the *action* of the winning case of argmax match-action table. And the comparison with 0 is executed while reading/updating the counter that stores the number of ambiguous packets in stage 8. If the counter exceeds T_{esc}, the packet is escalated to IMIS.

Escalation Flag. Due to the limited number of stages in the ingress pipeline of the Tofino 1 switch, we must use both the ingress and egress pipelines in our prototype. However, this poses a challenge because the egress port of a packet must be determined in the ingress pipeline, but we cannot determine whether a packet should be escalated to IMIS until all operations in the egress pipeline have completed. To address this challenge, we store an escalation flag in the ingress pipeline so that the egress port for a packet can be properly determined. We update the escalation flag through egress-to-egress mirroring and recirculating.

A.2.2 Implementation of IMIS

The architecture of IMIS is plotted in Figure 13. We use Intel Data Plane Development Kit (DPDK version 20.11.9 LTS) [18] to enable multiple NIC RX/TX queues, each of which is bound to one analysis module. The Receive Side Scaling (RSS) [32] is enabled to efficiently distribute traffic to each analysis module.

Analysis Module. The parser engine uses DPDK APIs to parse flow identifier (*e.g.,* 5-tuple) and the raw bytes from the input traffic. It stores the parsing results into a lock-free ring buffer consumed by the pool engine to maintain per-flow state and perform batch arrangement. After obtaining the parsing result of a packet, the packet is sent to another lock-free ring buffer consumed by the buffer engine to perform egress queuing according to model inference results. As our transformer-based model only uses the first 5 packets in a flow for inference, the subsequent packets sent by the flow will be forwarded to the buffer engine directly without raw bytes extraction.

The pool engine translates the streamed parsing results into batched data to facilitate model inference. Specifically, it continuously fetches the raw byte features of packets from the lock-free ring buffer linked to the parser engine, and organizes them as per-flow state. When it receives a call from the analyzer engine, the pool engine selects flows according to their timestamps to form a batch of inputs, and sends the batch to the analyzer engine for inference. If a selected flow has fewer than 5 packets, the pool engine pads its data with zeros. The inference result obtained for this flow is considered to intermediate, and the pool engine may select this flow again in the next round.

To accelerate model inference, the analyzer engine uses CUDA (version 11.7) [38] to interact with the auxiliary GPU card. Specifically, it continuously requests input batches from the pool engine. Upon receiving a batch, the analyzer engine executes inference on the GPU and sends the results to a lock-free ring buffer consumed by the buffer engine.

The buffer engine continuously fetches the latest inference results from the analyzer engine and uses the results to release packets. Upon receiving a packet from the parser engine, the buffer engine checks if the inference result for the packet's flow has been determined. If so, the packet is released immediately. Otherwise, the packet is placed in the egress queue for its flow to wait for the inference result. When the buffer engine receives a flow inference result from the analyzer engine, it releases all packets in the egress queue for that flow.

The buffer engine keeps fetching the latest inference results from the analyzer engine, and uses the results to release packets. Upon receiving a packet from the parser engine, the buffer engine checks if the inference result for the packet's flow has been determined. If so, the packet is released immediately. Otherwise, the packet is placed in the egress queue for its flow to wait for the inference result. When the buffer engine receives a flow inference result from the analyzer engine, it releases all packets in the egress queue for that flow.

A.3 Additional Details about Testbed

We use a Wedge 100BF-32X programmable switch with 2 pipes and 32×100 Gbps ports to deploy the on-switch RNN in BoS. The version of SDE is 9.7.0. The off-switch IMIS is hosted on server with two Intel(R) Xeon(R) Gold 6348 CPUs (2×28 cores), Ubuntu 20.04.1, 512 GB memory, and one Mellanox 100 Gbps NIC with two ports that support DPDK (version 20.11). We reserve 160 GB memory as huge pages for DPDK (80 GB/NUMA Node), and an NVIDIA A100 GPU is attached to IMIS. All physical cores for parser engines, pool engines and buffer engines are on the same Node with NIC, and all the physical cores for analyzer engines are on the same Node with an auxiliary GPU.

Scaling the On-switch Analysis Beyond One Pipe. The on-switch analysis of our current prototype is implemented using one switch pipe. The complexity of scaling the analysis beyond one pipe depends on whether cross-pipe traffic forwarding is allowed. Specifically, if the traffic forwarding for each pipe is self-contained (*i.e.,* the traffic ingressing from one pipe will only exit from this pipe), we can easily operate

multiple pipes independently, where each pipe hosts an instance of our on-switch RNN and processes traffic in parallel. However, when the flows entering from different pipes can eventually exit via the same pipe, we have to deploy our flow management modules in both the ingress and egress pipes (as we have discussed in § A.1.4). In this case, the flows allocated dedicated per-flow storage within their ingress pipes may still end up using the per-packet model if their storage indices collide when exiting from the same egress pipe. This would lead to the underutilization of storage resources initially reserved for these flows within their ingress pipes.

Runtime Programmability. The on-switch analysis model of BoS can be programmed in runtime. Specifically, the weights of RNN layers, the escalation thresholds, the number of classification classes, and widths of the inputs and outputs of each layer (*i.e.,* the number of binary neurons) are all programmable via the control plane. For instance, the weights can be reconfigured by updating the table entries from the control plane.

On-switch Statistics Collection. To collect the evaluation results from our testbed, we use the second pipe on our switch to implement a result collection module. Specifically, we allocate registers to count the numbers of escalated packets, packets analyzed by per-packet model, packets analyzed by binary RNN, and pre-analysis packets. Further, we allocate a register array for reporting the on-switch analysis precision and recall for each class, using the combination of ground-truth label and predict label as index. We read these registers from the control plane to obtain the raw data for calculating the macro-F1 scores.

Flow Replayer. To generate traffic according to our pcap files, we investigate both tcprelay and DPDK pktgen. We choose to use pktgen because it can generate high-throughput traffic that saturates the physical 100 Gbps NIC on our testbed. Yet, the key problem of pktgen is that it fails to honor the packet timestamps when sending traffic. However, the on-switch RNN relies on inter-packet delays for inference. To work around this issue, we embed the desired timestamp of each packet within the MAC address field of its Ethernet frame. The on-switch analysis pipeline reads this field for flow management and inference. We create 32 pcap files throughout the evaluation. When the flow replayer sends an excessively large pcap file that cannot be loaded into the memory at once, it breaks the file into smaller slices and replays these slices sequentially.

Stress Test of Standalone IMIS. To stress test the system performance of IMIS (§ 7.3), we generate flows on a server with DPDK packet generator (pktgen version 23.06). These flows are sent directly to the server where we deploy IMIS, bypassing the on-switch analysis. To generate a burst of concurrent flows, the packet generator repeatedly sends packets within a group of selected 5-tuples and the packet size is fixed as 512 bytes.

A.4 Additional Details about Datasets

Data Pre-processing. For every dataset used in our evaluations, we collect flow records from the original pcap files using the following procedure. (i) We collect the original pcap files for each class in the dataset separately, and all the flow records extracted from a pcap file are labelled as the class of this file. (ii) For each pcap file, we collect the TCP and UDP packets of IPv4, and remove other irrelevant packets, *e.g.,* packets of Domain Name System (DNS), Address Resolution Protocol (ARP), Dynamic Host Configuration Protocol (DHCP) and so on. (iii) We split a clean pcap file by five tuple, and further split packets of the same five tuple into flow records by inter-packet delays. Specifically, if the inter-packet delay between two packets is greater than 256 ms, we consider the latter packet as the first packet of a new flow record. This is consistent with our online inference where we consider a flow is completed if we do not receive new packets for the flow for 256 ms. (iv) 80% of flow records in a dataset are used as the training set and the remaining records are used as testing set.

Traffic Analysis Tasks. We evaluate BoS using the following tasks.

- Encrypted traffic classification on VPN. This task classifies traffic encrypted by Virtual Private Networks (VPNs). We use the ISCXVPN2016 [12] dataset, which contains 7 categories of communication applications captured through the Canadian Institute for Cybersecurity in both VPN and non-VPN. We process the original pcap files for 6 classes of VPN flows, including Email, Chat, Streaming, FTP, VoIP, and P2P. We exclude the Browsing class in our evaluation because some of the applications used for generating Email, Streaming, VoIP packets are web-based, resulting in significant noises, as explained in [12]. The number of flows in each of the six classes is 613, 2350, 375, 1789, 3495, and 1130, respectively.
- Botnet traffic classification on IoT. This task classifies different botnet traffic collected from the Internet of Things (IoT) systems. We process the original pcap files for 4 classes of flows (Data Exfiltration, Key Logging, OS Scan, Service Scan) from the BOTIOT [22] dataset, collected in a realistic network environment deployed in the Cyber Range Lab of UNSW Canberra. The number of flows in each class is 353, 427, 1593, and 7423, respectively.
- Behavioral analysis of IoT Devices. This task classifies traffic generated by IoT devices in different working states. We collect the original pcap files for 3 classes (Power, Idle, Interact) from the CICIOT2022 [10] dataset, which contains 40 devices of audio, camera, home automation and so on. We process the original pcap files for the Power and Interact classes, and select one day from the 30 days of Idle pcap

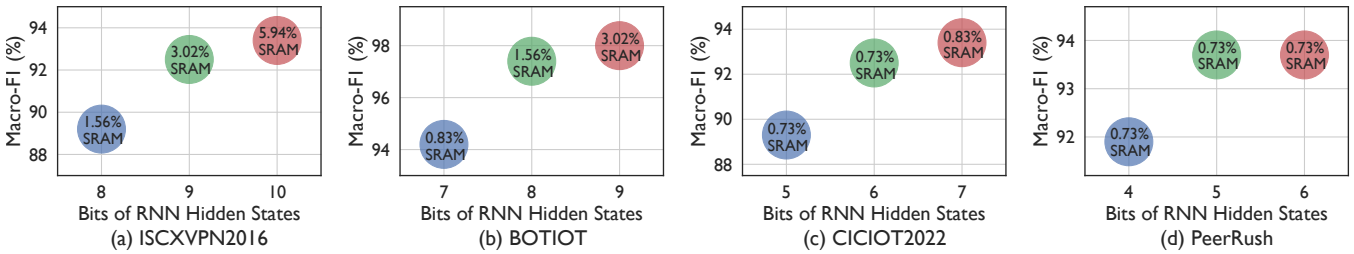


Figure 14: [Testbed] The traffic analysis accuracy given different binary RNN model sizes.

files. The number of flows in each class is 1131, 4382, and 1154, respectively.

- P2P application fingerprinting. This task classifies P2P application traffic. We process the original pcap files for 3 classes (eMule, uTorrent, and Vuze) from the PeerRush dataset [40]. Each class captures one hour of traffic. The number of flows in each class is 20919, 9499, and 7846, respectively.

A.5 Reproducing [71] and [51]

We reproduce two recent art NetBeacon [71] and N3IC [51] for evaluation.

- NetBeacon [71]: a reproduced version of NetBeacon, which deploys multi-phase tree-based models on switch using both flow-level features and per-packet features. We use the same per-packet features as in [71], and use the max, min, mean, and variance of the packet size and IPD as flow-level features. The inference points are located at the $\{8^{\text{th}}, 32^{\text{nd}}, 256^{\text{th}}, 512^{\text{nd}}, 2048^{\text{th}}\}$ packet. For each phase we train a 3×7 (3 trees with max depth 7) Random Forest model (their largest model).
- N3IC [51]: a reproduced version of N3IC, which deploys binary MLP on SmartNIC using both statistical flow-level features and per-packet features. We use the same features and phases as NetBeacon for fair comparison, and for each phase the number of neurons in the hidden layers is [128, 64, 10] (their largest model). Note that N3IC deploys binary MLP on SmartNIC but the model cannot be deployed on P4 switches due to hardware resource constraints. Thus, we simulate the switch-side traffic management logic and the binary MLP inference in software to to obtain the traffic analysis results for N3IC.

A.6 Binary RNN Model Complexity

The number of bits allocated to store the RNN hidden states determines both the performance of our RNN model and the size of match-action table for a GRU layer. In Figure 14, we present the performance of BoS under different bit lengths. The default bit lengths used in four tasks are 9, 8, 6, and 5, respectively. This is because further increasing the bit lengths does not significantly improve F1 scores, while it does increase the SRAM consumption (especially for the first two tasks). When the bit length is smaller than 6, the size of one GRU table is smaller than the minimum allocation unit of SRAM. Thus, further reducing bit lengths will not reduce SRAM consumption.

The Eternal Tussle: Exploring the Role of Centralization in IPFS

Yiluo Wei[1] Dennis Trautwein[2] Yiannis Psaras[2] Ignacio Castro[3]
Will Scott[2] Aravindh Raman[4] Gareth Tyson[1]

[1]*Hong Kong University of Science and Technology (GZ)* [2]*Protocol Labs*
[3]*Queen Mary University of London* [4]*Brave Software*

Abstract

Web centralization and consolidation has created potential single points of failure, *e.g.,* in areas such as content hosting, name resolution, and certification. The "Decentralized Web", led by open-source software implementations, attempts to build decentralized alternatives. The InterPlanetary File System (IPFS) is part of this effort and attempts to provide a decentralized layer for object storage and retrieval. This comes with challenges, though: Decentralization can increase complexity, overhead, as well as compromise performance and scalability. As the core maintainers of IPFS, we have therefore begun to explore more hybrid approaches. This paper reports on our experiences building three centralized components within IPFS: (*i*) ***InterPlanetary Network Indexers***, which provides an alternative centralized method for content indexing; (*ii*) ***Hydra Boosters***, which are strategic DHT nodes that assist IPFS in content routing; and (*iii*) ***HTTP Gateways***, which are a public access point for users to retrieve IPFS-hosted content. Through this approach, we trade-off the level of decentralization within IPFS in an attempt to gain certain benefits of centralization. We evaluate the performance of these components and demonstrate their ability to successfully address the challenges that IPFS faces.

1 Introduction

Driven by powerful economies of scale, the centralization and consolidation of the web seems unstoppable [11, 19]. For example, website administrators will often choose to rely on large cloud platforms such as Amazon EC2, third-party DNS providers like GoDaddy, content delivery networks such as Akamai, and certificates issued by Let's Encrypt. These are well-engineered, performant services. Yet, they are also a potential single point of failure. Disruptions and outages in these centralized entities (*e.g.,* a cloud platform) can result in enormous financial losses, such as the reported loss of over $60,000 per minute by Amazon's eCommerce platform due to an outage [13, 15, 29, 33].

The "Decentralized Web" is a response to this growing concentration. The Decentralized Web refers to a group of technologies that aim to decentralize control away from major players. These technologies rely on open-source, community-led software implementations that decentralize traditional web functions, such as name lookup, object storage, and certification. As anyone can use and contribute to the software, the effort strives to reduce barriers to participation and reduce current trends towards web consolidation [19]. Note, we draw a clear distinction between *decentralized*, and *distributed*. Although many web system implementations are distributed, their control and operation remain centralized.

The *InterPlanetary File System (IPFS)* is part of this decentralization effort. IPFS is a storage layer for the Decentralized Web. It is a decentralized content-addressable object storage and retrieval platform. IPFS is a community-driven, open-source project, which covers 200 git repositories with 67809 stars and 12407 forks. In total, there are 83.5 K commits by 1352 code contributors, covering 400+ organizations including universities, start-ups and large corporations.

IPFS has seen widespread uptake with more than 1 B web client accesses and more than 250 k unique nodes participating in its peer-to-peer (P2P) network every week. Critically, IPFS underpins various other Decentralized Web applications, including social networking and discussion platforms (Discussify, Matters News), data storage solutions (Space, Peergos, Temporal), content search (Almonit, Deece), messaging (Berty), content streaming (Audius, Watchit), and e-commerce (Ethlance, dClimate) [2].

Although these figures point to the success of IPFS, our experience has shown that decentralization comes at a cost (§2.2). Decentralization can increase complexity and overhead due to the need to coordinate a large number of decentralized entities. Consequently, performance can be compromised, scalability can be challenging, and this can lead to a less practical system. Measuring, managing and debugging such systems is also more challenging. As part of the group of core maintainers, our operational experience has identified three key challenges faced by IPFS and similar systems:

1. **Massive content publication**: IPFS uses a distributed hash table (DHT) for content indexing and publication. However, our measurements show that publications can take over 1 minute, hindering the publication of large amounts of content. This is caused by the (*i*) decentralized routing process of node discovery in the DHT; and (*ii*) the need to replicate content indexing data across many nodes to mitigate the impact of churn.
2. **Content retrieval performance**: IPFS's overall retrieval delay is approximately 3x slower than HTTPS. By deploying a range of applications, we have found that IPFS's fully decentralized model is well suited to serving

delay-tolerant objects like file hosting, yet struggles with real-time applications such as live video streaming.

3. **Adoption**: The content-addressed IPFS protocols differ from traditional location-based protocols. This makes adoption difficult for both web developers and browser manufacturers, who mostly do not support IPFS retrieval. As IPFS retrievals tend to be slower than HTTPS, it also makes it difficult to integrate HTTP and IPFS objects within a single webpage. Further, full decentralization means that all clients should ideally install the IPFS node software. Yet, this requires technical skills, and IPFS software is currently too heavyweight for mobile devices. This presents barriers to adoption.

Our initial strategy was to tackle all these challenges in an entirely decentralized manner. However, practical constraints make this difficult, primarily driven by the inherent performance issues associated with DHTs, and the limited resources of (some) peers operating in the system. Thus, we have since been experimenting with the introduction of centralized components to complement the IPFS decentralized architecture. We argue that this can be an effective middle-ground, while decentralized technologies continue to be developed. The idea is to tradeoff the degree of decentralization of IPFS in order to attain potential benefits. Although this may result in a more efficient system, this also means that IPFS is partially centralized, raising certain issues which require exploration. For example, users may interact exclusively with centralized components during object retrieval, placing significant additional power in the hands of operators. This brings risks related to security, robustness, privacy, alongside raising philosophical questions related to the role of centralization in the Internet. Thus, we strive to design centralized components that are optional, allowing the system to continue operation even in the failure of centralized aspects (albeit with degraded system properties). With this tussle in mind, we report on our operational experiences at Protocol Labs in deploying a set of three complementary centralized components within the wider IPFS network.

First, we present the **InterPlanetary Network Indexers**. These provide an alternative centralized method for content indexing and lookup. Each Indexer acts as a server-based key store, complementing the role of the fully decentralized DHT. This avoids the overheads associated with peer-to-peer routing. Importantly, any entity can operate an Indexer with the ability (and expectation) to synchronize indexing data, thereby reducing the risks of consolidation. Second, we present the **Hydra Boosters**, a small set of high-performance DHT nodes that assist IPFS in content routing, content provision, and peer routing. The Hydra Boosters are strategically spread across the DHT key space and try to provide one-hop access to data records. Third, we present the **HTTP Gateways**, a set of proxy servers that offer an HTTP bridge into the IPFS network. These help reduce barriers to adoption by allowing clients to access IPFS content without installing the fully decentralized IPFS node software. Additionally, the centralized gateways benefit from aggregation in demand, allowing us to deploy caching of content in a way that was not possible when operating in a fully decentralized fashion. No privileges are required to establish these components, and anybody can contribute to the effort.

The contributions are as follows:

1. We present the design and implementation of three new centralized components deployed within the IPFS infrastructure. For each component, we explore the challenges it addresses and how we have tackled those challenges.
2. Based on real-world operational data, we present an evaluation of the three complementary components. We demonstrate that these centralized components do bring large benefits to IPFS, addressing the challenges discussed above.
3. We explore security, privacy, and other risks that could arise from the centralized nature of these components. We discuss the associated trade-offs that must be considered when designing hybrid solutions such as IPFS.

2 Background and Motivation

2.1 IPFS Fundamentals

We start by providing a brief overview of IPFS. We redirect interested readers to [41] for full details.

Content Addressing. IPFS is a decentralized object store. Much like prior information-centric networks, it uses self-certifying hash-based Content Identifiers (CIDs) to decouple content names from their storage location. When content is added to IPFS, it is split into chunks and each chunk is assigned its own CID, which is the result of hashing its content and adding metadata. These CIDs are then used to construct a Merkle Directed Acyclic Graph (DAG) of the file. The root node of the DAG combines all the CIDs of its descendant nodes and forms the final content CID, which allows for chunk de-duplication and eliminates the need to store or transmit the same content twice. Merkle DAGs are agnostic to where the content is stored, so they do not need to be updated when a file is replicated on or deleted from nodes in the network.

Peer Addressing. Each peer has a unique ID, generated as a hash of its public key, and represented as a Multihash [7]. A multiaddress is then associated with each PeerID. Multiaddresses are a simple data structure that allows multiple protocols and address types to be included for each peer.

Content Indexing. IPFS relies on a fully decentralized Kademlia DHT for content indexing. The DHT does not store content itself but, instead, hosts (*i*) *peer records* that map PeerIDs to the multiaddresses that can be used to contact the peer; and (*ii*) *provider records* that map the CIDs of content to the PeerIDs of the peers who provide the content. This indexing allows clients to map their desired content to a peer

that can provide it and to discover the multiaddresses that can be used to contact the peer.

Content Publication. To publish content, the host first generates a provider record that maps the CID to its own Peer ID. It then pushes it to the DHT. To ensure availability, this record is stored on the 20 closest peers in terms of their PeerIDs' XOR distance from the SHA256 hash of the CID. Note, peers retrieving the content can volunteer to become temporary or permanent content providers by publishing a provider record pointing to their own node on the DHT. By doing this, it avoids the original source becoming a single point of failure.

Content Retrieval. Content retrieval involves four steps: content discovery, peer discovery, peer routing, and content exchange. Content discovery is the process of looking up the provider record using the CID of the content. Before entering the DHT lookup, the requesting peer asks all peers it is already connected to for the desired content in an opportunistic fashion, using a protocol called BitSwap [3]. If this initial attempt is not successful, content discovery falls back to the DHT. After getting the PeerID of the provider from the provider record, peer discovery then involves querying the DHT and retrieving the PeerID's peer record. Recall, the peer record contains the peer's multiaddresses, listing the protocols and physical addresses that can be used to reach the node. To streamline this process, each IPFS node maintains a local address book of up to 900 recently seen peers. Once the PeerID is resolved to a peer record, the requesting node uses the list of Multiaddresses to connect to the desired peer, a process called peer routing. Finally, content exchange is carried out using the Bitswap protocol. Importantly, all the above steps take place across a fully decentralized infrastructure.

2.2 Challenges of Decentralization in IPFS

Our experience in operating the above decentralized setup has highlighted three key challenges that we discuss in this paper.

Massive Content Publication. IPFS utilizes a Kademlia distributed hash table (DHT) for publishing and locating content. Although Kademlia enables seamless distribution, this brings additional overhead and high delay when compared against simple centralized key stores (*e.g.,* SplinterDB). To highlight this, Figure 1 shows the content publication time as measured in our experiments (detailed in §4.2). The overall publication process across all regions takes 11.81s, 40.81s, and 66.73s at the 50th, 90th, and 95th percentiles, respectively. This is a clear cost of decentralization, which is not experienced by well-resourced centralized database lookups The delay is dominated by the DHT walk to find the nodes to publish provider records to (covering 83.37% of the overall delay), due to the need to distribute records to 20 different peers. Furthermore, the IPFS DHT requires re-publication of records every 24 hours. In practice, this takes an excessive amount of time, generates large traffic volumes, and consumes a substantial amount of storage space for peers to store provider records.

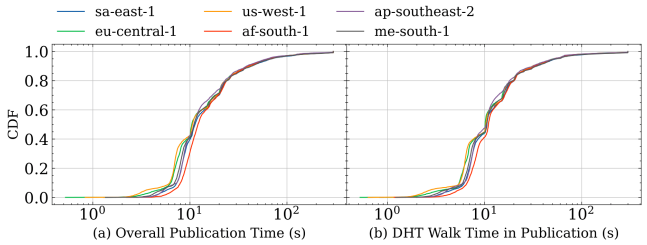


Figure 1: CDF for content publication for each AWS region: a) The overall publication duration, and b) the DHT walk.

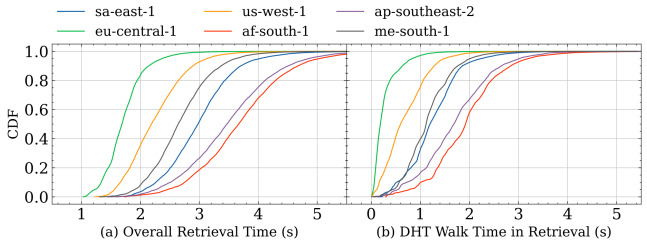


Figure 2: CDF for content retrieval for each AWS region: a) The overall retrieval duration, and b) the DHT walk.

Content Retrieval Performance. The decentralized nature of IPFS results in slower content retrieval speeds compared to well-resourced centralized systems. This is because all content must first be mapped to an appropriate source, via (several) routing hops in the DHT. Figure 2 shows the content retrieval time measured in our experiments (detailed in §4.2). The overall retrieval process across all regions takes 2.72 seconds, 4.03 seconds, and 4.42 seconds in the 50th, 90th, and 95th percentiles, respectively. This is slower than HTTP and is not suitable for certain delay-sensitive applications, such as live video streams. Moreover, the decentralization of storage across many independent nodes hinders the aggregation of demand, making it difficult to use techniques such as caching to improve the retrieval of frequently accessed content.

Adoption. To access IPFS-hosted content, users must run the IPFS node software, thereby participating in the storage and distribution of files. This is vital for the self-scaling properties of our decentralized setup, ensuring that no single node becomes overly powerful. However, setting up an IPFS node requires specific skills, which is a barrier for some users. Due to its footprint, the IPFS node software cannot yet run on mobile devices. This is problematic because 58% of website traffic comes from mobile devices, and approximately 92% of Internet users access the web using their smartphone [4]. There are also adoption challenges for web developers wishing to embed IPFS-host content within traditional HTML/HTTP websites. This is because IPFS retrievals tend to be slower than HTTP, creating usability issues with mixing HTTP and IPFS-hosted objects. Unfortunately, few browser implementations can retrieve IPFS content, further disincentivizing integration.

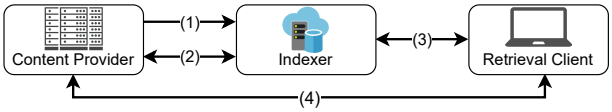


Figure 3: The publication and retrieval steps for Indexers. (1) Content provider publishes announce message. (2) Indexer synchronizes advertisements with the content provider. (3) Retrieval client queries Indexer using CID or multihashes and gets the provider record. (4) Retrieval client gets data from content providers according to the provider record.

3 Design and Implementation

To tackle the above challenges, we have experimented with deploying three new centralized elements to IPFS. These are all open-source, allowing anybody to set them up and contribute to IPFS. Through this, we strive to address the above challenges. These centralized components are:

1. ***InterPlanetary Network Indexers*** provide an alternative centralized approach to content indexing, complementing the DHT. It primarily targets the massive content publication challenge by making it faster to push new provider records. It also enhances content retrieval performance by making lookups faster.
2. ***Hydra Boosters*** introduce strategically placed reliable routing nodes in the DHT, to improve performance in a coordinated manner. This offloads work from the remaining nodes. Thus, the primary goal of Hydra Boosters is to enhance publication routing performance, as well as retrieval performance. It does this in two ways: (*i*) by hosting a large number of provider records in a stable fashion; and (*ii*) by providing stable routing nodes with the DHT.
3. ***HTTP Gateways*** provide an HTTP bridge into the IPFS network. Its primary focus is on improving adoption, by allowing access to IPFS without a full stack IPFS installation. Further, gateways also serve as a centralized point of request aggregation, enabling us to improve content retrieval performance via caching.

Although Protocol Labs has built and deployed these components, we emphasize that any stakeholder could adopt their usage. As such, whereas each individual component is centralized, there can be many instantiations by different operators. The rest of this section describes the components in detail.

3.1 InterPlanetary Network Indexers

Recall that the first challenge of decentralization in IPFS is massive content publication. We define this as the ability for providers to publish large numbers (*i.e.,* millions) of objects in an efficient fashion. This is currently difficult because IPFS's decentralized routing requires protocol exchange with *at least* 20 nodes to publish each object. Thus, in contrast to a centralized index, the overhead is dominated by the DHT hops that must be undertaken to publish provider records.

Overview of Indexers. To address this issue, we have built the InterPlanetary Network Indexers (aka "the Indexers") to complement the DHT. Put simply, an Indexer is a high-performance keystore server that indexes provider records. The provider records in the Indexers differ from those in the DHT. They comprise the identity of the content provider, its physical address, and the protocols required for retrieving the data. As a result, there is no need to resolve the PeerID to a physical address. Providers can push their provider records directly to the Indexer, and clients can directly retrieve the records from the Indexer using the appropriate CID. As this can be done within a single protocol exchange, it substantially reduces the overhead of storing provider records with the decentralized DHT. Importantly, the Indexer is optimized for bulk publication with certain requirements that publishers must adhere to. Hence, it is recommended for large content providers (and not for ordinary users) to publish via an Indexer. We emphasize that the Indexers do not replace the DHT but, instead, complement it by offering an additional content indexing approach. Figure 3 illustrates an overview of the publication and retrieval process, as detailed below.

Preparation of Advertisements. Each provider is required to record their own available content using a chain of immutable *advertisements*. An advertisement is a data structure asserting the publication or deletion of content by a given provider. An advertisement may contain multiple provider records. Importantly, by constructing an immutable chain of advertisements, it becomes possible for any Indexer to audit what content has been provided over time by each provider. Note, to achieve incremental verifiability, all advertisements are signed by the provider. To publish content, the provider adds a new advertisement to its chain and notifies the Indexers.

Sending Announcement Message. To notify the Indexers of newly published content, the provider sends an announcement message (Figure 3 (1)). An announcement message contains the CID of the advertisement, and the publisher's address (where to retrieve the advertisement from). By default, the announcement messages are sent via Gossipsub [42] (a P2P protocol to broadcast messages to nodes in a network). Through this, the providers gossip announcement messages among all Indexers. The Indexers can also receive such messages via an HTTP RESTful API call. The announcement messages can be shared among Indexers when configured to do so: they can re-publish HTTP announcements to other Indexers, and relay Gossipsub announcements.

Synchronize Advertisements. Once an Indexer receives an announcement message from a provider, it connects to the provider and traverses the advertisement chain to construct the up-to-date provider records (*i.e.,* advertisement synchronization, Figure 3 (2)). Since the advertisements are immutable,

the Indexers can recognize which portions of the chain are new and only traverse those sections. Once the provider records have been constructed, we use Pebble [8], a high-performance key-value store, to index all records.

Note, unlike the IPFS DHT, which only allows one file to be published at a time, the Indexers allow multiple provider records to be packed into a single advertisement. Additionally, content does not need to be repeatedly republished by providers, because there is no time-to-live for records in Indexers. Instead, only changes (additions or removals) are published, and the Indexers can reconstruct all provider records based on the advertisement chain. This makes the Indexer unsuitable for ephemeral publishers. Indexers must, therefore, configure their own policies for the removal of stale content. This design is based on our prior experiences with scalability and is optimized for bulk publication, directly addressing the previous massive publication challenge. Specifically, batching large amounts of provider records within a single advertisement makes it highly efficient to publish multiple objects at once and avoids republishing unless changes occur.

Retrieval of Content. Clients that wish to retrieve provider records interact with the Indexers via a RESTful HTTP query API. The identities of Indexers are currently public, with a default set distributed with client binaries. Clients can issue queries containing a single CID or several CIDs. In both cases, the Indexers return a list of provider records that match the query (Figure 3(3)). Retrieval clients can then retrieve the content directly from the providers based on the provider records (Figure 3(4)). To exploit the aggregation of queries at the Indexers, they utilize a frontend server to perform caching on the HTTP API calls. This caching stores frequently accessed provider records, allowing these requests to be directly served without querying the Indexer's Pebble database. This further reduces response latency and enhances overall performance.

3.2 Hydra Boosters

The second challenge of decentralization in IPFS pertains to content retrieval performance. This is because the DHT walk often needs to pass through multiple intermediate nodes, which can take a considerable amount of time.

Overview of Hydra Boosters. To address this issue, we have developed a set of Hydra Boosters. These are highly connected DHT nodes, which offer "shortcuts" through the routing space to provider records. They also help address the challenge of massive content publication, as they accelerate routing for publishing records.

The Hydra Boosters consist of: (*i*) Hydra Head DHT nodes, comprising thousands of virtual nodes in the DHT, such that every other peer has at least one Hydra Head in its routing table; and (*ii*) A shared Hydra Database that stores all provider records in the DHT and can be accessed by all Hydra Heads. This allows Hydra Boosters to serve other peers with the provider records immediately through the directly connected Hydra Heads, thereby improving content retrieval performance. Our key idea is that it does not require IPFS itself to adopt any infrastructure that is hard-coded into the protocol, or controlled by a few parties. Instead, Hydra Head nodes operate at the same level as any other node in the network, without any privileged status, making it a complement to the regular content routing operation.

Hydra Heads. The Hydra Boosters introduce the concept of Hydra Heads. Each head is perceived by other network participants as a conventional, distinct peer without special privileges. Each head has its own unique PeerID, routing table, and peer store. However, these heads are strategically positioned within the Kademlia XOR keyspace. The strategic positioning strives to make sure every peer has at least one Hydra Head within its 20-peer XOR proximity. If this goal is achieved, every provider record should be stored on at least one Hydra Head. Thus, giving the Hydra Boosters full vantage on all provider records. To further accelerate performance, records are stored in a distributed database shared across *all* Hydra Heads. As a result, any DHT walk to find a provider record will have a high chance of immediately getting the result from the Hydra database through the nearest Hydra Head in the routing table.

Peer ID Assignment. The main design challenge is to achieve this optimal placement. To address this, it is necessary to determine the total number of participating network peers and generate an appropriate set of peer identities to cover the entire keyspace. Initially, we considered relying on randomly generated peer identities, but in practice, this resulted in an uneven distribution. To mitigate this issue, we rely on the "power of two choices" [31]. This relies on a biased random algorithm and has been demonstrated to be effective at balancing loads when each load balancer has an incomplete or delayed view. First, all Hydra Head PeerIDs are tracked in an XOR trie. For each new head's PeerID, two choices are generated. The one that decreases the trie's depth the least is selected. This approach ensures an average depth of $\log N$, with a maximum depth of $\log N + \log\log N$. Importantly, the trie's depth correlates with proximity to existing nodes. If a new Hydra head node were inserted at depth D, then its closest node is at a distance 2^{-D}. Having all nodes at similar depths means they are implicitly equidistant in the XOR metric.

Hydra Database. The Hydra Booster system stores recent records in a shared Amazon DynamoDB key-value store. All heads can access this database, allowing every individual head to have full vantage on all provider records known by any head. Importantly, recall our earlier distinction between decentralized and distributed. Although DynamoDB is distributed, it sits under the control of Protocol Labs. Note, the Hydra database is not related to the peer routing functionality of Hydra heads. Thus, even if DynamoDB fails, the heads can continue to assist with peer routing similarly to any other node (although they cannot directly serve provider records).

3.3 HTTP Gateways

The third challenge of decentralization in IPFS pertains to adoption, particularly when compared to the simplicity of browser-based access using HTTP. Accessing IPFS content necessitates running an IPFS node, which requires technical expertise. Further, due to the computational cost, the implementation is currently not compatible with mobile systems. Additionally, due to the use of incompatible protocols, integrating IPFS with conventional (HTML/HTTP) websites is challenging. In part, this is also because IPFS experiences longer retrieval delays than traditional client-server HTTP, creating usability challenges for composing a webpage from both HTTP and IPFS-hosted objects.

Overview of Gateways. To address this issue, we have deployed a small set of gateway servers. A gateway works as a simple HTTP bridge into the IPFS network. We implement this as a Nginx web server front end, co-located with a full IPFS node installation. Clients interact with a gateway by issuing a `GET` request using the CID as a URL parameter. The gateway then retrieves the requested object from the IPFS network on behalf of the client before returning it via the HTTP connection. Note, as our software is open-source, any entity can deploy their own gateway. Indeed, other organizations such as Cloudflare have done precisely that.

Caching. To exploit the centralized aggregation of demand, and improve retrieval performance, we also enable caching on the gateways. Specifically, we use Nginx's default web cache with a Least Recently Used replacement strategy. Thus, any requested objects are cached within the default HTTP cache of the gateways. The gateways also allow authorized third parties to actively push content objects into its local object stores (*i.e.,* its node cache). To date, this is used by NFT.storage and Web3.storage. Web3.storage stores over 40M objects, while NFT.storage stores over 91M objects. This serves as a significant benefit of centralization: By aggregating user requests within the gateways, we can serve popular objects from the cache, avoiding repeatedly retrieving the same content from the IPFS network (as the gateway only forwards a request when it is not stored or cached locally). Alongside improving performance for the client, it also offloads unnecessary requests from the remainder of the network.

4 Evaluation Methodology & Data

To explore the efficacy of the above centralized components and their impact on IPFS's decentralization efforts, we rely on operational data collected from the components managed by Protocol Labs. See Appendix A for ethics discussion.

4.1 InterPlanetary Network Indexer Data

The below IPNI datasets are obtained from the Indexer managed by Protocol Labs (`https://cid.contact`).

Providers and Provider Records. Our first dataset covers the indexed records in the Indexer's data storage. As of `2023-04-24 UTC` , the dataset consists of 173,998,039,712 (approximately 0.17 trillion) provider records.

Index Ingestion Performance Data. Our second dataset covers the performance of our Indexer. We collect data from the operational logs of the Indexer. Because the Indexer processes approximately 5.3 billion provider records per day, we take a sample of advertisement synchronization operations covering 16,757,817 provider records on `2023-04-24`.

Index Query Performance Data. Our third dataset covers the Indexer query performance. The Indexer employs Amazon CloudFront. Our dataset comprises HTTP requests made to Amazon CloudFront, along with cache hit information and the request complete time. The dataset covers a period of one day, from `2023-04-18 09:05 UTC` to `2023-04-19 09:05 UTC`. During this time, 62,611,156 successful requests were made, and 246.77 GB of traffic was served.

4.2 Hydra Boosters Data

Protocol Labs has deployed 135 Hydra Booster nodes on Amazon ECS in the `us-east-1` region. Each of these Hydras has between 10 and 15 heads, resulting in a total of 2,015 Hydra heads.

Active Retrievals with Hydra Boosters. To evaluate the performance benefit of the Hydra Boosters, we perform a set of active measurements with the Hydra Boosters enabled. We utilize six machines that are located in different regions on Amazon AWS (see Table 1). On each machine, we run an IPFS DHT server node (Kubo implementation v0.16.0).

In each iteration, we randomly select a single node to publish a new 0.5 MB object (CID) to the network. Subsequently, all other nodes retrieve that object. Once all the nodes complete this process, they disconnect to prevent the next retrieval operation from being resolved through Bitswap. It is important to note that this is the closest one can get to a controlled test in the public IPFS network.

Active Retrievals without Hydra Boosters. To evaluate the benefit of Hydra Boosters, it is also necessary to perform the above measurements *without* the Hydra Boosters (as a baseline). Therefore, we repeat the above active measurements, whilst performing a *controlled deactivation* of Hydra Boosters. Specifically, we unplug the common database from the Hydra Boosters for a period of 7 days. We leave Hydra Heads in the network but configure them to behave differently:

- ADD-PROVIDER queries are ignored. Since Hydra Heads make about 10-15% of the network, this means we are artificially decreasing the k-replication to 19 [30].
- GET-PROVIDER queries are answered with only closer peers (*i.e.,* no provider records) because the common database is not there anymore.

The active retrieval experiment starts on `2022-11-25 15:27`

UTC to 2022-12-01 17:30 UTC. We have deployed the change of unplugging the database from the Hydra Boosters on 2022-12-01 17:30 UTC, thus the active retrieval experiment without Hydra starts on 2022-12-01 17:30 UTC to 2022-12-08 20:00 UTC. This gives us approximately 6 days of data with the Hydra Booster database, and 7 days without it. Table 1 summarizes the number of retrievals from each region before and after unplugging the database.

	Retrieval		Publication	
	with	without	with	without
af_south_1	9590	12084	2579	3284
ap_southeast_2	10375	13143	2579	3282
eu_central_1	10180	13233	2578	3289
me_south_1	10183	12814	2579	3280
sa_east_1	10192	12904	2578	3291
us_west_1	10162	12821	2579	3286

Table 1: The number of retrievals and publications from each region with and without the Hydra Boosters database.

4.3 HTTP Gateway Data

Finally, we gather a dataset from the public IPFS gateway (https://ipfs.io) managed by Protocol Labs. The dataset comprises of all HTTP GET requests made on 2023-03-15 UTC+0. Each record in the dataset represents the request and response details, covering remote IP address, request timestamp, user agent, HTTP referrer, request complete time, response size, response latency, and cache hit/miss information. In total, the dataset covers 69,339,954 successful requests from 1,645,611 IP addresses, with response sizes equalling 91,869 GB.

5 Evaluation

We now assess the effectiveness of our centralized components in addressing the challenges outlined in Section 2.2. We focus on (*i*) The effectiveness of the Indexers in supporting massive content publication; (*ii*) The effectiveness of the Indexers, the Hydra Boosters, and gateways in improving content retrieval performance; (*iii*) The effectiveness of the gateways in increasing the adoption of IPFS.

5.1 Challenge 1: Massive Content Publication

The first challenge of IPFS's decentralization pertains to the massive publication of objects. This is caused by the need to traverse many decentralized routing hops and replicate provider records to mitigate churn. To overcome this, we have introduced the Indexer, a centralized key store. Therefore, we start by evaluating the efficacy of our centralized components in improving the publication process.

Indexer Adoption. We first investigate the adoption of Indexers by content providers, by checking the number of provider records that have been uploaded to our Indexer server. Figure 4a illustrates the total number of provider records in the Indexer and IPFS DHT, alongside the number of provider records offered by the top 100 content providers in both sub-systems. We get the DHT data from our Hydra Booster database, as discussed in §3.2. The Indexer stores 173,998,039,71 provider records, over 100x more than the number stored in DHT. Additionally, we observe that the top providers in the Indexer offer 100-1000x more provider records than the top providers in the DHT, with most of them hosting around 10^9 provider records. Initially, one might assume that the Indexers have therefore eclipsed the DHT, and created a fully centralized environment. Closer inspection reveals a more nuanced situation though. Although the Indexers host many records, these are uploaded by just 604 major providers, with most of them running NFT or storage services. In contrast, 56k providers use the DHT, confirming the majority still choose to use the DHT. This confirms that large content providers *do* see benefit in using the Indexer system, while outside of these large providers, the remaining IPFS users continue to publish via the DHT.

We further examine the adoption of Indexers by retrieval clients. Our data shows that the Indexer receives 2.5k lookup requests per second, while the IPFS DHT receives around 5k (estimated using the number of GET-PROVIDER requests received by Hydra Boosters). These statistics confirm that, although the DHT plays a larger role, the Indexers successfully support massive content publication activities by this small number of major publishers. Overall, the results indicate that the Indexer is fulfilling its intended purpose of supporting massive content publications from large content providers and that clients do have interest in such content.

Publication Performance. The above confirms the adoption. We next assess if the Indexer indeed improves content publication performance. Figure 4b plots the CDF of advertisement synchronization time for the Indexer and compares it with the DHT publication time. We see that the advertisement synchronization time is faster than the DHT publication time at all percentiles. It is approximately twice as fast at the 50th percentile. Unsurprisingly, this confirms that the performance of the centralized Indexer is faster than the DHT publication.

However, an advertisement can contain multiple provider records (see §3.1). This implies that the time taken by the Indexer to index each provider record may be significantly shorter than what is depicted in Figure 4b. To explore this, Figure 4c presents the relationship between the synchronization time of advertisements and the number of provider records in the advertisement. Indeed, we find that the majority of advertisements contain 16K to 20K provider records (compared to 1 record for the DHT publications). Additionally, the synchronization time of advertisements is almost agnostic to the number of provider records, confirming that the Indexer is well suited for bulk publication.

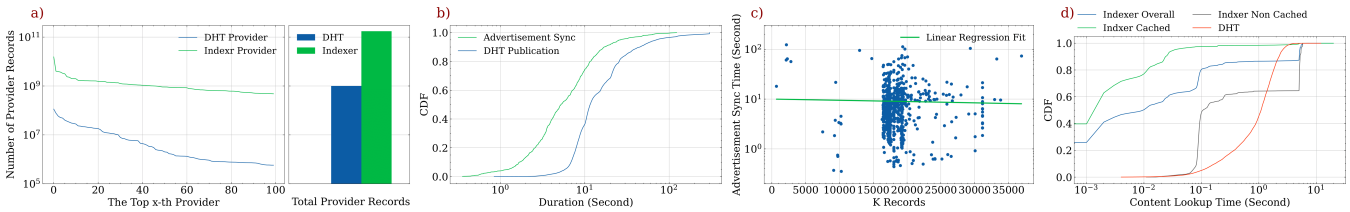


Figure 4: (a) The number of provider records offered by the top 100 content providers in the Indexer and IPFS DHT, alongside the total number of provider records in the Indexer and IPFS DHT; (b) CDF of the advertisement synchronization time for the Indexer, and the DHT publication time of an IPFS node for comparison; (c) Advertisement synchronization time vs. the number of provider records in the advertisement; (d) CDF of the content lookup time of the Indexer, and of the DHT for comparison.

Discussion & Takeaways. The above shows that Indexers effectively support massive content publication, with 604 large content providers hosting provider records 100x more than those in the DHT. Moreover, it offers a significantly faster indexing speed of approximately 500μs per provider record.

However, the Indexers also introduce centralization that brings certain trade-offs. Most noteworthy is the significant monetary cost of operating Indexers. In our experience, an Indexer server costs \$5–10K in capital expenditure, with approximately \$1K per month in operating expenditure. In contrast, IPFS's original publication mechanism (via the DHT) is designed such that the load is spread evenly (thereby removing the need for individuals to host expensive hardware). Moreover, the Indexer introduces a single point of failure as centralized failures in the Indexer can make significant volumes of provider records inaccessible. This is because some provider records are only stored within Indexers and is not replicated in IPFS DHT. We further discuss this in §6.

5.2 Challenge 2: Content Retrieval Performance

The second challenge of IPFS's decentralization pertains to content retrieval performance. This is driven by the need for clients to perform distributed lookups across (many) nodes. The Indexer, Hydra Boosters, and gateways are all geared towards addressing this, by complementing the distributed lookups and retrievals with centralized acceleration. We use the content lookup time as a metric to evaluate content retrieval performance. It refers to the duration between the point when the retrieval client launches a query and the point when the content is found. Across the three components, this metric corresponds to the time to get the first provider record in the DHT walk; the request completion time in an Indexer query; and the response latency in a gateway query.

Indexer Content Lookup Time. Figure 4d plots the CDF of the content lookup time of the Indexer. We also plot the content lookup time in a vanilla DHT lookup for comparison, as measured in our Hydra Booster experiments (§4.2). As expected, the Indexer's overall content lookup time is significantly shorter than the decentralized DHT lookup time. Our Indexer servers achieve approximately 10x better performance than the DHT lookup time at the 50th percentile.

One reason for the improvement is that the centralization allows us to exploit caching in the Indexer server (§3.1). On average, the cache hit rate is 65.22%. This means that the majority of queries hit the Indexer's cache, which is approximately 100x faster than the DHT lookup time in the 50th percentile. This is a significant benefit of centralization, as it allows us to aggregate demand into a small set of central points. Doing the same across our DHT is much less effective. We also note that, even for the 34.78% of requests that result in a cache miss, the content lookup times still outperform the DHT, as shown by the black line in Figure 4d. The performance around the 65th percentile is due to an optional cascading lookup which makes the Indexer query the IPFS DHT. Thus, even without the cache mechanism, the centralized Indexer still offers better content retrieval performance than the DHT.

Hydra Booster Lookup Time. The Hydra Boosters also improve content retrieval performance by shortening the DHT's provider record lookup. To evaluate this, recall that we performed a controlled experiment running IPFS both with and without the Hydra Booster active (see §4.2).

Figure 5a displays the content lookup time across all probed AWS regions, with and without the Hydra Booster database. We observe a clear performance improvement in `us-west-1`, `af-south-1`, `ap-southeast-2`, `sa-east-1`, and `me-south-1`, with the median improved by 36.5%, 25.1%, 16.4%, 11.7%, and 3.8% respectively. Although this is not as impactful as the Indexer speedups, it still constitutes a notable improvement. However, there is one outlier: the Hydra Booster database provides no performance improvements in `eu-central-1`. Our investigation suggests that this is because the retrieval performance in Europe is already optimized. As shown in Figure 5a, it is significantly better than in other regions, leaving limited room for further improvement. This suggests that the decentralization vs. centralization trade-off may differ based on region, and raises questions about whether such speed-ups warrant the high monetary cost.

We further investigate how these speedups are achieved. Recall, the Hydra Boosters improve performance by: (*i*) the strategic placement of stable Hydra Heads across the DHT

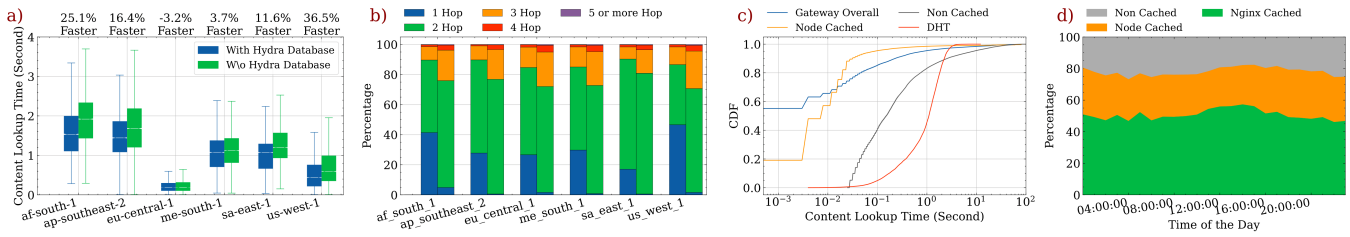


Figure 5: (a) The content lookup time across all probed AWS regions with and without the Hydra database, the values at the top refer to the medians; (b) Breakdown of the hop count per provider record lookup across all probed AWS regions with (left bar) and without the Hydra database (right bar); (c) CDF of the content lookup time of the Gateway and of the DHT for comparison; (d) Cache hit rate of the Gateway throughout the day, aggregated in 1-hour intervals.

hash space so that every peer has a Hydra Head in its 20-proximity; and (*ii*) the storage of provider records in the shared database, allowing many records to be retrieved within one hop. To confirm this, we examine the closest neighbors of each peer by taking a DHT routing table snapshot [40]. Confirming our expectation, we find a total of 16,208 peers in the network, with 96.6% having at least one Hydra Head in their 20-proximity. To quantify how this impacts path length, Figure 5b shows the hop count of DHT walks with and without the Hydra database. We separate the data into each AWS region. With the hydras, many DHT walks reach provider records within one hop. This is the case for 41.45% in `af-south-1`, 27.81% in `ap-southeast-2`, 26.68% in `eu-central-1`, 29.79% in `me-south-1`, 17.04% in `sa-east-1`, and 46.64% in `us-west-1`. In contrast, without the Hydra database, under 5% in `af-south-1` and 2% in other regions of queries achieve single hop routing, and about 20% of DHT walks require three or more hops to access the provider record in all regions.

Gateway Content Lookup Time. We finally inspect the gateway, which also improves retrieval performance by introducing centralized caching, circumventing the need to traverse the DHT or Indexers. Figure 5c displays a comparison between the gateway content lookup time for IPFS node-stored content and non-cached content. As a baseline, it also shows the DHT content lookup time when retrieving from an IPFS node, as obtained from our Hydra Booster data (§4.2). Note that Nginx cached content is not plotted separately because the latency is too short (near 0) to be plotted for comparison.

We see that introducing this centralized caching brings significant retrieval performance benefits. At the 50th percentile, the gateway achieves approximately 100x better performance than the DHT. This is because most requests hit either the Nginx cache or the node cache, as plotted in Figure 5d. On average, the Nginx cache hit rate is 50.3% and the average IPFS node storage hit rate is 27.6%. The gateway lookup time for cached objects is just 24 ms at the 95th percentile.

That said, there is little improvement for the remaining objects that are not cached. The non-cached content shows a similar trend to DHT lookup in Figure 5c. This is not surprising since, for non-cached objects, the gateway's co-located IPFS node also needs to perform a DHT lookup. On top of this, the co-located IPFS node needs to retrieve the content and then forward it to the retrieval client, thus introducing extra overhead (which causes the disparity between the non-cached line and the DHT line). This highlights the complexity of inter-linking these centralized components with the larger (slower) decentralized system.

Discussion & Takeaways. The centralized Indexer, Hydra Boosters, and gateways improve content retrieval performance from different perspectives. The Indexer offers better content lookup time in 90% of cases, while the gateway aggregates user demand to provide better content retrieval via caching, in 95% of cases compared to DHT lookup time. The Hydra Boosters reduce the number of hops during DHT lookups to achieve faster lookup times. Overall, deploying these components successfully improves content retrieval performance compared to our earlier fully decentralized design. That said, the introduction of these components leads to clear trade-offs, most notably between performance and privacy. As centralized instances, they allow operators to monitor and even censor usage. However, our experience and previous research [10, 37] has shown that these attacks are still possible in DHTs, regardless of whether Indexers or gateways are present. We further discuss this in §6.

5.3 Challenge 3: Adoption

Client Adoption. The gateways are our main mechanism to improve adoption, by providing direct integration with HTTP. Thus, we first inspect its success by measuring the number of gateway users as a metric of adoption. For this, we categorize all user-agents into three categories: desktop browsers, mobile browsers, and other net tools [9] or libraries such as *wget*. Access from both desktop browsers and mobile clients is a crucial metric for evaluating improvements in adoption because IPFS node is not yet implemented on mobile devices, and setting up an IPFS on a desktop can be complicated.

Figure 6a depicts the number of requests, number of users (we treat each IP address as an individual user), and total traf-

fic served for different user-agents. Figure 6b also presents a breakdown of the file types requested. Confirming uptake, 771,977 users access the gateway from mobile clients, 776,286 from desktop user-agents, and 195,854 using net tools. 31.8% of the total traffic and 18.8% of the total requests come from mobile devices, whereas 32.9% of the total traffic and 29.8% of the total requests originate from desktop browsers. This suggests that our gateway has successfully expanded access to IPFS.

For completeness, Figure 6b presents a breakdown of the file types requested. Note, `stream` refers to media streaming. We observe a wide range of object types. Excluding the unknown objects, for desktop agents, the most popular type of content is `stream` (20.47%) and `json` (12.99%). For mobile agents, the most accessed is `other` content (26.42%) followed by `web` content (14.57%). This confirms that the gateway has been adopted for various applications and content types.

Web Developer Adoption. To assess the adoption of the gateway by traditional websites, we look into the HTTP referer. This tells us from which website a request has been generated. We measure the number of requests with and without a referral, as well as the number of users and the total traffic served. The results show that 51.6% of the total traffic and 29.8% of the total requests come from a referral website, suggesting significant integration by web developers. This is similar to the percentage of traffic (48.3%) without a referral and lower than the requests (70.1%) without a referral. The requests with a referer come from 1,312,066 users, while the requests without a referer come from 405,087 users, which is more than 3x fewer, confirming that the majority of users use gateways due to integration with websites. This suggests that the gateway *has* supported web developer adoption, with the placement of IPFS-hosted objects in their pages.

We finally check the top 100 referral sites (based on the number of requests received) and manually label them with their website category. Figure 6c displays the request volumes among different referral types, the number of referral sites, total traffic served, and user count. Note, the IPFS type consists of IPFS official sites and tool sites such as the Gateway Checker [6]. We see that NFT and online video are the most common types of referral for the gateway, while other referral types only account for a small fraction. The former is likely because a large amount of NFT content is stored on IPFS, whereas the latter may simply be an effort to offload traffic from a developer's own server. Figure 6d also displays the top 100 referral sites, ranked by the number of requests and the number of users respectively. We observe a long-tail distribution, with 70.23% of the requests originating from just 23 websites. The findings arguably point to a level of additional centralization, with the majority of referred requests coming from a small number of top websites.

Discussion & Takeaways. The gateways enhance the adoption of IPFS for both clients and web developers, with a significant number of users accessing IPFS through our gateway on their mobile devices (which would not be possible without it). Additionally, over half of the traffic to our gateway is generated by referrals from existing HTML-based sites. The introduction of this centralization, therefore, has clear benefits. However, the use of such HTTP gateways does sacrifice the end-to-end cryptographic validation of IPFS content, which may enable man-in-the-middle attacks. Moreover, as centralized instances, gateways enable their operators to monitor usage, raising privacy concerns. We further discuss these concerns in §6. We also note that gateways may create other risks. For example, ideally, nodes who retrieve content also volunteer to serve it for others. This enables self-scaling, naturally creating more replicas of popular content. However, the use of the gateways prevents this and may risk an over-reliance on centralized elements. Thus, even though one can always circumvent any gateway, a lack of decentralized participants could hamper this in the longer term.

6 Discussion

The above solutions were introduced to mitigate challenges related to content publication, retrieval, and system adoption. This creates a trade-off, potentially undermining certain benefits of decentralization. We explore this below.

6.1 Security

Hydra Booster Security. One concern regarding the Hydra Boosters is that a centralized entity controlling multiple heads might theoretically gain excessive influence over IPFS routing, leading to issues such as result poisoning and Eclipse attacks [36]. However, we posit that these security concerns are unlikely to manifest. First, it is difficult for false content to be returned, as IPFS relies on self-validating CIDs. Additionally, while Hydra Boosters can supply the provider record, they do not serve the content itself. Thus, the most severe action Hydra Boosters could engage in is presenting a false provider record, forcing another query. This means that such an attack would only slow retrieval, rather than prevent it.

Second, as outlined in §3.2, the primary objective of the Hydra Boosters is to achieve a uniform distribution of Hydra heads. Our measurement (See Appendix B for details) shows that the mean number of hydra heads within a peer's 20-proximity is 2.36, and 99.5% peers have no more than five peers in the 20-proximity. Given the presence of just a few Hydra heads within 20-proximity, the ability to manipulate routing or execute eclipse attacks is constrained. This is because peers connect to not just the Hydra heads, but also and primarily to other peers. Thus, as long as the ordinary nodes in the IPFS network continue to function as intended, the routing of IPFS should remain resilient against potential disruption from Hydra Boosters. That said, the disruption would inevitably lead to a performance decline. The performance

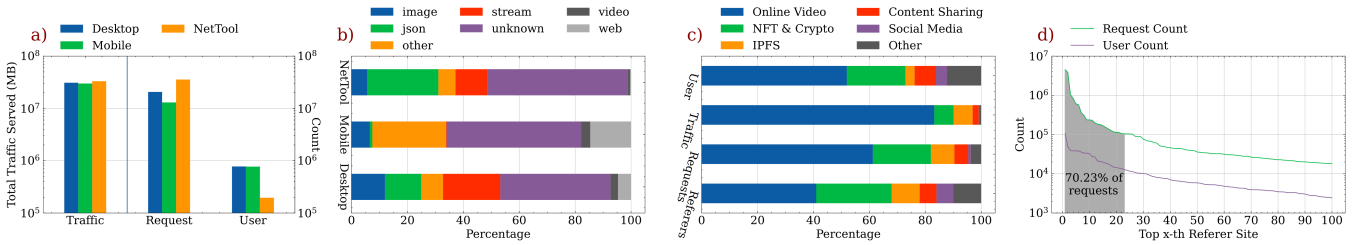


Figure 6: (a) Number of requests, number of unique users, and total traffic served for different user-agents; (b) Breakdown of requested file types for different user-agents; (c) Breakdown of referers for number of referral sites, the number of requests, total traffic served, and user count of the different referral types across the top 100 sites ranked by requests count; (d) The requests count and user count of the top 100 referer sites, ranked by request count and by user count, respectively.

impact, in this case, is similar to the situation where all Hydra Boosters are inactive, a topic we discuss further in §6.2.

Gateway Security. The HTTP gateway sacrifices the end-to-end cryptographic validation of content, which raises potential man-the-middle attacks that IPFS strives to prevent. To overcome this, retrieval client browsers could compute the hash of the retrieved file to verify whether they have received the correct content. However, this process entails the use of additional functionality in the browser, which may undermine the goal of improving adoption.

6.2 Robustness

Hydra Booster Robustness. The robustness of the Hydra Boosters could impact IPFS in two ways: (*i*) failing to return provider records (*e.g.,* due to the failure of the DynamoDB instance); or (*ii*) failing to correctly participate in IPFS routing (*e.g.,* due to failures on AWS EC2). Despite this, we emphasize that the failure of either component will result in IPFS peers falling back to the traditional decentralized model. Thus, although performance will suffer, availability will not. Confirming this, §5.2 demonstrated the scenario where Hydra Boosters stop returning provider records. This situation mirrors the conditions that arise when DynamoDB experiences a shutdown. Here, we observe a performance decrease by up to 36.5%, yet the content availability is not affected.

An additional risk is that the failure of the Hyrda Boosters will negatively impact routing. To test this, we repeat the methodology in §4.2, but prevent peers from utilizing any Hydra Boosters for routing (detailed in Appendix B). We do not observe any failed retrieval, indicating that the availability of provider records remains unaffected even when the peers do not interact with Hydra Boosters.

Indexer Robustness. When using the Indexers, the availability of provider records is entirely dependent on the availability of the Indexers. Unlike the DHT, which replicates provider records across 20 peers, centralized failures in the Indexer can make significant volumes of provider records inaccessible. This is mitigated by the ability of Indexers to gossip announcement messages among themselves. However, gossipsub (see §3.1) offers no hard guarantees on real-time synchronization. There are already 7 Indexers available for public use, and Protocol Labs is working on a protocol for automatic discovery of alternative Indexers. As more stakeholders operate Indexers, we hope the risk of Indexer unavailability decreases.

Briefly, it should be noted that the failure of an indexer does not necessarily result in a permanent loss of provider records. As explained in §3.1, Indexers obtain provider records from the advertisement chain of content providers. Therefore, provider records are inherently replicated by providers' advertisement chains. If an Indexer loses any provider records, it can restore them by re-synchronizing with the original providers, assuming such providers still wish to make their content available.

6.3 Privacy

Another concern is that, as centralized instances, these components may enable their operators to monitor usage. For instance, the Indexers and gateways can monitor the queried CIDs from the clients. Similarly, the Hydra Boosters can observe almost every DHT query in IPFS because most peers are directly connected to a Hydra Head. This asymmetry of power is a key concern and makes it necessary for clients to trust the operators with their data. We conjecture that this is one of the reasons why most IPFS clients use an instance of these components operated by Protocol Labs. Despite this, we acknowledge this is not a sustainable solution, with the risk of Protocol Labs becoming an overly central dependency. Although not discussed here, we also briefly note that the Indexers use a double-hashing technique to make this monitoring more difficult [5].

Our experiences within IPFS have also flagged that many of these attacks are possible in the decentralized DHT routing subsystem, agnostic to the presence of Indexers or gateways. For instance, a well-placed node in the DHT can learn which files are requested by a client during the routing process. By performing a Sybil attack, a well-resourced adversary could even gain global vantage on the routing process. Recent works

also show that privacy attacks [10] and censorship [37] are possible in the purely decentralized IPFS DHT. Despite that, it is clear that greater decentralization reduces the vantage of individual nodes, raising the barrier to such attacks.

6.4 Incentives

In previous subsections, we mention that issues with robustness can be mitigated by multiple stakeholders running these centralized components. This, however, requires a clear incentive model to encourage third-party operators to contribute their resources. To date, there are 7 Indexers and over 80 Gateways, and the stakeholders running them typically have three types of incentives: (*i*) Stakeholders that provide storage services utilizing IPFS can benefit from running Indexers and/or Gateways. These ease the discovery of their stored content, broaden adoption, and improve their service's performance. Such stakeholders include NFT.storage, Web3.storage, DSS, SXX, FilSwan, PikNik, Ken Labs, etc. (*ii*) Stakeholders that provide other applications built on top of IPFS benefit from running an Indexer and/or a Gateway to improve the stability and performance of their service. Such stakeholders include LeewayHertz, 4everland, Aragon, Pinata, etc. (*iii*) Stakeholders that have a broader interest in decentralized web applications have a motivation to contribute to the environment as it aligns with their long-term interests. Such stakeholders include CloudFlare and Infura.

We acknowledge that none of the aforementioned reasons provide formal incentives by design. It is yet to be determined how wide uptake is among third-party stakeholders without a formal incentive model in place. Our future plans involve implementing incentive mechanisms, and there are several potential options available [24].

7 Related Work

Decentralized Web. The IPFS network has grown alongside other Decentralized Web technologies, particularly the "fediverse". This is composed of server-based federated services, such as Mastodon [32], Pleroma [23], and Diaspora [22]. The service closest to IPFS is Nextcloud, which offers a federated file storage platform that integrates IPFS with server-local storage [1]. This functions similarly to IPFS gateways. However, IPFS can operate without gateways, while fediverse apps rely entirely on the uptime of federated servers [32].

P2P Networks. There have been numerous P2P overlay architectures, with dozens of DHT structures proposed, including Chord [38], Tapestry [44], Koorde [26], Pastry [34], and others [25]. Various applications have been built atop, such as large-scale content delivery platforms [14] and decentralized social networks [21], among others. Rather than creating an entirely new system, IPFS uses the Kademlia DHT for content indexing [30]. IPFS is built upon these technologies and is currently one of the largest "Decentralized Web" technologies deployed in the world. BitTorrent, which also uses Kademlia [14], is another example of a large-scale deployment. Note, as part of our decentralization efforts, IPFS aims to be resistant to censorship, much like other platforms such as Freenet [12] and Wuala [28]. These platforms achieve their goal by storing encrypted content across a random subset of peers. In contrast, IPFS follows a BitTorrent-like approach where nodes store only the content they are interested in.

DHT Optimization. Various attempts have been made to enhance the performance and usability of DHTs by employing caching [35], network-aware peer selection [27], and parallelizing lookups [39]. We also take inspiration from prior Information-Centric Networking designs that use DHTs for content indexing [17, 18]. We build on these prior works with our deployment of hydra boosters. These improve performance while offloading traffic from the DHT.

P2P System Evaluations. Closest to our evaluation are studies that measure operational IPFS [41]. Moreover, in [20], the authors measure DHT lookup latencies in the range of tens of seconds, but it was not a controlled experiment. There have been various performance evaluations of P2P systems. For example, [16] and [43] evaluate BitTorrent's implementation of Kademlia, to find a significant number of failed nodes that adversely affected the lookup times. Further, Stutzbach and Rejaie [39] model Kademlia's performance and propose several improvements. We borrow from these measurement studies to inform our methodologies in §4.

8 Conclusion

This paper has presented our experiences of deploying three centralized components within IPFS, namely Indexers, Hydra Boosters, and Gateways. Using real-world operational data, we have demonstrated their effectiveness in addressing the challenges faced by IPFS: massive content publication, content retrieval performance, and adoption. We have also discussed trade-offs, related to security, privacy, and other risks caused by the compromises of prior decentralization. Overall, our findings highlight the persistent challenges that exist in deploying operational decentralized systems with tight performance constraints. Equally, we have highlighted the challenges of overcoming traditional issues with centralized deployments (*e.g.,* single points of failure and privacy). Thus, we emphasize that developers must make judgment calls about how to balance such trade-offs. In the future, we plan to explore the remaining challenges discussed in §6. We will continue working on the research and development of decentralized protocols, and we hope that IPFS can eventually perform similarly even without these centralized parts.

References

[1] IPFS for Nextcloud. https://apps.nextcloud.com/, 2020.

[2] Ipfs ecosystem directory. https://ecosystem.ipfs.io/, 2022.

[3] Bitswap. https://docs.ipfs.tech/concepts/bitswap/, 2023.

[4] Internet traffic from mobile devices (apr 2023). https://explodingtopics.com/blog/mobile-internet-traffic, 2023.

[5] The ipfs dht reader privacy upgrade. https://discuss.ipfs.tech/t/the-ipfs-dht-reader-privacy-upgrade/16279, 2023.

[6] Ipfs gateway checker. https://ipfs.github.io/public-gateway-checker/, 2023.

[7] Multihash. https://github.com/multiformats/multihash, 2023.

[8] Pebble database. https://github.com/cockroachdb/pebble, 2023.

[9] User-agent net tool ua strings. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers, 2023.

[10] BALDUF, L., HENNINGSEN, S. A., FLORIAN, M., RUST, S., AND SCHEUERMANN, B. Monitoring data requests in decentralized data storage systems: A case study of IPFS. *CoRR abs/2104.09202* (2021).

[11] BOMMELAER DE LEUSSE, C., AND GAHNBERG, C. The global internet report: Consolidation in the internet economy. *Internet Society* (2019).

[12] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. Freenet: A distributed anonymous information storage and retrieval system. In *Designing privacy enhancing technologies* (2001), Springer.

[13] CLAY, K. Amazon.com Goes Down, Loses $66,240 Per Minute, 2013.

[14] COHEN, B. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems* (2003), vol. 6, Berkeley, CA, USA, pp. 68–72.

[15] COLDEWEY, D. Cloudflare dns goes down, taking a large piece of the internet with it, 7 2020.

[16] CROSBY, S. A., AND WALLACH, D. S. An analysis of bittorrent's two kademlia-based dhts. Tech. rep., Rice Technical Report, 2007.

[17] DANNEWITZ, C., D'AMBROSIO, M., AND VERCELLONE, V. Hierarchical dht-based name resolution for information-centric networks. *Computer Communications 36*, 7 (2013), 736–749.

[18] DANNEWITZ, C., KUTSCHER, D., OHLMAN, B., FARRELL, S., AHLGREN, B., AND KARL, H. Network of information (netinf)–an information-centric networking architecture. *Computer Communications 36*, 7 (2013), 721–735.

[19] DOAN, T. V., VAN RIJSWIJK-DEIJ, R., HOHLFELD, O., AND BAJPAI, V. An empirical view on consolidation of the web. *ACM Transactions on Internet Technology (TOIT) 22*, 3 (2022), 1–30.

[20] FALKNER, J., PIATEK, M., JOHN, J. P., KRISHNAMURTHY, A., AND ANDERSON, T. Profiling a million user dht. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement* (New York, NY, USA, 2007), IMC '07, Association for Computing Machinery, p. 129–134.

[21] GRAFFI, K., GROSS, C., STINGL, D., HARTUNG, D., KOVACEVIC, A., AND STEINMETZ, R. LifeSocial. KOM: A secure and P2P-based solution for online social networks. In *CCNC* (2011).

[22] GUIDI, B., CONTI, M., PASSARELLA, A., AND RICCI, L. Managing social contents in Decentralized Online Social Networks: A survey. *Online Social Networks and Media 7* (2018).

[23] HASSAN, A. I., RAMAN, A., CASTRO, I., ZIA, H. B., DE CRISTOFARO, E., SASTRY, N., AND TYSON, G. Exploring content moderation in the decentralised web: The pleroma case. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies* (2021), pp. 328–335.

[24] IHLE, C., TRAUTWEIN, D., SCHUBOTZ, M., MEUSCHKE, N., AND GIPP, B. Incentive mechanisms in peer-to-peer networks – a systematic literature review. *ACM Computing Surveys* (Jan. 2023).

[25] ISDAL, T., PIATEK, M., KRISHNAMURTHY, A., AND ANDERSON, T. Privacy-preserving p2p data sharing with oneswarm. In *Proceedings of the ACM SIGCOMM 2010 Conference* (New York, NY, USA, 2010), SIGCOMM '10, Association for Computing Machinery, p. 111–122.

[26] KAASHOEK, M. F., AND KARGER, D. R. Koorde: A simple degree-optimal distributed hash table. In *International Workshop on Peer-to-Peer Systems* (2003), Springer, pp. 98–107.

[27] KAUNE, S., PUSSEP, K., LENG, C., KOVACEVIC, A., TYSON, G., AND STEINMETZ, R. Modelling the internet delay space based on geographical locations. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing* (2009), IEEE, pp. 301–310.

[28] MAGER, T., BIERSACK, E., AND MICHIARDI, P. A measurement study of the wuala on-line storage service. In *2012 IEEE 12th International Conference on Peer-to-Peer Computing (P2P)* (2012), IEEE.

[29] MATHEWS, E. Amazon cloud outage hits major websites, streaming apps, 12 2021.

[30] MAYMOUNKOV, P., AND MAZIERES, D. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems: First InternationalWorkshop, IPTPS 2002 Cambridge, MA, USA, March 7–8, 2002 Revised Papers*. Springer, 2002, pp. 53–65.

[31] MITZENMACHER, M. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst. 12*, 10 (oct 2001), 1094–1104.

[32] RAMAN, A., JOGLEKAR, S., CRISTOFARO, E. D., SASTRY, N., AND TYSON, G. Challenges in the decentralised web: The mastodon case. In *Proceedings of the Internet Measurement Conference* (2019), pp. 217–229.

[33] ROSEMAIN, M., AND SATTER, R. Millions of websites offline after fire at french cloud services firm, 3 2021.

[34] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing* (2001), Springer, pp. 329–350.

[35] SALEH, O., AND HEFEEDA, M. Modeling and caching of peer-to-peer traffic. In *Proceedings of the 2006 IEEE International Conference on Network Protocols* (Nov. 2006), IEEE.

[36] SINGH, A., CASTRO, M., DRUSCHEL, P., AND ROWSTRON, A. Defending against eclipse attacks on overlay networks. p. 21.

[37] SRIDHAR, S., ASCIGIL, O., KEIZER, N., GENON, F., PIERRE, S., PSARAS, Y., RIVIÈRE, E., AND KRÓL, M. Content censorship in the interplanetary file system. *arXiv preprint arXiv:2307.12212* (2023).

[38] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review 31*, 4 (2001), 149–160.

[39] STUTZBACH, D., AND REJAIE, R. Improving lookup performance over a widely-deployed dht. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications* (2006), IEEE, pp. 1–12.

[40] TRAUTWEIN, D. Nebula – A crawler for networks based on the libp2p DHT implementation, 7 2021.

[41] TRAUTWEIN, D., RAMAN, A., TYSON, G., CASTRO, I., SCOTT, W., SCHUBOTZ, M., GIPP, B., AND PSARAS, Y. Design and evaluation of ipfs: A storage layer for the decentralized web. SIGCOMM '22, Association for Computing Machinery, p. 739–752.

[42] VYZOVITIS, D., NAPORA, Y., MCCORMICK, D., DIAS, D., AND PSARAS, Y. Gossipsub: Attack-resilient message propagation in the filecoin and eth2.0 networks, 2020.

[43] WOLCHOK, S., AND HALDERMAN, J. A. Crawling BitTorrent DHTs for fun and profit. In *4th USENIX Workshop on Offensive Technologies (WOOT 10)* (Washington, DC, Aug. 2010), USENIX Association.

[44] ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., AND KUBIATOWICZ, J. D. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications* 22, 1 (2004), 41–53.

A Ethics

All data used within this paper is collected as part of Protocol Lab's operational activities. Although we observe IP addresses, we do not attempt to map these back to personal identities, as such analysis is not within the scope of this study. The IPFS gateway data also contains user information, as it covers requests from web clients. In all cases, we anonymize IP addresses, and do not perform lookups on the CIDs to infer the nature of the content exchanged. We perform no per-user analysis, and focus only on overall system analysis. All information is collected as part of our routine operations, and in line with IPFS policies. This paper has not triggered additional data collection.

B Additional Experiments

B.1 Hydra Heads in 20-Proximity

We examine the closest neighbors of each peer by taking a DHT routing table snapshot [40]. Figure 7 shows the CDF of the number of hydra heads in a peer's 20-proximity.

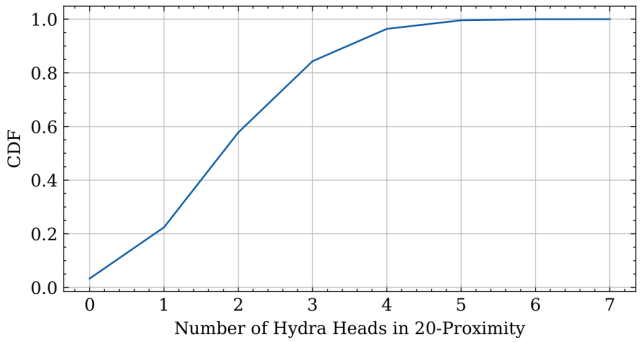


Figure 7: CDF of the number of hydra heads in a peer's 20-proximity.

B.2 Active Retrieval Ignoring Hydra Boosters

We conduct another experiment using the same methodology as described in §4.2. However, in this experiment, we utilize modified IPFS nodes that disregards all Hydra heads. This modification incurs minimal overhead since we possess a list of the PeerIDs of all Hydra heads. This simulation mirrors the conditions where a peer choose not to interact with any Hydra head. The experiment starts on `2022-11-28 16:17 UTC` to `2022-12-09 20:00 UTC`. Table 2 list the number of retrievals and publications from each region with modified IPFS nodes that ignore Hydra Boosters. Figure 8 shows the performance comparison by plotting the content lookup time across all probed AWS regions with and ignoring the Hydra database.

	Retrieval	Publication
af_south_1	7533	1674
ap_southeast_2	7800	1674
eu_central_1	8003	1677
me_south_1	7578	1674
sa_east_1	7529	1675
us_west_1	7886	1571

Table 2: The number of retrievals and publications from each region with modified IPFS nodes that ignore Hydra Boosters.

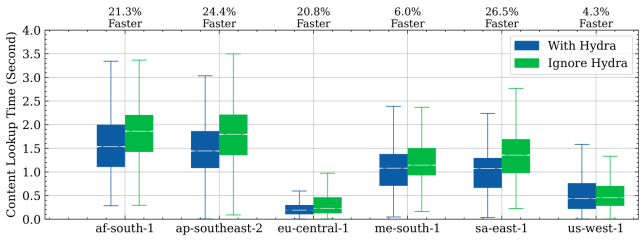


Figure 8: The content lookup time across all probed AWS regions with and ignoring the Hydra database, the values at the top refer to the medians.

BBQ: A Fast and Scalable Integer Priority Queue for Hardware Packet Scheduling

Nirav Atre, Hugo Sadok, Justine Sherry
Carnegie Mellon University

Abstract

The need for fairness, strong isolation, and fine-grained control over network traffic in multi-tenant cloud settings has engendered a rich literature on packet scheduling in switches and programmable hardware. Recent proposals for hardware scheduling primitives (e.g., PIFO, PIEO, BMW-Tree) have enabled run-time programmable packet schedulers, considerably expanding the suite of scheduling policies that can be applied to network traffic. However, no existing solution can be practically *deployed* on modern switches and NICs because they either do not scale to the number of elements required by these devices or fail to deliver good throughput, thus requiring an impractical number of replicas.

In this work, we ask: is it possible to achieve priority packet scheduling at line-rate while supporting a large number of flows? Our key insight is to leverage a scheduling primitive used previously in software – called Hierarchical Find First Set – and port this to a highly pipeline-parallel hardware design. We present the architecture and implementation of the Bitmapped Bucket Queue (BBQ), a hardware-based integer priority queue that supports a wide range of scheduling policies (via a PIFO-like abstraction). BBQ, for the first time, supports hundreds of thousands of concurrent flows while guaranteeing 100 Gbps line rate (148.8 Mpps) on FPGAs and 1 Tbps (1,488 Mpps) line rate on ASICs. We demonstrate this by implementing BBQ on a commodity FPGA where it is capable of supporting over 100K flows and 32K priorities at 300 MHz, 3× the packet rate of similar hardware priority queue designs. On ASIC, we can synthesize 100K elements at 3.1 GHz using a 7nm process.

1 Introduction

Packet scheduling, the problem of deciding what *order* and/or *time* network packets ought to be served or transmitted, has long occupied a prominent position in networking literature. Prior work in this space has resulted in a rich repertoire of packet scheduling algorithms with a variety of different properties: fairness and starvation avoidance [9], attack resilience [7], burst reduction [34], optimal flow completion time [4], *etc.* At the heart of many of these algorithms is a *priority queue* data-structure, which allows the scheduler to map packets' relative order (or scheduling time) to unique priorities, sort them, and subsequently extract the highest-priority item (i.e., the next packet to schedule) from the queue.

Despite these strong theoretical foundations, network switches and NICs have historically failed to provide anything beyond a small suite of simple scheduling algorithms [41]. The key problem is the *complexity associated with implementing a fast, scalable, and generic priority queue in hardware*. PIFO [41] was a foundational paper in articulating the importance of priority queueing and providing a practical implementation of a hardware priority queue. Yet, as we will discuss further in §2, PIFO falls short of achieving suitable performance for deployment either on an ASIC (as in a traditional switch) or on an FPGA (as in modern SmartNICs). Today, line rates run at 100+ Gbps, and comprise tens or even hundreds of thousands of concurrent flows, but PIFO can support at most 2048 concurrent flows while guaranteeing line-rate processing.

This performance bottleneck arises because implementing an exact priority queue involves comparison-based sorting a sequence of n arbitrary elements, which imposes a theoretical lower bound of $\Omega(n \log n)$ on the required number of comparator operations. While PIFO reduces this complexity to $O(n)$ by observing that priority can be determined at enqueue time, and the spatial nature of switch or NIC hardware allows this computation to be parallelized (i.e., by unrolling the operations in *space* instead of *time*), spatial parallelism only scales so far: as the number of comparator operations increases, so does the complexity of the resulting datapath circuit, causing the maximum operating frequency of the switch or NIC to drop dramatically. Although prior work has since improved upon PIFO using more sophisticated parallelization techniques [40, 47]), they are nonetheless limited in terms of performance, scalability, or both. Moreover, many of these solutions give up logical partitioning, a key feature of PIFO that is essential for practical switch deployment. For instance, the state-of-the-art priority queue design, BMW-Tree [47], would require 1,056 replicas in order to support a 32×400 Gbps output-queued switch (§2.1.1)!

In this paper, we ask: *is it possible to achieve priority packet scheduling on NICs and switches at state-of-the-art line rates with 100K+ flows without sacrificing accuracy or the expressiveness that PIFO offers?*

As we look to build a highly performant and scalable priority queue for hardware packet scheduling, we are inspired by data structures with constant worst-case time complexity. In particular, we find that *integer priority queues* (IPQs) are well suited to the task: they subvert the complexity barrier imposed by comparison-based sorting, so performance and scalability are no longer at odds. Further, there should be no loss in precision so long as the relative ordering of packets can be encoded in the priority span of the IPQ.

In this work, we present the design and implementation

of the *Bitmapped Bucket Queue* (BBQ),[1] a scalable, high-performance IPQ for hardware packet scheduling. The data structure underlying BBQ is a Hierarchical Find-First Set (HFFS) queue [38, 45], which guarantees constant worst-case time complexity for standard heap operations. While HFFS is a well-known data structure that has found applications in several software systems,[2] our key insight is that *HFFS queueing is amenable to a highly efficient, fully-pipelined hardware implementation*. As a consequence, BBQ enables, for the first time, packet scheduling at 100 Gbps line rate using minimum-sized packets on a commodity FPGA-based SmartNIC, and can be incorporated into state-of-the-art 32 × 400 Gbps switches [33] with as little as 12 replicas.

Although efficient pipeline parallelism is key to BBQ's high performance and scalability, incorporating this parallelism introduced new challenges in avoiding data hazards (parallel reads and writes to the same data) and correctness. As we will discuss in §4.2, hazards manifested in our design in three ways and required careful design to disentangle parallel access to shared data *without* sacrificing performance. Nonetheless, BBQ ultimately makes one sacrifice to correctness in that it cannot support back-to-back dequeues of packets from the same flow; in §5 we show that by combining BBQ with a tiny PIFO instance we can guarantee the *performance* of BBQ with the *correctness* of PIFO.

The rest of this paper is organized as follows. In §2 we provide background and discuss the challenges of incorporating a programmable scheduler into modern switches and NICs. In §3, we provide an overview of the BBQ design, followed by a more detailed description of the architecture in §4, focusing on the challenges due to parallelism and hazards. In §5, we explore an augmentation to the BBQ design that counteracts the latency artifacts introduced by pipelining. In §6 we evaluate BBQ, followed by a discussion on how to incorporate the design into modern switches and NICs in §7. We then describe the related work in §8, discuss limitations and future work in §9, and conclude in §10.

2 Background and Motivation

More than ever, there is a need for programmable packet scheduling in hardware. Switches have long offered a limited set of packet scheduling algorithms and NICs are increasingly taking over dataplane tasks traditionally performed in software [37], including end-host packet scheduling [26, 34, 42]. In the case of switches, having a programmable packet scheduler could not only vastly expand the catalog of scheduling algorithms available to network operators, but also pave the way for faster innovation and customization [11, 41]. With NICs, system administrators already *expect* to be able to customize the packet scheduler that runs on the end host [37].

[1] Available at https://github.com/cmu-snap/BBQ.

[2] The Linux kernel uses FFS-based priority queueing for process scheduling. Eiffel [38] demonstrates how HFFS queueing can be used to realize high-performance *software* packet scheduling, and [45] even gives a sketch for a hardware design.

2.1 Lack of Support for Emerging Use-Cases

PIFO [41] is a seminal work in this regard. It makes the observation that, when considering a single node [28], all scheduling algorithms can be expressed in how they make two decisions: *which* packet to schedule next and *when* to schedule it. The authors observe that for many scheduling algorithms this behavior can be captured simply with a hardware priority queue. This priority queue can be used in one of two ways: it can implement work conserving algorithms by sorting flows by rank, or it can implement non-working conserving algorithms by sorting flows by scheduling time. While PIFO was initially conceived to run on switches, it has also been shown to be a useful primitive to schedule packets on NICs [26, 42].

Unfortunately, existing solutions for programmable packet scheduling in hardware fail to meet the requirements for both state-of-the-art NICs and switches. In what follows, we elaborate on these two use cases, highlighting how their stringent performance requirements are at odds with existing proposals for programmable hardware packet schedulers.

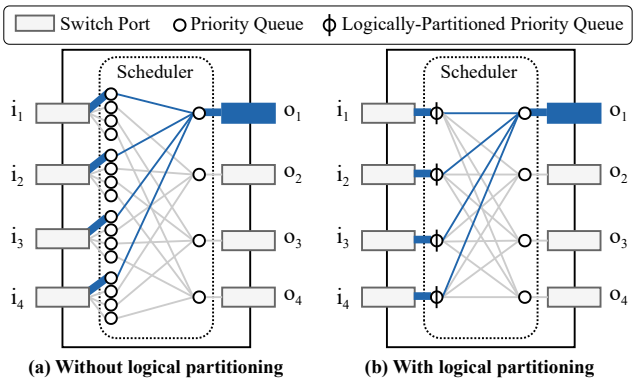
2.1.1 Line-Rate Switches

In order to support packet scheduling without impeding the rest of the switch's dataplane functionality, any realistic proposal for a packet scheduler ought to satisfy two key requirements:

(A1) Match the switch's aggregated packet rate: For scheduling in output-queued switches, the worst-case throughput demand corresponds to the scenario where ingress traffic from all ports is incident on a single egress port (i.e., incast behavior). In order to avoid backpressuring the switch fabric, the packet scheduler must be able to process packets at the same rate as the switch backplane (i.e., its aggregated packet rate) at *any* given port. For instance, in NVIDIA's Spectrum SN4700 (a state-of-the-art 400 GbE switch with 32 ports), the packet scheduler for a single port must be able to handle an aggregated packet rate of 8.4 *billion* pps [33].

(A2) Allow every port to address all buffered packets: In order to efficiently utilize on-chip memory, switches use a shared packet buffer that is dynamically partitioned between its output ports [41]. In the worst case, every packet in the shared buffer might be destined for the same output port, and each packet might map to a different flow to be scheduled. If the packet scheduler at a given port could only address a subset of buffered elements, it would impose additional constraints on the switch's ability to handle such bursts. Thus, a second key requirement for schedulers is the ability to allow any output port to address *every* buffered packet. In modern switches, shared packet buffers are provisioned for *hundreds of thousands* of packets [33].

In 2016, a single physical PIFO instance could handle the full aggregated packet rate for a 64-port 10 GbE switch (1 Bpps) [41]. Today, state-of-the-art switches, *e.g.* the SN4700, offer 400 GbE line rates with 32 ports (20× higher



(a) Without logical partitioning **(b) With logical partitioning**

Figure 1: Scheduler architecture using priority queues without (left), and with (right) support for logical partitioning. In a k-port switch using priority queues without logical partitioning, we need k^2+k priority queues in order to both implement output queueing *and* absorb incast traffic from all the ports. Support for logical partitioning reduces the number to $2k$.

aggregated throughput). Given the large (and ever-widening) gap between network and processing speeds, a single priority queue instance can no longer sustain aggregated packet rates. Consequently, to satisfy (A1), packet schedulers for modern switches must "scale out" using a mesh of priority queues.

Unfortunately, *priority queue designs that do not support logical partitioning, i.e., the ability to multiplex several independent queues atop a single physical queue, require prohibitively large meshes.* To the best of our knowledge, PIFO and PIEO are the only existing design that support logical partitioning, while more recent proposals (*e.g.*, BMW-Tree [47]) do not. Generously assuming that a single priority queue instance could handle 400 Gbps line-rate input in a k-port switch, these designs would require, at minimum, a (k^2+k) mesh of instances to realize per-output-port scheduling while supporting the full aggregated throughput due to incasts; an example mesh for $k=4$ is depicted in Figure 1(a). Further, *every instance would have to be provisioned with hundreds of thousands of queue elements* to satisfy (A2). Overall, a 32-port switch operating at 400 Gbps line-rate would require at least 32^2+32 BMW-RPU instances, each provisioned with 100K+ entries, to implement priority queueing alone. Based on synthesis numbers reported by the authors and considering that a switch chip area ranges from 200 mm^2 to 800 mm^2 [41] this corresponds to $1.5-6\times$ the total area.

The ability to logically partition a physical priority queue enables a significantly simpler mesh architecture. Assuming, again, that each priority queue instance can sustain 400 Gbps input, we would only need $2k$ instances, as depicted in Figure 1(b). Each physical instance in the first layer ingests packets from a single ingress port, but enqueues into one of k *logically independent queues* corresponding to the k possible destination ports. The second layer then periodically schedules packets among their k inputs, feeding traffic to their respective egress ports. *Despite the fact that PIFO benefits from this architecture, its inability to scale beyond 2,048 elements means that it does not satisfy (A2). Conversely, while PIEO scales marginally better, it does not provide the requisite performance, violating (A1).*

Consequently, we find that **no existing hardware priority queue design is suitable for modern switches** due to a fundamental limitation on either scaling (*e.g.*, PIFO), performance (*e.g.*, PIEO), or their ability to provide logical partitioning (*e.g.*, BMW-Tree).

2.1.2 SmartNICs in the Public Cloud

Another key driver for programmable packet schedulers in hardware are SmartNICs in the public cloud, either ASIC [26, 34, 37, 42] or FPGA-based [6, 15, 16, 20, 27, 35, 37]. Packet schedulers for such NICs ought to satisfy three requirements:

(B1) Scale to tens of thousands of flows: The packet scheduler on the SmartNIC may need to implement scheduling policies for a large number of active (concurrent) flows. This might seem surprising because NIC packet buffers are typically orders of magnitude smaller than those used in switches; however, unlike switches, modern SmartNICs may be required to make scheduling decisions for flows not just in their local TX or RX queues, but also those residing in *host memory*. This is a popular theme in the cloud setting where, in order to save valuable CPU cycles, the hypervisor dataplane (including scheduling functionality) is offloaded to the NIC [15]. The packet scheduler aboard the NIC is then responsible for deciding which backlogged flow queues in host memory to serve at any point, with the number of scheduling candidates scaling as (tenants $\times$ flows per tenant). For instance, across 1M+ VMs in Azure, VFP [14] reports 4.8K active connections *per VM* at the 99th percentile, and as high as 12K at P99.9. We expect the scalability problem to become all the more apparent given trends of increasing core counts [5] (and therefore potential for multitenancy), and as more services that traditionally used multiple physical NIC queues (*e.g.*, RDMA) become amenable to virtualization [19].

(B2) Sustain 100GbE+ line-rates: While state-of-the-art NICs have lower throughput requirements compared to switches, they still need to support line rates of 100 Gbps and beyond [32]. This is particularly relevant in the context of public clouds because network bandwidth is a commoditized resource and an important driver for many high-performance cloud-based applications [35].

(B3) Implement scheduling both across and within tenants: Finally, in the context of multi-tenant clouds, the NIC scheduler should be able to provide, at minimum, the ability to schedule traffic *across* tenants (to enforce cloud providers' policy requirements, *e.g.*, fairness or bandwidth quotas) and *within* tenants (to provide application-level priority queueing), without imposing significant resource overhead.

Recent priority queue designs (*e.g.*, PIEO [40], BMW-Tree [47]) symbolize a concerted effort towards addressing the *scalability* requirement outlined in (B1). For instance, we

can synthesize a PIEO instance with up to 64K entries, and a BMW-RPU instance with 350K entries on a state-of-the-art FPGA (§6.2). Unfortunately, *these designs do not meet both the performance (B2) and provider policy (B3) requirements.*

In the context of (B2), the problem with existing designs is that performance degrades rapidly as the number of elements increases, a fundamental tradeoff associated with comparison-based sorting. Moreover, scaling out using a mesh is not feasible because NICs are considerably smaller and more resource-poor than switches; as such, single-instance performance is the key factor in determining feasibility. For example, [47] reports that a single BMW-RPU instance can sustain 200 Mpps with 85K elements using a 28nm ASIC process; this is sufficient to drive line rate at 100 GbE (148.8 Mpps), but not at 200 GbE. The picture is even more dire for packet scheduling on FPGA-based SmartNICs [15, 20]. For instance, as we will show in §6.2, BMW-Tree achieves a packet rate of 55 Mpps for 85K elements on a state-of-the-art, Intel Stratix 10 MX FPGA, 37% of line rate even at 100 Gbps.

A second, more fundamental problem with these designs is that it is impossible to disentangle their *function* (implementing priority queueing) from their *form* (a fixed tree [10, 25, 47] of queue elements). As a result, implementing n distinct priority queues (*e.g.*, for n tenants) requires duplicating the underlying data structure, imposing significant resource overhead, fragmentation of queue memory, or both. As before, the key enabler for (B3) is *logical partitioning*.

Once again, we find that **existing hardware priority queues are not viable alternatives for packet scheduling in modern SmartNICs** because they do not provide the necessary performance or logical queueing functionality.

2.2 Exploring a Different Tradeoff

In this work, we seek to explore a different tradeoff that allows us to circumvent the performance-scalability barrier: sacrificing a small amount of *precision* to achieve the best of both worlds. In this regard, we are motivated by prior work's observation that a large fraction of networked applications *do not* require particularly high precision [2, 39]. For instance, both VLAN and DSCP support up to 8 traffic classes (3-bit priority tags), priority-qdisc (*tc-prio*) in the Linux kernel provides at most 16 priority bands, and state-of-the-art commercial switches support up to 32 strict-priority queues [2]. These, in turn, provide sufficiently fine-grained priority queueing to support a variety of higher-level abstractions: transport protocols and frameworks (*e.g.*, Homa [30], PASE [31], PIAS [8]), congestion and interference controllers (RC3 [29], QJUMP [18]), and high-performance overlay networks (*e.g.*, GRIN [1], SLIM [49]). In what follows, we describe the design of a highly scalable and performant priority queue exploring this tradeoff.

3 BBQ Overview

BBQ is a new hardware-based priority queue architecture for packet scheduling that is designed with three goals in mind: (1) *scalability*, the maximum number of concurrent flows that the queue can support, (2) *performance*, the maximum steady-state packet rate that the queue can sustain, and (3) *logical partitioning*, the ability to multiplex several logical queues atop a single physical queue. In this section, we give an overview of BBQ's design before diving into the architectural details in §4. We start with a brief introduction to the data structure underlying BBQ in §3.1, followed by a high-level description of the BBQ primitive in §3.2. Finally, in §3.3, we describe the challenges the hardware architecture must address in order to meet our system goals.

3.1 Data Structure

Integer Priority Queueing: BBQ leverages an *Integer Priority Queue* (IPQ) scheme to alleviate the tension between scalability and performance (§2). Unlike traditional priority queues where elements can have arbitrary priorities, an IPQ requires elements to map to a *finite set of integer priorities*, called its *priority span*; for an IPQ that supports P integer priorities, the span is represented by the set $\{0, ..., P-1\}$. Quantizing the priority range allows IPQs to implement a simple counting sort-like algorithm: the IPQ maintains an array of P *priority buckets*, each representing a unique priority in its span; when a new element is enqueued, it is inserted into the bucket indexed by the element's priority.

The only remaining challenge is to find the right priority bucket to dequeue from. Since only a subset of buckets may be occupied at any time, dequeueing entails finding the *highest-priority bucket containing at least one element*. A naive approach is to sequentially check buckets in decreasing order of priority, stopping at the first non-empty bucket; however, this may turn out to be expensive, necessitating $O(P)$ sub-operations in the worst case.

Building efficient IPQs using Find-First Set: One approach to improve the run-time efficiency of dequeue operations is to encode the occupancy of the IPQ's priority buckets as a P-bit wide bitmap, with '0's representing empty buckets, and '1's representing buckets containing at least one element. Then, the *most-significant set bit* (MSSb) in the bitmap yields the required bucket to dequeue from. Finding the MSSb, an operation known as *Find First Set* (FFS), can be performed with $\Theta(\log P)$ simple bit operations (bit-shifts and additions) using a binary search algorithm. Unfortunately, scaling to large values of P (*e.g.*, 32K) using FFS Queues quickly becomes impractical due to constraints on the maximum word size that the hardware can efficiently operate upon. For example, general-purpose processors provide FFS intrinsics for at most 64-bit words; similarly, implementing FFS on bitmaps larger than 64 bits would result in low operating frequency even in more specialized circuits (*e.g.* ASICs or FPGAs).

Scaling to larger priority spans using Hierarchical FFS:

[45] novelly observed a recursive structure to the problem: given an array of *bitmaps* ordered by priority, the *highest-priority non-zero bitmap* can also be identified via a single FFS operation using a schema similar to the one described above. This observation naturally leads to the notion of a *Hierarchical* FFS (HFFS) Queue.

The idea is to construct a tree of bitmaps, with leaf nodes representing regular FFS Queues. The bitmaps are encoded such that, at any level in the tree, *a '1' in any bit position indicates a non-empty priority bucket in the subtree rooted at that node*. Now, to dequeue the highest-priority element, we recursively perform FFS at each level of the tree starting with the root, following the MSSb until we arrive at the required priority bucket. BBQ uses a variant of the HFFS Queue as its underlying data-structure.

3.2 The BBQ Primitive

At the heart of BBQ is a priority index structure inspired by HFFS Queues: a perfect w-ary tree of w-bit bitmaps representing the queue occupancy. The bitmap tree in a BBQ can be composed of an arbitrary number of levels, which in turn dictates the queue's priority span. In general, for a BBQ with w-bit bitmaps and $D \geq 1$ levels, the number of supported priorities is $P = w^D$. Figure 2 depicts a BBQ with $w = 3$ and $D = 2$, representing $P = 3^2 = 9$ unique priorities.

The bitmap tree has a recursive structure: at the leaf level of the tree (*e.g.*, the L_2 bitmaps in Figure 2), each bit maps to a unique *priority bucket*; at non-leaf levels (*e.g.*, L_1), each bit maps to a unique subtree of bitmaps. In either case, we maintain the invariant that a bit in any bitmap is 1 *if and only if* there is a priority bucket containing at least one element in the corresponding subtree. In BBQ, we additionally associate with every bit a *subtree occupancy counter* (StOC) that indicates the total number of elements in that subtree; a StOC is non-zero if the corresponding bit is 1, and vice versa. As we will see in §4, the design choice of storing additional counters enables us to achieve stall-free execution of the BBQ pipeline, yielding high performance (i.e., full pipelining) with a relatively small memory overhead.

IPQs group queue elements (QEs) with identical priority in the same priority bucket (PB). In BBQ, PBs are implemented as *doubly-linked lists* of QEs. In particular, each PB stores a pair of pointers: `HEAD` and `TAIL`, pointing to the first and last QE in the linked list, respectively. QEs themselves are composed of two attributes: (1) a `DATA` field to store arbitrary, user-supplied identifiers[3], and (2) a pair of pointers (`PREV` and `NEXT`) to other QEs, allowing them to interface with the PBs' doubly-linked lists.

The final component of the BBQ is the *Free List* (FL): a FIFO queue containing pointers to QEs that are currently unallocated (i.e., *not* enqueued in the BBQ). Table 1 depicts the operations supported by BBQ.

[3]BBQ, like most priority queues, is agnostic of the data contained in the QEs. The `DATA` attribute has a configurable bit-width and could be used to store a pointer to a packet, flow ID, or even a reference to another BBQ. Unless specified otherwise, we will assume that QEs represent *flows* [41].

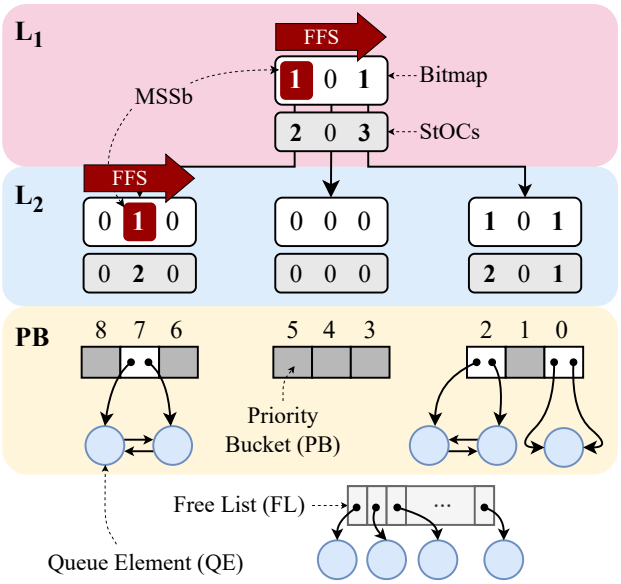


Figure 2: 2-level BBQ with $w = 3$ bit bitmaps. To dequeue the highest-priority element, we recursively perform FFS at each level of the tree starting with the root, following the most-significant set bit (MSSb) until we arrive at the required priority bucket.

3.3 Goals and Challenges

Recall that we sought out to build BBQ with three key goals in mind: *scalability*, *performance*, and *logical partitioning*. In this section, we describe how BBQ meets these goals, and the challenges the underlying hardware architecture must address to achieve them.

3.3.1 Scalability

Using an IPQ-based design breaks the dependence between run-time complexity of operations and queue size, allowing BBQ to support a large number of QEs without necessitating a fundamental performance trade-off. In many ways, scalability "falls out" of this high-level design choice, allowing us to explicitly optimize for the other goals.

3.3.2 Performance

While the data structure underlying BBQ is conceptually simple, realizing this functionality in hardware in a manner that achieves high performance is a challenging proposition.

The overall performance of the hardware queue (measured in packets per second) is the product of two independent metrics: (C_1) f_{max} or the maximum frequency that the queue operates at, which is dictated by its *critical path* (i.e., the worst-case combinational delay in the hardware circuit), and (C_2) the number of operations that can be issued every cycle. Given the logical complexity of the queue operations (i.e., `ENQUEUE` or `DEQUEUE`), it is impractical to perform them in a single hardware clock cycle because it would significantly

Operation	Description
`ENQUEUE(X, p)`	Inserts the given data, `X`, into the BBQ with priority p.
`DEQUEUE(t)` $\rightarrow$ `(Y, p)` or $\varnothing$ `t` $\in \{$`MAX`,`MIN`$\}$	Extracts the data, `Y`, corresponding to either the *highest-priority* QE (when `t` is `MAX`) or the *lowest-priority* QE (when `t` is `MIN`) currently enqueued in the BBQ. Returns $\varnothing$ if empty.

Table 1: Priority Queue operations supported by BBQ.

throttle f_{max}, hurting (C_1). Instead, operations must be divided into a sequence of n *stages*, where each stage involves a smaller quantum of work. While this improves f_{max} of the resulting circuit by reducing combinational delay, doing so naively (*e.g.*, trying to avoid concurrency problems by issuing one operation every n cycles) would slash (C_2) by a factor of n, once again degrading performance.

The key to high performance – and simultaneously the most challenging aspect of BBQ's architectural design – is *pipelined parallelism*. In this context, pipelining refers to designing the hardware architecture such that *multiple stages* may simultaneously be active at the same time, thereby allowing operations to be issued fewer than n cycles apart. Unfortunately, there are several sources of complexity that make it non-trivial to achieve a high degree of pipelined parallelism: practical limitations on the number of R/W ports on physical memory blocks, data hazards (i.e., ephemeral memory dependencies between active pipeline stages), and control hazards (i.e., logical and algorithmic dependencies between stages).

Our key finding in this context is that by carefully architecting BBQ's hardware pipeline, we can, in fact, achieve fully-pipelined execution (i.e., guaranteed throughput of 1 operation per cycle) *without* compromising on the maximum clock frequency. This is realized by: (a) employing deep pipelining to preserve f_{max}, and (b) using a variety of architectural techniques (speculation, write-forwarding, and instruction coloring) to handle pipeline hazards without stalling or discarding operations. We describe the BBQ pipeline in detail in §4.

3.3.3 Logical Partitioning

Full decoupling between BBQ's queue memory (i.e., its QEs) and its priority index structure (i.e., the bitmap tree) gives BBQ a unique opportunity to provide logical partitioning with no resource overhead. The idea is to *treat the bitmap tree as a collection of n disjoint subtrees, each of which maps to an independent BBQ*. This effectively partitions the original BBQ's priority span into n disjoint regions; then, in order to `DEQUEUE` an element from a *logical* BBQ, we simply "mask" the appropriate bits in the bitmap tree while performing FFS such that we only traverse down the corresponding subtree.

This technique allows us to fully reuse *all* of the physical BBQ's resources without any performance degradation or fragmentation of queue memory. There is, however, a cost in terms of precision, because each logical BBQ can only address $\frac{1}{n}$'th of the priority span of the underlying instance. For use-cases where the number of logical partitions is not too large (*e.g.*, 32-64 port switches [33], or cloud servers hosting 16-128 tenants), this is simply a matter of appropriately provisioning the priority span of the underlying BBQ.

Since logical partitioning is, (a) a key enabler for building efficient priority queue meshes for line-rate switches and realizing hierarchical scheduling in multi-tenant cloud NICs (§7), and (b) adds negligible overhead in the BBQ datapath, we natively support this feature in the BBQ primitive. We describe logical partitioning (both for prior work, as well as BBQ) in more detail in Appendix A. Logical partitioning also enables us to extend the BBQ primitive to operate over dynamic priority ranges with zero overhead, an idea we describe in Appendix D.

4 BBQ Architecture

In this section, we describe the architectural details that enable us to map BBQ's design to hardware in a manner that achieves our performance goals. We begin by describing the hardware pipeline in §4.1, followed by a description of the hurdles that arise while trying to fully pipeline the design.

4.1 Hardware Pipeline

In principle, enqueueing and dequeueing follow a similar blueprint, yielding an intuitive algorithm for mapping them to a unified datapath: for each level of the tree starting with the root (i.e., L_1), compute the bitmap index (for `DEQUEUE(t)`, the index is computed by performing FFS on the bitmap, while for `ENQUEUE(X, p)`, it is computed using simple bit manipulations on p), increment/decrement the corresponding StOC, then update the bitmap; finally, enqueue into or dequeue from the doubly-linked list corresponding to the target priority bucket. To maximize performance, we take a careful two-pronged approach.

(1) Maximizing f_{max}: Our first objective is to maximize f_{max}, or the maximum clock frequency at which the BBQ circuit can operate. To do this, we use a *deep pipeline* where individual stages are designed to do both *little* and *roughly equal* amounts of work. Table 2 depicts the events that occur at cycle-level in a 11-stage pipeline for a 2-level BBQ.[4] By load balancing expensive operations across stages, we minimize the number and severity of same-stage dependencies (depicted by $\hookrightarrow$ and $\rightsquigarrow$). For instance, chaining FFS and StOC updates (multi-bit addition or subtraction) would result in a large combinational delay, so we split this work across stages

[4] Since the L_1 level has a small memory footprint, we choose to store the associated metadata (bitmaps and StOCs) in registers with single-cycle access latency. The larger arrays (*e.g.*, L_2 bitmaps and StOCs, priority buckets, and queue elements) use substantially more memory and involve more complex address decoding logic; as such, we store these in SRAM with a 2-cycle access latency in order to optimize f_{max}.

Cycle	Description	PHR
0	Register inputs If ENQUEUE: $F \leftarrow$ FreeList.POP() // Pop free list	
1	Compute L_1 bitmap index ↳ Read the corresponding L_1 StOC	L_1
2	Compute, Write: L_1 StOC ⇝ L_1 bitmap Read L_2 bitmap	L_2
3	// Read delay for L_2 bitmap	
4	Compute L_2 bitmap index ↳ Read the corresponding L_2 StOC	
5	// Read delay for L_2 StOC	
6	Compute, Write: L_2 StOC ⇝ L_2 bitmap Read the corresponding PB	PB
7	// Read delay for PB	
8	If DEQUEUE: (a) Read $X \leftarrow$ QE_{DATA}[PB.TAIL^{new}] (b) Read $Y \leftarrow$ QE_{PREV}[PB.TAIL^{new}]	
9	// Read delay for QE_{DATA} and QE_{PREV}	
10	If ENQUEUE: // Enqueue at the HEAD (a) $QE_{DATA}[F] \leftarrow$ Data to enqueue (b) Write $QE_{NEXT}[F] \leftarrow$ PB.HEAD (c) Write QE_{PREV}[PB.HEAD] $\leftarrow F$ (d) Write PB.HEAD$^{new} \leftarrow F$ If DEQUEUE: // Dequeue from TAIL (a) FreeList.PUSH(PB.TAIL) (b) Write PB.TAIL$^{new} \leftarrow Y$ (c) Output X	

Table 2: 11-stage hardware pipeline for a 2-level BBQ (without operation coloring) highlighting independent pipeline hazard regions (PHRs). ↳ and ⇝ indicate same-stage dependencies, which result in more complex combinational logic. In general, a BBQ with $D > 1$ levels entails a pipeline depth of $p = 7 + 4 \times (D-1)$ stages.

(*e.g.*, cycles 1 and 2).

(2) Maximizing operations per cycle: As described in §3.3.2, high f_{max} is only useful if we are not rate-limited by the pipeline latency or even a portion of it. Our second objective is to *fully pipeline* the BBQ design so it can concurrently process as many operations as there are pipeline stages, thereby achieving its maximum rate of 1 op/cycle. There are several challenges we encounter in this process, which we address in detail next.

4.2 A Fully-Pipelined Architecture

Pipeline Hazard Regions: The first key enabler for BBQ's stall-free architecture is the design choice of associating every bit in the bitmap tree with a *subtree occupancy counter*. Recall that for a given bit, the associated StOC indicates the *total number of elements contained in the corresponding subtree*.

To understand how this enables pipelining, consider a strawman design with n pipeline stages where we *only* store the bitmaps for each level of the HFFS tree, but not the associated StOCs. Here, the earliest time we know whether a DEQUEUE operation causes a priority bucket to become empty is when the operation is committed (i.e., the final pipeline stage). Now, consider what happens if this causes a bit in any bitmap along the path to that priority bucket to flip (i.e., become '0'). Any subsequent DEQUEUE operations in the pipeline may have been routed along the tree based on stale state, creating an incorrigible *control hazard*. As a result, we would have to either: (a) discard and re-issue the subsequent DEQUEUE operation(s), hurting worst-case performance, or (b) only issue DEQUEUEs every n cycles, defeating the purpose of pipelining altogether.

Instead, StOCs allow us to divide the BBQ pipeline into independent *pipeline hazard regions* (PHRs) mapping to different levels of the bitmap tree, as shown in Table 2. When exiting a PHR, the outcome of every operation (either an ENQUEUE or a DEQUEUE) is committed to the StOC. This has two implications: (1) two operations can only be conflicted (i.e., have data or control dependencies between them) if they are in a PHR at the same time, and (2) conflicts are limited to intra-PHR state (*e.g.*, the bitmap or StOC data at that tree level). As a result, we only have to address intra-PHR hazards (i.e., dependencies between active operations in the same PHR), which are far more localized – and therefore more tractable – than hazards spanning the entire pipeline. We characterize our implementation of StOCs in detail in Appendix B.

While StOCs *enable* us to achieve stall-free operation, they are not sufficient to guarantee a fully-pipelined design on their own. In what follows, we describe three types of intra-PHR hazards we encountered in our endeavor to fully-pipeline BBQ, and the architectural techniques used to address them.

(H_1) Data Hazards: The simplest form of hazards we encounter are *data* hazards, where one stage of the pipeline either: (a) issues a memory read, or (b) waits on completion of a memory read at an address that is concurrently updated by a different pipeline stage.

For instance, consider the BBQ pipeline depicted in Table 2. If Stage 2 issues a read for an L_2 bitmap that is simultaneously being modified by Stage 6, it will receive either a stale or invalid value 2 cycles later.[5] Similarly, if Stage 3 has a read in progress for the same memory address, it will receive a stale value on the next cycle. This is a common problem in processor design, where the standard solution – and the one we use here – is to perform ***write forwarding*** from a later pipeline stage to its predecessors when a read-after-write conflict occurs.

[5] Certain hardware platforms may guarantee a consistent memory view, but this is not true in general (*e.g.*, FPGA SRAM).

In order to resolve data hazards, we need to first compute *whether* and *which pairs of* operations in a PHR access the same state (*e.g.*, bitmaps, StOCs, PBs). BBQ exploits the hierarchical nature of the queue to make this computation efficient: since bitmap and StOC addresses also have a hierarchical structure to them (*e.g.*, the address of an L_3 bitmap is generated by splicing together the address of its L_2 parent and its own index in the parent bitmap), we can both reduce address comparator logic and improve f_{max} by memoizing address conflicts at higher levels and propagating them down the pipeline; then, when we need to compute address conflicts for lower levels, we reuse the memoized results, necessitating comparison of only the lower address bits.

(H_2) Non-atomic bitmap accesses: A second type of intra-PHR hazard arises due to the non-atomic nature of bitmap accesses, causing back-to-back `DEQUEUE` operations to be routed down incorrect paths in the bitmap tree.

To illustrate this problem, consider the example depicted in Figure 3. Initially, at ①, both the MSb and LSb of an L_2 bitmap are set, and the corresponding StOC values are 1 and 2, respectively. Now, consider two `DEQUEUE-MAX` operations, OP_A and OP_B, issued one cycle apart. Since the highest-priority (left-hand) subtree has just one element and becomes empty after the first `DEQUEUE` operation (i.e., OP_A), we would *expect* to see OP_B to be routed down the right-hand subtree (corresponding to the LSb). Instead, we find that *both* operations are incorrectly routed down the left-hand subtree. The problem arises at ③, the moment OP_B and OP_A reach Cycles 4 and 5 of the pipeline, respectively. At this point, OP_A is waiting on the read for the MSb's StOC (issued one cycle earlier, at ②) to complete, while OP_B computes the same MSSb as OP_A and issues a read for the same StOC. It is only on the following cycle – at ④, when OP_A decrements the MSb's StOC down to 0 – that we discover that the left-hand subtree becomes empty, and that OP_B *should* have been steered to the right-hand subtree instead; unfortunately, it is far too late by this point. Note that this is not simply a rare performance issue that can be addressed by, *e.g.*, discarding trailing operations in case of conflicts; rather, it is a *correctness* bug. In this case, since OP_B may have already been "committed" to StOCs earlier in the pipeline, the operation cannot simply be discarded. Once again, we would have to either: (a) stall the pipeline, hurting worst-case performance, or (b) enforce that `DEQUEUE` operations are issued at least 2 cycles apart, throttling the queue's `DEQUEUE` throughput.

To address this problem, we adopt another technique from the architecture literature: ***speculation***. The idea is as follows: within the L_i PHRs, if we have two back-to-back `DEQUEUE-MAX` operations (say, OP_A and OP_B, issued in that order, respectively) that access the same bitmap, *we compute the bitmap index for OP_B speculating that OP_A causes the MSSb to become '0'*. Consequently, OP_B will issue a read for the next-MSSb. On the following cycle, if we find that OP_A indeed caused the MSSb to flip (i.e., the corresponding StOC

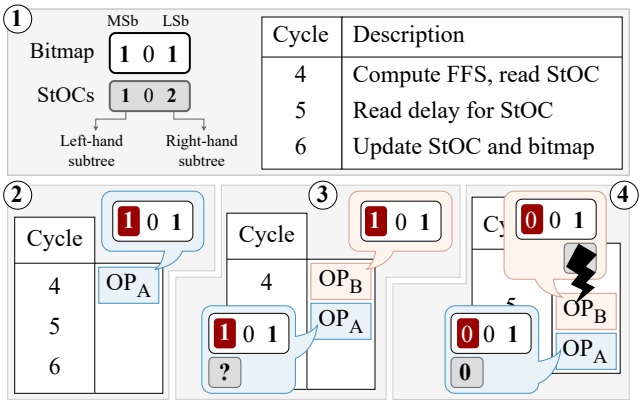


Figure 3: Non-atomic read-modify-write accesses to bitmaps cause OP_B (the second of two consecutive `DEQUEUE-MAX` operations) to be incorrectly routed to the left-hand subtree.

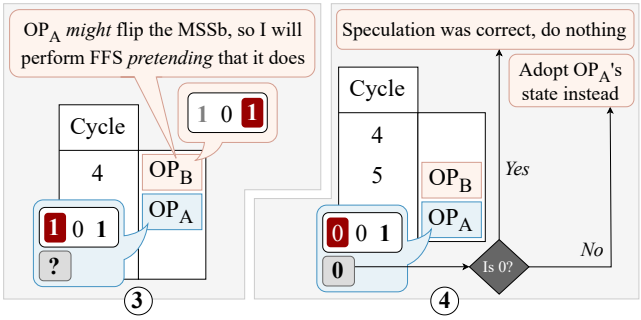


Figure 4: If there are conflicting operations in the L_i PHRs, operations issued later compute bit indices *speculating* that earlier operations will change the bitmap.

becomes 0), our speculation was correct, and OP_B proceeds as usual. Otherwise, OP_B simply discards its own state, and adopts both the MSSb index and StOC values computed by OP_A for the remainder of the pipeline. The speculation logic is illustrated in Figure 4.

The key observation here is that instead of squandering OP_B's one available read on data that are going to be available anyway (via inter-stage forwarding), we can "hedge" our bet on multiple bits simultaneously, ensuring that at least one of them yields the desired outcome. We also note that, while the description presented here involves only `DEQUEUE-MAX` operations for the sake of simplicity, the same technique generalizes to any combination and order of operations (i.e., `ENQUEUE`, `DEQUEUE-MIN`, and `DEQUEUE-MAX`).

(H_3) Non-atomic PB accesses: Conceptually similar to (H_2), the third and final type of hazard arises due to the non-atomic nature of priority bucket accesses, causing back-to-back `DEQUEUE` operations to potentially corrupt state within the *PB* PHR. To see why, consider stages $6-10$ of the BBQ pipeline. In Stage 10, `DEQUEUE` operations cause the PB's `TAIL` pointer to be updated (now denoted by PB.$\texttt{TAIL}^{\text{new}}$). In stage 8, `DEQUEUE` operations perform a read that is *supposed to be addressed* by the most up-to-date `TAIL` pointer for the

corresponding PB. However, consider what happens when two back-to-back DEQUEUE operations (say, OP_A and OP_B, issued in that order, respectively) land at the same priority bucket. At Stage 9, OP_B is waiting on completion of the read addressed by PB.TAIL available on the previous cycle. However, OP_A, now at Stage 10, modifies the TAIL pointer, causing the read issued by OP_B (for QE_{DATA} and QE_{PREV}) to become stale. Unfortunately, write-forwarding does not help here because the stale variable (PB.TAIL) is being used to address other state, creating a control hazard.

To address this problem, we introduce the notion of ***operation coloring*** (inspired by *graph coloring*, problems where vertices in a graph must be assigned colors such that no two adjacent vertices have the same color), which works as follows. First, we tag each operation with a *Color* attribute, which assumes one of two values: *Purple* (■) or *Orange* (●). An operation's color determines which end of the PB's doubly-ended linked-list it interacts with: operations colored purple operate on the HEAD of the PB, while those colored orange operate on the TAIL. All ENQUEUE operations are always colored purple, while DEQUEUE operations are, by default, colored orange. Finally, we add a single constraint on color: *a DEQUEUE operation must not have the same color as a conflicting operation issued immediately before it (i.e., one cycle earlier)*. In the first cycle of the PB PHR (Cycle 7), if the active operation is a DEQUEUE that conflicts with another operation in the subsequent pipeline stage, it is recolored. Table 3 depicts the relevant portion of the BBQ pipeline post operation coloring (extraneous details are omitted for the sake of brevity).

C	Description	
7	// Color operation	
8	If DEQUEUE ■: Rd QE_{DATA}[PB.HEAD^{new}] Rd QE_{NEXT}[PB.HEAD^{new}]	If DEQUEUE ●: Rd QE_{DATA}[PB.TAIL^{new}] Rd QE_{PREV}[PB.TAIL^{new}]
9	// Read delay	
10	If ENQUEUE ■: Update PB.HEAD ^{new}	
	If DEQUEUE ■: Update PB.HEAD ^{new}	If DEQUEUE ●: Update PB.TAIL ^{new}

Table 3: Updated stages 7 – 10 showing operation coloring.

Observe that, so long as the DEQUEUE operations are colored differently from any operation *immediately preceding them in the pipeline*, they will not incur stale reads. The key idea here is that each operation (whether an ENQUEUE or a DEQUEUE) only affects one pointer in the PB's (HEAD, TAIL) pair.[6] As a result, picking a mutually exclusive color also guarantees exclusivity on the data structure itself. Operations spaced more than one cycle apart can be safely handled via write forwarding. An artifact of this design choice is that back-to-back DEQUEUEs landing at the same priority bucket will not dequeue elements in FIFO order; however, since DEQUEUEs are bound to be interleaved with ENQUEUEs during typical operation, we do not expect this case to arise often.

Together, these optimizations realize a fully-pipelined priority queue architecture with both high f_{max}, and a guaranteed operation throughput of 1 op/cycle independent of workload.

[6]The only situations in which *both* pointers are affected is when the priority bucket becomes empty (i.e., due to a DEQUEUE), or a priority bucket that was *previously* empty becomes non-empty (i.e., due to an ENQUEUE). Speculation precludes the first possibility (attempting to DEQUEUE an empty PB), so we are left with the second corner case, which we handle explicitly.

5 BBQ$_{◉}$: A Latency-Free BBQ

While deep pipelining is key to BBQ's high performance, the resulting *pipeline latency* (i.e., the number of clock cycles that elapse between when an operation is issued and when it completes) introduces a new source of error in the relative ordering of elements compared to an "ideal" priority queue.

The problem manifests due to a confluence of two factors: (a) since it takes several cycles for an operation to traverse the pipeline, in order to use BBQ at full throughput (1 op/cycle), multiple operations need to be issued concurrently; and (b) for a pipeline of depth p, *the minimum delay* for a high-priority element to be dequeued, served, and re-enqueued into the queue is also p. *Consequently, any DEQUEUE operations that are issued in the p cycle interval that the highest-priority element is not present in the queue might ultimately dequeue lower-priority elements*. A concrete example of the problem is described in §C.1. The aforementioned problem is inextricably tied to our decision of using a pipelined architecture, implying that the BBQ primitive alone cannot guarantee absolute accuracy at full throughput.

However, we find that a simple augmentation to the BBQ primitive allows us to hide this latency and avoid the accuracy issues that come with it: use a tiny PIFO as a "cache" in front of the BBQ to hold the highest-priority elements. Whenever this tiny PIFO overflows, it "leaks" the lowest priority element to the BBQ. This PIFO only needs to be able to hold as many elements as the BBQ's pipeline depth (order of tens of elements). Because of its small size, this instance does not face the scalability limitations associated with the PIFO architecture and only adds a small footprint to the design.

The augmented design, BBQ$_{◉}$, *provably guarantees zero loss in accuracy (i.e., any dequeued element is always the highest-priority one in the system at that time) while providing full throughput (1 op/cycle)*. We can show this by proving the sufficient condition in Theorem 1: with a PIFO whose size exceeds the pipeline latency of the BBQ, the highest-priority element is always served from the PIFO, such that we never experience the accuracy or latency artifacts introduced by the accompanying BBQ. We describe BBQ$_{◉}$ in detail in §C.2.

Theorem 1 (Priority Set Invariant for BBQ$_{◉}$). *In a BBQ$_{◉}$ instance composed of a BBQ with pipeline latency p cycles and a PIFO of size $k > p$, the top $(k-p)$ highest-priority elements are always in the PIFO.*

The proof can be found in §C.3.

6 Evaluation

We now evaluate BBQ. Our main goal is to understand BBQ's *performance* and *viability* for both ASICs and FPGA designs. We also compare BBQ with PIFO [41], PIEO [40], and BMW-Tree [47]. PIFO is the state-of-the-art hardware priority queue in terms of *throughput* while BMW-Tree is the state of the art in terms of *scalability*. Throughout this evaluation, we show that BBQ can surpass PIFO's throughput while achieving similar scalability to BMW-Tree.

6.1 Setup and Methodology

We implement BBQ in SystemVerilog. Given the recurring and composable structure of the design, we implement a Python script to automatically generate different configurations of BBQs by stitching together modular blocks of hand-written SystemVerilog code. Users can specify the number of levels in the tree (D), the bitwidth of every node (W), and the maximum number of elements (N). We synthesize BBQ targeting both an FPGA (Intel Stratix 10 MX FPGA [24]) as well as an ASIC. The Stratix 10 MX contains 702,720 Adaptive Logic Modules (ALMs), 140 Mb of SRAM, and two 100 Gb Ethernet ports. For comparison, we also synthesize PIFO, PIEO, and BMW-Tree targeting the same board. For each design and configuration, we conduct a bisection search to find the maximum clock frequency achievable with 3 MHz precision, picking the best synthesis across 10 seeds. To synthesize the FPGA we use Intel Quartus [21]. To synthesize the ASIC, we use Synopsys Design Compiler [43] using a 7 nm Standard Cell Library [44] based on the ASAP7 PDK [13].

All the designs we evaluate have deterministic performance that is independent of the workload. As such, our analysis focuses on the packet rate that each design is able to sustain, as well as the cost [36] (in terms of die area and FPGA resources).

6.2 FPGA

As mentioned in §2.1.2, FPGA-based NICs are increasingly used as a way to achieve programmable offloads on the NIC [15, 16, 23, 35, 37]. Having a programmable packet scheduler on the NIC would allow administrators to change the packet scheduling algorithms at run time. In this section, we evaluate how BBQ and the baseline designs perform, in terms of throughput (§6.2.1) and FPGA resources (§6.2.2). We also explore how the different BBQ design parameters affect its performance when running on an FPGA (§6.2.3).

6.2.1 Throughput Scalability

Queue capacity can influence throughput by increasing the hardware critical path, which in turn reduces the maximum clock frequency that we can achieve with the design (f_{max}). To understand this effect, we synthesize BBQ (with 8-bit bitmaps) and the baselines while changing both the queue capacity and the number of bits used to express the priorities (precision). We report both the clock frequency as well as the overall throughput achievable by each design.

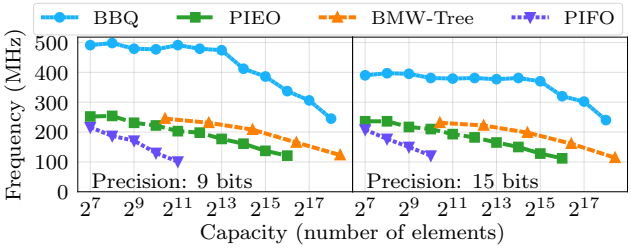


Figure 5: Clock frequency as we scale the queue capacity.

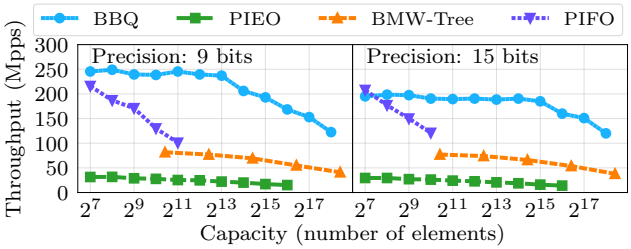


Figure 6: Throughput as we scale the queue capacity.

Figure 5 shows the clock achievable by each design when we increase the queue capacity. BBQ achieves a clock as high as 500 MHz with 9-bit precision and 400 MHz with 15-bit precision, significantly higher than the baseline designs. Moreover, BBQ is able to scale to up to 2^{17} elements while still sustaining a 300 MHz clock. In comparison, PIFO can only scale to up to 2^{11} elements.

In addition to achieving a higher clock frequency, BBQ's fully pipelined design allows it to execute an operation (enqueue or dequeue) every clock cycle. As a result, the throughput difference is even higher compared to BMW-Tree (that consumes 1 cycle to enqueue and 2 cycle to dequeue) and PIEO (that consumes 4 cycles to enqueue and 4 cycles to dequeue). However, different from the other designs, PIFO is able to execute both an enqueue and a dequeue operation in the same cycle, allowing its packet rate to match its clock frequency. Figure 6 shows the throughput of the different designs for different queue capacities. BBQ is able to drive 100 Gbps line rate (148.8 Mpps) with as many as 2^{17} elements. Also note that PIFO is able to achieve similar throughput to BBQ, but only for small queues (128 elements or less).

6.2.2 Resource Scalability

We also evaluate how the different designs scale in terms of FPGA resources. We report both ALM utilization and SRAM blocks, both as a fraction of the overall number of resources available in the target FPGA. Figure 7 shows how the ALM utilization scales as we increase the number of elements that the queue can support. In BBQ, scaling the queue capacity has little effect on the logic utilization. This is a direct consequence of BBQ's use of an integer priority queue, which lets it avoid comparison-based sorting. BMW-Tree's

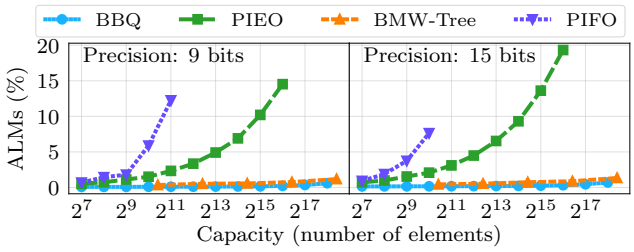


Figure 7: ALM utilization as we scale the queue capacity.

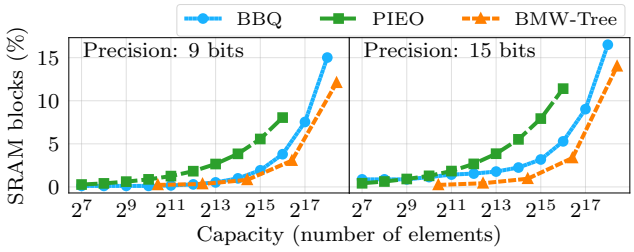


Figure 8: SRAM block utilization as we scale the queue capacity. PIFO is not included in the plot as it does not use SRAM.

hierarchical design also gives it much better scalability, using only slightly more ALM resources than BBQ. In contrast, both PIFO and PIEO use significantly more resources as we scale the capacity.

While BBQ's ALM utilization remains low even for 131,072 elements (2^{17}), BBQ, like PIEO and BMW-Tree, relies on SRAM to store its elements. As a result, we expect SRAM utilization to increase with the queue capacity for all these designs. Figure 8 shows this effect. Note that BBQ's SRAM utilization is in between PIEO's and BMW-Tree's. However, PIEO and BMW-Tree require multiple copies in order to match BBQ throughput, which causes them to use vastly more SRAM if provisioned to meet the same performance target.

6.2.3 BBQ Sensitivity Analysis

To understand the impact of BBQ's configuration on performance, we perform a sensitivity analysis of the bitmap tree parameters: the number of levels (D), and the bitmap width (W) (recall that the number of priorities is computed as $P = W^D$), while keeping the number of elements fixed.

Figure 9 depicts how BBQ's throughput behaves as a function of the bitmap width. We sweep the number of levels in the bitmap tree from $D = 3$ to 15 (or until we reach 2^{15} priorities) for different bitmap widths. Then we plot the attained throughput for the corresponding priority count. We find that bitmap widths between 2 and 8 yield similar performance (with 4 being optimal), but this deteriorates as the bitmap width increases. In particular, starting with $W = 16$, *FFS computation* becomes the primary f_{max} bottleneck. We can similarly infer from the same graph that level count has little impact on performance; for instance, we observe that a $(D = 4, W = 2)$ BBQ achieves the same throughput as a $(D = 8, W = 2)$ BBQ despite the latter containing 16× as

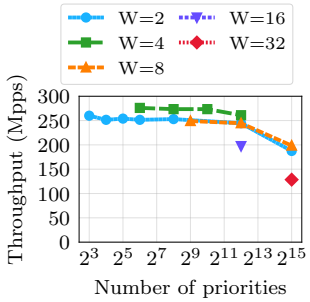


Figure 9: BBQ throughput as we increase the number of priorities when using different bitmap widths (W).

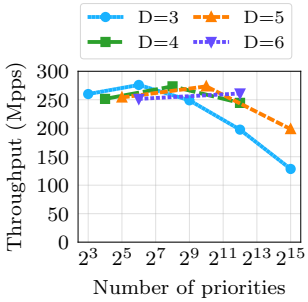


Figure 10: BBQ throughput as we increase the number of priorities when using different number of levels (D).

many priorities. Figure 10 shows the complementary view of the data, depicting change in throughput as a function of priorities for different numbers of tree levels.

6.3 ASIC

We now evaluate BBQ in the context of designs targeting ASICs. Here we compare BBQ with PIFO for two reasons: (1) it is one of the two baseline designs that supports logical partitioning, which, as we discussed in §2, is essential to allow them to be efficiently incorporated into state-of-the-art switches, and (2) it offers the best throughput among all the baseline designs.

We synthesize both BBQ and PIFO using a 7 nm process. PIFO only meets timing at 1 GHz with up to 2^{11} elements, which is consistent with [41]. BBQ meets timing at 3.1 GHz with 2^{17} elements, but we did not try scaling beyond this point. The difference in the clock frequency achieved by BBQ and PIFO means that BBQ is able to run at 55% higher throughput. To evaluate the cost of the design, we compare the chip area when synthesizing a single queue. We use the synthesis results to calculate the area used by the logic gates and estimate the SRAM area using the cost of 0.027mm^2/Mb reported by TSMC for their 7 nm process [12, 46].

Table 4 shows the chip area breakdown, divided in logic and SRAM for both BBQ and PIFO, using 9b and 15b priorities. BBQ uses very little area with logic; most of its area is taken up by SRAM. BBQ is not only able to scale to many more elements than PIFO, but also consumes less area when both are provisioned for the same capacity.

Design	Elements	Priorities	Clock	Area (mm^2)		
				Logic	SRAM	Total
PIFO	2^{11}	2^9	1 GHz	0.043		0.043
	2^{11}	2^{15}	1 GHz	0.058		0.058
BBQ	2^{11}	2^9	3.1 GHz	0.00071	0.0029	0.0037
	2^{11}	2^{15}	3.1 GHz	0.0011	0.035	0.036
	2^{17}	2^9	3.1 GHz	0.00095	0.24	0.24
	2^{17}	2^{15}	3.1 GHz	0.0014	0.29	0.29

Table 4: Chip area for the different designs. BBQ uses little logic, causing its area to be primarily determined by SRAM.

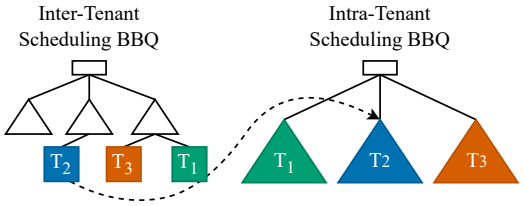


Figure 11: A two-level hierarchical NIC scheduler using BBQ. The first BBQ schedules traffic *across* tenants. The second BBQ (which is logically partitioned) is used to schedule traffic *within* each tenant.

7 Applications

In Section 2, we motivated the need for a hardware priority queue capable of supporting packet scheduling for two rapidly emerging use-cases: terabit-scale switches, and NICs in multi-tenant cloud datacenters. Having evaluated its scalability, throughput, and resource usage, in this section we describe how BBQ can fill these application-level gaps.

7.1 Packet Scheduling on Switches

As described in §2.1.1, a key enabler for priority queue deployment in modern switches is the ability to realize *logical partitioning*, allowing the packet scheduler to leverage the simpler priority queue mesh architecture depicted in Figure 1(b). BBQ achieves this by treating disjoint subtrees in its priority index structure as independent queues (§3.3.3), enabling multiplexing of the underlying instance; as noted earlier, this comes at a cost in terms of precision (each logical queue gets $\frac{1}{k}$'th the original priority range), but with no resource overhead, performance penalty, or fragmentation of queue memory.

In terms of *performance* and *scalability*, we can synthesize BBQ instances with 100K+ entries and 32K priorities at 3.1GHz using a 7nm ASIC process; at two operations per packet, each BBQ instance can sustain a packet rate of 1.55Bpps. Our target switch (NVIDIA SN4700, 400GbE x32) supports an aggregated packet rate of 8.4Bpps, implying that a total of $\lceil \frac{8.4}{1.5} \rceil \times 2 = 12$ BBQs that are logically partitioned among the 32 output ports in a shared scheduler pipeline are sufficient to match the switch fabric's processing speed. Based on our ASIC synthesis results for a single BBQ (§6.3), we estimate that provisioning each of these 12 instances with 131K queue entries and 32K priorities (1K priorities per port) would require a total area of 3.48mm^2. Considering that a switch chip area ranges from 200mm^2 to 800mm^2 [41] this corresponds to 0.4–1.74% of the total chip area.

Thus, BBQ's scalability, performance, and ability to be logically partitioned make it, for the first time, a plausible candidate to realize priority queueing in modern switches.

7.2 Packet Scheduling on Cloud SmartNICs

A key requirement for NIC-based packet schedulers in public clouds is the ability to independently perform scheduling both *across* tenants and *within* each tenant (§2.1.2). By allowing several logical BBQs (each representing one tenant) to share a single BBQ instance, we can efficiently use the available resources (*e.g.*, queue memory) without having to provision one priority queue per tenant. In order to enforce cross-tenant traffic policies, we instantiate another, smaller BBQ that stores references to the tenant BBQs. Conceptually, this corresponds to the two-level hierarchical scheduler depicted in Figure 11, with the lower and upper levels handling intra- and inter-tenant scheduling decisions, respectively.

Previously, we evaluated the feasibility of operating BBQ on an ASIC in the context of switches (§7.1). We now frame our discussion about scalability and performance for SmartNICs in the context of the more resource-constrained device family: FPGAs. On an Intel Stratix 10MX we can synthesize a BBQ instance with 100K+ entries and 32K priorities that meets timing at 302MHz, and uses 0.45% of the available ALMs and and 9% of the total FPGA SRAM, respectively. Consequently, a single instance can sustain 151Mpps, surpassing 100GbE line rate with minimum-sized packets (148.8 Mpps).

On FPGAs, scaling to higher line rates (*e.g.*, 400GbE [22]) is beyond the capability of any single priority queue instance, and would require augmenting the scheduler pipeline with multiple BBQs. However, given the resource cost of each instance relative to total FPGA resources, this is currently impractical; any priority queue design would have to significantly reduce its SRAM footprint (*e.g.*, offloading elements to DRAM) in order to make scaling out on FPGAs practical.

8 Related Work

Counting priority index: The data structure used in BBQ is similar to the counting priority index (CPI) proposed by Wang and Lin [45]. They were the first to hypothesize that an integer priority queue could be used to speed up packet scheduling in both software and hardware. Unfortunately, CPI is not implementable in hardware in its original form as it fails to account for the many practical issues that arise when building a pipelined hardware design, e.g., memory access latency, hazards, and limited memory. As we discussed in §3 and §4, the challenging aspects of BBQ's design stem from these very issues. BBQ is also orthogonal to Eiffel [38], which deals with the practical issues of using a priority index to schedule packets in *software*.

Hardware priority queueing: PIFO [41] is the current state-of-the-art priority queue implementation in terms of throughput, and BMW-RPU [47] is the current state-of-the-art in terms of scalability. As confirmed in our evaluation, BBQ is able to match PIFO's throughput while scaling beyond BMW-RPU's maximum capacity. Another notable hardware priority queue design is pHeap [10], unfortunately pHeap can only process an operation every two clock cycles, which makes it unsuitable for line-rate switches. Further, besides PIFO and PIEO, none of the designs that close the gap in terms of scalability (including BMW-Tree and pHeap) support logical partitioning efficiently, making them impractical for

deployment in both switches and SmartNICs in the cloud setting. We characterize the efficiency that existing designs achieve in implementing logical partitioning in §A.1.

Approximate priority queueing: There is also a line of work that proposes approximating different scheduling algorithms to make them amenable to hardware implementation [2, 3, 17, 39, 48]. BBQ borrows from these works the observation that a small priority set (at the hardware level) is sufficient for most use cases. These works provide interesting theoretical insights and a path to implement some scheduling policies on existing programmable switches. However, BBQ's design shows that it is unnecessary to sacrifice accuracy in order to achieve scalability and speed.

9 Discussion

Limitations: A key limitation of IPQ-based designs such as BBQ is that they operate over priority ranges that are both *finite*[7] and *static*. While we can, in fact, augment the vanilla BBQ primitive to support *dynamic* priority ranges (Appendix D), boundedness of the priority span remains an immutable constraint. The fundamental reason is that we must map every priority in BBQ to a bucket in physical memory, so SRAM usage scales linearly with the priority span. Notably, this scheme becomes altogether impractical when the required precision grows beyond a certain threshold (*e.g.*, supporting 2^{32} priorities would require over 500MB of SRAM *just* to store bitmaps). In §6, we demonstrated the feasibility of synthesizing a BBQ instance with 15-bit priority tags (32K priorities), but we don't expect this number to scale much further. Thus, the ideal operating point for BBQ corresponds to a setting where we need to support a *large number of queue entries falling in a small (possibly dynamic) priority range*. Some priority queue architectures also enable richer abstractions (*e.g.*, PIEO's predicate-based filtering allows scheduling based on eligibility criteria such as virtual or wall-clock time [40]), which BBQ does not support.

Future Work: Today, we are at an inflection point with regard to the scalability of hardware priority queue designs. On the one hand, support for 100K+ queue entries is the culmination of a decade-long concerted effort towards jointly optimizing scalability and performance. On the other hand, this appears to be the end of the scalability roadmap: since every queue element must be stored *somewhere*, we are ultimately bottlenecked by available memory. For the sake of performance, today's designs exclusively use SRAM, which offers deterministic, single-cycle memory access. However, SRAM is a scarce resource, and even highly scalable designs such as BBQ and BMW-Tree [47] would require over 10% of the available FPGA SRAM (§6.2) to support 200K queue entries – a highly impractical proposition. However, given the trend of increasing multi-tenancy in datacenters, it is not far fetched to believe that schedulers will some day need priority queueing for 1M+ flows. A natural question then is: *how do we get there?* We believe the key to this lies in offloading queue entries to DRAM, which provides much slower (and non-deterministic) access latencies compared to SRAM, but is a far more abundant memory resource. The clean decoupling between BBQ's priority index structure and its queue memory (i.e., BBQ's ability to locate the highest-priority entry without needing access to the entry itself) makes it feasible to offload queue memory to DRAM, but there are several challenges that need to be addressed along the way. We leave it to future work to realize this lofty goal.

[7] Technically, priority ranges are *always* finite (regardless of the underlying priority queue design) because they are ultimately upper-bounded by the maximum precision afforded by the priority tag (i.e., number of priority bits). However, the point here is that comparison-based priority queue designs (*e.g.*, PIFO) can, in principle, create the *illusion* of an infinite priority range using large priority tags; for instance, a time-based PIFO scheduler that uses nanosecond-granularity timestamps as priorities would require well over 500 years to exhaust a 64-bit priority range ($2^{64} = 1.8 \times 10^{19}$ priorities). Conversely, IPQs hit their priority scaling limits far earlier than any reasonable interpretation of infinity.

10 Conclusion

PIFO's vision—a programmable packet scheduler that operates at line rate even on high-throughput switches—has been hampered by throughput and scalability issues of existing priority queue designs. In this paper, we presented BBQ, a new priority queue design that is both scalable and fast. At the heart of its design is an integer priority queue that allows BBQ to avoid the complexity barrier imposed by comparison-based sorting. While this paper shows the usefulness of BBQ for performing packet scheduling in both FPGA SmartNICs and line rate switch ASICs, we expect such a high-performance priority queue to find use in many other contexts. We look forward to future work that creatively use BBQ for other purposes.

Acknowledgements

We thank our shepherd, Mina Arashloo, and the anonymous reviewers for their insightful feedback. We are also indebted to Anup Agarwal for inspiring us to write this paper, Andrew Boutros for his guidance on ASIC design, Weina Wang for her help with the theory, Vyas Sekar for shaping the story, and the Azure Host Networking and Scheduler teams (particularly Mahmoud Elhaddad, Idan Regev, and Dongwook Lee) for valuable discussions on design and use-cases. This work was funded by Intel and VMware through the Intel/VMware Crossroads 3D-FPGA Academic Research Center, a VMWare Systems Research Award, NSF Award 2007733, ONR Award #N00014-24-1-2059, a Cylab Presidential Fellowship, and a Google Research Gift.

References

[1] Alexandru Agache, Razvan Deaconescu, and Costin Raiciu. Increasing datacenter network utilisation with GRIN. In *12th USENIX Symposium on Networked*

Systems Design and Implementation (NSDI 15), pages 29–42, 2015.

[2] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. SP-PIFO: Approximating Push-In First-Out behaviors using Strict-Priority queues. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 59–76, Santa Clara, CA, February 2020. USENIX Association.

[3] Albert Gran Alcoz, Balázs Vass, Gábor Rétvári, and Laurent Vanbever. Everything matters in programmable packet scheduling. *arXiv preprint arXiv:2308.00797*, 2023.

[4] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 435–446, New York, NY, USA, 2013. Association for Computing Machinery.

[5] AMD. AMD EPYC 4th gen 9004 & 8004 series server processors – details, 2023. https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series.html#specs.

[6] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 93–109, Santa Clara, CA, February 2020. USENIX Association.

[7] Nirav Atre, Hugo Sadok, Erica Chiang, Weina Wang, and Justine Sherry. SurgeProtector: Mitigating temporal algorithmic complexity attacks using adversarial scheduling. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 723–738, New York, NY, USA, August 2022. Association for Computing Machinery.

[8] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Weicheng Sun. Pias: Practical information-agnostic flow scheduling for data center networks. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, page 1–7, New York, NY, USA, 2014. Association for Computing Machinery.

[9] J. C. R. Bennett and Hui Zhang. WF2Q: Worst-case fair weighted fair queueing. In *Proceedings of IEEE INFOCOM '96. Conference on Computer Communications*, volume 1 of *INFOCOM '96*, pages 120–128 vol.1, 1996.

[10] R. Bhagwan and B. Lin. Fast and scalable priority queue architecture for high-speed network switches. In *Proceedings IEEE INFOCOM 2000 Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Society*, volume 2 of *INFOCOM 2000*, pages 538–547 vol.2, 2000.

[11] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. Association for Computing Machinery.

[12] Jonathan Chang, Yen-Huei Chen, Wei-Min Chan, Sahil Preet Singh, Hank Cheng, Hidehiro Fujiwara, Jih-Yu Lin, Kao-Cheng Lin, John Hung, Robin Lee, Hung-Jen Liao, Jhon-Jhy Liaw, Quincy Li, Chih-Yung Lin, Mu-Chi Chiang, and Shien-Yang Wu. A 7nm 256Mb SRAM in high-k metal-gate FinFET technology with write-assist circuitry for low-VMIN applications. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 206–207, 2017.

[13] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal*, 53:105–115, 2016.

[14] Daniel Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17, pages 315–328, Boston, MA, March 2017. USENIX Association.

[15] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pages 51–66, Renton, WA, April 2018. USENIX Association.

[16] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. Corundum: An open-source 100-Gbps

NIC. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '20, pages 38–46. IEEE, 2020.

[17] Peixuan Gao, Anthony Dalleggio, Yang Xu, and H. Jonathan Chao. Gearbox: A hierarchical packet scheduler for approximate weighted fair queuing. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '22, pages 551–565, Renton, WA, April 2022. USENIX Association.

[18] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 1–14, 2015.

[19] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. MasQ: RDMA for virtual private cloud. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, pages 1–14, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Intel. Intel, Baidu drive intelligent infrastructure transformation, 2020. https://www.intel.com/content/www/us/en/newsroom/news/baidu-intelligent-infrastructure-transformation.html#gs.5vl4ru.

[21] Intel. FPGA Design Software – Intel Quartus Prime, 2023. https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html.

[22] Intel. Intel Agilex 7 FPGAs and SoCs product brief, 2023. https://www.intel.com/content/www/us/en/content-details/762901/intel-agilex-7-fpgas-and-socs-product-brief.html.

[23] Intel. Intel infrastructure processing unit (Intel IPU) platform (codename: Oak Springs Canyon), 2023. https://www.intel.com/content/www/us/en/products/platforms/details/oak-springs-canyon.html.

[24] Intel. Intel Stratix 10 MX 2100 FPGA, 2023. https://ark.intel.com/content/www/us/en/ark/products/210297/intel-stratix-10-mx-2100-fpga.html.

[25] Aggelos Ioannou and Manolis G. H. Katevenis. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Transactions on Networking*, 15(2):450–461, April 2007.

[26] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A high-performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 243–259. USENIX Association, November 2020.

[27] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From luna to solar: The evolutions of the compute-to-storage networks in Alibaba Cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 753–766, New York, NY, USA, 2022. Association for Computing Machinery.

[28] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Universal packet scheduling. In *13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '16, pages 501–521, 2016.

[29] Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Recursively cautious congestion control. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 373–385, Seattle, WA, April 2014. USENIX Association.

[30] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

[31] Ali Munir, Ghufran Baig, Syed Mohammad Irteza, Ihsan Ayyub Qazi, Alex X Liu, and Fahad Rafique Dogar. Pase: synthesizing existing transport strategies for near-optimal data center transport. *IEEE/ACM Transactions on Networking*, 25(1):320–334, 2016.

[32] Nvidia. ConnectX-7 400G Adapters: Smart, accelerated networking for modern data center infrastructures, 2023. https://nvdam.widen.net/s/csf8rmnqwl/infiniband-ethernet-datasheet-connectx-7-ds-nv-us-2544471.

[33] Nvidia. Nvidia spectrum sn4000 series switches, 2023. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/br-sn4000-series.pdf.

[34] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. SENIC: Scalable NIC for end-host rate limiting. In *11th USENIX Symposium on Networked Systems*

Design and Implementation, NSDI '14, pages 475–488, Seattle, WA, April 2014. USENIX Association.

[35] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. Ensō: A streaming interface for NIC-application communication. In *17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '23, pages 1005–1025, Boston, MA, July 2023. USENIX Association.

[36] Hugo Sadok, Aurojit Panda, and Justine Sherry. Of apples and oranges: Fair comparisons in heterogenous systems evaluation. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, HotNets '23, pages 1–8, New York, NY, USA, 2023. Association for Computing Machinery.

[37] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S. Berger, James C. Hoe, Aurojit Panda, and Justine Sherry. We need kernel interposition over the network dataplane. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 152–158, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. Eiffel: Efficient and flexible software packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, pages 17–32, Boston, MA, February 2019. USENIX Association.

[39] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 685–699, Santa Clara, CA, February 2020. USENIX Association.

[40] Vishal Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 367–379, New York, NY, USA, 2019. Association for Computing Machinery.

[41] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 44–57, New York, NY, USA, 2016. Association for Computing Machinery.

[42] Brent Stephens, Aditya Akella, and Michael M. Swift. Loom: Flexible and efficient NIC packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, pages 33–46, Boston, MA, February 2019. USENIX Association.

[43] Synopsys. Design Compiler, 2023. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[44] Vinay Vashishtha, Manoj Vangala, and Lawrence T. Clark. ASAP7 predictive design kit development and cell design technology co-optimization: Invited paper. In *2017 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD, pages 992–998, 2017.

[45] Hao Wang and Bill Lin. Per-flow queue management with succinct priority indexing structures for high speed packet scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 24(7):1380–1389, 2013.

[46] Shien-Yang Wu, C.Y. Lin, M.C. Chiang, J.J. Liaw, J.Y. Cheng, S.H. Yang, C.H. Tsai, P.N. Chen, T. Miyashita, C.H. Chang, V.S. Chang, K.H. Pan, J.H. Chen, Y.S. Mor, K.T. Lai, C.S. Liang, H.F. Chen, S.Y. Chang, C.J. Lin, C.H. Hsieh, R.F. Tsui, C.H. Yao, C.C. Chen, R. Chen, C.H. Lee, H.J. Lin, C.W. Chang, K.W. Chen, M.H. Tsai, K.S. Chen, Y. Ku, and S. M. Jang. A 7nm CMOS platform technology featuring 4th generation FinFET transistors with a 0.027um2 high density 6-T SRAM cell for mobile SoC applications. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 2.6.1–2.6.4, 2016.

[47] Ruyi Yao, Zhiyu Zhang, Gaojian Fang, Peixuan Gao, Sen Liu, Yibo Fan, Yang Xu, and H. Jonathan Chao. BMW tree: Large-scale, high-throughput and modular PIFO implementation using balanced multi-way sorting tree. In *Proceedings of the ACM SIGCOMM 2023 Conference*, SIGCOMM '23, pages 208–219, New York, NY, USA, 2023. Association for Computing Machinery.

[48] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 179–193, New York, NY, USA, August 2021. Association for Computing Machinery.

[49] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS kernel support for a low-overhead container overlay network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 331–344, 2019.

Appendix A Logical Partitioning in Practice

In §2, we motivated *logical partitioning* (i.e., the ability to multiplex several logical priority queues atop a single physical queue) as a key requirement for deployability in modern switches and SmartNICs. In this section, we first characterize the extent to which existing priority queue designs can realize logical partitioning (§A.1), followed by a detailed description of the mechanisms that enable BBQ to achieve this functionality (§A.2).

A.1 Existing Designs

The goal of logical partitioning is to allow a single, physical priority queue to emulate a collection of multiple, *logically-independent* priority queues. Simply realizing this abstraction is not particularly challenging, but doing so *efficiently* turns out to be a major impediment for most priority queue designs. We can evaluate efficiency along three axes: (1) *queue fragmentation*, or the worst-case fraction of queue elements (QEs) lost to external fragmentation when an instance is partitioned q ways; (2) *performance overhead*, or the throughput degradation resulting from logical partitioning; and, (3) *resource overhead*, or the resource cost (*e.g.*, logic, memory) required to support q logical partitions relative to an unpartitioned priority queue.

PIFO supports logical partitioning with zero queue fragmentation, a small performance overhead, and a resource overhead that scales linearly with queue size: Consider the example depicted in Figure 12, where a physical PIFO (provisioned with $N = 8$ QEs) is partitioned into $q = 2$ logical PIFOs. Per usual, PIFO maintains a sorted list of QEs ordered by priority [41]. To realize logical partitioning, PIFO annotates every QE with a *Logical PIFO ID* (in this case, q_0 or q_1) at enqueue time. Then, in order to dequeue from the i'th logical PIFO, it first "selects" the subset of elements with the corresponding ID (q_1 in our example), then performs priority decoding to extract the highest-priority element from that subset. Since every QE in the physical instance can always be independently addressed by every logical PIFO, queue memory is fully multiplexed, yielding zero fragmentation. Every element must be annotated with a $\log_2 q$ bit wide ID, resulting in a resource overhead that scales with queue size. Finally, selecting the appropriate subset of elements involves an extra comparator per element, incurring a small performance cost.

With modest changes, PIEO can support logical partitioning with zero queue fragmentation, and performance/resource overheads that scale with queue size: Architecturally, PIEO is organized as a matrix: an array of $2\sqrt{N}$ *sublists*, each consisting of $\sqrt{N}$ QEs sorted by priority. Besides standard priority queue operations, PIEO allows specifying *eligibility predicates*: a programmable function that "selects" a subset of elements to dequeue from. In principle, this is similar to the mechanism PIFO uses to implement logical partitioning (Figure 12), and an appropriate predicate

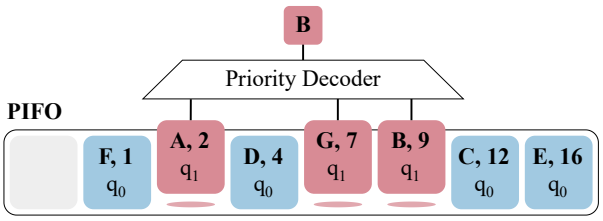


Figure 12: A single physical PIFO partitioned into 2 logical PIFOs: q_0 consisting of 4 elements, and q_1 consisting of 3 elements. Available queue memory is fully multiplexed among the logical PIFOs, resulting in zero fragmentation.

function can be used in PIEO to the same effect. Unfortunately, the *vanilla PIEO design does not allow QEs belonging to different logical PIEOs to coexist in the same sublist*. The reason is that, as a first step, PIEO must perform predicate filtering at sublist level, which only supports range-based queries (*e.g.*, $a \leq f \leq b$) but not set queries (*e.g.*, $f \in X$) that are required for logical partitioning. Consequently, it would incur external fragmentation at the granularity of sublists. However, we note that this is not a fundamental limitation, and with minor changes to the design (*e.g.*, by annotating every sublist with a q-bit bitmap representing the logical PIEO QEs contained therein), PIEO can, in theory, achieve logical partitioning with no external fragmentation while incurring a resource/performance cost similar to PIFO.

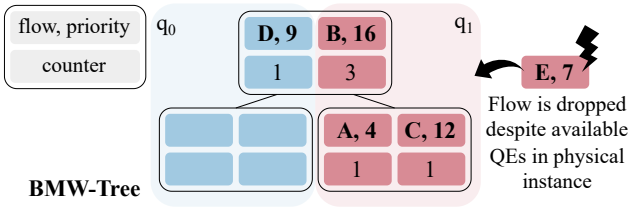


Figure 13: A 2-level 2-way BMW-Tree instance partitioned into 2 logical queues. Each logical queue must be mapped to a *physical* subtree, resulting in external fragmentation. Once a subtree becomes full, enqueues into the corresponding logical queue are impossible even if there are available QEs in other subtrees.

Other comparison-based priority queue designs cannot efficiently implement logical partitioning: PIFO and PIEO both satisfy a property that is key to achieving zero fragmentation in fixed-layout priority queues: maintaining a total ordering over elements at all times. *In their effort to leverage spatial and pipelined parallelism, other designs violate this property, inducing significant queue fragmentation*. We illustrate this point using the 2-way BMW-Tree depicted in Figure 13. To implement partitioning between $q = 2$ logical BMW-Trees, each logical queue must be statically mapped to a *physical* subtree as shown in the figure;[8] any other arrangement might result in inadvertently dequeueing from the wrong logical queue. The result is that, for a BMW-Tree instance of size N, each logical queue can address only $\frac{N}{q}$

[8] Heap property is intentionally violated at root level to enable partitioning.

QEs, incurring a fragmentation cost that scales with q. For instance, with $q = 8$ partitions, we would risk losing up to 87% of the queue memory to external fragmentation (and still not be able to support N flows). Instead, over-provisioning the physical instance to account for the worst case (all N elements being enqueued in a single logical queue, while the other 7 queues remain empty) would incur a 700% memory overhead. Other tree-like priority queue designs, such as pHeap [10] and Pipelined Heap [25], encounter precisely the same issue.

Overall, we find that besides the designs that maintain a total ordering over elements (PIFO and PIEO), *realizing logical partitioning in comparison-based designs incurs prohibitively high resource overhead, queue fragmentation cost, or both.*

A.2 Logical Partitioning in BBQ

Given a BBQ instance with w-bit bitmaps, we illustrate how to partition it into q logical BBQs by means of two exemplar configurations: one where $q \leq w$, and another where $q > w$.

(1) Fewer logical partitions than the bitmap width ($q \leq w$): Consider a 2-level BBQ with 4-bit bitmaps (i.e., $w = 4$, $D = 2$) that we would like to partition into $q = 2$ logical BBQs. The physical BBQ has a priority span of $P = 4^2 = 16$ priorities. As a first step, we partition this range equally between the two logical instances, allocating priorities $[0,7]$ to queue q_0, and $[8,15]$ to q_1. Observe that, in order to facilitate this split, the L_1 bitmap also needs to be partitioned as shown in Figure 14, with the lower two bits corresponding to q_0, and the upper two bits corresponding to q_1. Enqueueing an entry, X, into logical queue $i \in [0,1]$ with relative priority $j \in [0,7]$ is simple: first, we compute the *absolute* priority corresponding to the physical BBQ as $p = (i \times 8) + j$,[9] then perform `ENQUEUE(X, p)` as usual. In order to dequeue the highest (or lowest) priority entry from the i'th logical queue, we first mask the bits in the root bitmap *not* corresponding to q_i (*e.g.*, for $i = 1$, we would apply the mask `0xC`), perform FFS on them, then proceed down the bitmap tree as in a typical `DEQUEUE` operation.

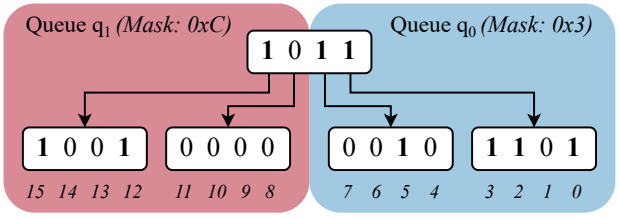


Figure 14: Bitmap tree for a 2-level BBQ with 4-bit bitmaps partitioned into 2 logical BBQs, q_0 and q_1. Each logical BBQ is allocated disjoint ranges of 8 priorities. To `DEQUEUE` from a logical BBQ, we first apply the corresponding *mask* to the L_1 bitmap before performing FFS on it.

(2) More logical partitions than the bitmap width ($q > w$): Consider again a ($w = 4$, $D = 2$) BBQ that we would now like to partition into $q = 8$ logical BBQs. Observe that each bit in the root bitmap now corresponds to *two* different logical BBQs, and therefore does not offer any discriminatory power.[10] Consequently, we eliminate this level of the bitmap tree; in its place, we insert a single pipeline stage that steers operations to their respective subtrees based on the logical queue index (*e.g.*, `ENQUEUE` and `DEQUEUE` operations on $i \in \{2,3\}$ are steered to the second-from-right subtree). The updated bitmap tree structure is depicted in Figure 15. Each L_2 bitmap now maps to $q' = 2$ different logical BBQs, and we apply the idea described in (1) (since $q' \leq w$) to achieve this partitioning.

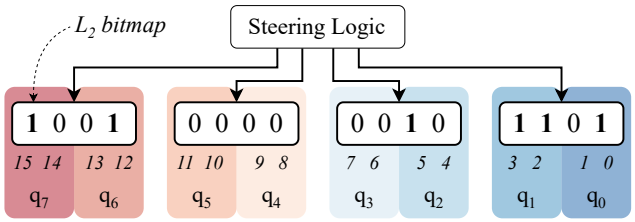


Figure 15: Bitmap tree for a 2-level BBQ with 4-bit bitmaps partitioned into 8 logical BBQs. The root (L_1) bitmap no longer adds any value, so we replace it with a *steering stage* that simply routes operations on logical BBQs to the corresponding subtree.

Thus, with nominal changes to the BBQ pipeline, we can support a broad range of partitioning configurations without any performance overhead. In contrast to PIFO and PIEO, the resource cost (corresponding to over-provisioning the priority range) scales with the *degree of logical partitioning* rather than queue size. Finally, full decoupling between its priority index structure (i.e., the bitmap tree) and QEs enables BBQ to achieve zero queue fragmentation. Overall, these techniques make it possible for BBQ to *efficiently* realize logical partitioning.

Appendix B Using StOCs in Practice

In §4.2, we described how every bit in BBQ's bitmap tree is associated with a StOC, which represents the *total number of elements contained in the corresponding subtree*. StOCs are an important component in BBQ because they are a key enabler for its fully-pipelined architecture. In this section, we characterize two practical details regarding our implementation of StOCs: how they are sized (§B.1), and a general optimization technique that improves their performance (§B.2).

B.1 Sizing

In order to handle the worst case (i.e., all elements in the BBQ being contained in a single priority bucket), every StOC must be provisioned to represent the range $[0, N]$, where N is the number of supported queue elements. Conventionally, N is configured to be a power of two (i.e., 2^k) to avoid wasting resources such as pointer address bits. However, in the case

[9] Logically, this corresponds to simply concatenating together i and j.

[10] Since either of the logical BBQs contained therein may be empty, a '1' in any bit position of the L_1 bitmap provides no guarantee that a `DEQUEUE` operation on that subtree will succeed.

of BBQ, naively provisioning the queue with $N = 2^k$ elements would entail $(k+1)$-bit StOCs, with the most-significant bit (MSb) only ever being used to encode the maximum occupancy of 2^k. Instead, in BBQ, we snap N to a value of the form $(2^k - 1)$, allowing us to use k-bit StOCs. Thus, carefully sizing the queue (and deliberately wasting one element's worth of address space) saves 1 bit per StOC, yielding a sizeable reduction in memory footprint.

B.2 Waterlevel Bit Optimization

Since StOC bit-widths scale with the queue size (§B.1), performing arithmetic or logical operations on these counters can be expensive for large BBQ instances. As we will see, these operations sometimes need to be chained together with other combinational logic in a single pipeline stage, which in turn inflates the critical path and significantly degrades f_{max}. In this section, we describe a general optimization technique that alleviates counter-related performance bottlenecks, yielding up to 17% higher f_{max} for some BBQ configurations.

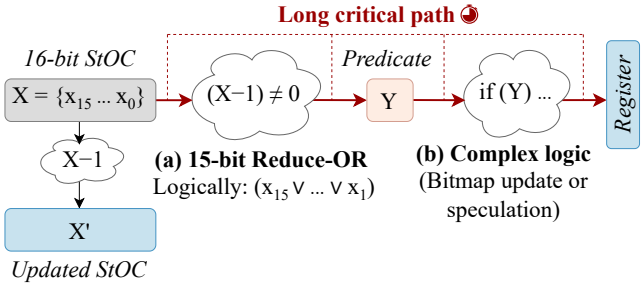


Figure 16: Dependency chain involving 16-bit counter logic for a `DEQUEUE` operation. The critical path comprises of (a) a 15-bit Reduce-OR (to determine whether the StOC becomes 0), chained with (b) more combinational logic (which uses (a) as a predicate).

To illustrate the problem, consider a `DEQUEUE` operation in stage 6 of the BBQ pipeline depicted in Table 2. This stage comprises of several sub-operations, two of which we will focus on here: (a) decrementing an L_2 StOC and checking if the resulting value becomes zero, and (b) updating the L_2 bitmap predicated on the result of (a). Note that (a) operates on a $log_2(N+1)$-bit counter and (b) operates on a w-bit bitmap, and chaining these sub-operations together inevitably puts them on the critical path. The problem is further exacerbated if (b) entails more complex combinational logic (*e.g.*, resolving speculation outcomes, which, as shown in step ④ of Figure 4, follows the same blueprint). The critical path for this sequence of sub-operations is depicted in Figure 16.

To address this performance bottleneck, we augment every StOC in BBQ with an additional bit called the **waterlevel bit** (WLb). We maintain the invariant that the WLb is set to '1' if the current StOC value is greater than or equal to 2, otherwise it is set to '0'. The key idea is that *the WLb opportunistically memoizes the future result of (a)*,[11] allowing

[11] Observe that, for positive values of X, the expression evaluated by (a), $(X-1) \neq 0$, is logically equivalent to $X \geq 2$, the value encoded in the WLb.

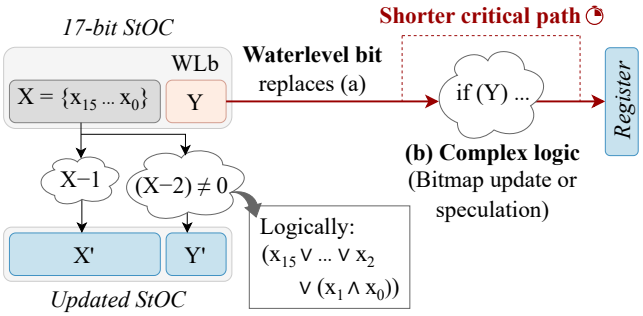


Figure 17: The waterlevel bit (WLb) replaces (a), improving f_{max} by removing the counter operations from the critical path.

it to directly serve as the predicate for (b) instead of having to compute it from scratch (Figure 17). This avoids chaining the expensive Reduce-OR computation with other complex combinational logic, thereby shrinking the critical path. Moreover, the updated value of the WLb (corresponding to the current StOC value minus 2) can be efficiently computed using another 16-bit operation; however, since this is not chained with additional logic, it does not appear on the critical path.

In the context of BBQ, this optimization yields between $5-17\%$ higher f_{max} on the Stratix 10 MX FPGA for configurations with $(2^{17}-1)$ queue entries (i.e., 17-bit StOCs). This does come at a resource cost, since every StOC must now be one bit wider to accommodate the WLb (corresponding to approximately 4.75kB higher SRAM usage for a BBQ that supports 32K priorities with 8-bit bitmaps). However, we find that the resulting performance improvements justify this resource overhead, and we enable this optimization by default in the BBQ artifact.

Finally, we note that the underlying technique – memoizing useful counter arithmetic results in the counter structure itself – is a general one that may benefit *any* design which employs occupancy counters that might ultimately appear on the critical path (*e.g.*, BMW-Tree [47]).

Appendix C BBQ$_\odot$: A Latency-Free BBQ

In §5, we briefly described the latency artifacts that arise due to pipelining, and how they might cause the packet schedule produced by BBQ to deviate from that of an "ideal" priority queue. In this section, we describe BBQ$_\odot$, an augmentation of BBQ that counteracts the latency issue. We start with a concrete example motivating the problem (§C.1), then dive into BBQ$_\odot$'s design (§C.2), followed by a proof of its correctness (§C.3).

C.1 Motivating Example

Consider the scenario depicted in Figure 18, where we use a BBQ instance with a pipeline latency of $p = 4$ cycles to implement strict priority scheduling at a bottleneck switch. There is a single high-priority flow, A, competing with 3 lower-priority flows (B, C, and D); if none of the flows are application-limited, we expect A to receive the full share of

bandwidth, while the other flows should starve (i.e., never be served). Assume now that a single packet can be transmitted every other cycle (i.e., line rate corresponds to one packet every two cycles). Since it takes 4 cycles for the BBQ instance to complete a DEQUEUE request and yield the appropriate flow to schedule, we are faced with two alternatives for managing priority queue state.

First, when a flow is scheduled, we might re-enqueue the flow in the BBQ and immediately issue another DEQUEUE request, wait 4 cycles for the BBQ to respond with the next flow to schedule, and so on. This guarantees accuracy (i.e., the scheduled flow is always the highest-priority one), but implies that flows can only be scheduled every 4 cycles, wasting half the link bandwidth.[12]

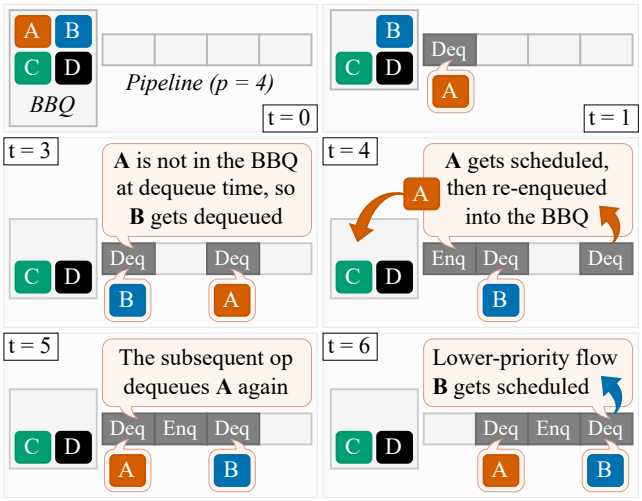


Figure 18: Issuing concurrent DEQUEUE requests, in combination with BBQ's pipeline latency, incorrectly causes a lower-priority flow, B, to be extracted (at $t = 3$) and scheduled (at $t = 6$).

The second option is to preemptively maintain *multiple* concurrent DEQUEUE requests in flight such that a flow is *always* available to be scheduled. In our example, this corresponds to issuing DEQUEUE operations 2 cycles apart such that a flow gets dequeued every other cycle (*e.g.*, at $t = 5$ in Figure 18). While this saturates the link bandwidth, it also implies that *not all active flows are enqueued in the BBQ at dequeue time*. For instance, at $t = 1$, the first issued DEQUEUE operation extracts flow A from the BBQ. Consequently, at $t = 3$, the next DEQUEUE operation results in the extraction of a lower-priority flow, B. Flow A eventually returns to the BBQ at $t = 4$, but it is far too late by this point: at $t = 6$, the second DEQUEUE operation completes, yielding B. In effect, this violates the strict priority scheduling requirement we sought to enforce.

[12] Alternatively, we might schedule *bursts* of packets at a time so as to hide the pipeline latency, but this effectively imposes a leaky bucket atop the underlying scheduling policy, which may not always be desirable.

C.2 BBQ$_{\odot}$ Design

BBQ$_{\odot}$ is composed of two components: a BBQ and a PIFO instance, which are connected as shown in Figure 19. To simplify the theoretical analysis of this system (§C.3), we assume that both components run at the same clock frequency (say 250MHz), and the system clock, denoted by CLOCK$_{sys}$ runs at half the frequency. On each CLOCK$_{sys}$ cycle, we can insert one element into the BBQ$_{\odot}$, extract the highest-priority element, or both.

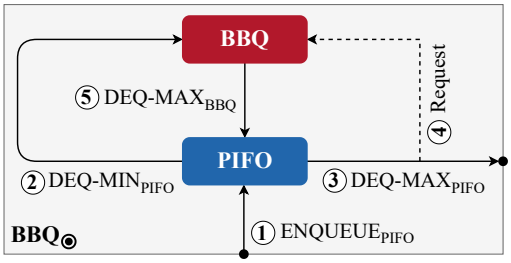


Figure 19: BBQ$_{\odot}$ design.

At a high level, our goal is to keep the PIFO full, only inserting into the BBQ when low-priority elements "spill over" from the PIFO. When a new element arrives at the system, it is first ENQUEUE'd into the PIFO ①; if this causes the PIFO to overflow its capacity (k), we perform a DEQUEUE-MIN$_{PIFO}$ operation ②, inserting the resulting element into the BBQ. Extracting the highest-priority element from the system involves two steps: (a) we perform a DEQUEUE-MAX$_{PIFO}$ operation to get the highest-priority element in the PIFO ③, which completes immediately, and (b) we issue a DEQUEUE-MAX$_{BBQ}$ request to fetch the highest-priority element from the BBQ ④, which takes p timesteps to complete (where p is the pipeline latency of the BBQ in units of CLOCK$_{sys}$ cycles). Finally, when a DEQUEUE-MAX$_{BBQ}$ operation completes ⑤, we insert the resulting element into the PIFO; as before, if this causes the PIFO to overflow, we move the lowest-priority element in the PIFO to the BBQ. Observe that on every CLOCK$_{sys}$ cycle (corresponding to 2 clock cycles for each component), we issue at most 2 BBQ operations and 4 PIFO operations (2 ENQUEUEs and 2 DEQUEUEs), which matches their respective operation throughputs.

C.3 Proof of Theorem 1

In this section, we prove a sufficient condition for BBQ$_{\odot}$ to guarantee zero accuracy loss when using an appropriately-sized PIFO. We start by proving a lower bound on PIFO occupancy when the associated BBQ is not empty, followed by an invariant regarding the subset of priorities contained in the PIFO at any time. In the remainder of the section, we use $\mathcal{P}(t)$ to denote the set of elements contained in the PIFO at time t, and $|\mathcal{P}(t)|$ to denote the cardinality of this set (and transitively the PIFO occupancy).

Lemma 1 (Lower-Bound on PIFO Occupancy). *In a BBQ$_{\odot}$ instance composed of a BBQ with pipeline latency*

p cycles and a PIFO of size $k > p$, if $|\mathcal{P}(t)| < (k-p)$, the BBQ is empty at time t.

Proof. The intuition behind the Lemma is that, so long as the BBQ is *not* empty, it will prevent the PIFO occupancy from dropping below a certain threshold (corresponding to $k-p$). We prove this claim via contradiction, showing that if $|\mathcal{P}(t)| < (k-p)$ starting with a full PIFO (a necessary condition for the BBQ to be non-empty), at least one DEQUEUE-MAX$_{\text{BBQ}}$ request resulted in $\varnothing$ after p timesteps, implying that the BBQ is empty at time t.

Let t_1 denote the *latest* time that the PIFO was full, and let t_2 denote the *earliest* time such that $|\mathcal{P}(t_2)| < (k-p)$. Note that no elements are inserted in the BBQ in the period $[t_1, t_2]$; otherwise $\exists\, t_1' > t_1$ where the PIFO is still full, implying that t_1 was not the latest time. Assume towards a contradiction that a total of n_1 DEQUEUE-MAX$_{\text{PIFO}}$ operations and $n_2 \geq 0$ ENQUEUE$_{\text{PIFO}}$ operations were performed in $[t_1, t_2]$, and n_3 DEQUEUE-MAX$_{\text{BBQ}}$ operations completed in the same period, all of which returned non-$\varnothing$ values.

$$n_1 - (n_2 + n_3) > p, \tag{1}$$

$$n_3 \geq \max(0, n_1 - p) \tag{2}$$

where (1) is true because the difference between the number of departures from the PIFO (n_1) and the number of arrivals to the PIFO ($n_2 + n_3$) in $[t_1, t_2]$ must correspond to an occupancy drop from k to $|\mathcal{P}(t_2)| < k-p$, i.e., exceeding p. (2) is true because a DEQUEUE-MAX$_{\text{BBQ}}$ request is issued for every DEQUEUE-MAX$_{\text{PIFO}}$ operation, and the maximum number of requests still outstanding is at most p. Substituting (1) into (2), we get: $n_3 \geq (n_1 - p) > n_2 + n_3$, a contradiction. □

We are now ready to prove Theorem 1 (restated below for reference).

Theorem 1 (Priority Set Invariant for BBQ$_{\odot}$). *In a BBQ$_{\odot}$ instance composed of a BBQ with pipeline latency p cycles and a PIFO of size $k > p$, the top $(k-p)$ highest-priority elements are always in the PIFO.*

Proof. Given Lemma 1, we only need to consider the scenario where $(k-p) \leq |\mathcal{P}(t)| \leq k$. Assume that the BBQ is not empty, otherwise all $|\mathcal{P}(t)| \geq (k-p)$ elements in the PIFO are trivially the highest-priority ones. Now, assume towards a contradiction that only the $m < (k-p)$ highest-priority elements in the system are in the PIFO at a certain time t_2. It follows that the highest-priority element in the BBQ, x, has a higher priority than the remaining $|\mathcal{P}(t_2)| - m$ elements in the PIFO at t_2. Observe that x may only have been inserted in the BBQ if, at the time of insertion, t_1: (1) the PIFO was full, and (2) all k elements in the PIFO had higher priority than x.

Now, for x to be the $(m+1)$'th highest-priority element in the system at time t_2, we must have performed $n = (k-m) > (k-(k-p)) = p$ number of DEQUEUE-MAX$_{\text{PIFO}}$ operations since t_1, implying that n $(> p)$ DEQUEUE-MAX$_{\text{BBQ}}$ operations were issued in the interval $[t_1, t_2]$. Since at most one DEQUEUE-MAX$_{\text{BBQ}}$ operation can be issued every timestep and each such operation takes exactly p timesteps to complete, it follows that *at least one* DEQUEUE-MAX$_{\text{BBQ}}$ operation completed and returned x. Thus, x is in the PIFO at time t_2, a contradiction. □

Appendix D Dynamic Priority Ranges

As described in §3.1, the standard BBQ primitive operates over a priority range that is both *finite* and *static*. While this is sufficient in some contexts (*e.g.*, strict priority scheduling), many policies implicitly assume an infinite priority set (*e.g.*, fair queueing). IPQs are fundamentally incapable of upholding this assumption (§9). Fortunately, prior work has shown that in most cases, a *dynamic* – albeit finite – priority range is sufficient to realize these policies [38, 39]. In this section, we describe how BBQ can be extended to provide the abstraction of a *rolling priority window*.

To handle dynamic priority ranges, we directly adapt Eiffel's [38] idea of using a *Circular Hierarchical FFS-based Queue* (cFFS). The idea is to have two independent HFFS queues, each with priority span P, working in tandem: a primary HFFS queue, q_0, that stores elements with priorities $[0, P)$, and a secondary HFFS queue, q_1, mapping to elements with priorities in $[P, 2P)$ (i.e., just outside q_1's range). Together, these queues represent a *logical* priority window of $[0, 2P)$. Once the primary queue becomes completely empty, the logical priority window advances by P, and the queue designations are swapped, with q_0 – now the secondary queue – buffering elements with priorities in $[2P, 3P)$, and so on and so forth.

We follow precisely the same blueprint for BBQ, with logical partitioning enabling us to multiplex both q_0 and q_1 atop a single physical BBQ instance with no resource overhead (or modification to the primitive, for that matter). The only additional component required is a simple controller to orchestrate the two logical queues. We do not implement this feature as part of our research artifact (yet), but we expect it to have little to no impact on BBQ's performance.

Sirius: Composing Network Function Chains into P4-Capable Edge Gateways

Jiaqi Gao[†]*, Jiamin Cao*[†]*, Yifan Li, Mengqi Liu, Ming Tang, Dennis Cai, Ennan Zhai*
Alibaba Cloud

Abstract

Alibaba Cloud designs and deploys P4-capable gateway to accelerate the processing of the diverse business traffics in the edge cloud. Since the programmable ASIC in the gateway only accepts a monolithic, pipelined P4 program, the dozens network function chains for different business traffics have to be composed into one. This is non-trivial due to the contention between the complexity of network function chains and the limited resource in the programmable ASIC. In this paper, we present Sirius, a system that automates network function chain composition process. Sirius synthesizes tables to identify which business traffic the input packet belongs to, pipelines loops in the merged network function graph via recirculations, and partitions the graph between programmable ASIC and CPU when the required memory consumption exceeds the ASIC's capability. So far, Sirius has automated network function arrangement in hundreds of gateways, and has effectively decreased our programmers' workload by three orders of magnitude, from weeks to minutes.

1 Introduction

As a cloud provider, Alibaba operates hundreds of edge clouds to deliver fast services (*e.g.*, game and video) to global end users. To maintain reasonable costs, each of these small-scale edge clouds contains a pair of gateways, a few switches, and tens of light-weight servers with tight space constraints and CPU compute limitations. Our typical edge cloud topology is shown in Figure 1(a). Depending on the services deployed, the edge cloud can serve more than 10 types of business traffics (load balancer, proxy, Virtual Private Cloud (VPC), *etc.*) Illustrated in Figure 1(b), each business traffic requires its own network function processing chain, different directions of the same business traffic may traverse network functions in different orders. As services today constantly evolve, it has become increasingly difficult for these resource-limited edge clouds to handle the ever-growing traffic and CPU overhead.

Recent advances in programmable switch ASICs have enabled us to offload network functions from edge cloud servers to the programmable switch ASICs. We designed and built our own P4-capable edge gateway. Shown in Figure 1(c), the gateway is equipped with a programmable switch ASIC and server-grade CPU with dozens of cores, connected via a pair of non-programmable NICs with hundreds of gigabits throughput. The gateway sits on the border of the edge cloud between the ISP and edge cloud switches. Deploying network function chains onto the P4-capable gateway significantly reduces the usage of constrained server CPU resources and improves the performance of our edge clouds.

Despite the P4-capable gateway's performance and flexibility, we face a tough programming challenge. Recent compiler works such as Lyra [5], Cetus [13] allow programmers to develop *one* network function chain fast and resource-optimized. However, programmers struggle with **composing all network function chains into the gateway while satisfying the throughput requirement.** More specifically, since the programmable ASIC is performant but limited in resources, the programmers have to merge all network function chains into a graph by reusing overlapping network functions, and maximize the network functions assigned to the programmable ASIC to process hundreds of gigabits of traffic flowing through the gateway. Network function chain composition involves three major challenges:

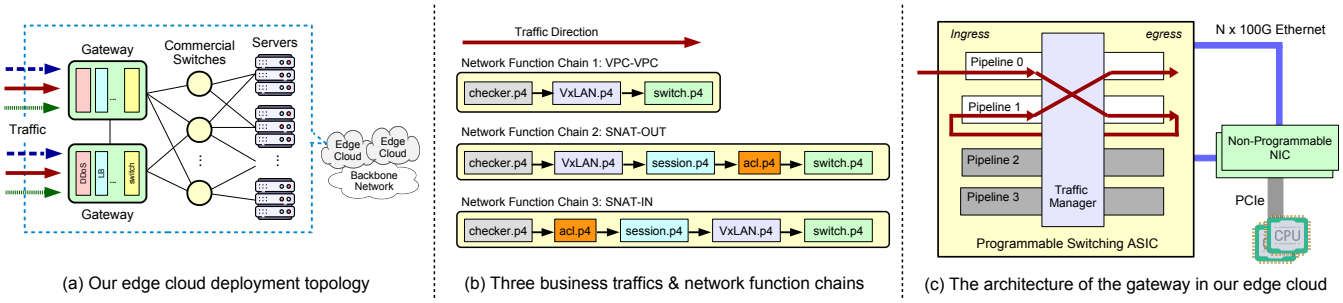
Identification. When a packet arrives, the first step is to identify which business traffic it belongs to and which network functions to execute. While there are many potential ways of identifying a packet, the solution should be generic and can be implemented with reasonable resource overhead.

Pipelining. In real-world applications, it is inevitable that different business traffics execute network functions in different orders. Given the ASIC's pipeline architecture, it is infeasible to compose these different processing chains directly. Such ordering conflicts can be resolved by inserting recirculations but with substantial human efforts.

Partitioning. Even after resolving ordering conflicts, the composed program containing all required network functions may not fit on the programmable ASIC due to its resource constraints. Thus, some components in the composed program have to be moved to the gateway's CPU, at the expense of lower performance and degraded traffic throughput. Such partitioning between programmable ASIC and the CPU is non-trivial because it requires careful balancing between the deployment capability and the overall throughput.

State of the art. To the best of our knowledge, no prior compiler work [3, 5–7, 9, 10, 12, 13, 19, 20, 23–25, 27] targeted the above-mentioned network function chain composition problem. μP4 [21] and P4 Weaver [4] enables modular P4 programming and composes multiple P4 modules into a directed acyclic graph (DAG) without reusing any common code block across modules. P4Visor [28], HyperV [26], Dejavu [25], and HyPer4 [8] merge multiple P4 programs into

[†] Both authors contributed equally to this paper



(a) Our edge cloud deployment topology (b) Three business traffics & network function chains (c) The architecture of the gateway in our edge cloud

Figure 1: Our edge cloud's topology, business traffic examples, and architecture of our P4-capable gateway.

one but do not handle ordering conflicts, or partition the program across ASIC and CPU. These approaches incur high memory overhead and cannot handle production-scale complexity with tens of business traffics and hundreds of match action tables.

Our system: Sirius. In this paper, we present Sirius, a system that automates network function chain composition in Alibaba's edge clouds. Sirius takes the following inputs: (1) P4 programs that define the network function chains for each business traffic, (2) a traffic identification database that describes flow characteristics for sets of business traffics, and (3) edge cloud topology and the throughput requirement for each traffic. Sirius then returns a composed P4 program that can handle all the above traffics and can deploy on the programmable ASIC. It also outputs the list of modules that must be moved to the CPU, so that our developers can later implement them as C++ programs.

Overall, this paper makes the following contributions:

- A synthesis algorithm that generates a memory-efficient P4 traffic identification table, along with corresponding guard conditions to identify each business traffic (§4).
- A pipelining algorithm that finds all possible candidates for resolving ordering conflicts in composing business traffic processing chains (§5).
- A new resource encoding paradigm and iterative searching approach to find the best pipelining and partitioning plan to minimize CPU load. (§6)

We have been using Sirius in production for one year, and it has automated the arrangement of network functions for hundreds of gateways in our global edge clouds. Sirius has effectively decreased our network function arrangement workload by three orders of magnitude (from weeks to minutes). In §7, we share our experience in using Sirius, representative cases solved by Sirius, and lessons we learned. We also evaluate Sirius's performance in §8.

2 Background

Edge clouds are deployed closer to end users and deliver dozens of services (*e.g.*, IoT, cloud gaming, CDN, and storage) with lower latency. Gateways on the edge cloud host network functions to process all services' business traffic at hundreds

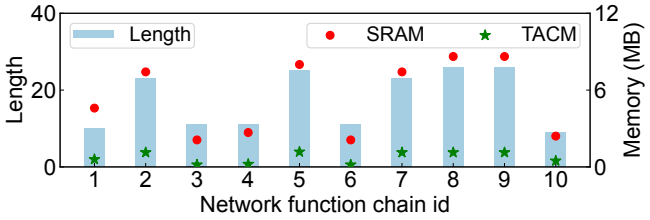


Figure 2: The length and resource consumption of overlapping tables of each network function chain.

of gigabits per second, such as balancing e-payment load, translating network address for VPC, collecting statistics for QoS and billing, *etc.* Traditionally, the network functions are implemented as software programs running on the CPU. As the scale of the edge cloud and the complexity of deployed services grow, the CPU-based solution struggles to keep up. We spent hundreds of CPU cores on an edge cloud site only processing the traffic traversing through, resulting in huge overhead both in cost and space. Therefore, we built our own P4-capable gateway and accelerated our network functions via the programmable ASIC. Depending on the services deployed, the gateway serves traffic at hundreds of gigabits to multi-terabits per second.

Developing atop the P4-capable gateway is challenging due to the scale and complexity of our network functions. To quantify the complexity, we examined four P4 programs for different edge cloud sites and summarized the line of codes and resource consumption in Table 1. We can see that all four programs occupy all 12 stages of the programmable ASIC, and all have high Packet Header Vector (PHV) usage. The largest program has 248 tables and consumes all four pipelines in the gateway's ASIC.

We experience such complexity and scale because each P4 program is composed of tens of network function chains processing different business traffic. We further dive into the 'Medium 1' program, which is composed of 10 network function chains. We examined the length of each network function chain and the result is recorded in Figure 2. The length varies from 9 to 26, which means some chain occupies at least 26 stages in the programmable ASIC. We also observe a huge amount of table overlapping between the chains. We recorded the SRAM and TCAM memory size of the tables shared with other chains and the result is shown in Figure 2. Each chain at least reuses 1MB of SRAM memory and 180KB

Scale	LoC	# of Table	# of Pipe.	PHV (%)	SRAM (%)	TCAM (%)	Stage
Small	4155	81	2	81.1%/68.8%	78.0%/98.8%	28.1%/10.1%	12/12
Medium 1	11870	197	2	88.9%/87.0%	51.7%/49.3%	64.6%/25.3%	12/12
Medium 2	9996	156	2	69.1%/85.4%	34.9%/47.2%	27.8%/29.2%	12/12
Large	16190	248	4	85.9%/97.3%/59.4%/87.4%	39.5%/67.2%/86.8%/66.9%	26.4%/33.0%/17.0%/35.4%	12/12/12/12

Table 1: The resource usages of four P4 programs at different scales.

of TCAM memory. If we choose not to reuse the tables and duplicate them, the overall memory consumption explodes by 3.56X in SRAM and 4.93X in TCAM, which way exceeds the ASIC's capability. Thus, it is necessary for us to compose network function processing chains that overlap with each other.

To the best of our knowledge, no prior work can handle the complex challenges we face. We believe the fundamental reason is that current abstractions either assume there exists only one huge network function chain (P4) or multiple chains are independent of each other (Lyra [5]). Such assumptions do not hold in our edge cloud scenario, as explained above.

Now we use the example network function chain shown in Figure 1(b) to illustrate the network function chain composition process and the challenges. The three network function chains create a simple edge cloud instance that allows bidirectional communication between private and public networks. *SNAT-IN* and *SNAT-OUT* chains process traffic going in and out of the edge cloud respectively, and *VPC-VPC* chain process traffic flowing between private networks. The expected throughput is 300 Gbps for all business traffics.

It is impossible to compose the three chains by directly merging their network functions into a single P4 program. As illustrated in Figure 3(a), each chain defines a mandatory ordering among its network functions. The orderings imposed by different chains conflict with each other, resulting in the two loops (highlighted in red) in the graph. In particular, *SNAT-IN* and *SNAT-OUT* traffic flow in reverse directions, thus, the two network function chains execute modules in reversed orders as well. Such conflicts are inconsistent with the pipelined architecture of the programmable ASIC. To successfully compose the three chains onto our P4-capable gateway, the following three challenges must be addressed.

Challenge 1: Pipelining network functions. Ordering conflicts commonly exist when composing diverse network function chains. We employ the recirculation feature to resolve these conflicts. Recirculation allows a packet to go through the programmable ASIC one more time, at the cost of reducing the overall processing throughput. For example, Figure 3(b) shows a pipelining plan that follows the order of the SNAT-OUT chain. This allows SNAT-OUT traffic to be processed in one pass, while introducing two recirculations for SNAT-IN traffic (*i.e.*, a packet visits checker and acl in round 0, session in recirculation round 1, and VxLAN and switch in recirculation round 2). As a result, SNAT-IN traffic's maximum throughput is reduced to 1/2 of the recirculation channel's bandwidth since every packet goes through the recirculation channel twice. Assuming that the recirculation channel has 400 Gbps bandwidth, this pipelining plan violates the throughput requirement (*i.e.*, 300 Gbps). Instead, a feasible pipelining plan is shown in Figure 3(c), where both SNAT-IN and SNAT-OUT traffics recirculate only once and guarantees 400 Gbps maximum bandwidth. Note that this is not the only feasible pipelining plan, swapping `VxLAN.p4` and `session.p4` in Figure 3(b) also satisfies the throughput requirement.

Challenge 2: Identifying business traffics. Despite the fact that all network functions are present in the composed programmable ASIC, different business traffics visit different network functions defined by their processing chains. This requires us to identify which business traffic an input packet belongs to and only execute relevant P4 modules. Figure 3(d) shows a naive solution that inserts different identification tables before each module. For example, the two identification tables marked in red ignore *VPC-VPC* traffic according to the network function chain. The rest of the identification tables marked in blue allow all traffic to go through but also ensure that the table only executes at the correct recirculation round. However, this solution incurs high memory overhead and may lead to a longer execution chain of P4 tables. According to findings in Cetus [13], this approach does not scale.

Challenge 3: Partitioning network functions. In many cases, the composed P4 program simply requires too much memory resource and does not fit on the programmable ASIC. We have to partition the program and move some tables to the CPU. This requires careful balancing between resource consumption on the programmable ASIC and the throughput degradation caused by moving to the CPU. Furthermore, since the CPU sits on the recirculation path, a careless partitioning plan can introduce additional recirculations. For example, if we assign `switch.p4` to the CPU based on the pipelining plan in Figure 3(c), all three business traffics have to recirculate once more time to be forwarded out of the gateway. A better plan is to assign `session.p4` to the CPU since it already sits on the recirculation path, which we will detail in §6.1.

The network function composition problem causes significant overhead during gateway development. It often takes weeks for our programmers to find an arrangement plan, which is subject to change when the network function implementation or the gateway configuration (*e.g.*, set of supported business traffics) changes. Thus, it is necessary to build a system to automate the network function composition process.

3 Sirius Overview

Figure 4 presents Sirius's architecture. Sirius offers our programmers a set of high-level interfaces to automatically address the network function composition problem. The inter-

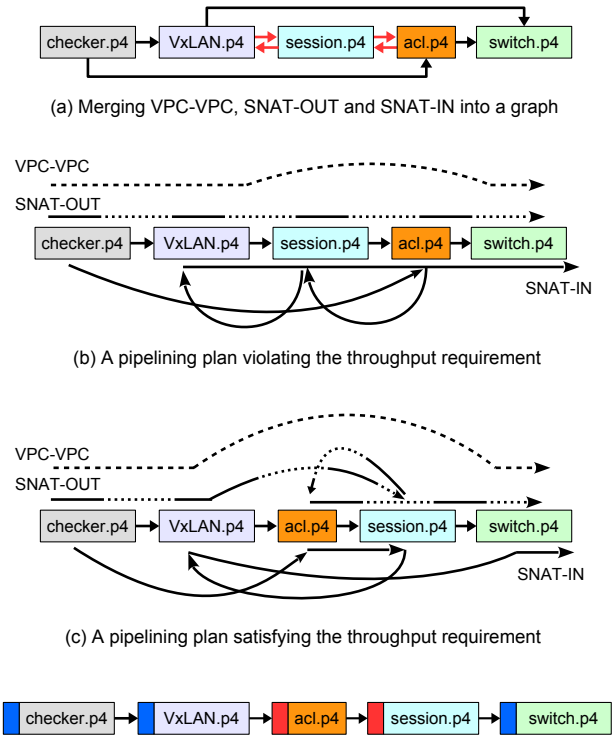


Figure 3: An example of a merged graph and two pipelining plans for this graph.

faces require the following input: (1) P4 programs that specify all the individual network functions and how these network functions are chained for diverse business traffics, (2) a traffic identification database, which contains flow characteristics rules (flow predicates) that distinguish sets of business traffics, (3) edge cloud topology and throughput requirements, which specify the throughput needs of different business traffics.

As shown in Figure 5, given a set of original P4 code for each network function chain (Figure 5(a)), Sirius follows three phases to produce the final composed P4 program that can compile and deploy to a programmable ASIC.

In the first phase, Sirius leverages the input traffic identification database to generate a *traffic identification table* that tags each packet according to a selected set of flow predicates. In addition, it generates guard conditions that utilize these tags to distinguish each business traffic (Figure 5(b)), such that a network function is only visited by traffic chains it belongs to. To accommodate memory constraints on the programmable ASIC, we describe in §4 our algorithm for synthesizing a memory-efficient traffic identification table.

In the second phase, Sirius generates solutions for resolving network function ordering conflicts among different processing chains. This is a unique challenge for composing network function chains onto a programmable ASIC, which follows a pipelined architecture. Sirius resolves such conflicts by inserting recirculations after certain network functions. To find

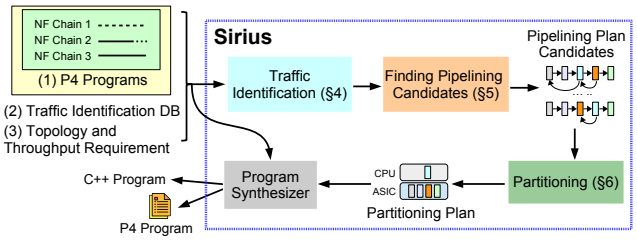


Figure 4: Sirius's system architecture overview.

pipelining plan candidates (*i.e.*, candidate recirculation points) with a satisfied number of recirculations *efficiently*, Sirius models this pipelining process as a feedback arc set problem and introduces an algorithm as explained in §5. Figure 5(c) illustrates the deployment of a candidate solution, where the guard condition before network functions are extended with predicates on the recirculation count.

In the third phase, if no pipelining candidate can fit within the switch resource, Sirius searches for P4 tables to assign to the CPU. We propose a novel *logical stage* encoding paradigm to transform this search into a satisfiability problem, and employ an iterative strategy to generate the optimal partitioning between the programmable ASIC and the CPU (§6). As illustrated in Figure 5(d), a solution may involve moving the session module to the CPU and adjusting recirculation destinations accordingly.

In this way, Sirius generates a composed P4 program that accommodates all original network function processing chains and can compile to the programmable ASIC.

4 Traffic Identification

The composition of multiple traffic processing chains requires the integrated program to distinguish business traffics from each other and add corresponding guard conditions at the entry of each module. This ensures a packet only visits modules that belong to its processing chain. Because the programmable switch has limited resources, it is necessary to reduce the memory usage overhead of traffic identification and to fit as much business traffic processing logic as possible.

In this section, we first show the input to Sirius, *i.e.*, the traffic identification database that records how header fields and metadata identify different business traffics (§4.1). Then, we introduce the insight that Sirius uses to synthesize the traffic identification table and add guard conditions on the programmable switch (§4.2).

4.1 Traffic Identification Database

Each business traffic has its unique flow characteristics. For example, VPC-VPC traffic travels within the internal network, and thus its source and destination IP are both in the internal IP range, while for SNAT-IN traffic, its source IP belongs to

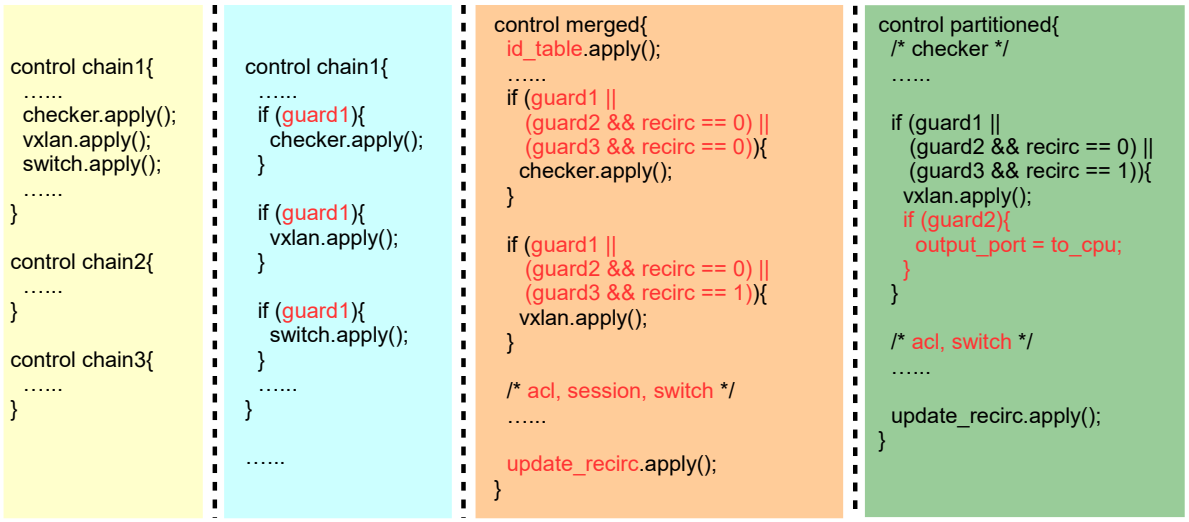


Figure 5: Changes on the P4 implementation after each phase. (a) depicts the original P4 code for each network function processing chain. (b) depicts chain 1 (VPC-VPC) after going through the traffic identification phase, where guard1 denotes the synthesized guard condition for it. (c) depicts the merged P4 code after going through the pipelining phase, assuming the pipeling solution shown in Figure 2(d). Here, recirc is a variable recording the number of recirculations experienced by a packet. (d) depicts the P4 code after going through the partitioning phase. Here, traffics belonging to chain 2 will be directed to the CPU after the vxlan module, which implements the solution in Figure 7(b).

Rule ID	Flow Predicate	Business Traffics	Entries
Rule 1	src_ip ∈ Public IP	SNAT-IN	100
Rule 2	src_ip ∈ VPC IP	VPC-VPC ∪ SNAT-OUT	10
Rule 3	dst_ip ∈ Public IP	SNAT-OUT	100
Rule 4	dst_ip ∈ VPC IP	VPC-VPC ∪ SNAT-IN	10
Rule 5	phy_port ∈ Internal Port	VPC-VPC ∪ SNAT-OUT	48
Rule 6	(src_ip, dst_ip) ∈ VPC IP PAIR	VPC-VPC	50

Table 2: Our example's traffic identification DB.

the public IP range. Sirius relies on the traffic identification database to maintain this information.

Sirius's traffic identification database adopts a flow-predicate-centric approach, *i.e.*, the primary key of the database is the membership relation of header fields (such as the source IP field) or metadata (such as the physical port ID on the switch) that distinguishes a subset of the business traffics (being a necessary and sufficient condition). An example traffic identification database is shown in Table 2. Rule 2 means if a traffic's source IP belongs to the VPC IP set, then it is either VPC-VPC or SNAT-OUT traffic, and vice versa. Besides the basic flow predicate and the corresponding business traffic set, the database also provides the number of entries in each rule. Three factors together decide the memory consumption of each rule. For example, Rule 6 occupies 10× more memory than Rule 2 in Table 2 because Rule 6 checks both source and destination IP and has 5× more entries[1].

We rely on the composition of flow predicates to distinguish each business traffic. For example, (Rule 2 AND Rule 4) separates VPC-VPC traffic from the rest, and (Rule 4 AND (NOT Rule 1)) also achieves the same goal. The reason for such a design is twofold. On the one hand, it is not always feasible in practice to find a single predicate to identify each business traffic. On the other hand, although Rule 6 uniquely defines VPC-VPC traffic, it requires more memory resources than composing Rules 2 and 4.

Implementation: parallel matching on all predicates We use parallel matching to implement the above flow predicate composition in a generic way. As illustrated in Figure 6, when a packet arrives at the programmable switch, we match it against all flow predicates in parallel and store the results in corresponding flags. Upon subsequent network functions, guard conditions are generated by composing these predicates, *e.g.*, `hdr.p2 && hdr.p4` denotes (Rule 2 AND Rule 4), which uniquely identifies VPC-VPC traffic.

4.2 Synthesis of Traffic Identification Tables

As discussed above, we draw flow predicates from a developer-provided traffic identification database (*e.g.*, Table 2) and use their logical compositions to identify each business traffic. This section describes how we choose the set of flow predicates (*i.e.*, rules) that is both *complete* (able to distinguish each business traffic) and memory efficient when implemented on the programmable switch[2].

[1] Rule 6's size is smaller than 90 (*i.e.*, 10 × (10 - 1)) because some source and destination IP pairs are not possible in reality.

[2] We ignore other types of resources (especially PHV) because (1) according to our experience, memory is the major bottleneck, and (2) it is challenging to quantify other resources' overhead and hard to choose one solution over another.

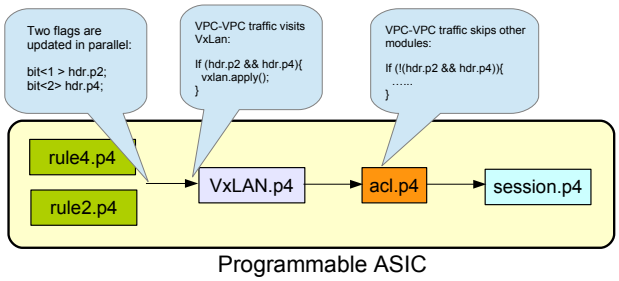


Figure 6: Implementation of flow predicates: rule 2 and rule 4

Our synthesis algorithm takes the traffic identification database as input and returns a set of rules as output that is complete and incurs minimum resource usage. The pseudo-code is listed in Algorithm 1. First, the algorithm iterates through each rule and builds a search tree from all the rules behind, where each node in the tree represents a subset of the rules in the database, and each branch adds one more rule to the subset (Line 3-4). There are $|\mathcal{R}|$ trees in total. Next, we iterate through all nodes in the search tree and find a node that represents the minimum valid set of rules (Line 5-15).

Because the resource consumption increases monotonically as rules are added, we apply three more optimizations to prune the search tree. Before searching, we sort the rules in the database by memory size and traverse them in ascending order (Line 2). Within a tree, we sort the nodes in the tree by their resource consumption in ascending order (Line 4) and stop searching when we find the first valid set (Line 19-22). Between trees, we stop searching the current tree when the current set has higher resource consumption than the global best solution (Line 16-18).

Example. Assume the business traffics shown in Figure 1 and the traffic identification database shown in Table 2. When searching directly without the optimizations, we need to check the search tree of each rule. For example, in the first round, we check the search tree of $\{Rule\ 1\}$ and obtain three traffic identification tables: $\{Rule\ 1, Rule\ 3\}$ with $(100\times32+100\times32)$ bits, *i.e.*, 800 bytes of memory usage, $\{Rule\ 1, Rule\ 4\}$ with $(100\times32+10\times32)$ bits, *i.e.*, 440 bytes of memory usage, and $\{Rule\ 1, Rule\ 6\}$ with $(100\times32+50\times64)$ bits, *i.e.*, 800 bytes of memory usage. After we check all search trees in all rounds, we choose $\{Rule\ 2, Rule\ 4\}$ with 80 bytes of memory usage generated in the second round. With the optimizations, we directly check the search tree of $\{Rule\ 2\}$ and obtain $\{Rule\ 2, Rule\ 4\}$, since $\{Rule\ 2\}$ has the smallest memory usage and $\{Rule\ 2, Rule\ 4\}$ has the smallest resource consumption in the tree.

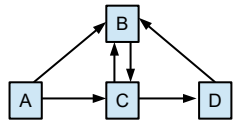
5 Finding Pipelining Candidates

A unique challenge in composing multiple network function processing chains is the inevitable ordering conflicts between individual network functions. Manifested as loops in the merged network function graph, these ordering conflicts commonly exist because business traffic going in and out of the gateway usually visits network functions in opposite directions. A network function graph with loops cannot be deployed on the programmable ASIC directly due to the ASIC's pipelined architecture. Sirius solves this problem by inserting recirculations in the merged network function graph. In this section, we first explain that the problem resembles a feedback arc set problem (§5.1), and next, we show how Sirius solves the problem and finds all pipelining candidates (§5.2).

Algorithm 1: Traffic identification table synthesis

Input: $\mathcal{T}$: Business traffic.
Input: $\mathcal{R}$: Rules in traffic separation DB.
Output: $\mathcal{F}$: Rules to synthesize traffic identification tables.

1 $\mathcal{F} \leftarrow \mathcal{R}$
2 $\mathcal{R} \leftarrow SizeSortAscending(\mathcal{R})$
3 **foreach** $r_i \in \mathcal{R}$ **do**
4 $\mathcal{C} \leftarrow SizeSortAscending(Combinations(\{r_i, r_{i+1}, \cdots, r_{|\mathcal{R}|}\}))$
5 **foreach** $\mathcal{R}' \in \mathcal{C}$ **do**
6 $\mathcal{X} \leftarrow \{\{t, t \in \mathcal{T}\}\}$
7 $\mathcal{S} \leftarrow \emptyset$
8 $terminate \leftarrow False$
9 **foreach** $r_k \in \mathcal{R}'$ **do**
10 **foreach** $x_j \in \mathcal{X}$ **do**
11 $tmp_1 \leftarrow x_j \cap r_k.action$
12 $tmp_2 \leftarrow x_j - tmp_1$
13 **if** $((tmp_1 \cup tmp_2) - \mathcal{X}) \neq \emptyset$ **then**
14 $\mathcal{X} \leftarrow tmp_1 \cup tmp_2$
15 $\mathcal{S}.append(r_k)$
16 **if** $RuleSize(\mathcal{S}) \geq RuleSize(\mathcal{F})$ **then**
17 $terminate \leftarrow True$
18 **break**
19 **if** $AllElementSizeOne(\mathcal{X})$ **then**
20 $\mathcal{F} \leftarrow \mathcal{S}$
21 $terminate \leftarrow True$
22 **break**
23 **if** $terminate$ **then break**
24 **return** $\mathcal{F}$



(a) An example merged network function graph

// All existing processing loops
L = { [B, C, B], [B, C, D, B] }
// Key-Value Table V

Execution Sequence	Loops
B → C	{ [B,C,B] , [B,C,D,B] }
C → B	{ [B,C,B] }
D → B	{ [B,C,D,B] }
A → C	{ }
… …	… …

(b) Vector and Key-Value Table

Figure 7: An example for our insight and modeling.

5.1 Modeling Pipelining Process

It is impossible to deploy a set of network functions containing ordering conflicts (*i.e.*, loops in the merged graph) on

the programmable ASIC directly. We address this issue by inserting recirculations at the end of certain network functions, which effectively breaks the immediate ordering (*i.e.*, removes the edges) after these network functions. Until there is no longer ordering conflict (*i.e.*, the merged graph becomes acyclic), this set of network functions becomes deployable on the programmable ASIC. Figure 5(c) illustrates that multiple processing chains can be merged into a single P4 program after resolving all ordering conflicts.

We use Figure 7(a) to explain this process. In Figure 7(a), there are four network functions *A*-*D*, and each directed edge represents an immediate ordering between two network functions, such as $B \rightarrow C$, required by the network function processing chain it belongs to. There are two processing loops, $B \rightleftarrows C$ and $B \rightarrow C \rightarrow D \rightarrow B$. If we remove the ordering $B \rightarrow C$, the two processing loops disappear. Removing $D \rightarrow B$ only removes one processing loop $B \rightarrow C \rightarrow D \rightarrow B$.

This tells us that we can leverage the recirculation feature (a commonly used feature in P4 for processing a packet multiple times) to "remove" a network function ordering, which eventually removes processing loops. At the same time, we should use recirculations as little as possible to avoid throughput drop. Therefore, our goal is to remove all processing loops with minimal recirculations, making the network function graph a directed acyclic graph.

This resembles the feedback arc set problem [2], which removes edges from a directed graph until it becomes acyclic. However, the main difference is that the original problem formulation focuses on minimizing the number of edges removed or by minimizing a certain weight. As a comparison, we ultimately focus on the traffic throughput, which is determined by a lot more factors, including the number of edges removed on each processing chain (*i.e.*, the number of recirculations).

5.2 Generating Pipelining Candidates

There are multiple pipelining candidates (*i.e.*, set of edges to remove) for the same merged graph. For example, both $\{B \rightarrow C\}$ and $\{C \rightarrow B, D \rightarrow B\}$ are valid solutions for Figure 7(a). However, it is hard to tell which one is better without going through the partitioning phase. In order to reduce the number of candidates entering the partition phase, Sirius employs the following two pruning strategies during the search. (1) The search backtracks if the current solution violates the **throughput** requirement, since inserting more recirculations would only result in lower throughput. (2) The search backtracks when a **valid** solution is found, such that unnecessary recirculations are prevented.

To generate the above sets of candidates, Sirius first computes a set L, which contains all processing loops, and a key-value table V. Figure 7(b) illustrates an example for L and V. In V, each key (say k_i) represents an execution sequence, and the corresponding value records all the processing loops that disappear once k_i is removed. It then searches for all sets of edges that collectively remove all loops in the graph, *i.e.*, for any set of edges $\mathcal{K}$, $\bigcup_{k \in \mathcal{K}} V[k]$ must equals L.

Sirius introduces a dynamic programming approach to explore the search space and applies heuristics based on the throughput specifications. The algorithm works as follows:

- (1) Remove the keys with an empty value from the key-value table V, and sort the keys based on the number of elements in the mapped values in descending order. For example, execution sequence $\{B \rightarrow C\}$ has two elements.
- (2) In the ranked V, traverse each key k_i and add it to $\mathcal{K}$, which denotes that the edge k_i is removed.
  - (2.1) Then (still for k_i), iterate each network function chain and compute the expected throughput given the current $\mathcal{K}$. When a chain contains n keys in $\mathcal{K}$, the chain's throughput is reduced to T/n, where T is the recirculation channel's throughput. If the expected throughput is violated, remove key k_i from $\mathcal{K}$, and jump to (3).
- (3) Select key k_{i+1} from the remaining elements in the ranked V, continue step (2) until the values of $\mathcal{K}$ covers L, then $\mathcal{K}$ is a solution (*i.e.*, a pipelining candidate). Continue the iteration until all solutions are found.

6 Partitioning

Given a set of pipelining plan candidates that have resolved all ordering conflicts and also satisfied the throughput requirements considering all inserted recirculations (generated from §5), we now need to compile and fit them into the programmable ASIC. Based on our one-year experience, none of the merged P4 code (*e.g.*, Figure 5(c)) can directly compile to the programmable ASIC due to its limited resources. Sirius thus proposes a partitioning approach that tries to move some tables or an entire network function to the CPU so that the resulting P4 code can be successfully compiled.

Similar to Lyra [5] and Cetus [13], our partitioning approach leverages the SMT solver to search for a satisfied result (*i.e.*, tables or network functions to move to the CPU); however, our scenario needs to take into account many different factors such as the recirculation feature and the CPU. Existing SMT solver-based approaches do not address these unique challenges.

In this section, we present a novel logical stage encoding approach (§6.1) that transforms partitioning problems into SMT problems. Then, we explain how we use iterative searching to find the best partitioning plan that minimizes the load on the CPU (§6.2).

6.1 Partitioning Encoding

Moving some tables or the entire network function to the CPU may add additional recirculation to the input network function chain, because a packet needs to first leave the programmable ASIC, then enter the CPU, and finally enter the programmable

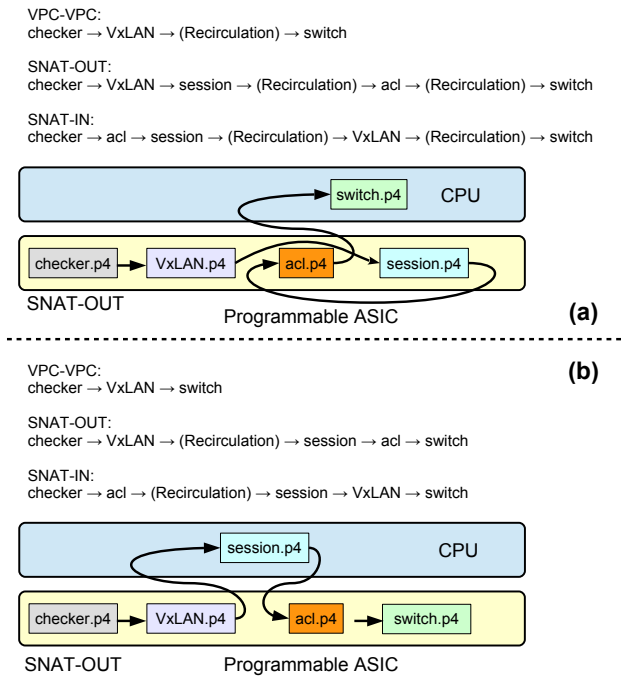


Figure 8: Recirculation situations resulting from two different partitioning plans.

ASIC again in a recirculation. If a partitioning plan is not well-chosen, the entire network function chain may include additional recirculations (introduced by the CPU), violating the throughput requirement. Figure 8 shows an example.

Suppose a programmable ASIC can only hold four network functions in Figure 3(a), and we have a pipelining result shown in Figure 3(c). Because the entire network function chain shown in Figure 3(c) cannot be put in the programmable ASIC, we need to move a network function to the CPU. Figure 8(a) shows a partitioning result that moves the `switch.p4` to the CPU. In such a partitioning plan, unfortunately, VPC-VPC, SNAT-OUT, and SNAT-IN have one, two, and two recirculations, respectively. Namely, this partitioning additionally adds a recirculation to each of these three chains. As a result, the throughput is significantly decreased. Figure 8(b) shows a better partitioning plan that moves `session.p4` to the CPU. In this case, all three network function chains achieve the desired throughput without any additional recirculation caused by the CPU.

Encoding. It is common to address resource allocation problems by encoding relevant constraints as math formulas and invoking SMT solvers to find a satisfying solution. However, encoding the effect of moving P4 tables to the CPU is not straightforward due to its intertwining with existing recirculations generated in §5.

We propose a concept called *logical stage* to address this challenge. While a physical stage describes where a *P4 table* is actually deployed, the logical stage depicts its execution sequence, *i.e.*, physical stage + N × recirculation rounds, where N denotes the number of physical stages in the switch.

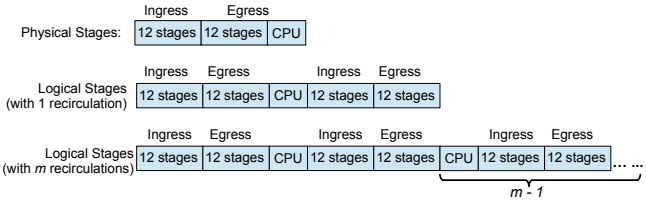


Figure 9: Physical stages and their logical stages.

As illustrated in Figure 9, Sirius duplicates m ingress and egress pipelines to model the logical execution sequence that spans m recirculation rounds. Further, Sirius inserts a new CPU stage between each egress and ingress pipeline to model the execution on the CPU.

The partitioning phase takes the merged P4 code (*e.g.*, Figure 5(c)) as input and calculates the set of tables to move to the CPU that enables the successful deployment of the remaining P4 tables. We encode this phase as a satisfiability problem as below.

Input: We use k to denote the total number of stages on the target ASIC. For the particular pipelining candidate, we use r_c to denote the number of recirculations for chain c.

Output: For each table t, s_{pt} and s_{ct} represents its physical and logical stage index, respectively.

Constraints:

- ASIC stage constraint: a table can either deploy at a physical stage or be moved to the CPU (the $(k+1)^{\text{th}}$ stage). Thus, $0 \le s_{pt} \le k$ must hold.
- A physical stage must correspond to a logical stage at a particular recirculation round: $(s_{ct} < k+1 \implies s_{pt} = s_{ct}) \wedge (k+1 \le s_{ct} < 2k+2 \implies s_{pt} = s_{ct} - k - 1) \wedge \cdots$ must hold.
- Moving tables to the CPU does not result in more recirculations: for each table t belonging to chain c, $0 \le s_{ct} \le (r_c+1) \times (k+1)$ must hold.
- Ordering constraint: if a chain orders table t after t', $s_{ct} > s_{ct'}$ must hold.

In addition to the above partitioning constraints, we also apply resource constraints defined in Lyra [5] and Cetus [13] to determine whether the remaining tables can indeed deploy to the programmable ASIC.

6.2 Finding the Partitioning Result

Given the above encoding, we invoke an SMT solver to search for a partitioning result with the optimization goal of **minimizing the CPU usage**. The performance of a program running on the CPU is affected by many factors and is hard to predict. Recent works such as Bolt [11] require packet traces to predict performance at reasonable accuracy. Instead, Sirius uses two heuristic metrics to achieve the goal: (1) minimizing the number of tables assigned to the CPU[3], and (2) when two

[3]We choose not to assign TCAM tables to the CPU since ternary matching's complexity is $O(n^{\log_3 2})$ [1] while LPM and exact match's complexity is only O(1) [18]

plans assign the same number of tables, choose the one with lower traffic volume.

In particular, Sirius adopts iterative searching and increases the allowed number of tables to move to the CPU gradually until a feasible plan is found by the SMT solver.

Iterative searching. The iterative searching reduces an optimization problem into a series of satisfiability problems, which the SMT solver is good at. Atop the existing encoding, for each table t, Sirius introduces one 0-1 variable i_t that denotes whether the table should be deployed on the CPU, which must satisfy: $(s_{pt} = k+1 \implies i_t = 1) \wedge (s_{pt} < k+1 \implies i_t = 0)$. Then the summation of all such variables $I = \sum_t i_t$ means the total number of tables assigned to the CPU. Then, Sirius limits the summation $I == m$ starting from $m == 0$, and checks whether there exists a partitioning plan $Part_m$. If not, this means no solution can be found when assigning m tables to the CPU, and then Sirius increases m by one and calls the solver again. If $Part_m$ exists, Sirius records it as a candidate. In case multiple solutions exist, we remove $Part_m$ from the solution set by negating it atop the encoded formula E, and check whether $E \ \& \ \neg Part_m$ is solvable. We continue the above process until all candidates under m are found. Among all the pipelining candidates that assign m tables to the CPU, Sirius selects the result with minimal CPU usage.

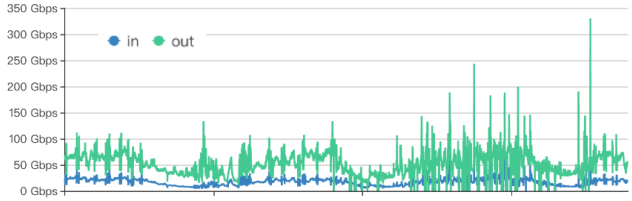
7 Experience

This section shares the deployment experience of Sirius (§7.1), real cases with Sirius (§7.2), and our lessons (§7.3).

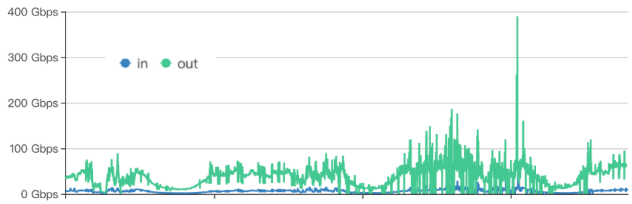
7.1 Deployment Experience

We started to deploy the edge clouds in 2017. As the number of business traffics and offloaded functions grew, it became increasingly difficult to arrange network functions in our edge clouds manually. We, therefore, started building Sirius in 2020. So far, Sirius has been used for one year. The network functions arranged by Sirius have been used to carry $O(10)$ types of business traffics, including streaming, games, IoT devices, and e-payments. With Sirius's assistance, we have built $O(100)$ gateways for edge-cloud services, $O(100)$ CDN nodes, and nearly 100 nodes for security in the past year. The peak throughput was higher than 10 Tbps across all edge-cloud instances.

Performance of gateways arranged by Sirius. An important metric for evaluating the effectiveness of Sirius is throughput, namely, whether the throughput of network functions arranged by Sirius meets our expectations. Figure 10 randomly selected two gateways where we used Sirius to arrange network functions automatically. These two gateways have been deployed in two edge clouds, respectively, and are mainly used to carry the streaming service. The throughput requirements for all the traffics on these two gateways are 500 Gbps. Figure 10 shows the performance of the gateways within one week. We



(a) The performance of gateway in the edge cloud A.



(b) The performance of gateway in the edge cloud B.

Figure 10: One-week performance of two edge clouds (for streaming service) where the network functions are arranged by Sirius. In - Traffic going into the edge cloud. Out - Traffic going out of the edge cloud.

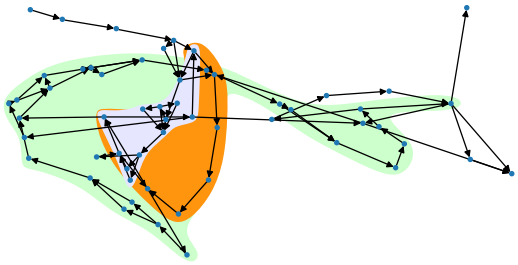
can observe that the peak traffic within one week was 300-400 Gbps, and the network functions arranged by Sirius can handle the traffic with very stable performance.

Development workload saved by Sirius. In terms of development efficiency, we combined Sirius with Lyra [5] to directly generate the compilable P4 programs that meet our specified throughput requirement. For the CPU-side code, our programmers have internal scripts to automatically generate C++ code. Before using Sirius, our gateway engineering team (more than twenty persons) spent more than two weeks analyzing and discussing a network function arrangement plan. After the initial network function arrangement version, it further took more than two weeks for adaptation and resource optimization. On the contrary, using Sirius, our programmers only need to write simple, high-level P4 programs and specify the throughput requirement. Sirius generates the expected results in an efficient and automatic way (within a few minutes). In the past year, Sirius decreased our programmers' workload by three orders of magnitude (from weeks to minutes).

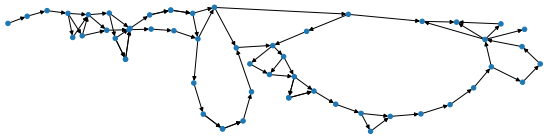
7.2 Real Cases Addressed by Sirius

We now describe three representative cases to show the practicality of Sirius in real world.

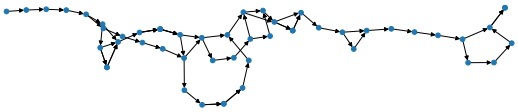
Merging and arranging network function chains for $O(10)$ business traffics. We now present a real network function arrangement process produced by Sirius in Figure 11. In one of our edge clouds, there are $O(10)$ network function chains (each for one sub-business traffic) needed to be offloaded onto the single programmable gateway in this edge cloud. These business traffics contain our mainstream services, including games and streaming. These $O(10)$ network function chains are merged into a network function graph shown in Figure 11a. This graph contained 127 processing loops. Figure 11a only colors three of these loops.



(a) Merging $O(10)$ network function chains results in a network function graph containing 127 processing loops. We only color three of the processing loops in this figure.



(b) A pipelining plan output by Sirius. There are only two recirculations in this result.



(c) The pipeline after the partitioning (this figure shows the part offloaded onto the programmable ASIC).

Figure 11: A real network function arrangement in our production. Over 10 network function chains types of traffics are arranged into a single gateway. Each back node in the above figures represents a snippet of P4 code containing around ten tables. Each $\rightarrow$ denotes the execution sequence.

Figure 11b shows a pipelining result output by Sirius. This pipelining result only contains two recirculations in three subchains. Our specified throughput for all business traffics is 0.5 Tbps, so this candidate meets the specifications. More importantly, our programmers spent one week creating a traffic identification table for forwarding different business traffics to the corresponding network functions. This manually-written table occupied 15.7% of SRAM memory and 38.1% of TCAM memory of the entire data plane program. With Sirius, we got this traffic identification table within a few minutes, and the table only used 11.5% of SRAM memory and 42% of TCAM memory of the manually-written one.

Figure 11c shows a partitioning result generated by Sirius. Some of the programs in Figure 11b have been moved to the CPU. Each node in the graph shown in Figure 11 represents a snippet of code—containing around 10 match-action tables if it is implemented in P4, and each arrow denotes the execution sequence. **The entire network function arrangement process, with Sirius, was finished within 10 minutes.**

Safely and easily network function updating. Another benefit Sirius offers is to ease our network function updating across edge clouds. Different edge clouds serve different businesses; thus, the data plane programs running on different edge clouds are diverse. However, before Sirius was deployed, our programmers used to update network functions across

```
table snat_session {
  key = {
  #ifdef IPv4
    hdr.ipv4.src_ip: lpm;
  #endif
  #ifdef IPv6
    hdr.ipv6.src_ip: lpm;
  #endif
  ...
  }
  size = SNAT_SESSION_NUM;
  actions = {
  #ifdef IPv4
    snat_ipv4_rewrite;
  #endif
  #ifdef IPv6
    snat_ipv6_rewrite;
  #ifdef VXLAN
    snat_vxlan_rewrite;
  #endif
  #endif
  }}
```

Figure 12: Example code snippet using macro extensively.

edge clouds via macros in P4. On different gateways, our programmers turned on different macros, and the compiler can remove the rest code. The macro solution is, nevertheless, hard to maintain and error-prone, because the P4 programs end up with $O(10)$ different macros and $O(100)$ copies of them spread across different files. Figure 12 shows a real table definition in our edge cloud, which adds many macros to change its key and actions based on the actual needs. We can observe that multiple macros are chained and nested together. A failure event that occurred two years ago in one of our edge clouds was caused by incorrectly updating macros, since our programmers made a mistake when they updated nested macros. Sirius solves this problem. For different gateways, the programmers only need to specify network function chains in P4 and provide the correct traffic separation DB, freeing them from the "tangled" macros.

Real partitioning case. Without Sirius, even though our programmers generate a pipelining plan with the satisfied number of recirculations, it is very hard to squeeze this single pipeline into the programmable ASIC due to the limited hardware resources. Thus, our programmers used to reduce the size of some tables in some network functions that they thought would experience low volume for a while to ensure the tailored pipeline complies with the hardware constraints. After a while, when the traffic volume went back up, they moved them back and shrank some tables in another network function. The above situations occurred on almost every gateway before Sirius was built. In a recent case, our programmers spent two weeks squeezing a merged graph (with Source NAT, destination NAT, and load balancer network functions) into the programmable ASIC. They tried to reduce the size of the routing tables but still failed to compile the entire graph. Sirius helped them to automatically generate a partitioning result that guided them to move the `source_NAT_session_table` to the CPU, efficiently solving this problem. Moving `source_NAT_session_table` to the CPU does not affect our throughput, since for that edge instance, the source NAT traffic volume is low while the session table sits in a critical position in the network function pipeline. Assigning it to the CPU not only frees up a lot of memory resources but also significantly shortens the length of the table dependency chain [13], which creates more headroom and makes the program easier to compile.

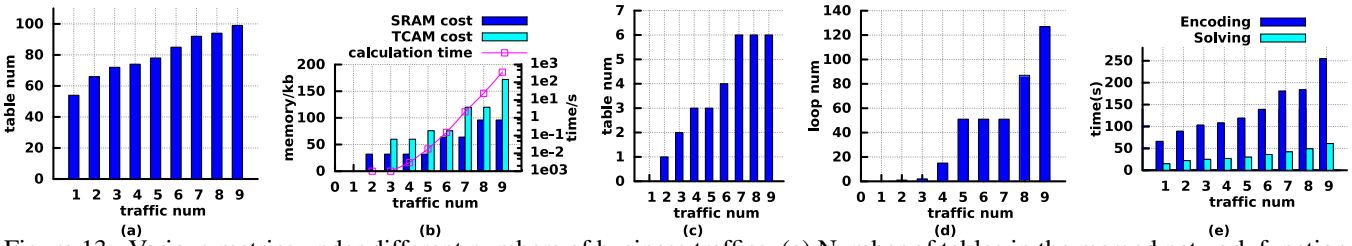


Figure 13: Various metrics under different numbers of business traffics. (a) Number of tables in the merged network function graph. (b) Memory cost and execution time of constructed traffic identification tables. (c) Number of traffic identification tables. (d) Number of loops in the merged graph. (e) Time to encode and solve the SMT formula.

7.3 Lessons and Discussions

We now share our lessons in using Sirius in our edge clouds, and also discuss the limitations of Sirius and open questions.

Can any of the pipelining candidates directly comply with the hardware constraints? Our one-year experience shows that none of the pipelining plan candidates can directly comply with the programmable ASIC's constraints. In other words, the partitioning between the programmable ASIC and the CPU is a must for the network function arrangement in our edge clouds. This situation results from (1) the limited hardware resources in programmable ASICs, and (2) our large-scale production programs.

Can a pipelining plan with more processing loops produce a better partitioning result? This may happen. We selected pipelining plan candidates that meet the specified throughput, and then partitioned these candidates. In other words, the pipelining plans that violate the specification are ineligible to join the partitioning phase, because they cannot be better than the pipelining candidates meeting the specification, no matter how to partition them. However, for the pipelining candidates, a candidate with more processing loops might be partitioned to a result with less CPU usage than another candidate with fewer processing loops.

Duplicating the overlapping network functions to remove processing loops is impractical. We now use the recirculation to remove processing loops. Another option is to duplicate the overlapping network functions to remove the loops. While such a solution, in principle, works and does not sacrifice the throughput, it is impractical due to the limited hardware resources. Since a network function needs about 5-10 stages, additional 5-10 stages for one processing loop is too expensive to the precious hardware resources.

Optimization target. Sirius currently minimizes the total number of tables assigned to the CPU, which is not always the best optimization target. For example, in some cases, our programmers want to pin some tables or network function chains onto the programmable ASIC to achieve consistency or low processing latency. Sirius can support such cases by only searching through candidates that satisfy the requirement.

Gateways in edge clouds v.s. Gateways in data center networks. We have equipped gateways with programmable ASICs in both edge clouds and data center networks. Compared with the data center case, deploying programmable data planes in the edge clouds is much more challenging.

First, each gateway in a data center network needs to hold only one network function, unlike the edge cloud, where a gateway needs to hold a large number of network functions. This is because there are $O(100)$ gateways in a data center network in our global network, and multiple network functions can be horizontally distributed across these gateways without squeezing all of them into a single gateway [16]. On the contrary, each edge cloud only contains two gateways (for redundancy) since edge clouds are typically deployed in the rented cheap, small-size machine rooms close to end users. Each gateway in our edge clouds needs to hold $O(100)$ network functions, and squeezing so many network functions into a programmable ASIC is very hard due to limited hardware resources. For the same reason, it is impossible to deploy multiple groups of gateways in the edge clouds.

Second, network functions in the edge clouds are logically more complex. In the edge clouds, the gateways process packets from over 10 types of business traffics. On the contrary, gateways in the data center just forward packets to the corresponding switches. We typically developed the P4 program for data center network gateways by just tailoring `switch.p4`.

Partitioning across switching ASIC, smartNIC, and CPU. Offloading network functions to smartNIC, in principle, can alleviate programmable ASIC hardware resources issue while maintaining the throughput, It can replace CPU and reduce the overall cost of the gateway. We are developing the next-generation gateway that replaces CPU with SmartNIC. We are also working on building a partitioning approach to distribute the code across ASIC, smartNIC, and CPU in order to balance the trade-off between performance and cost. While FlightPlan [23] has presented a good effort toward this direction, deploying such a system in production remains challenging.

8 Evaluation

Our evaluation mainly focuses on presenting the scalability of each component in Sirius. We chose 9 different business traffics, added them one by one, and recorded the time of execution and other metrics we were interested in. All experiments were performed on a server with a 2.5GHz CPU and

768GiB RAM.

Figure 13(a) shows the program scale. There are two observations. Firstly, different business traffics had many tables in common. Because the number of tables only doubled when we added 8 more business traffics. Secondly, the difference between different business traffics is similar, as the number of tables grows gradually as the traffic number increases.

Traffic identification table construction. We recorded the SRAM and TCAM usage under different numbers of network functions, and the time Sirius took to solve the problem. Figure 13(b) shows the result. Both types of memory increased as the number of traffics increased, which showed the overhead of distributing flows to different network functions. Also, the time it took to compute the minimal traffic identification table increased dramatically as the number increased. This is because Sirius had to try more combinations to find the optimal result. When there were 9 business traffics, Sirius took 371 seconds to construct the distribution tables. We also recorded the number of tables constructed in Figure 13(c). We observed that even though the memory increased, the number of tables did not always increase. This is because Sirius chose multiple rules that shared the same match field so that they could share the same table.

Pipelining. In the pipelining process, the algorithm Sirius used to solve the set covering problem is efficient enough to finish all experiments within a second. So we mainly present the number of processing loops in the network function graph as more business traffics are added. Figure 13(d) shows the result. The number of processing loops increases quickly as the business traffics become more complicated, which also demonstrates the programmers' workload before Sirius is deployed. Programmers had to plan carefully to make sure each business traffic visit functions in its desired order.

We also noticed that when adding the 6th and 7th business traffic, the number of loops stayed the same. This is because those business traffics are similar, and they shared the same code snippet that was in the loop.

Partitioning. In the partitioning phase, we mainly evaluated the time for Sirius to encode the SMT formula and for the SMT solver to solve it. The result is shown in Figure 13(e). Both the encoding and solving time increase as the number of business traffics increases. We can see that the encoding time grows faster than linear because the encoding is built atop the network function pipeline, whose complexity grows faster than linear as the business traffic is added. Solving the formula is faster than encoding it. This is because Sirius leverages the constraint encoding optimizations proposed by Cetus [13]. Overall, the partitioning finishes within minutes.

9 Related Work

Partitioning hardware code to CPU. Closed to Sirius's partitioning goal, Gallium [27] automatically translates a part of software-version middlebox programs into a P4 program running on a programmable ASIC. Gallium does not support recirculation modeling and mainly targets a specific switch-CPU architecture. In addition, Sirius's partitioning goal is also different from Gallium's. FlightPlan [23] deployed P4 programs across the switch, FPGA, and CPU to benefit bandwidth and heterogeneity. FlightPlan relies on programmers to explicitly split the program and profile each code block's performance on each platform. We cannot apply FlightPlan, since the above requirements do not hold in our scenario. Some of the prior work targeted partitioning between the SmartNIC and the CPU, such as iPipe [14], Clara [15], and Floem [17]. These efforts are quite different from Sirius's focused goal and assumptions. Dejavu [25] leverages recirculation to compose multiple network function chains to a single ASIC but it cannot handle the three challenges Sirius solves.

Compiler for programmable data planes. The state-of-the-art compiler systems for P4 [3,5,9,10,19,22] aim to optimize resource usage in programmable ASICs or simplify programmers' tasks on expressing their coding intent. For example, P4All [9,10] optimizes resource usage by leveraging reusable data structures. P4visor [29,30] optimizes resources by merging redundant code fragments (*e.g.*, header parser and tables). μP4 [22] and P4 Weaver [3] propose P4 modular programming to write P4 code from scratch or in an incremental way.

Extend switch memory with host memory. TEA [12] addresses the memory constraint issue on the programmable ASIC by extending ASIC memory with the DRAM on server machines. Even though memory occupation is one of our major concerns, as shown in Figure 2 and Cetus [13], our long network function chain is another major concern. TEA reduces memory footprint by introducing more computation logic in the programmable ASIC. This worsen our network function chain dependency issue and does not solve our problem.

10 Conclusion

This paper has shared our design of Sirius, and our experience with Sirius in our edge clouds after one year of use. Sirius is the first system capable of automating the network function arrangement problem. Compared with manual solutions to the network function arrangement problem, Sirius effectively decreased our workload for deploying and updating our edge clouds by three orders of magnitude from weeks to minutes.

This work does not raise any ethical issues.

Acknowledgement

We thank our shepherd, Ming Liu, and NSDI'24 reviewers for their insightful comments. Jiamin Cao was supported by Alibaba Group through Alibaba Research Intern Program. Ennan Zhai is the corresponding author.

References

[1] ASAI, H. Palmtrie: A ternary key matching algorithm for ip packet filtering rules. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies* (New York, NY, USA, 2020), CoNEXT '20, Association for Computing Machinery, p. 323–335.

[2] EADES, P., LIN, X., AND SMYTH, W. F. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters 47*, 6 (1993), 319–323.

[3] FATTAHOLMANAN, A., BALDI, M., CARZANIGA, A., AND SOULÉ, R. P4 weaver: Supporting modular and incremental programming in P4. In *Symposium on SDN Research (SOSR)* (2021).

[4] FATTAHOLMANAN, A., BALDI, M., CARZANIGA, A., AND SOULÉ, R. P4 weaver: Supporting modular and incremental programming in p4. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)* (2021), pp. 54–65.

[5] GAO, J., ZHAI, E., LIU, H. H., MIAO, R., ZHOU, Y., TIAN, B., SUN, C., CAI, D., ZHANG, M., AND YU, M. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous ASICs. In *ACM SIGCOMM (SIGCOMM)* (2020).

[6] GAO, X., KIM, T., VARMA, A. K., SIVARAMAN, A., AND NARAYANA, S. Autogenerating fast packet-processing code using program synthesis. In *18th ACM Workshop on Hot Topics in Networks (HotNets)* (2019).

[7] GAO, X., KIM, T., WONG, M. D., RAGHUNATHAN, D., VARMA, A. K., KANNAN, P. G., SIVARAMAN, A., NARAYANA, S., AND GUPTA, A. Switch code generation using program synthesis. In *ACM SIGCOMM (SIGCOMM)* (2020).

[8] HANCOCK, D., AND VAN DER MERWE, J. HyPer4: Using P4 to Virtualize the Programmable Data Plane . In *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2016).

[9] HOGAN, M., FEIBISH, S. L., ARASHLOO, M. T., REXFORD, J., AND WALKER, D. Modular switch programming under resource constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2022).

[10] HOGAN, M., FEIBISH, S. L., ARASHLOO, M. T., REXFORD, J., WALKER, D., AND HARRISON, R. Elastic switch programming with P4All. In *19th ACM Workshop on Hot Topics in Networks (HotNets)* (2020).

[11] IYER, R., PEDROSA, L., ZAOSTROVNYKH, A., PIRELLI, S., ARGYRAKI, K., AND CANDEA, G. Performance contracts for software network functions. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (USA, 2019), NSDI'19, USENIX Association, p. 517–530.

[12] KIM, D., LIU, Z., ZHU, Y., KIM, C., LEE, J., SEKAR, V., AND SESHAN, S. TEA: enabling state-intensive network functions on programmable switches. In *ACM SIGCOMM (SIGCOMM)* (2020).

[13] LI, Y., GAO, J., ZHAI, E., LIU, M., LIU, K., AND LIU, H. H. Cetus: Releasing P4 programmers from the chore of trial and error compiling. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2022).

[14] LIU, M., CUI, T., SCHUH, H., KRISHNAMURTHY, A., PETER, S., AND GUPTA, K. Offloading distributed applications onto smartNICs using iPipe. In *ACM SIGCOMM (SIGCOMM)*. 2019.

[15] NARAIN, S., LEVIN, G., MALIK, S., AND KAUL, V. Declarative infrastructure configuration synthesis and debugging. *J. Network Syst. Manage. 16*, 3 (2008), 235–258.

[16] PAN, T., YU, N., JIA, C., PI, J., XU, L., QIAO, Y., LI, Z., LIU, K., LU, J., LU, J., ET AL. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (2021), pp. 194–206.

[17] PHOTHILIMTHANA, P. M., LIU, M., KAUFMANN, A., PETER, S., BODIK, R., AND ANDERSON, T. Floem: A programming system for NIC-Accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2018).

[18] SCHOLZ, D., STUBBE, H., GALLENMÜLLER, S., AND CARLE, G. Key properties of programmable data plane targets. In *2020 32nd International Teletraffic Congress (ITC 32)* (2020), pp. 114–122.

[19] SIVARAMAN, A., CHEUNG, A., BUDIU, M., KIM, C., ALIZADEH, M., BALAKRISHNAN, H., VARGHESE, G., MCKEOWN, N., AND LICKING, S. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), pp. 15–28.

[20] SONCHACK, J., LOEHR, D., REXFORD, J., AND WALKER, D. Lucid: A language for control in the data plane. In *ACM SIGCOMM (SIGCOMM)* (2021).

[21] SONI, H., RIFAI, M., KUMAR, P., DOENGES, R., AND FOSTER, N. Composing dataplane programs with μp4. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 329–343.

[22] SONI, H., RIFAI, M., KUMAR, P., DOENGES, R., AND FOSTER, N. Composing dataplane programs with μp4. In *ACM SIGCOMM (SIGCOMM)* (2020).

[23] SULTANA, N., SONCHACK, J., GIESEN, H., PEDISICH, I., HAN, Z., SHYAMKUMAR, N., BURAD, S., DEHON, A., AND LOO, B. T. Flightplan: Dataplane disaggregation and placement for P4 programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2021).

[24] WINTERMEYER, P., APOSTOLAKI, M., DIETMÜLLER, A., AND VANBEVER, L. P2GO: P4 profile-guided optimizations. In *The 19th ACM Workshop on Hot Topics in Networks (HotNets)* (2020).

[25] WU, D., CHEN, A., NG, T. S. E., WANG, G., AND WANG, H. Accelerated service chaining on a single switch ASIC. In *18th ACM Workshop on Hot Topics in Networks (HotNets)* (2019).

[26] ZHANG, C., BI, J., ZHOU, Y., DOGAR, A. B., AND WU, J. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)* (2017), pp. 1–9.

[27] ZHANG, K., ZHUO, D., AND KRISHNAMURTHY, A. Gallium: Automated software middlebox offloading to programmable switches. In *ACM SIGCOMM (SIGCOMM)* (2020).

[28] ZHENG, P., BENSON, T., AND HU, C. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies* (2018), pp. 98–111.

[29] ZHENG, P., BENSON, T., AND HU, C. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2018).

[30] ZHENG, P., BENSON, T. A., AND HU, C. Building and testing modular programs for programmable data planes. *IEEE J. Sel. Areas Commun. 38*, 7 (2020), 1432–1447.

Empower Programmable Pipeline for Advanced Stateful Packet Processing

Yong Feng[1†], Zhikang Chen[1†], Haoyu Song[2], Yinchao Zhang[1], Hanyi Zhou[1],
Ruoyu Sun[1], Wenkuo Dong[1], Peng Lu[1], Shuxin Liu[1], Chuwen Zhang[1], Yang Xu[3] and Bin Liu[1]

[1]*Tsinghua University,* [2]*Futurewei Technologies,* [3]*Fudan University*

Abstract

Programmable pipeline offers flexible and high-throughput packet processing capability, but only to some extent. When more advanced dataplane functions beyond basic packet processing and forwarding are desired, the pipeline becomes handicapped. The fundamental reason is that most stateful operations require backward cross-stage data passing and pipeline stalling for state update and consistency, which are anomalous to a standard pipeline. To solve the problem, we augment the pipeline with a low-cost, yet fast side ring to facilitate the backward data passing. We further apply the speculative execution technique to avoid pipeline stalling. The resulting architecture, RAPID, supports native and generic stateful function programming using the enhanced P4 language. We build an FPGA-based prototype to evaluate the system, and a software emulator to assess the cost and performance of an ASIC implementation. We realize several stateful applications enabled by RAPID to show how it extends a programmable dataplane's potential to a new level.

1 Introduction

Dataplane devices equipped with high-performance, programmable switch chips are changing the landscape of networks in a profound way. More and more potentials, from realizing customized forwarding and middlebox functions to enabling in-network computing applications, are unleashed. The high-throughput demand makes the hardware Match-Action Table (MAT) pipeline [23] the chief choice of the switch chip architecture. For example, Intel Tofino [7] and Broadcom Trident [2] are both pipeline-based. Pipeline is also used in high-performance NIC (e.g., PANIC [38], nanoPU [33], and RingLeader [37]) for packet processing. Although a pipeline has unmatched throughput, it assumes a forward processing flow, impeding the efficient support for stateful functions essential to many valuable applications.

[†]Co-first authors.

A stateful dataplane function can be generalized as an Extended Finite State Machine (EFSM) [42]. The state of a packet is read from a flow state table; the corresponding actions are executed based on the current state and input; the action may result in a state update, which is written back to the state table. We discuss several real use cases in Sec. 2.1.

Trivial stateful functions (e.g., counter) can be realized as atomic operations using registers in a single pipeline stage. However, Most stateful functions need state writeback beyond the capability of a pipeline. For such functions, determining the next state often requires multiple actions and table accesses, causing delayed cross-stage state writeback. For certain applications [35,40], the size of state tables size surpasses the memory capacity of a single stage, compelling cross-stage table writebacks even with straightforward logic. To preserve state consistency, potentially impacted packets are blocked until writeback completion. The only recourse for data writeback is recirculation (i.e., looping the data back to the head of the pipeline). Both these inefficiencies can reduce the pipeline throughput to an unacceptable level.

Pure pipelines therefore falter in supporting stateful packet processing. There is a clear call for a new chip architecture that prevents pipeline stalls while facilitating fast, unobtrusive packet and data backtracking. One tentative method decouples the stage processors by placing all tables in a separate memory pool, allowing stages to interface with the same table, obviating writeback paths. However, pipeline stalling persists, accompanied by notable interconnection expenses and table access scheduling intricacies [26].

In this paper, we target a cost-effective, high-performance solution for arbitrary stateful functions via the MAT pipeline. We augment the pipeline with a simple side ring and make each pipeline stage interface with it. On the ring, data flows in the reverse direction of the pipeline, providing a fast backward communication path. The new architecture, **R**ing-**A**ugmented **PI**peline **D**ataplane (RAPID), is illustrated in Fig. 1. RAPID introduces a new "dataplane writable table" abstraction to enhance the programming language such as P4 [22] for flexible and easy stateful function composition and implementation.

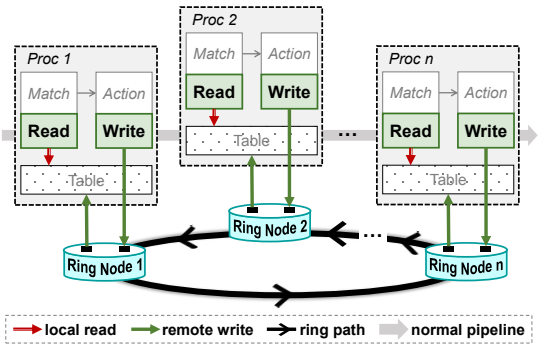


Figure 1: RAPID high-level architecture.

The ring serves two main purposes. First, it provides a fast path for new state writebacks to an earlier pipeline stage. Second, it is used to handle the failures of speculative execution. The speculative execution allows packets of the same flow to enter the pipeline without being blocked. But once the state is changed, the packets that read the stale state are resubmitted to the first stage of the stateful function through the ring for reprocessing, avoiding expensive packet recirculation.

This paper presents the design, implementation and evaluation of RAPID. Our key contributions are as follows:

• ***Cross-stage table writeback***: The architecture support and language abstraction enable programming stateful functions on a pipeline-based dataplane;

• ***Speculative execution***: RAPID uses it and ring-based failure resubmission to ensure state consistency and good pipeline throughput.

• ***Multi-level consistency support***: RAPID supports different consistency levels based on application needs.

We prototype RAPID on FPGA and evaluate its cost and performance. Estimations for area and power are made for a 45nm ASIC setup [10]. We enhance the P4 language for stateful functions, developing its compiler. A software ASIC emulator is used to emulate RAPID's behavior for parameter tuning and performance insights. Several use cases demonstrate the capability of RAPID.

The remaining of the paper is organized as follows. Sec.2 outlines stateful processing needs and our architectural rationale. Sec.3 reviews prior work. The architecture of RAPID is elaborated in Sec.4. Sec.5 presents the enhanced P4 language and compiler issues. Implementation and evaluations appear in Sec.6 and Sec.7. We discuss design choices in Sec.8 and conclude in Sec.9.

2 Background

2.1 Motivating Stateful Functions

Stateful functions are ubiquitous in network applications. The generic support for stateful functions enriches the programmable network dataplane devices, as embodied in the following motivating use cases.

• **Stateful Load Balancer.** Load balancers (LBs) are pivotal in cloud networks, where stateful LBs preponderate over stateless ones due to their flexibility [19, 34, 40, 62]. LBs need to ensure connection affinity. If a packet matches an existing flow in the state table, it adopts the old connection; otherwise, it triggers a strategy to set up the connection with a new server, and subsequently updates the state table. The strategy may entail intricate logic. Typically, the state table is updated through a controller, inducing queue buildup and lower throughput. Recent efforts [19, 40] champion dataplane-centric solutions, but when applied on a pipeline architecture, these necessitate cross-stage data writebacks.

• **DDoS Detection and Mitigation.** DDoS attacks (e.g., TCP SYN floods) persist as major security concerns. Conventional server or middlebox solutions are costly and constrained in throughput [63]. Thus, in-network dataplane solutions start to gain traction [31, 36, 39, 59, 63]. However, prevalent switch-driven DDoS defenses [1, 12, 27, 63] often lean on controllers or middleboxes for detection. We advocate a pure dataplane solution for better performance and efficiency. Packets undergo initial categorization using list tables (denylist, allowlist, graylist) and subsequent analysis by a detection module. This module, possibly a sketch or header-check series, re-categorizes suspicious packets from the allowlist to graylist. Such a method necessitates cross-stage writebacks in a pipeline to update tables.

• **Traffic Shaping and Policing.** Traffic shaping and policing are essential in enforcing Quality of Service (QoS) policies and ensuring optimal bandwidth allocation. During the process, packets are first classified by user-defined header fields, and then evaluated by rate control or scheduling algorithms based on the current queue or link status. Next, packets are queued or dropped, and the queue status is updated (i.e., a writeback to the evaluation module). In this case, the writeback would be from the egress pipeline to the ingress pipeline. Since the seminal work of PIFO [51], many efforts have been made to make packet scheduling programmable [18, 45–47, 61]. However, most of them cannot avoid data writebacks. On today's pipeline-based chips, these schemes have to use packet recirculation or approximation which influences the scheduling performance or accuracy.

• **Stateful Firewall with Connection Tracking.** A stateful firewall tracks active connections, discerning malicious traffic via flow context and a finite state machine. It consults a table to determine if a packet belongs to a current connection, updating connection details as necessary. For new packets, predefined rules dictate connection permission. Approved connections prompt state table updates with details like flow ID. Given complex state transitions and inter-module messaging, a direct dataplane pipeline cannot be realized without cross-stage data writebacks.

• **Heavy Hitter Detection.** Top-K heavy hitters aid in traffic routing, engineering, and real-time monitoring [20, 32, 52, 56]. For in-dataplane detection, the sizable hash table is spread

across multiple pipeline stages. Packets traverse each stage to find an appropriate location, and then the flow data is written back to a chosen stage. Present pipeline devices lean on controllers or packet recirculation for these tasks. A data writeback channel would be both practical and efficient.

2.2 Architecture Considerations

We can dissect a stateful function as state lookup, state calculation, and state update processes as shown in Fig. 2. State consistency issues may arise when a packet is updating a state from s' to s while another packet of the same flow reads the stale state s'. Two major chip architectures (pipeline-based and RTC-based) use different strategies to ensure consistency but so far none are ideal.

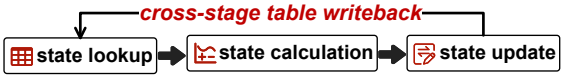


Figure 2: Stateful function abstraction.

• **Pipeline-based Architecture.** For a pipeline, there is no direct dataplane path for state updates if the function occupies more than one stage. Single-stage stateful functions [42] or atomic stateful functions [51] limit the scope of applications. Due to the unpredictable complexity of stateful functions, it is implausible to try to design a single stage processor to support an arbitrary stateful function which could make the resource intractable.

The existing architectures (e.g., FlowBlaze [42]) assume that all packets entering a stateful function may change the state, and to guarantee state consistency (i.e., avoid reading stale states), they block all the subsequent packets during the stateful processing of a packet, which creates a performance bottleneck. However, stateful functions often target a flow subset; distinct flows may access separate states, and packets within the same flow might not modify the state [41, 42, 63]. Therefore, it is rare for back-to-back packets to concurrently read and update the same state. *Speculative execution* techniques can prevent pipeline stalls. Once a speculative failure happens, reprocessing affected packets is essential for state consistency, manifesting another backward data passing requirement. Packet recirculation, retracing the whole pipeline, or controller detours involving a slower path are suboptimal. A dataplane path for fast and direct data writeback is desired.

• **RTC Architecture.** The Run-to-Completion (RTC) mode on multi-core, multi-thread processors (e.g., Trio [60] and dRMT [26]) supports stateful functions, but requires complex access scheduling and table locking to ensure state consistency if shared memory is used. To prevent state inconsistency during state calculation, the conflict-avoidance algorithm in dRMT needs to schedule halts for the cores with data dependency. Trio uses read-modify-write engines to do stateful operations. However, a long latency in state access (from read to write) results in extended suspension for other threads. Alternatively, each thread can maintain its own copy of the state table, but this demands table synchronization and consumes much more memory.

• **Consistency Levels.** Depending on applications, state consistency can be either strict or loose. For instance, a stateful firewall may tolerate a transitory state inconsistency whereby some packets employ outdated states [30]; on the contrary, NAT and MAC learning demand strict consistency to guarantee accurate packet processing and forwarding [62]. Enforcing strict consistency requires more resource and may affect the throughput. A good design should be able to adapt to the application requirements with a sound tradeoff.

• **Ring Topology.** When unidirectional data passing is the predominant communication pattern, a ring is the simplest and most efficient interconnection topology. On a unidirectional ring, there exists a unique path for a node to send messages to another node. Although a crossbar or other interconnection networks can provide more flexible communication paths [26, 28], they are an overkill to our problem and incur high implementation cost.

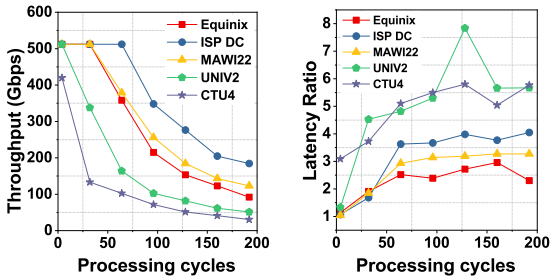
2.3 Traffic Trace Analysis

Before delving into the new architecture details, under various traffic traces and the assumption of different stateful processing latency cycles, we test the pipeline's throughput and latency using the conventional blocking scheme to expose the problem and motivate our work.

We collect 10 traffic traces (Table 8 in Appendix A) from campus, data center, and IoT networks and analyze the flows based on different specifications (i.e., five-tuple and sIP-dIP pair). Ignoring the timestamp, we feed the packets from the traces back-to-back into a 1GHz×64Byte pipeline. We define the Conflict Ratio (CR) as the ratio of the packets from the same flow which are spaced less than n clock cycles by packets of other flows, suggesting a potential consistency violation provided the state update latency of a stateful function is n. Table 8 in Appendix A shows the CR results for $n = 16$. As the value of n increases, maintaining state consistency through packet blocking and pipeline stalling reduces the pipeline throughput progressively.

Flow queues in front of a stateful function can mitigate the Head-of-line blocking (HOL) issue, enabling more flows to process packets in the stateful function pipeline, but may cause inter-flow packet reordering. Consider four queues, each holding 32 packets. The packets are scheduled in a round-robin manner as long as there is no packets of the same queue in the stateful function. We evaluate packet processing throughput and latency multiplication ratio over the stateless processing with five packet traces from Table 8 under varied state update latency cycles, shown in Fig. 3. With fewer cycles (e.g., 20-30 cycles), the queues buffer incoming packets effectively. As cycles rise, queues become overwhelmed,

leading to increased packet drops, reduced throughput, and heightened latency that plateaus.



(a) Throughput. (b) Latency ratio.

Figure 3: Packet throughput and latency under different stateful processing cycles (i.e., state update latency).

Our analysis, as well as the results from others [24], confirms that the state consistency does require pipeline to stall for real traffic. A coarser flow specification makes the problem even more severe. While more flow queues can reduce the HOL blocking probability, the system cost increases. All these point to the conclusion that the speculative execution is not only beneficial, but also imperative.

3 Related Work

There have been a long track of works concerning the stateful function support in programmable dataplane. The works range from new abstractions, new chip architectures, and solutions based on existing chips.

• **Abstraction.** OpenState [21] proposes a dataplane abstraction that uses eXtensible Finite State Machine (XFSM) to do stateful flow processing in switches. It relies on the OpenFlow protocol and has no real hardware support. Fast [41] uses multiple tables to simplify the flow-level state transition. The state transitions need data writeback to previous tables which can only be implemented in software.

• **Architecture.** Banzai [51] is the first hardware architecture for stateful functions. However, the stateful functions supported are limited to those that can be compiled as an atomic operation in a single pipeline stage. FlowBlaze [42] extends the OpenState abstraction to EFSM and modifies the stage processor of the conventional Reconfigurable Match Tables (RMT) pipeline to support stateful functions that can be implemented in a single stage. Due to the state processing latency, it needs to stall the pipeline for state consistency. SDPA [54] and SDP-CDP [29] use a co-processor to handle packets that need stateful processing and keep the pipeline for stateless processing only. Since a packet can only take one path, the scheme degrades into a software-based solution if most packets require stateful processing. dRMT [26] and Trio [60] are both multi-core-based architectures working in RTC mode. When supporting stateful functions, due to the state writeback and synchronization, their performance deteriorates. Banzai-based MP5 [49] further allows communication between multiple pipelines via crossbars, but it does not solve the cross-stage state writeback problem. Thanos [48] supports multidimensional packet filtering using a series of condition assertions in the pipeline, but it does not involve flow tables and thus cannot support functions that need state tables.

• **Solution.** The flexible match-action tables for the DPDK-based t4p4s target [50] are dataplane writable, so software-based stateful functions can be supported. Lucid [53] is a high-level programming language supporting event-driven dataplane packet processing. It relies on packet recirculation for stateful functions. Deterministic Finite Automaton (DFA) can be used to reduce certain complex stateful functions to basic atomic operations supported by the Banzai architecture [25]. However, its limited capability cannot support most of the use cases we discussed. RIBOSOME [44] leverages external CPUs or FPGAs to perform stateful packet processing. The programmable chip sends the packet headers and payloads to different external devices for processing, and then the processed packets are sent back to the pipeline for packet reassembly and forwarding. The processing latency and system cost are both high.

4 RAPID Architecture

4.1 Overview

While the concept of a side ring is simple and convenient, many details need to be considered to make it work. A stateful function involves a sequence of stage processors in a pipeline. The first processor (P_R) maintains the flow state table. It retrieves the states of incoming packets, and updates the states of flows as instructed by the last processor (P_W) after the intermediate processors finish the stateful processing and calculate the new states. P_W communicates with P_R through the ring. The "packet" processed in a pipeline is actually just a Packet Header Vector (PHV) which contains the parsed headers and other metadata pertaining to the packet. As shown in Fig. 4, a scheduler module can be configured as a ① Read Scheduler (*rd_sched*) in P_R or a ② Write Scheduler (*wr_sched*) in P_W. Each stage processor is attached with a ③ Ring Node Scheduler, which is responsible for resolving access conflicts between different types of data on the ring. Traversing a ring node requires only one cycle in most cases, much faster than the data flow on the pipeline.

We elaborate RAPID by answering the following critical questions:

- How to prevent packets who read stale states from being wrongly processed and forwarded? (Sec. 4.2)
- How to write back the updated states? (Sec. 4.3)
- How to ensure state consistency and in-order processing under speculative execution? (Sec. 4.4)
- How to support different levels of consistency? (Sec. 4.5)

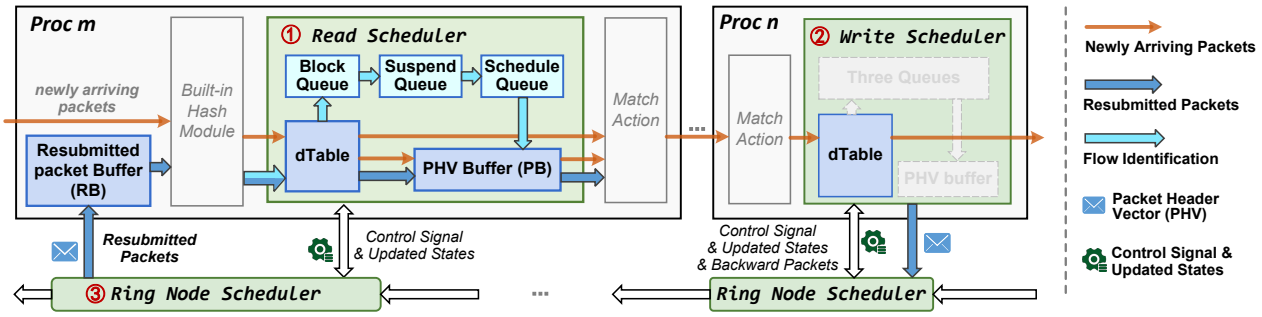


Figure 4: RAPID's architecture.

4.2 Stale State Safeguard

Before a new state is potentially calculated and written back, the state may be in a stale condition. Reading a stale state unattended could break the application. To prevent this, a small Content Addressable Memory (CAM) table, *dTable*, in both *rd_sched* and *wr_sched* is used to register the "dirty" flows which have packets currently under stateful processing. The search key of *dTable* is the flow's state table index (e.g., a flow ID's hash value). While *dTable* in *wr_sched* is a key-only table, *dTable* in *rd_sched* also contains associated data to keep the information for dirty flow handling (Sec. 4.4).

Once *wr_sched* realizes that a packet causes a change of its flow state, the flow is registered in its *dTable*. Meanwhile, *wr_sched* sends the updated state to *rd_sched*, as well as notifying *rd_sched* to register the flow in its own *dTable*. For any following packet under speculative execution, if it hits *dTable* at P_W, it means that the packet reads a stale state and it will be resubmitted to P_R through the ring.

A flow registered in *dTable* in *rd_sched* serves as the safeguard to prevent newly arriving packets of the flow from entering the stateful processing. During its residency, any already admitted packets of the flow would undergo a speculation failure and are resubmitted to P_R. When all the packets of the flow under speculation failure have been resubmitted, *rd_sched* sends a *cancel_dirty* signal to notify *wr_sched* to remove the flow from its *dTable*. The flow is removed from local *dTable* after all its backlogged packets are cleared.

For fairness, the packets of the flows which are not in *dTable* are processed without blocking. The resubmitted and blocked packets of a "dirty" flow are opportunistically scheduled at free pipeline cycles only. Therefore, a buffer is needed for the resubmitted and blocked packets. Under normal traffic conditions, the switch pipeline is only lightly loaded, leaving enough free cycles to handle the buffered packets.

4.3 Fast State Writeback

At P_R, the state of a packet is retrieved from a flow state table and stored as metadata which can be used as key for further table matching or as parameter for action execution. If the state is updated to a new value, the metadata is marked. If *wr_sched* finds the state is updated, the update is uploaded onto the ring for a writeback to P_R. Meanwhile, the flow is registered in *dTable* to detect speculation failures.

On the ring, the data can usually pass one node per clock cycle. However, it is possible that some other cross-stage table write operations exist, so there can be race condition (e.g., a function's resubmitted PHV may conflict with another function's *cancel_dirty* signal) delaying the data. A small buffer for the attached stage processor and another for the upstream ring node are allocated for a ring node. The ring node scheduler prioritizes control signals (including write-backs and *cancel_dirty* signals) over resubmitted PHVs, and merges the signals if possible.

At the destination ring node, the writeback data is offloaded to P_R. The flow is registered in *dTable* and the updated state is written into the flow state table, which is either a TCAM table or an SRAM-based hash table. Updating existing entries is straightforward, while some stateful functions (e.g., MAC learning) require generating and inserting new entries. While it is easy for an SRAM-based table (e.g., using a small stash [42]), it is complicated for a TCAM table. To resolve the priority order, significant calculation and entry relocation may be needed [57,58], incurring an intolerable delay. Fortunately, most use cases do not need TCAM table entry insertion. We leave the solution to this issue as future work.

4.4 Speculative Execution

The blocking scheme that blocks every packet before its predecessor completes the stateful processing is too conservative. The improvement that only blocks packets of the same flow for which a packet is under stateful processing is still not good enough: a stateful function using only one stage processor can still result in more than 20% throughput reduction [24].

Instead, we use speculative execution. If a packet is under the stateful processing, the following packets of the same flow can still enter the stateful processing pipeline until the flow is registered in *dTable* in *rd_sched*. The packets under speculation failure must be resubmitted and reprocessed. These measures have some performance impact. The worst case, which is unlikely in reality, happens when a long sequence of back-to-back packets come from the same flow and each

packet causes a state transition. Our evaluation shows that in normal cases RAPID maintains a high throughput thanks to the speculative execution.

A speculation failure is handled by three procedures: *packet backhauling*, *packet buffering*, and *packet releasing*, as illustrated in Fig. 6.

Packet Backhauling. Packet backhauling uses the ring to send packets back to P_R for reprocessing. As shown in Fig. 5, a 4096-bit data bus on the ring is sufficient for transmitting either a PHV or a state data which contains a stateful function ID, a flow state table index (*addr*), and an updated state value (e.g., our use cases only use 128 bits for the state value). The signal type is indicated by *ctrl_tag*: 0b00 for invalid signal, 0b01 for state writeback and *cancel_dirty*, and 0b10 for resubmitted PHVs. The destination stage processor's ID is encoded in the one-hot bitmap *dst2*.

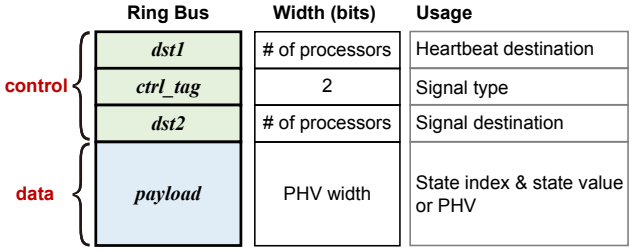


Figure 5: Ring bus composition.

Packet Buffering. The resubmitted packets need to be buffered in a FIFO queue at P_R and wait for free cycles to enter the pipeline. Since the match-action unit requires a PHV to calculate the table search keys in a stage using the hash module, the buffer for resubmitted packets - Resubmitted packet Buffer (RB), is located before the hash module (Fig. 4).

Newly arriving packets from the pipeline that hit *dTable* in *rd_sched* are kept in PHV Buffer (PB) in *rd_sched*. The packets for each flow are linked in a list, and the link pointers are maintained by the corresponding flow entries in *dTable*.

Once the first packet in RB gets a chance to enter the pipeline, it is moved from RB to PB. In PB, the resubmitted packets for each flow are also linked in a list, and their link pointers are maintained in *dTable* as well.

rd_sched can schedule a packet of a flow suffering speculative failure in PB only if **(1)** all resubmitted packets of the flow have arrived at P_R, and **(2)** all the resubmitted packets of the flow in RB have been moved to PB. Condition **(2)** can only be satisfied after condition **(1)** has been met. The former is guaranteed by the flow timers (described in the next subsection), and the latter is guaranteed by the resubmitted packet counters, both maintained in *dTable*. Each time P_R receives a resubmitted packet, the corresponding counter is incremented, and each time a resubmitted packet is moved from RB to PB, the corresponding counter is decremented. A counter value 0 means the condition **(2)** is met.

Fig. 6 illustrates an example of packet scheduling in *rd_sched*. A shows the *dTable* which has three flows f_1, f_2, and f_3, where "Timer Offset" means the elapsed time since the flow is last blocked or resubmitted (at most T). B shows the current status of RB and PB in which $p(i,j)$ means the jth packet of f_i. In this example, all the packets of f_1 have been in PB, but f_2 and f_3 each have one packet left in RB (i.e., $p_{2,2}$ and $p_{3,1}$). At this moment, f_1 and f_2's timers are both expired. In summary, only f_1 is legitimate to be scheduled.

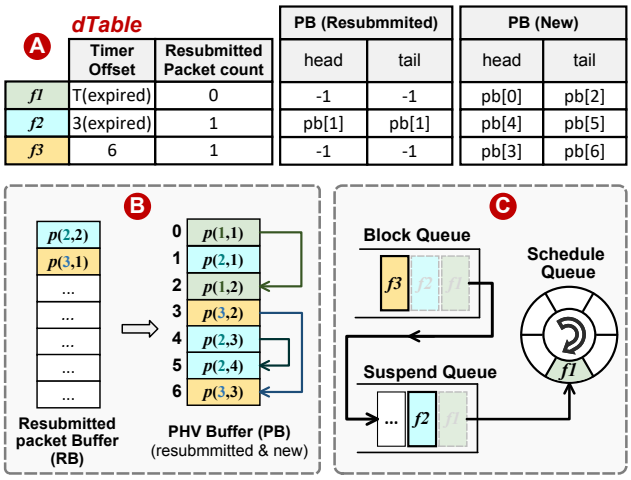


Figure 6: Scheduling dirty flows at *rd_sched*. (*f3* in Block Queue; *f2* in Suspend Queue; *f1* in Schedule Queue.)

Once a flow can be scheduled by *rd_sched* , the flow record should be removed from *dTable* in *wr_sched*. To achieve this, *rd_sched* sends a *cancel_dirty* signal to P_W by setting *ctrl_tag* to 0b01, the same as state writeback signal. Because the *cancel_dirty* signal carries only a 64-bit index while PHV has 4096 bits, multiple *cancel_dirty* signals can be merged with their indices placed in different locations in the payload in Fig. 5. Furthermore, *cancel_dirty* signals can also be merged into writeback data, so they share the same *ctrl_tag*. This mechanism resolved the potential conflict. Because the control signals have priority in the ring node scheduler, they are never delayed. Assume the pipeline has l stages and the stateful function occupies m stages, the *cancel_dirty* signal would traverse $l-m+1$ ring nodes to reach P_W. A reasonable design can guarantee the signal reach P_W earlier than any packet of the addressed flow[1], to avoid the case that a newly released packet hits the *dTable* in *wr_sched* again and is wrongly resubmitted.

Packet Releasing. From the moment *rd_sched* learns a flow changes its state, it needs at least $T = c(m)+m+2$ clock cycles to ensure that the speculative packets of the flow that read the stale state, if any, have all been received and buffered in *RB* at P_R, where m is the number of pipeline stages

[1]The *cancel_dirty* signal needs at most *processor_num* cycles to reach the destination processor, where *processor_num* < 16 in general designs. The number of cycles of a processor is greater than 20, and then the *cancel_dirty* signal can arrive earlier than the released packets.

between P_R and P_W (each ring node takes one clock cycle to pass), $c(m)$ is the pipeline processing latency on these stages, and 2 accounts for the cycles for uploading and downloading the PHV to and from the ring. This waiting time, T, must be observed by each flow in *dTable*.

Instead of maintaining a timer for each flow in *dTable*, we use a First-In-First-Out (FIFO) queue and two clocks to achieve the same goal: the DECREASE clock (D.clk) keeps the remaining waiting time for the first flow to be scheduled, and the INCREASE clock (I.clk) keeps the elapsed time since the arrival of the last packet from one of the flows in *dTable*. The reason we need only two clocks is that the flows are scheduled in the same order as they are registered in *dTable*.

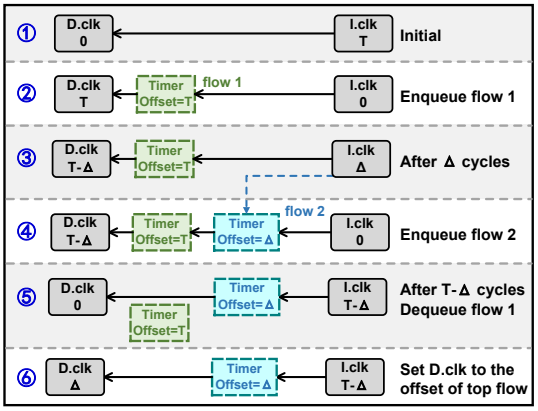


Figure 7: Achieve per-flow timer with two timers.

We use an example in Fig. 7 to show how this works. When the first packet of the first flow f_1 arrives, the initial value of I.clk, T, is copied to D.clk and f_1's timer offset field in *dTable*, and then I.clk is reset to 0 (①→②); both clocks start to work. When the first packet of flow f_2 arrives at the Δ-th cycle, the value of I.clk, Δ, is copied to f_2's timer offset (which means f_2 can be scheduled Δ cycles after f_1 is scheduled), and then I.clk is reset to 0 (③→④). After T-Δ cycles, D.clk decrements to 0, so f_1 meets its scheduling condition **(1)**. Now f_2 becomes the next flow to be considered (⑤). Its timer offset, Δ, is copied to D.clk (⑥). As a result, f_2 will meet its scheduling condition **(1)** after Δ cycles and its total waiting time is also T cycles.

During a race condition on the ring, a resubmitted packet may be delayed by the ring node scheduler. Thus, a constant value of T is no longer accurate. To address this problem, we introduce a heartbeat signal, *dst1*, on the ring bus as shown in Fig. 5. *dst1* is a one-hot bitmap. Each P_W sets the corresponding bit for P_R to one. At each cycle, if P_W generates another type of data, the heartbeat is piggybacked on it. If P_W has no other data to send to the ring and the ring is free, a heartbeat-only data is generated by setting *ctrl_tag* to 0b11; if the ring is not free (i.e., a data from an upstream node on the ring is scheduled), the heartbeat signal is overloaded to that data by setting the corresponding bit for P_R in *dst1* of the data. Thus, for each cycle of delay a resubmitted packet experiences, *rd_sched* will not receive the heartbeat for the flow for one cycle, which causes both I.clk and D.clk to halt for a cycle, so the scheduling time for the flow is compensated. A ring node can combine non-conflicting signals (e.g., different heartbeats or heartbeat with updated states) to use the ring bus bandwidth more efficiently.

Ⓐ and Ⓑ of Fig. 6 show that in PB, the resubmitted packets and the newly arriving packets of a flow in *dTable* are maintained in two linked lists (e.g., the resubmitted packet $p_{2,1}$ is in a linked list and the newly arriving packets $p_{2,3}$ and $p_{2,4}$ are in another linked list. When the timer of a flow expires and all the resubmitted packet of the flow are moved to PB, the resubmitted packet list is prepended to the newly arriving packet list, and the flow can be scheduled to release its packet from the newly arriving packet list. Ⓒ in Fig. 6 illustrates the flow scheduling process. The Block FIFO queue stores the flows in *dTable* which have not met the scheduling condition **(1)** . The Suspend FIFO queue stores the flows which have met condition **(1)** but not condition **(2)**. A flow removed from the Block queue may not enter the Suspend queue if its resubmitted packets are all moved to PB before its timer expires. Such a flow or a flow removed from the Suspend queue has met both conditions, so it is transferred to a circular Schedule queue. Flows in the Schedule queue are scheduled in a round-robin manner. When all the packets of a flow are scheduled, the flow is removed from the Schedule queue, and its record is removed from *dTable* as well. Thus, the process of a flow suffering speculation failure ("Block Queue - Suspend Queue - Schedule Queue" in the read scheduler of Fig. 4) is illustrated.

If a scheduled packet is resubmitted to P_R again before all the backlogged packets of the same flow are cleared, the flow is removed from the Schedule queue and inserted to the back of the Block queue. Its timer is restarted. The corresponding flow entry in *dTable* is updated accordingly. Appendix B illustrates that the probability of hash collisions in *dTable* is so negligible that it can effectively be disregarded.

4.5 Multi-level Consistency

SwiSh [62] categorizes stateful functions by their consistency needs: strict, weak, and bounded staleness. Applying strict consistency universally may sacrifice performance unnecessarily. In RAPID, an FSM guides *wr_sched* to accommodate these varied consistency levels.

For strict consistency, every packet hitting *dTable* in *wr_sched* should be resubmitted. With weak consistency, state changes trigger writebacks without *dTable* registration, avoiding future resubmissions. This model works for scenarios where the state can eventually converges even some packets read stale states (e.g., a flow rate limiter).

Bounded staleness consistency tolerates stale state reads for up to K packets. Once this limit is met, state synchronization is mandatory before continuing. Fig. 8 presents the FSM. In

the *Run* state, no flow exists in *dTable* of *wr_sched*. On a state change, the FSM shifts to *Expiring* with a counter set to K. Here, K packets can proceed without resubmission. At counter zero, the FSM enters *Expired*, registering the flow in *dTable* for strict synchronization. Only a *cancel_dirty* from *rd_sched* can deregister the flow and return to the *Run* state.

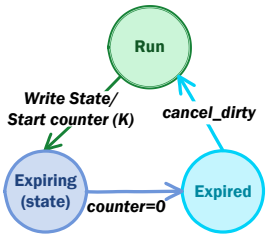


Figure 8: Multi-consistency support.

Key applications for this model are packet scheduling and sketch-based DDoS detection. Sketch-based DDoS detection risks measurement inaccuracies. Using weak consistency can delay state convergence, while strict consistency can burden the pipeline. Bounded staleness offers a balance between accuracy and efficiency with a configurable K threshold.

4.6 Handling High State Update Rate

Consider a flow with n back-to-back packets and a state update latency of T cycles between P_R and P_W. RAPID struggles with elevated state update rates, leading to more writebacks and speculation failures. When the state update rate exceeds $\alpha = \frac{2T-3}{2T+n}$, RAPID starts to exhibit even worse performance than the blocking scheme (Appendix C). Although this scenario is highly unlikely, it can be handled by introducing an additional field `resubmit_cycle` in *dTable* to downgrade the RAPID scheme to the blocking scheme once a high state update rate is observed on the per-flow basis.

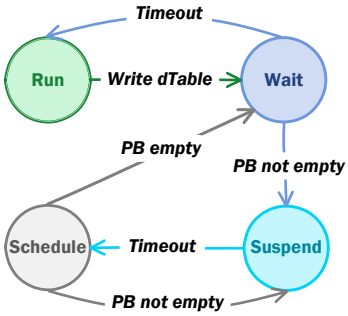


Figure 9: FSM for the blocking mode in RAPID.

When the packets of a flow have not been fully scheduled to be reprocessed, a new state for this flow may be written back, followed by some resubmitted packets again. In this case, the flow resets its timer to T (re-enqueued into the Block queue), and its `resubmit_cycle` increments by one. When the `resubmit_cycle` exceeds a threshold, the blocking scheme is applied to the flow. At the same time a *cancel_dirty* signal is transmitted. The blocking scheme lasts until PB of the flow is cleared. Consequently, the `resubmit_cycle` serves as an indicator of the state writeback rate. PB drives an FSM in the blocking mode. As depicted in the FSM in Fig. 9, if packets of the flow are present in PB, the flow enters a *Suspend* state, requiring the timer (no need for heartbeat) to expire before a packet can be scheduled. Once the packet is scheduled, if remaining packets of the flow still reside in PB (PB non-empty), the flow re-enters the *Suspend* state; otherwise, the flow transitions to the *Wait* state and is removed from *dTable* when the timer expires.

5 Programming Language and Compiler

5.1 P4 Language Enhancement

P4 [22] only supports atomic stateful operations using registers in a single stage (i.e., the Banzai architecture). NPL [11] does not support stateful functions at all. XL [55] requires intricate design to extract usable state transitions [25]. After evaluating the existing dataplane programming languages, we take $P4_{16}$ as the baseline and enhance it to support general stateful function programming on the RAPID architecture.

```
/* can be written by the data plane or the control plane;
   can be compiled as registers or flow state tables */
muTable packet_filter {
    keys = {
        hdr.ipv4.src_addr;
        hdr.ipv4.dst_addr;
        hdr.ipv4.protocol;
    }

    values = {
        bit<2> state;
    }

    type = exact; // ternary, lpm, direct
    consistency = STRICT; // WEAK, BS(K)
    size = 4096;
}

/* read out the current state */
cur_state = packet_filter.read(hdr);
/* get new state with calculations */
new_state = cal_state(...);
/* write back the new state */
packet_filter.write(hdr, new_state);
```

Figure 10: P4 language enhancement.

We introduce *muTable*, akin to *table* of P4, but modifiable by the dataplane, suitable for flow state tables. Unlike *table*, *muTable* leverages various memory types (e.g., SRAM, TCAM) for multi-stage functions or registers for single-stage atomic stateful functions. As depicted in Fig.10, `keys` index states, `values` store state data, `type` defines the match method (i.e., exact, ternary, LPM, or direct), and `consistency` sets consistency levels. *muTable* supports both `read` and `write`

primitives, enabling customized stateful processing logic. The compiler determines if a stateful function spans single or multiple stages based on complexity. A *port_knocking* example using *muTable* is presented in Fig. 25 of Appendix H.

5.2 Data Renaming

If the stateful processing involves modifying some data (e.g., decrementing *ipv4.ttl*), packet resubmission may cause the data to be modified more than once, leading to incorrect result. The compiler uses the "field renaming" technique to solve the problem, which is similar to the "register renaming" used for instruction-level parallelism in computer processor architecture [43].

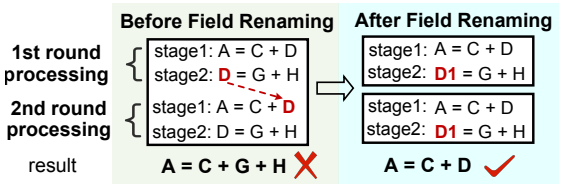


Figure 11: Field Renaming.

Fig. 11 shows an example. In the first packet processing round, *D* is used as an *rvalue* for calculating *A* before it is updated. Without field renaming, if this packet is resubmitted and reprocessed, *D* will use its new value to calculate *A* again. Instead, for a "Write-after-Read" (RAW) data, the compiler renames it to avoid such errors, and the new field *D*1 remains to be used for processing beyond the stateful function.

6 Implementation

6.1 Hardware Prototypes

We build hardware prototypes on a Xilinx UltraScale+ XCVU13P-based programmable switch. XCVU13P has up to 1.7M LUTs, 3.5M FFs, and 2.7K BRAMs. We implement PISA [23], Banzai [51], FlowBlaze [42], and RAPID with 2,584, 2,292, 3,627, and 4,676 lines of Scala code, respectively. For RAPID, 1,822 lines of code are dedicated to the ring. The prototypes of PISA, Banzai, and RAPID contain 4 physical pipeline stages; the prototype of FlowBlaze has only 3 stages due to its high resource consumption. All the prototypes use a 512-byte PHV. Each prototype has 4 hash modules, 4 parallel matching tables (FlowBlaze has one additional TCAM-based EFSM table), and a number of SRAMs and ALUs. We set 4 queues for FlowBlaze with each having a depth of 64. The RAPID ring bus is 4,114-bit wide (8-bit *dst1*, 8-bit *dst2*, 2-bit *ctrl*, and 4,096-bit *payload* for PHV or state data). The configuration of the RAPID schedulers is shown in Table 1. All prototypes run at 100 MHz clock frequency. A stage processor of PISA, Banzai, FlowBlaze, and RAPID takes 16, 16, 21, and 18 clock cycles, respectively.

Type	Size
Resubmitted Packet Buffer (RB)	16
dTable	64
PHV Buffer (PB)	32
Block Queue	64
Suspend Queue	64
Schedule Queue	64
Ring Buffer from pipeline	8
Ring Buffer from ring bus	8

Table 1: Configuration of RAPID schedulers.

6.2 Software ASIC Emulator

We develop software emulators for RAPID [14] (2,557 lines) and FlowBlaze (1,440 lines) using C++ on the Ubuntu 20.04 LTS server which can emulate the ASIC behavior. Users can write stateful packet processing programs using the the enhanced P4 language to test it on the emulator. The virtual clock of the emulators is set to 1 GHz. Since FlowBlaze only supports single-stage SRAM-based state updates, we test it with a simple case, the elephant flow detector. The emulators are configured to support 16x 100Gbps ports per pipeline.

6.3 Compiler and Controller

We develop a compiler (5,606 lines) in C++ to compile programs in the enhanced P4 language. The compiler generates a JSON file that describes configurations of every stage processor which can be directly used by the software target. For the hardware target, we further use Python to convert the JSON file into a binary file which can be downloaded to the FPGA. We also implement a controller (1,967 lines) in Python to communicate with the pipeline at runtime.

6.4 Implemented Use Cases

We implement and verify the following three use cases in both hardware and software. These use cases use 2, 3, and 4 stage processors, respectively.

• **Port knocking stateful firewall.** Port knocking [21] enables a firewall to accept connection attempts on a sequence of closed ports, and upon receiving the correct sequence, modifies rules to grant access. Using *muTable* in $Proc_1$, packets retrieve the current state. $Proc_2$, integrating this state and the port of the packet, updates the knocking state table in $Proc_1$. Packets matching the "pass" state are forwarded; others are dropped.

• **DDoS detection and mitigation.** We implement a cookie-based SYN flood detection and mitigation approach. $Proc_1$ keeps an Access Control List (ACL) table for admission control, and $Proc_2$ maintains a bloom filter to identify the first packet of a new flow. In $Proc_3$, for a new flow, the hash value of the packet's five-tuple is sent back to the sender as a cookie; otherwise, the packet is dropped, and the five-tuple is written back to $Proc_1$'s ACL for blocking. For a packet of monitored

TCP flow, if the bloom filter reports positive match and the packet carries the correct cookie with the right ACK, its flow will be added to $Proc_1$'s ACL as an admitted flow.

• **NAT with load balancing.** We adopt the method from SilkRoad [40]: at the load balancer, a packet looks up a ConnTable using the carried VIP. If found, the associated DIP is retrieved and used for forwarding; otherwise, the VIP is used to look up a VIPTable, selecting a DIP with the lowest load from a list. The load is then updated, and the new {VIP: DIP} mapping is added to the ConnTable. The ConnTable resides in $Proc_1$, with writebacks issued from $Proc_4$.

7 Evaluation

• **Methodology.** We use Vivado Design Suite [17] to synthesize the four prototypes (PISA, Banzai, FlowBlaze, and RAPID), obtaining the FPGA resources usage. To evaluate the throughput and latency performance, we feed three real traffic traces (ISP DC, Equinix, and MAWI22) to the FPGA prototypes. We feed three synthetic traces to the ASIC emulator to test RAPID's sensitivity to different factors.

• **Testbed.** The testbed comprises the switch prototype with four 100Gbps ports, a server as the controller, and two servers for traffic sending and receiving through two 100Gbps NICs.

7.1 FPGA Resource Consumption

Table 2 compares the FPGA resource consumption for a processor of different architectures. RAPID consumes more resources than PISA and Banzai, but less than FlowBlaze. Consider RAPID's capability, the overhead is well-justified.

Architecture	**LUT**	**FF**	**BRAM**
PISA	13.91%	1.71%	14.08%
Banzai	15.77%	1.73%	14.08%
FlowBlaze	27.32%	2.48%	25.99%
RAPID	20.22%	2.35%	19.55%

Table 2: Resources for different architectures.

Table 3 breaks down the resource of the scheduler components in a RAPID processor. As the three queues only use pointers, they consume a few LUTs and FFs. The BRAMs are mainly used for storing PHVs in RB and PB.

Component	**LUT**	**FF**	**BRAM**
RB	9.273%	5.255%	9.324%
dTable	6.869%	4.476%	0
PB	3.104%	0.543%	18.649%
Block Queue	0.050%	0.063%	0
Suspend Queue	0.101%	0.034%	0
Schedule Queue	0.055%	0.034%	0
Ring Node	9.003%	10.667%	0
Sum	28.455%	21.073%	27.973%

Table 3: Resource breakdown of scheduler components.

We can feed the traces at 100Gbps rate to the RAPID prototype running different use cases and find no packet drop.

7.2 ASIC Implementation Cost

We run Design Compiler [15] to synthesize the prototypes on an open-source 45nm ASIC technology library [10], and show the chip area and power consumption of single stage processor in Table 4. The clock frequency of the RAPID prototype reaches 1GHz. While FlowBlaze's cost is significantly higher than PISA, RAPID's cost is only slightly higher.

	Area (mm^2)	**Power** (mW)
PISA	94.33	65000
Banzai	95.17	66100
FlowBlaze	176.08	86900
RAPID	99.45	67500

Table 4: Area and power of different architectures.

Table 5 summarizes the cost breakdown of the scheduler components in a RAPID processor. The area and power overhead of the scheduler accounts for only 4.18% and 1.2% of the entire processor, exhibiting a low implementation cost.

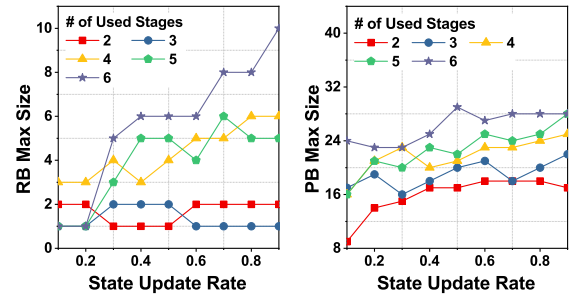
	Area (mm^2)	cost	**Power** (mW)	cost
RB	0.5263	0.5292%	108	0.1600%
dTable	0.0740	0.0744%	7.724	0.0114%
PB	2.4874	2.5009%	516	0.7644%
Block Queue	0.0074	0.0074%	1.56	0.0023%
Suspend Queue	0.0036	0.0036%	0.728	0.0011%
Schedule Queue	0.0036	0.0036%	0.729	0.0011%
Ring Node	1.0515	1.0573%	178	0.2637%
Total	4.1537	4.1764%	812.7	1.2041%

Table 5: Area and power overhead of scheduler components.

In both FPGA and ASIC, the hardware overhead (i.e., LUTs, FFs, ASIC area and power) to support stateful operations is moderate and constant. When there is no speculation failure, the pipeline latency and throughput remain unaffected. RAPID's *dTables* and the three extra queues are functioning in parallel with the main processing engine (MAUs), so the scheduling cycles do not incur extra clock cycles.

7.3 Parameter Setting

It is important to set the right size for the key performance and cost influencers (i.e., RB, PB, *dTable*, and Ring Node Buffer). We study this with the ASIC emulator. Assuming 90% pipeline throughput (i.e., 10% free pipeline cycles), we vary the number of stateful function stages and state update rates (i.e., the proportion of packets that trigger state updates to the total packets). We use three traces, ISP DC, Equinix, and MAWI22, for the test, and replicate each trace multiple times to get 1-minute worth of traffic. The results on Equinix are shown in Fig. 12, and the other results can be found in Appendix D. In general, the required component sizes increase



(a) RB. (b) PB. (c) dTable. (d) Ring Node Buffer.

Figure 12: Scheduler parameters under CAIDA Equinix trace.

as the state update rate and stateful function stage count grow. However, the sizes remain in an acceptable range. In real network scenarios, the pipeline throughput, the state update rate, and the stateful function stage are mostly at their low end of the scope, implying our configuration in Table 1 is sufficient.

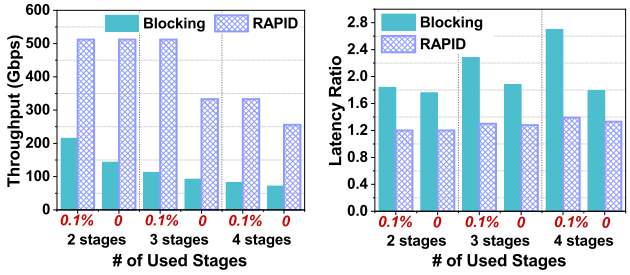
7.4 Influence of Stateful Function Stages

The length of a stateful function pipeline determines the state update latency. We test RAPID and the blocking scheme with different number of stateful function stages (2, 3 and 4) which take 36, 54 and 72 clock cycles, respectively. The blocking scheme uses 4 queues of size 32. We synthesize three traces with packet lengths of 256 bytes, 384 bytes, and 512 bytes, respectively. The characteristics of three traces are shown in Table 6. We orchestrate the packet headers for each trace to achieve a consistent state update rate of 30%.

Trace	packet size	spacing probability of two packets in a flow		
		< 36 cycles	< 54 cycles	< 72 cycles
Trace 1	256 B	31.42%	35.59%	39.33%
Trace 2	384 B	26.83%	31.42%	34.67%
Trace 3	512 B	21.65%	26.83%	31.42%

Table 6: Features of three synthetic traces.

We get the packet throughput and latency performance in terms of the number of function stages under Trace 2 as shown in Fig. 13 (other results are in Appendix E). As the number of stages increases, both the blocking scheme and RAPID exhibit a declining performance, but RAPID outperforms the blocking scheme and maintains a stable low latency.

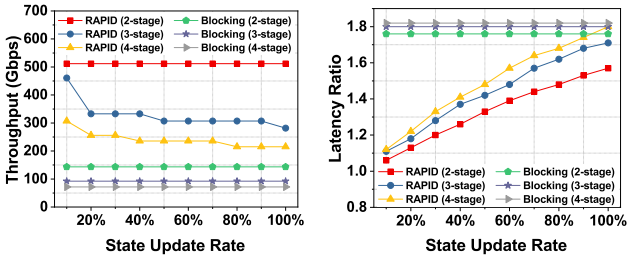


(a) Throughput. (b) Latency.

Figure 13: Performance with < 0.1% and 0 packet drop rate on Trace 2.

7.5 Influence of State Update Rate

A high state update rate predictably leads to a poorer pipeline throughput. To test how RAPID performs under different state update rate, we run Trace 2 on three hypothetical stateful functions with 2, 3, and 4 stages, and adjust the state update rate to examine its impact on the system. The throughput with no packet loss is derived. Fig. 14(a) shows that, as the state update rate increases, RAPID's throughput gradually declines, but its performance remains better than the blocking scheme. From Fig. 14(b) shows that RAPID's packet processing latency gradually rises with the increase of state update rate, and approaches the latency of the blocking scheme. The reason is that, when state update rate increases, packets may be resubmitted more than one time, leading to longer latency. Same conclusion can be drawn from results on Trace 1 and Trace 3 in Appendix F.



(a) Throughput. (b) Latency.

Figure 14: Performance on different state update rates of Trace 2.

7.6 Influence of Intra-flow Packet Gap

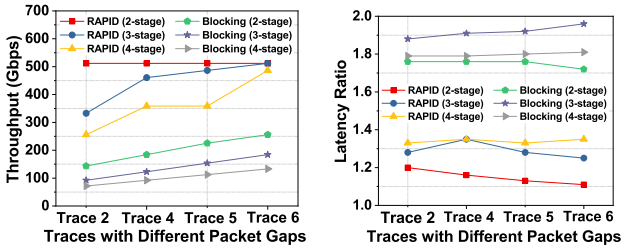
A larger intra-flow packet gap means that, for RAPID, it is less likely for the packets to be resubmitted or blocked, and for the blocking scheme, it is less likely for a packet to experience

HOL blocking. A gap larger than the state update latency can make both problems disappear. We devise experiments to validate this. Based on the Trace 2, we generate three additional traces (Table 7) with different intra-flow packet gap distribution. We set the state update rate to 30%. The result is shown

Trace	packet size	spacing probability of two packets in a flow		
		< 36 cycles	< 54 cycles	< 72 cycles
Trace 4	384 B	24.57%	26.83%	31.42%
Trace 5	384 B	21.65%	24.57%	28.62%
Trace 6	384 B	17.74%	21.65%	26.83%

Table 7: Features of three additional synthetic traces.

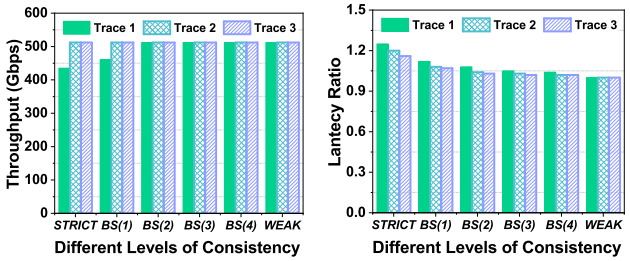
in Fig. 15. Fig. 15(a) demonstrates that as the packet gap within a flow increases, the throughput of both RAPID and the blocking scheme rises, which aligns with our hypothesis. RAPID approaches full throughput progressively. Fig. 15(b) confirms that the processing latency of the two approaches remains stable, and RAPID's performance gradually converges to that of stateless processing.



(a) Throughput. (b) Latency.

Figure 15: Performance on different intra-flow packet gaps.

7.7 Influence of Consistency Level



(a) Throughput. (b) Latency.

Figure 16: Performance on different consistency levels.

Different consistency levels affect packet resubmission rates and packet blocking. To compare RAPID's performance under different consistency levels, we test RAPID on a two-stage stateful function (i.e., 36-cycle state update latency). We run three traces with six different consistency level configurations: *STRICT*, *BS(1)*, *BS(2)*, *BS(3)*, *BS(4)*, and *WEAK*. The state update rate is fixed to 30%. As shown in Fig. 16, as the consistency requirement becomes looser, the throughput of RAPID keeps increasing until it reaches the full throughput, and the latency keeps decreasing. Results on 54-cycle (3-stage) and 72-cycle (4-stage) functions are shown in Appendix G, which lead to a similar conclusion.

8 Discussion

• **Potential Scalability of the Ring**: We investigate the application of the ring within a single pipeline. A ring can also be concurrently leveraged by multiple pipelines. For instance, if one pipeline utilizes only 10% of the Ring's bandwidth for data transfer, it would be feasible for two or more pipelines to share the ring. Similar with MP5 [49], this approach provides a viable design for collaborative task execution across multiple pipelines. Moreover, if the necessity for "cancel_dirty" communication between the Read and Write schedulers can be eliminated, allowing for independent scheduling on both sides, then the unidirectional ring could be evolved into a unidirectional backward path.

• **Priority of Resubmitted Packets**: When handling speculation failures, we currently give priority to new packets. We can also prioritize flows suffering speculation failures to speed up their processing to align with user-specific needs. This is reserved for future exploration.

• **Potential Optimization of RTC**: Compared to pipelines, RTC offers enhanced flexibility in function support. Moreover, RTC's multi-threading can leverage speculative execution too. Instead of stalling threads during state access conflicts, packets are reprocessed if errors are detected.

9 Conclusion

RAPID makes it possible to support advanced stateful packet processing functions on a pipeline-based programmable dataplane. A side ring is used to support cross-stage state write-back and speculative execution. The non-blocking speculative execution reduces the packet buffer requirements necessary for traditional stateful support, leading to diminished back-pressure and fewer congestion in the network. The application programming is enabled by a simple extension to the P4 language. When it comes to hardware, RAPID is not only efficient but also excels in performance across real-world use cases. As a result, tasks that traditionally require ASIC-CPU collaboration can now be executed by RAPID solely, saving the system cost and improving the system performance.

Acknowledgement. We thank the anonymous reviewers and our shepherd Nofel Yaseen for their insightful comments and suggestions which help improve this paper. The authors from Tsinghua University are supported by NSFC (62032013, 62272258) and NSFC-RGC (62061160489). Corresponding author: Bin Liu (lmyujie@gmail.com).

References

[1] ARBOR NETWORKS APS Data Sheet. https://www.netscout.com/sites/default/files/2018-04/DS_APS_EN.pdf.

[2] Broadcom Trident. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series.

[3] CAIDA Equinix Trace. https://catalog.caida.org/dataset/passive_2019_pcap.

[4] CIC-IOT Trace. http://205.174.165.80/IOTDataset/CIC_IOT_Dataset2022/CICIOT/5-Active.

[5] CTU2 Trace. https://mcfp.felk.cvut.cz/publicDatasets/CTU-Mixed-Capture-2.

[6] CTU4 Trace. https://mcfp.felk.cvut.cz/publicDatasets/CTU-Malware-Capture-Botnet-4.

[7] Intel Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[8] MAC Trace. https://download.netresec.com/pcap/maccdc-2012.

[9] MAWI22 Trace. http://mawi.wide.ad.jp/mawi/samplepoint-F/2022.

[10] NCSU FreePDK45. https://eda.ncsu.edu/freepdk/freepdk45/.

[11] NPL Specification. https://nplang.org/specifications.

[12] NSFOCUS Anti-DDoS System Datasheet. https://nsfocusglobal.com/wp-content/uploads/2018/05/Anti-DDoS-Solution.pdf.

[13] Simple Web Trace. https://www.simpleweb.org/wiki/index.php/Traces.

[14] Software ASIC Emulator. https://github.com/jijinfanhua/RPISA-sw.

[15] Synopsys Design Compiler. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[16] UNIV Trace. https://pages.cs.wisc.edu/~tbenson/IMC_DATA.

[17] Xilinx Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html.

[18] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues. In *NSDI*, pages 59–76, 2020.

[19] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostic, Gerald Q Maguire Jr, Panagiotis Papadimitratos, and Marco Chiesa. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency. In *NSDI*, pages 667–683, 2020.

[20] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 313–323. IEEE, 2018.

[21] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: Programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review*, 44(2):44–51, 2014.

[22] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[23] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[24] Carmelo Cascone, Roberto Bifulco, Salvatore Pontarelli, and Antonio Capone. Relaxing state-access constraints in stateful programmable data planes. *ACM SIGCOMM Computer Communication Review*, 48(1):3–9, 2018.

[25] Xiaoqi Chen, Andrew Johnson, Mengying Pan, and David Walker. Synthesizing state machines for data planes. In *Proceedings of the Symposium on SDN Research*, pages 81–88, 2022.

[26] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, et al. dRMT: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 1–14, 2017.

[27] Seyed K Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic ddos defense. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 817–832, 2015.

[28] Yong Feng, Zhikang Chen, Haoyu Song, Wenquan Xu, Jiahao Li, Zijian Zhang, Tong Yun, Ying Wan, and Bin Liu. Enabling In-situ Programmability in Network Data Plane: From Architecture to Language. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.

[29] Nadeen Gebara, Alberto Lerner, Mingran Yang, Minlan Yu, Paolo Costa, and Manya Ghobadi. Challenging the stateless quo of programmable switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 153–159, 2020.

[30] Mohamed G Gouda and Alex X Liu. A model of stateful firewalls and its properties. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 128–137. IEEE, 2005.

[31] Garegin Grigoryan and Yaoqing Liu. Lamp: Prompt layer 7 attack mitigation with programmable data planes. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, pages 158–159, 2018.

[32] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-wide heavy hitter detection with commodity switches. In *Proceedings of the Symposium on SDN Research*, pages 1–7, 2018.

[33] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *OSDI*, pages 239–256, 2021.

[34] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.

[35] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 90–106, 2020.

[36] Guanyu Li, Menghao Zhang, Cheng Guo, Han Bao, Mingwei Xu, Hongxin Hu, and Fenghua Li. Imap: Fast and scalable in-network scanning with programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 667–681, 2022.

[37] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E Stephens, Hassan Wassel, and Aditya Akella. Ringleader: Efficiently offloading intra-server orchestration to nics. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1293–1308, 2023.

[38] Jiaxin Lin, Kiran Patel, Brent E Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A high-performance programmable NIC for multi-tenant networks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 243–259, 2020.

[39] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches. In *USENIX Security Symposium*, pages 3829–3846, 2021.

[40] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[41] Masoud Moshref, Apoorv Bhargava, Adhip Gupta, Minlan Yu, and Ramesh Govindan. Flow-level state transition as a new switch primitive for SDN. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 61–66, 2014.

[42] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al. Flowblaze: Stateful packet processing in hardware. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019*, pages 531–547. USENIX ASSOC, 2019.

[43] Elham Safi, Andreas Moshovos, and Andreas Veneris. Two-stage, pipelined register renaming. *IEEE transactions on very large scale integration (VLSI) systems*, 19(10):1926–1931, 2010.

[44] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostic, and Marco Chiesa. A High-Speed Stateful Packet Processing Approach for Tbps Programmable Switches. In *Proceedings of the 20th USENIX Symposium on*

Networked Systems Design and Implementation (NSDI 23), 2023.

[45] Naveen Kr Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 1–16, 2018.

[46] Naveen Kr Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable Calendar Queues for High-speed Packet Scheduling. In *NSDI*, pages 685–699, 2020.

[47] Vishal Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 367–379. 2019.

[48] Vishal Shrivastav. Programmable multi-dimensional table filters for line rate network functions. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 649–662, 2022.

[49] Vishal Shrivastav. Stateful multi-pipelined programmable switches. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 663–676, 2022.

[50] Manuel Simon, Henning Stubbe, Dominik Scholz, Sebastian Gallenmüller, and Georg Carle. High-Performance Match-Action Table Updates from within Programmable Software Data Planes. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, pages 102–108, 2021.

[51] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 15–28, 2016.

[52] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 164–176, 2017.

[53] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 731–747, 2021.

[54] Chen Sun, Jun Bi, Haoxian Chen, Hongxin Hu, Zhilong Zheng, Shuyong Zhu, and Chenghui Wu. SDPA: Toward a stateful data plane in software-defined networking. *IEEE/ACM Transactions on Networking*, 25(6):3294–3308, 2017.

[55] Angelo Tulumello, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, and Giuseppe Bianchi. Pushing services to the edge using a stateful programmable dataplane. In *2019 European Conference on Networks and Communications (EuCNC)*, pages 389–393. IEEE, 2019.

[56] Belma Turkovic, Jorik Oostenbrink, and Fernando Kuipers. Detecting heavy hitters in the data-plane. *arXiv preprint arXiv:1902.06993*, 2019.

[57] Ying Wan, Haoyu Song, Hao Che, Yang Xu, Yi Wang, Chuwen Zhang, Zhijun Wang, Tian Pan, Hao Li, Hong Jiang, et al. FastUp: Fast TCAM Update for SDN Switches in Datacenter Networks. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 887–897. IEEE, 2021.

[58] Zhijun Wang, Hao Che, Mohan Kumar, and Sajal K Das. CoPTUA: Consistent policy table update algorithm for TCAM without locking. *IEEE Transactions on Computers*, 53(12):1602–1614, 2004.

[59] Jiarong Xing, Wenqing Wu, and Ang Chen. Ripple: A programmable, decentralized link-flooding defense against adaptive adversaries. In *USENIX Security*, 2021.

[60] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. Using trio: juniper networks' programmable chipset-for emerging in-network applications. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 633–648, 2022.

[61] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 179–193, 2021.

[62] Lior Zeno, Dan RK Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. SwiSh: Distributed Shared State Abstractions for Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 171–191, 2022.

[63] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *the 27th Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

Traces	# of pkts (K)	average pkt size (B)	# of flows, average # of pkts per flow, and conflict ratio					
			five-tuple	avg.	CR	sIP-dIP	avg.	CR
CTU4 [6]	24392	93	1422109	17.2	68.81%	8655	2818.3	84.27%
MACCDC [8]	6598	142	73071	90.3	8.76%	9594	687.8	43.38%
Equinix [3]	31540	482	1085931	29	1.47%	870685	36.2	1.49%
CIC-IOT [4]	1836	484	78799	23.3	46.46%	3774	486.6	56.81%
MAWI22 [9]	121308	606	13607494	8.9	9.53%	9627481	12.6	11.18%
ISP DC	33514	642	1199269	27.9	11.16%	873011	38.4	20.79%
UNIV1 [16]	912	673	10945	83.4	16.66%	2719	335.8	17.98%
CTU2 [5]	685	750	24274	28.2	27.21%	2312	296.6	29.51%
UNIV2 [16]	11772	780	33987	346.4	23.27%	19006	619.4	23.30%
Simple Web [13]	2636	836	51194	51.5	26.08%	4917	536.2	27.34%

Table 8: Statistics of the packet traces.

A Trace Statistics

Table 8 lists the statistics of the 10 traffic traces. Specifically, the ISP DC trace is collected from the data center of an ISP.

B Hash collision in *dTable*

dTable records the hash values of flows. In the case of hash collision, a flow might be mistakenly blocked and its packets resubmitted, but this will not cause catastrophic consequences. The victim flow only experiences an increased latency.

Given that a flow is hashed to a M-bit hash value, the probability of hash collision among this flow and another incoming flow is $P = \frac{1}{2^M}$. Assuming M=64, the collision probability is 5.42×10^{-20}, which is low enough to make its impact negligible to the overall performance.

C Writeback Rate Calculation

Assuming that within T clock cycles, a flow has n back-to-back packets arriving, and the writeback rate is α. We investigate the performance similarity point between the blocking scheme and RAPID by considering the maximum scheduling latency experienced by the last packet in both approaches. Set the latency of the blocking scheme is $t_b(T,n,\alpha)$ and RAPID is $t_R(T,n,\alpha)$.

Based on the properties of the Blocking scheme, it is evident that its performance remains stable across varying writeback rates, i.e.,

$$t_b(T,n,\alpha) = (n-1)T \tag{1}$$

Suppose $\alpha = \frac{1}{n}$, i.e., only one out of n packets updates states, the latency of the last packet being scheduled is: the cycles it will pass through (T) and the cycles it waits to be scheduled (n). So the time is

$$t_R(T,n,\frac{1}{n}) = n+T \tag{2}$$

When $\alpha > \frac{1}{n}$, we assume that the first packet updates states and the later packets update state evenly. We can get the latency in terms of α:

$$t_R(T,n,\alpha) = \sum_{n\alpha}^{i=1}(\frac{i}{n\alpha}n+T) = n\alpha T + \frac{n\alpha+1}{2}n \tag{3}$$

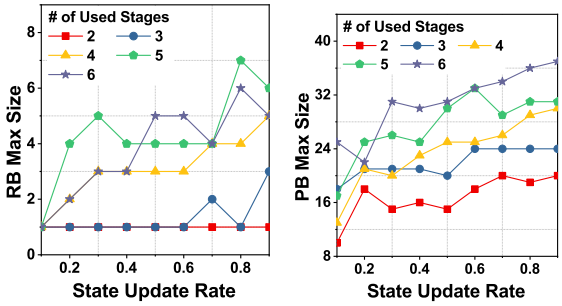
Combining Eq. 1 and Eq. 3, we can get the α in which two schemes perform similarly:

$$\alpha = \frac{2T-3}{2T+n} \tag{4}$$

When $n = 60$ and $T = 60$, packets will experience similar latency for both RAPID and the blocking scheme if the writeback rate is 65%.

D Parameter Setting

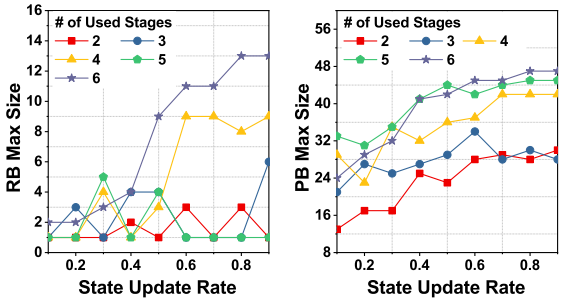
Fig. 17 and Fig. 18 show the scheduler parameters under ISP DC trace and MAWI22 trace, respectively. Based on these two traces, we can draw similar conclusions as in Sec. 7.3.



(a) RB. (b) PB.

(c) dTable. (d) Ring Node Buffer.

Figure 17: Scheduler Parameters under ISP DC trace.



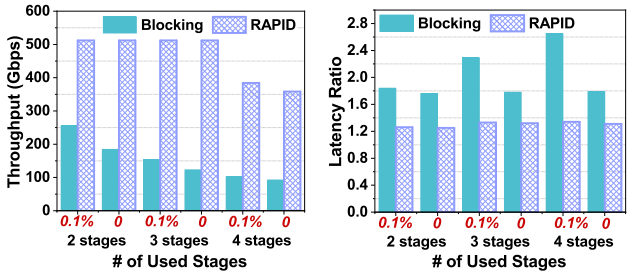
(a) RB. (b) PB.

(c) dTable. (d) Ring Node Buffer.

Figure 18: Scheduler Parameters under MAWI22 trace.

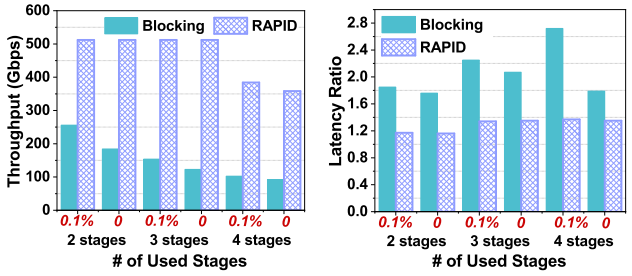
E Different used stages on Trace 1 and 3

The system throughput and latency performance in terms of the number of function stages under Trace 1 and Trace 3 can be seen in Fig. 19 and Fig. 20 respectively.



(a) Throughput. (b) Latency.

Figure 19: Throughput and latency with different consistency levels under Trace 1.

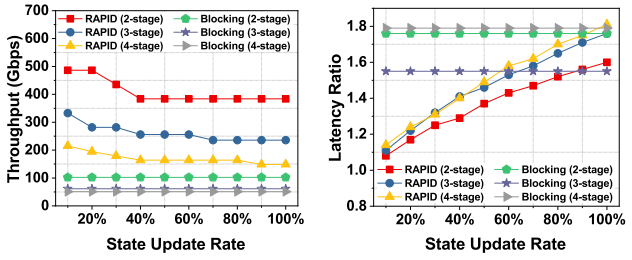


(a) Throughput. (b) Latency.

Figure 20: Throughput and latency with different consistency levels under Trace 3.

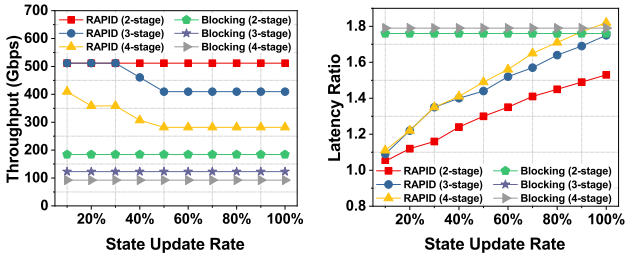
F Different State Update Rates on Trace 1 and Trace 3

The system throughput and latency performance in terms of different state update rates under Trace 1 and Trace 3 can be seen in Fig. 21 and Fig. 22 respectively.



(a) Throughput. (b) Latency.

Figure 21: Performance on different state update rates of Trace 1.

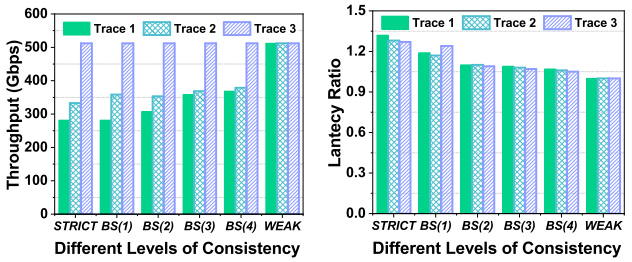


(a) Throughput. (b) Latency.

Figure 22: Performance on different state update rates of Trace 3.

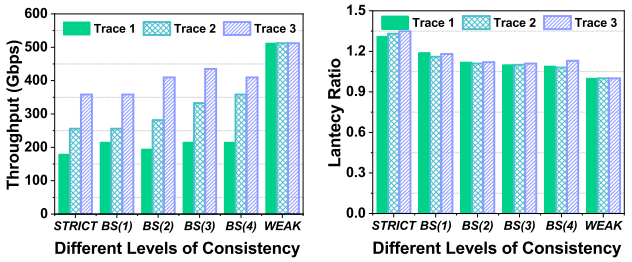
G Influence of Consistency Level under 54-cycle and 72-cycle

The system throughput and latency performance in terms of different consistency levels under 54-cycle and 72-cycle functions can be seen in Fig. 23 and Fig. 24.



(a) Throughput. (b) Latency.

Figure 23: Performance on different consistency level under 54-cycle functions.



(a) Throughput. (b) Latency.

Figure 24: Performance on different consistency level under 72-cycle functions.

H Simple Example with *muTable*

Fig. 25 shows a simple code example (port knocking [21]) written in the enhanced P4 language. The flow state table *port_knocking* stores the current state. The state can be read out with the stateful primitive `read`. Then with another stateless flow table, the packet can get the next state of the flow and write it back to the *port_knocking* state table.

```
/* define stateful table */
muTable port_knocking {
    keys = {
        hdr.ipv4.src_addr;
        hdr.ipv4.dst_addr;
    }

    values = {
        bit<8> state;
    }

    type = exact;
    consistency = STRICT;
    size = 4096;
}

table port_FSM {
    keys = {
        meta.cur_state;
        hdr.ipv4.dst_port;
    };

    actions = {
        get_new_state; // modify meta.new_state
    };
}

/* read out the cur_state */
meta.cur_state = port_knocking.read(hdr);
/* get new_state to look up table */
portFSM.apply();
/* write back the new_state */
port_knocking.write(hdr, meta.new_state);
```

Figure 25: A simple port knocking example with *muTable*.

GRACE: Loss-Resilient Real-Time Video through Neural Codecs

Yihua Cheng[1], Ziyi Zhang[1], Hanchen Li[1], Anton Arapin[1], Yue Zhang[1], Qizheng Zhang[2], Yuhan Liu[1], Kuntai Du[1], Xu Zhang[1], Francis Y. Yan[3], Amrita Mazumdar[4], Nick Feamster[1], Junchen Jiang[1]
[1]*The University of Chicago,* [2]*Stanford University,* [3]*Microsoft,* [4]*NVIDIA*

Abstract

In real-time video communication, retransmitting lost packets over high-latency networks is not viable due to strict latency requirements. To counter packet losses without retransmission, two primary strategies are employed—encoder-based forward error correction (FEC) and decoder-based error concealment. The former encodes data with redundancy before transmission, yet determining the optimal redundancy level in advance proves challenging. The latter reconstructs video from partially received frames, but dividing a frame into independently coded partitions inherently compromises compression efficiency, and the lost information cannot be effectively recovered by the decoder without adapting the encoder.

We present a loss-resilient real-time video system called GRACE, which preserves the user's quality of experience (QoE) across a wide range of packet losses through a new neural video codec. Central to GRACE's enhanced loss resilience is its *joint training of the neural encoder and decoder under a spectrum of simulated packet losses*. In lossless scenarios, GRACE achieves video quality on par with conventional codecs (e.g., H.265). As the loss rate escalates, GRACE exhibits a more graceful, less pronounced decline in quality, consistently outperforming other loss-resilient schemes. Through extensive evaluation on various videos and real network traces, we demonstrate that GRACE reduces undecodable frames by 95% and stall duration by 90% compared with FEC, while markedly boosting video quality over error concealment methods. In a user study with 240 crowdsourced participants and 960 subjective ratings, GRACE registers a 38% higher mean opinion score (MOS) than other baselines. We make the source codes and models of GRACE public at https://uchi-jcl.github.io/grace.html.

1 Introduction

Real-time video communication has become an integral part of our daily lives [29], spanning online conferences [3, 20], cloud gaming [11, 17], interactive virtual reality [6, 18], and IoT applications [16, 19]. To ensure a high quality of experience (QoE) for users, real-time video applications must protect against packet losses[1]. However, retransmitting lost packets across high-latency networks is not feasible due to stringent real-time latency requirements [57].

[1]In this study, we use the term "packet loss" to refer to both packets dropped in transit and those not received by the decoding deadline. Under this definition, a video frame could experience a high packet loss rate (e.g., 50%) even if the actual network loss rate is low [86].

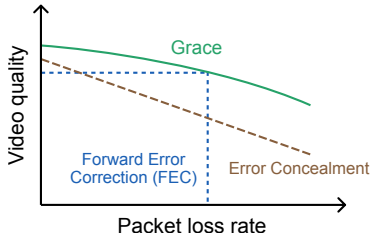


Figure 1: *Illustration of the video quality achieved by different loss-resilient schemes, operating under the same bandwidth budget, across varying packet loss rates. Actual experimental results are shown in Figure 8.*

Loss-resilient techniques generally fall into two categories. First is encoder-side forward error correction (FEC), such as Reed-Solomon codes [100], fountain codes [76, 77], and more recently, streaming codes [28, 86]. FEC incorporates redundancy into data prior to transmission. With a redundancy rate of *R%*—the percentage of redundant data relative to the total data size—up to *R%* of lost data can be recovered. Beyond that, the video becomes undecodable, rendering a sharp collapse in video quality (Figure 1). Increasing *R* protects against higher losses but also entails a higher bandwidth overhead, which in turn reduces video quality. Thus, determining the optimal *R* in advance poses a practical challenge.

The second category is decoder-side error concealment, which reconstructs portions of a video frame affected by packet losses, through handcrafted heuristics [63, 97, 115] or neural networks [59, 67, 79, 87, 102]. Nevertheless, implementing error concealment requires partitioning a video frame into independently decodable units (e.g., slices [99] or tiles [64]) first, thus reducing compression efficiency. Moreover, since the encoder is not optimized for loss resilience, the lost information cannot be effectively recovered by the decoder alone. As a result, the video quality tends to deteriorate rapidly with increasing packet loss, as illustrated in Figure 1.

In this paper, we present GRACE, a loss-resilient real-time video system designed to maintain the user's quality of experience (QoE) across a wide range of packet losses. Our key insight is that *jointly optimizing the encoder and decoder under a spectrum of simulated packet losses* considerably strengthens loss resilience. To facilitate this joint optimization, GRACE strategically employs a neural video codec (NVC) [73], integrating neural networks into the core components of a conventional video encoder and decoder. In contrast to FEC, GRACE's NVC is trained to handle *diverse* packet losses, eliminating the need to predict a loss rate beforehand and preventing the undecodable video under exceedingly high

losses. Unlike decoder-side error concealment, GRACE *jointly* trains the encoder and decoder, so that the encoder learns to properly distribute each pixel's information across multiple output elements in anticipation of losses, facilitating the decoder's frame reconstruction when packets are actually lost. Consequently, GRACE displays a more graceful quality degradation amid varying losses, while consistently delivering higher video quality than previous solutions (Figure 1).

To materialize the above benefits of GRACE's codec, our design of GRACE addresses three system challenges.

First, to ensure loss tolerance, each packet must be independently decodable. Existing solutions achieve this by dividing the frame into independently decodable units. However, this introduces a size overhead because the data in each unit follows different distributions and thus cannot be compressed efficiently. In response to this challenge, we train GRACE's neural encoder to regularize the values in its output to conform to the same distribution, thereby reducing the partitioning overhead. We also utilize reversible random mapping [8] during such partitioning, making it more amenable to NVCs. While training GRACE under packet losses, simulating random partitioning and packet losses is inefficient and precludes differentiability. Hence, we apply random zeroing to the encoder's output directly, simulating packet losses without actually dropping packets (§3).

Second, packet losses can lead to discrepancies between the reference frames at the encoder and decoder side, resulting in sustained quality degradation in the decoded video stream if synchronization is not maintained. Traditional remedies, such as retransmission or sending a new keyframe, fall short of seamlessly rectifying this inconsistency. GRACE introduces an innovative protocol to adeptly realign the encoder and decoder states without hindering video decoding. In the event of packet loss, the decoder leverages the loss resilience of GRACE to decode partially received packets. Simultaneously, the decoder communicates the loss details to the encoder. This feedback mechanism enables the encoder to swiftly adjust its recent reference frames to match those at the decoder end, eliminating the need for additional data transmission (§4.2).

Third, GRACE must be efficient to encode and decode video in real-time across various devices, from laptops to mobile phones. Existing NVCs, however, often employ expensive neural networks, particularly for motion estimation and post-processing. We show that by downscaling the image input for motion estimation and simplifying post-processing, GRACE accelerates the encoding and decoding by 4× without a noticeable impact on loss resilience. Moreover, with hardware-specific runtimes such as OpenVINO and CoreML, GRACE attains over 25 fps on CPUs and iPhones (§4.3).

Comprehensive experiments (§5) on a diverse set of videos and real network traces show that with the same congestion control logic, GRACE reduces undecodable frames by 95% and stall duration by 90% compared with state-of-the-art FEC baselines. It also boosts the visual quality metric of SSIM by 3 dB over a recent neural error concealment scheme (§5.1). Our IRB-approved user study with 240 crowdsourced participants and a total of 96 subjective ratings demonstrates a 38% higher mean opinion score (MOS) for GRACE, further attesting to its effectiveness. Regarding computational efficiency, GRACE achieves more than 25 fps on popular mobile devices (e.g., iPhone 14 Pro), meeting the real-time requirement.

Our contributions are summarized as follows. *(i)* We present GRACE, which, to the best of our knowledge, represents **the first effort to *jointly* train a neural video encoder and decoder under a spectrum of packet losses, aiming to improve loss resilience in real-time video** (§3). Different from other recent ML-based real-time video systems [107, 108, 114] that use ML-based rate adaptation to minimize packet loss, GRACE uses ML to make the video codec itself resilient to packet loss. *(ii)* We build the end-to-end video system to address the practical challenges associated with integrating a new NVC, developing optimization techniques related to packetization, encoder/decoder state synchronization, and runtime efficiency (§4).

2 Background

2.1 Real-time video coding

To help explain GRACE's design, we first introduce some key concepts in real-time video coding and streaming.

The sender encodes video at a specific frame rate and bitrate, e.g., with 25 fps (frames per second) and 3 Mbps, the encoder generates a 15 KB frame on average every 40 ms. An encoded video is composed of groups of consecutive frames, called a group of pictures (GoP). Each GoP starts with an I-frame (or key-frame), followed by multiple P-frames (or inter-frames)[2]. I-frames are independently encoded without referencing other frames, while P-frames encode only the differences relative to previous reference frames. In real-time video, the majority of frames are P-frames to minimize frame sizes, so our discussion here focuses on P-frames. Figure 2 shows the workflow of P-frame encoding and decoding. Given a new frame and a reference frame, the encoder (1) calculates motion vectors (MVs) and residuals for each macroblock (MB), e.g., 16×16-pixel samples, (2) transforms and quantizes the MVs and residuals, (3) performs entropy encoding on the transformed data, (4) divides the entropy-encoded data into packets, and (5) transmits these packets with congestion control, such as GCC [31]. Correspondingly, the receiver depacketizes and decodes the received data to reconstruct each frame from the received packets.

To reduce frame delay, which denotes the time from frame encoding to rendering, real-time video clients (e.g., WebRTC) commonly make two choices that differentiate them from video streaming (e.g., Netflix, ESPN Live):

[2]On-demand video also uses B-frames (bidirectional predicted frames), which refer to both past and future P-frames. However, real-time video rarely uses B-frames in order to render frames as soon as possible.

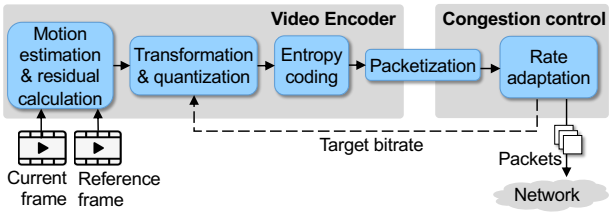


Figure 2: *A typical workflow of video frame encoding.*

- Real-time video employs notably *shorter* (tens of ms) buffers, as opposed to video streaming that utilizes several seconds of playback buffer for on-demand or live content. Thus, retransmission delay is difficult to conceal with such short buffers, especially in high-latency networks.
- To maintain a short buffer, real-time video sends each frame in a burst and decodes it as soon as its packets are received. As a result, *any* lost packets, whether due to drops or queuing, can affect frame decoding. In contrast, streaming video is transmitted in chunks (each with hundreds of frames) over the HTTP/TCP protocol.

Ideally, congestion control and bitrate adaptation (e.g., GCC [31], Salsify [45], and NADA [116]) are designed to handle bandwidth fluctuations, thereby avoiding congestion-induced packet losses. However, predicting bandwidth fluctuations in advance is challenging, making loss-resilient methods necessary when decoding frames under packet loss.

We define *packet loss per frame* as any packets not received before the receiver is expected to decode the frame. In other words, even if a packet is not dropped, it can still be counted as packet loss if it arrives too late. It is important to note that our notion of packet loss differs from network-level loss. Even with low network loss (which typically remains below 1%), real-time video may still encounter a high packet loss rate (e.g., over 50%) in certain frames, as corroborated by recent research in this space [35, 36, 86].

2.2 Related work

Various loss-resilient schemes have been studied.

Forward error coding (FEC) adds redundancy at the *encoder* before the data is sent to the network. This is also known as error-resilient channel coding. Examples include Reed-Solomon codes, LDPC [77], fountain and rateless codes [33, 76], streaming codes [28, 86], and recent ones based on DNNs [32, 50]. There are also hierarchical and multilevel FEC [94, 95], which organizes FEC into multiple layers and protects each layer with different redundancies. FEC is also used to protect frame metadata or the base layer in SVC (also known as UEP [25, 113]). However, in order to pick a suitable rate of redundancy, they need to estimate how many packets will be lost in advance. If the loss rate is underestimated, the redundancy will be insufficient to recover missing packets. On the other hand, adding excessive redundancy results in a higher bandwidth overhead and, in turn, a lower video quality.

Postprocessing error concealment reconstructs missing data in lost packets at the *decoder*. These methods generally consist of two components. First, the encoded packets should be decodable when only a subset of the packets is received. This is accomplished through INTRA-mode macroblock encoding [38], slice interleaving [56], or flexible macroblock ordering [66]. However, these approaches often compromise the encoder's ability to exploit redundancies across neighboring MBs, as adjacent MBs are either encoded in INTRA mode or split into different packets (in a checkerboard manner [64, 66] or based on ROI detection [93]). Therefore, these methods impair compression efficiency, causing the encoded frame size to inflate by 10%–50% [42, 64, 74, 99].

Then, the decoder reconstructs lost data based on the received packets, using classic heuristics (e.g., motion vectors interpolation [63, 97, 115] and intra-block refreshing [64] in H.264) or neural-network-based inpainting [59, 67, 79, 87, 102]. Recent work [67] use vision transformers [27, 43] to directly predict the missing bits in the lost packets before frame decoding. However, due to the encoder's lack of awareness of the decoder's postprocessing, each encoded packet contains limited redundancy and information that could aid in reconstructing missing motion vectors or residuals. As a result, the reconstruction process is forced to guess the missing data when a packet is lost. Even recent techniques still see a notable drop in video quality (e.g., PSNR drops from 38 dB to 25 dB at a 20% packet loss [81]).

GRACE takes a different approach than FEC and error concealment. Unlike error concealment that relies only on decoder-side postprocessing, GRACE jointly optimizes (via training) both the (neural) encoder and decoder. Unlike FEC that requires a pre-determined redundancy rate, GRACE's codec is optimized across a range of packet loss rates.

Other schemes: While there exist other techniques that might help mitigate the impact of packet loss, their primary goals are *not* loss resilience. Nevertheless, for the sake of completeness, we discuss some notable schemes here and also quantitatively compare GRACE against several of them in §5.

Scalable video coding (SVC) [40, 88, 89] and fine-granularity scalability (FGS) [68, 75] aim to optimize rate-distortion (RD) tradeoff—video quality achieved by a single encoded bitstream under different received bitrates. SVC encodes a video in multiple quality layers and sends data *layer by layer*. This is feasible for on-demand video [40, 72] but in real-time video, all packets of a frame are sent together to reduce frame delay (§2.1). When a packet loss occurs to a base layer, it will block the decoding of any higher layers. For this reason, SVC is rarely used to improve unicast real-time video (though it is used in multicast video to serve users with heterogeneous network capacities [88]).

There are a few alternatives to postprocessing error concealment. For instance, when loss occurs, Salsify [45] reverts to an older but reliably received frame—instead of the last frame—as the reference frame, so the decoder can safely skip a loss-affected frame without hurting subsequent frames. However, it needs more bits to encode the same quality than using the

last frame as the reference frame, e.g., the P-frames between every other frame are 40% greater in size than between two consecutive frames. Similar limitations apply to long-term reference frame (LTR) [112], which makes each P-frame individually decodable if the long-term reference is received, regardless of packet loss in between or not. Voxel [83] skips a loss-affected frame if the encoder indicates that skipping the frame does not affect video quality. It works well for on-demand video where B-frames can be safely skipped, and the impact of a skipped frame will stop at the next chunk within a few seconds. Unfortunately, neither applies to real-time video.

Recently, deep learning has been used in super resolution [61, 92, 106], SVC [40, 75], and postprocessing error concealment based on CNNs [59, 79, 87, 102] or transformers [44, 67]. Super-resolution can reduce packet losses by sending the video in a lower bitrate and enhancing the video quality on the receiver side. However, it still requires retransmissions to rectify frames impaired by packet loss. For SVC and postprocessing error concealment techniques, the aforementioned limitations inherent to these approaches remain, despite the use of deep learning.

Loss resilience has also been studied under specific assumptions, such as availability of multi-path [41, 80], early retransmission driven by router feedback [117], low-latency networks [85], and availability of video gaming states [52, 53, 101]. We do not make special assumptions in this work.

2.3 Neural video codec background

Our work is based on neural video codecs (NVCs), which use learned neural networks (NNs), instead of handcrafted logic, to encode and decode video frames [40, 55, 73]. Recent NVCs have demonstrated comparable or even better compression efficiency than traditional video codecs for two reasons:

- They leverage logical components commonly found in traditional video codecs, such as motion estimation, warping, and transformative compression (§2.1), replacing their handcrafted heuristics with NNs, which can learn more sophisticated algorithms from data.
- These NVCs exhibit remarkable generalization across a variety of video content because of training on a large corpus of videos (e.g., Vimeo-90K [103]). This capability to generalize is also observed in our evaluation (§5).

Despite their exceptional compression efficiency, NVCs have received little attention so far in the context of loss resilience. However, we believe NVCs have the potential to achieve greater loss resilience for the following reasons.

- First, unlike traditional codecs that map each pixel (or macroblock) to a distinct motion vector/residual, the highly parameterized NN of NVC's encoder can be trained to map the information of each pixel to multiple elements in output tensor, potentially making lost information recoverable.
- Second, the NVC's decoder, comprising convolutional NNs, can be trained to decode not only a direct encoder output but also tensors that resemble those with perturbations such as random noise or zeroing. In contrast, traditional codecs might fail to decode under similar circumstances.

Nevertheless, NVCs as is still lack tolerance to packet loss. Their standard training implicitly assumes that the encoder's output is identical to the decoder's input, so it does not prepare the NVC to handle data loss between the encoder and decoder. Meanwhile, entropy encoding used in conventional NVCs compresses the entire encoder output as a single bitstream, and thus any packet loss will render it undecodeable.

GRACE is an attempt at transforming NVCs to be resilient to different packet loss rates. Our work is related to an emerging line of work on deep joint source-channel coding [30, 37, 65], which trains an NVC to encode images in a representation robust to signal noises. GRACE differs with them on two key fronts. First, GRACE handles video frames, which cannot be treated separately as individual images because any error in one frame can propagate to future frames. Second, GRACE handles packet losses rather than physical-layer signal noises, which can be naturally modeled by differentiable linear transformations [30, 51].

In short, traditional error-resilient methods struggle to maintain video quality across a range of packet losses. Encoder-based forward error coding (FEC) optimizes quality only for a pre-determined maximum loss rate, whereas decoder-based postprocessing error concealment suffers from suboptimal quality especially at high loss rates. On the other hand, existing NVCs have the potential to tolerate data perturbations but are not explicitly designed to handle packet losses.

3 Training GRACE's neural video codec

This section outlines the training process of GRACE's neural video codec (NVC). At a high level, GRACE *jointly trains the neural encoder and decoder under a range of packet losses* to achieve enhanced loss resilience.

Basic NVC framework: Figure 3 depicts the workflow of GRACE's encoder and decoder (excluding entropy coding and packetization). The encoder follows a similar logical process as a traditional video encoder (Figure 2). It first employs a neural network (NN) to estimate motion vectors (MVs) and encodes them into a quantized tensor using an NN-based MV encoder. Subsequently, the tensor is decoded back into MVs to match those received by the decoder. Next, the encoder applies these MVs to the reference frame to generate a motion-compensated frame, and uses a frame smoothing NN to increase its similarity with the current frame before calculating the residual differences between them. Finally, an NN-based residual encoder encodes the residuals into another quantized tensor. When the encoded MV tensor and the encoded residual tensors are received by the decoder, they go through the NN-based MV decoder and residual decoder jointly trained with their respective encoders. Appendix A.1 provides more details of the tensors and NNs.

Although both the encoder and decoder of GRACE contain

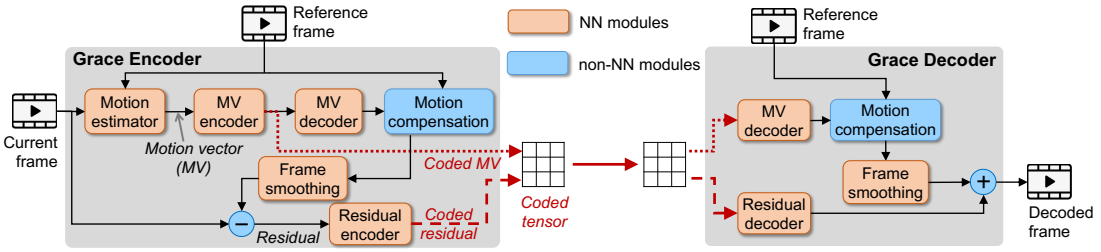


Figure 3: *Workflow of* GRACE*'s neural video codec.*

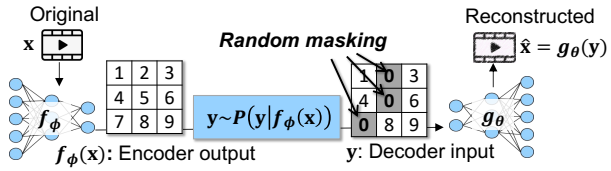


Figure 4: *Unlike traditional NVC training that assumes no data loss between the encoder and decoder,* GRACE *applies "random masking"—setting a fraction of randomly selected elements to zeros—to the encoder's output.*

multiple steps, they can be viewed as two differentiable models. We denote the encoder by f_ϕ (with its NN weights ϕ) and the decoder by g_θ (with its NN weights θ). The encoder encodes a frame $\mathbf{x}$ into a *coded tensor* $\mathbf{y} = f_\phi(\mathbf{x})$, and the decoder decodes $\mathbf{y}$ into a reconstructed frame $\hat{\mathbf{x}} = g_\theta(\mathbf{y})$. Traditionally, NVC seeks to minimize the following loss function:

$$\mathbb{E}_{\mathbf{x}}[\underbrace{D(g_\theta(\mathbf{y}),\mathbf{x})}_{\text{Pixel error}} + \alpha\cdot \underbrace{S(f_\phi(\mathbf{x}))}_{\text{Encoded size}}], \text{where } \underbrace{\mathbf{y} = f_\phi(\mathbf{x})}_{\text{No data loss}} \quad (1)$$

Here, $D(\hat{\mathbf{x}},\mathbf{x})$ is the pixel-level reconstruction error of the decoded frame $\hat{\mathbf{x}}$ (by default, L2-norm[3] of $\hat{\mathbf{x}}-\mathbf{x}$), and $S(\mathbf{y})$ is the entropy-coded data size of $\mathbf{y}$ in bit-per-pixel (BPP). The parameter α governs the size-quality tradeoff: a higher α leads to a smaller frame size, $S(x)$, but higher distortion (i.e., poorer quality) of the reconstructed frame $\hat{\mathbf{x}}$. As all the functions—f_ϕ, g_θ, D, and S (approximated by a pre-trained NN [73])—are differentiable, the NN weights ϕ and θ can be trained jointly via gradient descent to minimize Eq. 1.

Simulating packet loss during training: We begin by pre-training an NVC using Eq. 1, which we refer to as GRACE-P, and then fine-tune it by introducing simulated packet losses in the following manner. GRACE simulates the impact of packet losses by randomly "masking"—zeroing selected elements—in the encoder's output, $f_\phi(\mathbf{x})$, as shown in Figure 4. The fraction of zeroed elements is dictated by a distribution, $P(\mathbf{y}|f_\phi(\mathbf{x}))$, which represents the probability distribution of the resulting tensor $\mathbf{y}$ after random masking $f_\phi(\mathbf{x})$. For instance, with a 33% loss rate, $P(\mathbf{y}|f_\phi(\mathbf{x}))$ is the probability of $\mathbf{y}$ arising from the random masking of 33% of elements in $f_\phi(\mathbf{x})$, as illustrated in Figure 4. Formally, GRACE jointly

[3]Note that the L2-norm (or mean squared error) of $\hat{\mathbf{x}}$ and $\mathbf{x}$ is closely related to the PSNR of $\hat{\mathbf{x}}$. To avoid this bias, our evaluation in §5 measures the quality improvement in SSIM and subjective user studies.

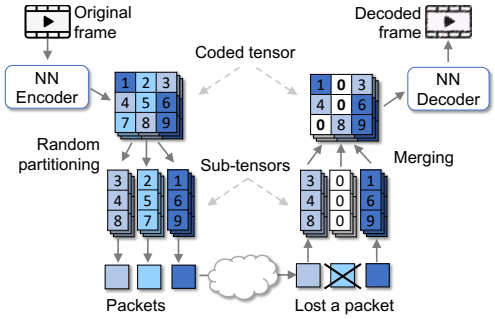


Figure 5: GRACE*'s reversible randomized packetization. The tensor elements mapped to a lost packet will be set to zeros.*

trains the encoder and decoder NNs to minimize:

$$\mathbb{E}_{\mathbf{x}}[D(g_\theta(\mathbf{y}),\mathbf{x}) + \alpha\cdot S(f_\phi(\mathbf{x}))], \text{where } \underbrace{\mathbf{y} \sim P(\mathbf{y}|f_\phi(\mathbf{x}))}_{\text{Simulate packet loss}} \quad (2)$$

The key difference from the traditional objective in Eq. 1 is the distribution function P (highlighted in blue), which captures the distribution of decoder input under packet loss.

To train the weights of ϕ and θ under the random perturbations of P, we employ the REINFORCE trick [62] (commonly used in reinforcement learning [84, 109]) to approximate the gradient through Monte Carlo sampling. A more detailed mathematical formulation is included in Appendix A.2.

Choosing simulated packet loss rates: To prepare GRACE's NVC to handle a wide range of loss rates, it is essential to simulate such losses in training. One approach is to select loss rates uniformly at random from [0, 100%) and apply them to the encoder output $f_\phi(\mathbf{x})$. However, the resulting NVC turns out to perform poorly, especially when dealing with low loss rates. Notably, even when high loss rates (e.g., over 80%) are introduced only in a small fraction of training samples, we empirically observe a significant drop in video quality under low loss rates while the quality improvement under high loss rates is only marginal. This phenomenon could be attributed to the encoder's tendency to incorporate more redundant information to prepare for high loss rates, adversely affecting video quality at low loss rates. Therefore, a practically effective distribution should cover both low and high loss rates, with a slight emphasis on low losses. Our final choice of loss rate distribution is described in §4.4.

Packetization during inferencing: Recall that during training, we simulate packet loss by applying random masking

rather than replicating the actual packetization and packet dropping process. Therefore, it is important to ensure that the impact of actual packet loss during runtime mirrors the effects of random masking. To achieve this, GRACE employs reversible randomized packetization as shown in Figure 5. GRACE's sender first splits the encoded tensor of a frame (both the encoded MVs and encoded residual) into multiple subtensors using a uniform random mapping. We use a reversible pseudo-random function to generate the mapping so that the receiver can correctly recover the original tensor with the same random seed. Specifically, we map the ith element to the $j = (i \cdot p \mod n)^{\text{th}}$ packet at the $\lceil (i \cdot p - j)/n \rceil^{\text{th}}$ position, where n is the number of packets and p is a prime number. If a packet is lost, the decoder assigns zero to each element whose position is mapped to the lost packet. Consequently, an $x\%$ packet loss rate has the effect of randomly zeroing $x\%$ of the values in the encoder's output tensor. [4] §4.1 explains how each subtensor is losslessly compressed via entropy encoding into the bitstream of a packet, but this lossless compression is bypassed during training for efficiency purposes.

Why is GRACE more loss-resilient? Unlike decoder-side error concealment, the joint training ensures that the *encoder* is also aware of packet losses. Empirically, we observe that GRACE's encoder tends to produce more non-zero values in its output than an NVC pre-trained on the same dataset but without simulated packet loss. This increase in non-zero values can be viewed as more "redundancy," as the encoder embeds each pixel's information into multiple elements in its output tensor, assisting the decoder in discerning loss-affected elements (from intended zeros) and reconstructing video better under packet losses. §5.4 empirically shows that training only the decoder with simulated loss cannot reach the same level of loss resilience (Figures 20 and 29).

4 Real-time video framework

With the training techniques detailed in §3, GRACE's NVC acquires the ability to withstand simulated packet losses. This section describes the integration of this NVC into a real-time video delivery framework: GRACE entropy-encodes the neural encoder's output into packets (§4.1), streams frames under packet loss (§4.2), and accelerates encoding and decoding across various devices (§4.3).

4.1 Entropy encoding the encoder's output

As mentioned in §3, GRACE splits the encoder's output into subtensors using a reversible-random function, with each subtensor corresponding to an individual packet. Similar to classic codecs such as H.265 and VP9, each subtensor undergoes lossless compression into a bitstream through arithmetic (entropy) coding. An arithmetic encoder uses an underlying symbol distribution to compress the values in the tensor. Instead of relying on hand-tuned heuristics (e.g., CABAC [4] in H.265), we adopt the method described in [73], training a distribution estimator in conjunction with the neural encoder and decoder to better estimate the symbol distribution of each encoder output. Since GRACE decodes individual packets independently, the symbol distribution of a packet must be sent as part of the packet to the decoder, which implies that the size overhead of symbol distributions increases with more packets.

GRACE reduces this overhead by employing a simpler symbol distribution that requires fewer bits to store within each packet. Specifically, GRACE trains the neural encoder to regularize the distribution of values in each encoder's output channel (224 channels in total) to conform to a zero-mean Laplace distribution. In doing so, the symbol distribution only needs to store the variance for each channel while still effectively compressing the encoder's output tensor. As a result, the symbol distribution now requires only ∼50 bytes per packet to store, a reduction from 40% of the packet size to 5%, without notably affecting the compression efficiency.

4.2 Streaming protocol

Basic protocol of GRACE: The encoder of GRACE encodes new frames at a fixed frame rate. When any packet for the next frame arrives, the decoder immediately attempts to decode the current frame. Unless all packets of the current frame are lost (which triggers a request for resending the frame), the decoder will decode the current frame using whatever packets have been received. We refer to a frame decoded using partially received packets as an *incomplete frame*. However, while GRACE can decode incomplete frames with decent quality, using these incomplete frames as reference images for decoding future frames causes the encoder's and decoder's states to be "out of sync," i.e., the next frame will be decoded based on a different reference image than the one used during encoding. This inconsistency causes error to propagate [96] to future frames even if all their packets arrive without loss.

One strawman solution to resolve error propagation is to synchronize the encoder and decoder on each frame. However, the encoding of each frame would be blocked until it knows which packets are used to decode the previous frame. This synchronization delay would render pipeline encoding, transmission, and real-time decoding infeasible.

Optimistic encoding with dynamic state resync: GRACE employs two strategies to prevent out-of-sync states from blocking the encoder or the decoder.

First, the encoder *optimistically* assumes all packets will be received and encodes frames accordingly, taking advantage of GRACE decoder's tolerance to packet losses for a small number of frames. For instance, §5.2 shows that GRACE is resilient against packet loss across 10 consecutive frames.

Second, when receiving an incomplete frame, the decoder, without stopping decoding new frames, requests the encoder to dynamically resynchronize the state in the following man-

[4] That said, such reversible random packetization requires a frame containing multiple packets. Therefore, GRACE's encoder controls the packet size such that each frame has at least 2 packets, since real-time video packets don't need to be as large as 1.5 KB [90] in practice.

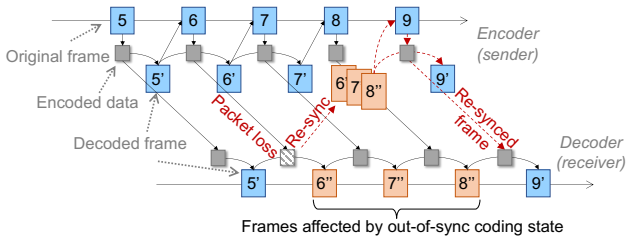
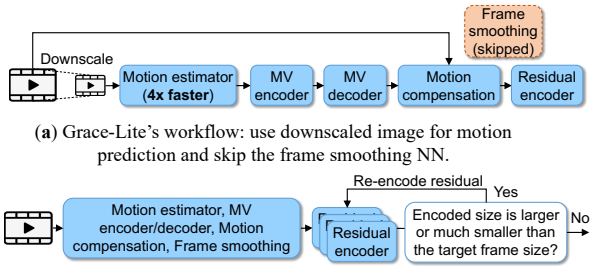


Figure 6: *Packet loss introduces discrepancies between the encoder's and the decoder's reference frames.* GRACE*'s state resync efficiently rectifies these discrepancies without causing interruptions for either the encoder or the decoder.*



(**a**) Grace-Lite's workflow: use downscaled image for motion prediction and skip the frame smoothing NN.

(**b**) Accurate bitrate control: When the frame size is too large or small, only the residual requires re-encoding.

Figure 7: GRACE *adapts the NVC for efficient execution on CPUs and accurate bitrate control.*

ner. Upon receiving a resync request, the encoder re-decodes the recent frames starting from the incomplete frame, using only the subset of packets received by the decoder (as indicated in the resync request), to compute the latest reference frame used by the decoder. As illustrated in Figure 6, if the encoder is about to encode the 9th frame and learns that the 6th frame has been decoded using partially received packets, it then quickly re-decodes frames from the 6th to the 8th. The 8th frame now aligns with the receiver's observation and thus is used as the reference frame for encoding the 9th frame.

A potential speed bottleneck is the re-decoding of frames during state resynchronization (e.g., the 6th to 8th frames in Figure 6). Fortunately, the encoder can re-decode these frames much faster than the regular decoding process by running only the motion decoder and the residual decoder. The insight is two-fold. First, motion estimation, motion encoding, and residual encoding can be skipped because these frames have already been decoded once at the encoder side, so the re-decoding only needs to estimate the incremental changes caused by the lost packets. Second, while skipping the frame smoothing NN may impact the compression efficiency of the last frame (e.g., the 9th frame in Figure 6), it only affects a single frame since the next frame will still be optimistically encoded. Appendix B.1 provides more details on the dynamics re-decoding, and §5.4 analyzes its runtime overhead.

GRACE's approach of optimistic encoding and dynamic state resynchronization capitalizes on a key advantage of GRACE's NVC—it does not need to skip or block the decoding processing for loss-affected frames; instead, it can decode them with decent quality *while* the encoder's and decoder's states are out-of-sync for a few frames, thus reducing frame delay. This approach differs from NACK (negative acknowledgement) in WebRTC [54], which requires blocking the decoding of loss-affected frames, and from Salsify's state synchronization [45], which skips all loss-affected frames.

4.3 Fast coding and bitrate control

Fast encoding and decoding: Using a standard GPU runtime with PyTorch JIT compiling [14], GRACE meets the latency requirement for real-time video communication on GPUs. As shown in §5.4, GRACE encodes and decodes 720p video at 31.2 and 51.2 fps respectively, on an NVIDIA A40 GPU (5× cheaper and 3× slower than A100). However, GRACE's NVC remains too heavy to run on laptops with CPUs and mobile phones. To address this, we develop GRACE-Lite, a lightweight version of GRACE that incorporates three optimizations (Figure 7a): *(i)* motion estimation NN operates on 2× downsampled frames, speeding up the motion estimation by 4×; *(ii)* frame smoothing NN is skipped; *(iii)* the floating point precision in NNs is reduced from 32 bits to 16 bits, making the inference 2× faster. These optimizations allow GRACE to encode and decode frames on an iPhone 14 Pro at 26.3 and 69.4 fps when compiled with the CoreML [5] library, while maintaining similar loss resiliency as GRACE (§5.4).

Accurate bitrate control: Video encoders are expected to encode frames to match the target frame sizes. Similar to Salsify [45], GRACE encodes a frame multiple times at different quality levels but in a faster way than encoding the frame from scratch each time (illustrated in Figure 7b). To achieve this, GRACE trains multiple neural encoders, each with a different α (in Eq. 2) to enable different quality-size tradeoffs. During the training phase, adjustments are confined to the residual encoders and decoders, leaving other NN weights fixed. Thus, once a frame is encoded, both the motion vector and residual are reusable, with the residual undergoing further encoding through different encoders, each producing a different frame size. This procedure, taking under 3 ms, can encode a frame multiple times using residual encoders with distinct α values. In practice, residual sizes can vary from 0.1× to 10× the MV size, allowing GRACE to cover a wide range of bitrates (§5.2).

4.4 Implementation of GRACE

GRACE is implemented in ∼2000 lines of Python code, including its NVC, packetization, bitrate adaptation, and state synchronization protocol.

Training: GRACE's NVC model architecture is based on a recent work called DVC [73]. We fine-tune GRACE from the pre-trained DVC model on the Vimeo-90K [103] dataset, under the following distribution of simulated per frame packet loss (§2.1): with an 80% probability, the loss rate is set to 0%; with a 20% probability, the loss rate is randomly selected

from {10%, 20%, 30%, 40%, 50%, 60%} [5]. By using this loss distribution, GRACE can be resilient to a wide range of loss rates without assuming the underlying network loss pattern. To achieve accurate bitrate control (§4.3), we first fine-tune an NVC with a default α (2^{-7}) using Eq. 2. Subsequently, we perform fine-tuning with 11 α values spanning from 2^{-8} to 2^{-15}, specifically to refine the residual encoder and decoder for bitrate adaptation. With a learning rate of 10^{-4}, each fine-tuning step takes about 1–2 hours on an Nvidia A40 GPU.

Delivery: We use *torchac* [13] for entropy encoding and decoding. GRACE utilizes PyTorch JIT compilation [14] when running on GPUs, while GRACE-Lite leverages CoreML [5] for inferencing on mobile devices. Both GRACE and GRACE-Lite operate using 16-bit floats at runtime. GRACE uses BPG [21] to encode and decode I-frames every 1000 frames, and can be integrated with any congestion control (CC) algorithms. Due to space limitations, we provide more details about I-frames and CCs in Appendix B.2 and B.3.

5 Evaluation

Our key findings are as follows:

- **Loss resilience:** GRACE's quality under no packet loss is on par with H.264/H.265 and gracefully declines with higher loss rates. Under 20–80% packet loss, GRACE improves the SSIM by 0.5–4 dB compared with other loss-resilient baselines across diverse videos.
- **Better video smoothness:** Under bandwidth fluctuations in real network traces, GRACE reduces the number of video freezes over 200 ms by up to 90%, tail frame delay by up to 2–5×, and non-rendered frames by up to 95%. Our user study also confirms a 38% rated score for GRACE.
- **Speed:** Our implementation of GRACE encodes/decodes 480p video at 65.8 fps/104.1 fps and 720p video at 33.6 fps/44.1 fps using Nvidia A40 GPU, 1.5–5× faster than recent neural video codecs [40, 73, 91, 105]. With the optimization detailed in §4.3, GRACE can encode/decode 720p video at 26.2 fps/69.4 fps on an iPhone 14 Pro with marginal quality degradation.

5.1 Setup

Testbed implementation: Our testbed 2 Nvidia A40 GPUs to run the video encoding and decoding with GRACE's NVC (each using one GPU). We use a packet-level network simulator to compare GRACE with baselines under various network conditions. The simulator uses a configurable drop tail queue to mimic congestion-induced packet losses and uses a token bucket scheme to simulate bandwidth variation every 0.1 seconds. Google Congestion Control (GCC) [31], a standard WebRTC algorithm widely used in real-time video applications, is used to determine the target bitrate of video codecs at each frame. It is worth noting that GCC is responsive to bandwidth drops and packet losses, as it tends to send data conservatively to avoid video delays and stalls caused by packet losses. The simulator includes encoding, packetization, rate adaptation, and decoding. We set the default frame rate at 25 fps (on par with typical RTC frame rates [78]), though GRACE can encode at a higher frame rate (§5.4). Instead of replaying stationary traffic/loss traces, the testbed can simulate dynamic packet loss rates under real-world bandwidth fluctuations. It records each decoded frame and its delay, including encoding, transmission, and decoding. We have confirmed our simulator's accuracy regarding frame delay via a real-world validation experiment in the appendix (§C.3).

[5]The packet loss rate should follow a uniform distribution covering a continuous range of losses (e.g., [0, 60%]). However, we empirically observe that using a discrete loss rate distribution makes the model converge faster without sacrificing the loss resilience.

Dataset	# of videos	Length (s)	Size	Description
Kinetics	45	450	720p 360p	Human actions and interaction with objects
Gaming	5	100	720p	PC game recordings
UVG	4	80	1080p	HD videos (human, nature, sports, etc.)
FVC	7	140	1080p	In/outdoor video calls
Total	**61**	**770**		

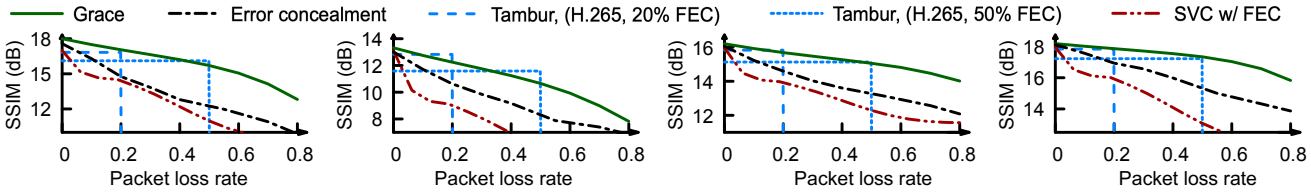
Table 1: *Dataset description.*

Test videos: Our evaluation uses 61 videos randomly sampled from four public datasets, summarized in Table 1. The total content length is 770 seconds where each video is 10–30 seconds long, matching the setup of similar works [39, 45, 86]. Importantly, these videos are obtained from entirely different sources than the training set, and they span a range of spatial complexity and temporal complexity (detailed in Appendix C.4), as well as multiple resolutions. This diversity allows us to assess GRACE's average performance across different contents and study how content affects its performance.

Network traces: We test GRACE and the baselines on 16 real bandwidth traces, eight of which are LTE traces from the Mahimahi network-emulation tool [9, 82], and the rest are broadband traces from FCC (July 2021) [10]. The traces are in the format of bandwidth timeseries. The bandwidth fluctuates between 0.2 Mbps to 8 Mbps in the traces. By default, we set the one-way propagation delay to 100 ms and the queue size to 25 packets. We also vary these values in §5.3.

Baselines: We employ H.265 (through FFmpeg v4.2.7) as the underlying video codec for all baselines (except for NVC-based ones) since H.265 is recognized with comparable or better compression efficiency than VP8/9 and H.264 [15, 49] (as confirmed in Appendix C.1). We compare GRACE against a range of loss-resilient baselines that cover various approaches outlined in §2.2 (more details in Appendix C.2).

- **Forward error correction**: We use Tambur [86], a state-of-the-art FEC scheme based on streaming codes [28]. Its redundancy rate dynamically adapts based on the mea-



(**a**) Kinetics dataset (**b**) Gaming dataset (**c**) UVG dataset (**d**) FVC dataset

Figure 8: *Video quality achieved by different schemes under varying packet loss rates at the same encoded bitrate (6 Mbps).*

sured packet loss in the preceding 2 seconds. Compared with regular FEC, streaming codes reduce the number of non-decodable frames when transmitting an equivalent amount of parity packets. We have also validated that Tambur outperforms WebRTC's default FEC scheme.

- **Decoder-side neural error concealment**: We use ECFVI [59], an NN-based error concealment method shown to outperform previous techniques relying on motion estimation recovery [87] or inpainting [34].[6] To ensure each packet is independently decodable, we apply flexible macroblock ordering (FMO) [42] to split the frame into 64×64-pixel[7] blocks and map them randomly to packets. This results in a 10% increase in the encoded frame size, in line with previous findings [64, 74, 99]. After decoding an incomplete frame, ECFVI uses NNs to estimate missing motion vectors and enhance the reconstructed frame through inpainting.
- **Scalable video coding (SVC)**: We implement an idealized SVC, designed so that when the first k layers arrive, it achieves the same quality as that of H.265 with the same number of received bytes. This idealized implementation surpasses the state-of-the-art NN-based SVC [40]. We also add 50% FEC to protect the base layer for SVC, following a common practice in real-time video applications [60].
- **Selective frame skipping**: Salsify [45] skips frames affected by loss at the decoder side after the encoder receives the packet loss indication and resends a new P-frame using the last fully received frame as a reference. Voxel [83] employs selective frame skipping to mitigate video rebuffering and improve the user's QoE.

We make another idealized assumption in favor of SVC, Salsify, and Voxel. We assume that their codec's output bitrate on every frame perfectly matches the target bitrate determined by the congestion control algorithm, i.e., no overshoots or undershoots. This idealization makes these baselines perform slightly better than they would under real-world conditions.

Variants of GRACE: To highlight the impact of different design choices, we evaluate **GRACE-P** and **GRACE-D**. They are trained the same way as GRACE, except that GRACE-P does not use simulated loss while GRACE-D freezes the encoder NN weights (i.e., fine-tuning only the decoder NN with simulated loss). They represent alternative ways to simulate packet losses during training. We also test **GRACE-Lite**, which incorporates the optimizations described in §4.3.

Furthermore, we present the quality improvement achieved by the state-of-the-art super-resolution (SR) model [70] when applied to GRACE and other baselines. It is important to note that SR can be applied to any decoded frames, making it *orthogonal* to GRACE's design space. Details of this experiment are provided in Appendix C.8.

Metrics: Following prior work on real-time video communication [26, 45, 47, 48, 86], we measure the performance of a video session across three aspects.

- *Visual quality* of a frame is measured by SSIM. Following recent work [45, 104], we express SSIM in dB, calculated as $-10\log(1-\text{SSIM})$ across all rendered frames.
- *Realtimeness* is measured by the 98th percentile (P98) of frame delay (time elapsed between the frame's encoding and decoding), and non-rendered frames (either undecodable due to insufficient FEC protection or exceeding 400 ms after the frame is encoded).
- *Smoothness* of the video is measured by video stall, defined as an inter-frame gap exceeding 200 ms, following the industry convention [78]. We report the average number of video stalls per second and the ratio of video stall time over the entire video length.

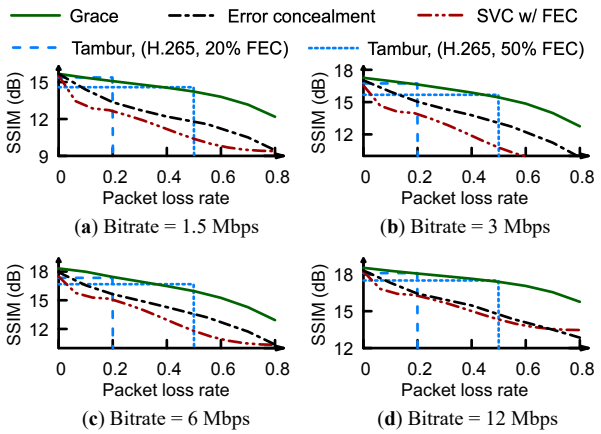
5.2 Compression efficiency and loss resilience

Loss resilience: In real world, packet loss per frame (defined in §2.1) can span a wide range from 0 to over 80% [86]. Figure 8 compares GRACE's video quality with the baselines under varying packet loss rates across different test video sets. For a fair comparison, we fix the encoded bitrate of all baselines at 6 Mbps (with actual differences under 5%) while ensuring that GRACE's encoded bitrate *never* exceeds that of the baselines. On average, the quality of GRACE drops by 0.5 dB to 2 dB in SSIM as the packet loss rate rises from 20% to 50%, and by up to 3.5 dB when the packet loss rate reaches 80%. These quality drops of GRACE are notably lower than the baselines, including FEC-based and neural error concealment schemes, at the same packet loss rates.

Figure 9 shows the average quality across all test videos

[6] A parallel effort, Reparo [67], demonstrates effective error concealment for a particular video type (talking head), but it lacks comparisons with any NN-based baselines and does not provide a public codebase for testing.

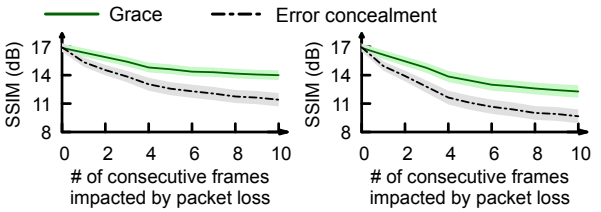
[7] A smaller block size such as 16×16 can greatly inflate the frame size [64, 74, 99], while a larger block size such as 256×256 hinders information recovery upon packet loss. We empirically choose the 64×64 block size to balance between frame size and quality.



(**a**) Bitrate = 1.5 Mbps (**b**) Bitrate = 3 Mbps

(**c**) Bitrate = 6 Mbps (**d**) Bitrate = 12 Mbps

Figure 9: *Video quality of each scheme under different packet loss rates when videos are encoded at different bitrates.*



(**a**) Loss rate = 30% (**b**) Loss rate = 50%

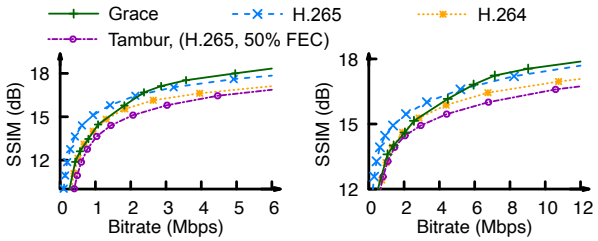
Figure 10: *Stress test of applying persistent packet loss on consecutive frames.*

when the encoded bitrates of all schemes are set to 1.5, 3, 6, and 12 Mbps. Compared with the baselines, GRACE achieves a more graceful and less pronounced quality decline as packet loss increases. Figure 10 further stress tests the loss resilience of GRACE against neural error concealment (the most competitive baseline), when a 30% or 50% packet loss is applied to 1 to 10 consecutive frames without the encoder and decoder synchronizing their states. Although the figure shows that both methods experience quality degradation, GRACE markedly surpasses the neural error concealment baseline in these extreme conditions. Figure 11 visualizes their decoded images after a 50% packet loss is applied to three consecutive frames, confirming that the image decoded by GRACE has less visual distortion.

Compression efficiency: We verify whether GRACE's compression efficiency under no packet loss is on par with H.264 and H.265, which are advanced video codecs designed for high compression efficiency rather than loss resilience. Figure 12 groups the test videos by resolution. On low bitrates, GRACE demonstrates similar compression efficiency as H.264 and marginally underperforms H.265 on both 720p and 1080p videos. On high bitrates (over 3 Mbps for 720p and 6 Mbps for 1080p), GRACE's compression efficiency matches or even surpasses H.265. Compared against Tambur with a persistent 50% FEC redundancy, GRACE achieves a better quality-bitrate tradeoff across the entire bitrate range.

Impact of video content on compression efficiency: To understand the impact of video content on GRACE's compression efficiency, we group the video content based on spatial index (SI) and temporal index (TI), which are established metrics for assessing the spatiotemporal complexity of videos [58]. Figure 13 presents the average gain of GRACE over H.264 in terms of SSIM for videos in each SI-TI combination, encoded at a bitrate of 5 Mbps. The results indicate that GRACE's compression efficiency has a higher advantage over H.264 for videos with low spatial complexity, but this advantage diminishes as the spatial index increases. For a more thorough understanding of GRACE's behavior, Appendix C.5 also shows an example where GRACE performs poorly.

Figure 11: *Sample images decoded by* GRACE *and error concealment under a 50% packet loss on three consecutive frames.* GRACE *achieves less image distortion.*



(**a**) Resolution = 720p (**b**) Resolution = 1080p

Figure 12: *Quality-size tradeoff of* GRACE *on videos with different resolution. Overall,* GRACE *is better than H.264 and slightly worse than H.265 in terms of compression efficiency.*

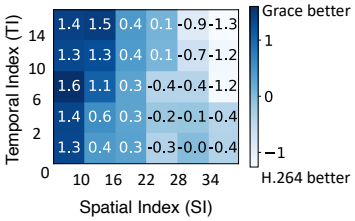
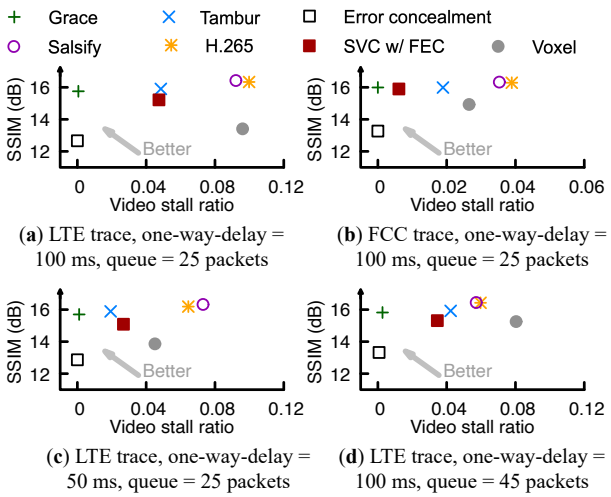


Figure 13: *Mean difference in SSIM (dB) between* GRACE *and H.264 on videos grouped by SI and TI. At the same bitrate (5 Mbps),* GRACE *achieves better video quality than H.264 on low-SI videos but lags behind H.264 on high-SI videos.*

5.3 Video quality vs. realtimeness/smoothness

Figures 14a evaluates GRACE against baselines in terms of average quality (SSIM) and video stall ratio (a smoothness metric) using the network traces from the LTE dataset, under a one-way network delay of 100 ms and a drop-tail queue of 25 packets. Although the SSIM of GRACE is slightly lower than that of the baselines with the highest average SSIM, GRACE



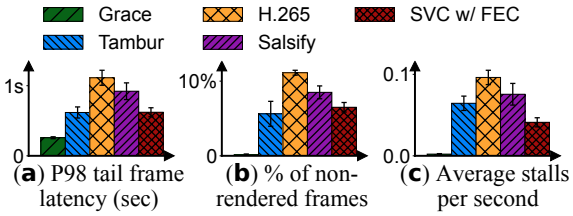
(a) LTE trace, one-way-delay = 100 ms, queue = 25 packets

(b) FCC trace, one-way-delay = 100 ms, queue = 25 packets

(c) LTE trace, one-way-delay = 50 ms, queue = 25 packets

(d) LTE trace, one-way-delay = 100 ms, queue = 45 packets

Figure 14: *End-to-end simulation results over different network traces, one-way delays and network queue lengths.*



(a) P98 tail frame latency (sec)

(b) % of non-rendered frames

(c) Average stalls per second

Figure 15: GRACE *outperforms other baselines on different metrics of realtimeness and smoothness. Updated this figure: added error bar and y-axis*

significantly reduces the video stall ratio.

We repeat the test on a different dataset (FCC) under the same network setup (Figures 14b), with a lower one-way network delay of 50 ms (Figure 14c), and with a longer queue length of 45 packets (Figure 14d). In all settings, GRACE maintains a video stall ratio below 0.5%, whereas the baselines have 4–32× more video stalls, except for the error concealment baseline, which yields a 3dB lower SSIM compared with GRACE. This is because when packet loss happens, GRACE can still decode the frame, while the baselines other than error concealment may experience video stalls due to either skipping frames (e.g., Salsify or Voxel) or waiting for retransmission packets (FEC and SVC).

Figure 15 compares GRACE with the baselines using other realtimeness and smoothness metrics, with the one-way delay set to 100 ms and the queue length set to 25 packets over the LTE traces. For clarity, we only include baselines with comparable average SSIMs in this figure (excluding Voxel and error concealment). While achieving similar video quality, GRACE reduces the 98th percentile frame delay by a factor of 2–5× and non-rendered frames by up to 95%. In §C.7, we also evaluate GRACE with a different congestion control algorithm—Salsify's CC [45].

Figure 16 provides a concrete example of GRACE's behavior. The bandwidth drops from 8 Mbps to 2 Mbps at 1.5 s,

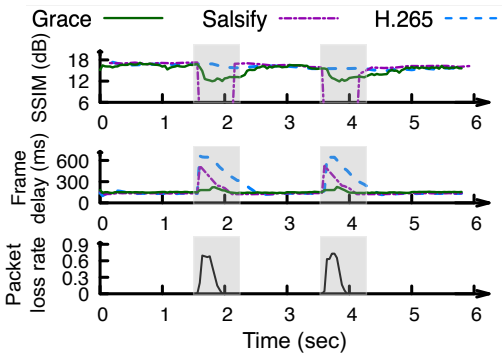


Figure 16: GRACE *achieves lower delay and maintains decent visual quality during sudden bandwidth drops: its delay is lower than both baselines while rendering more frames than Salsify without frame skipping or packet retransmission.*

lasting for 800 ms, before returning to 8 Mbps (another bandwidth drop occurs at 3.5 s and lasts for the same duration). During each drop, GRACE's delay does not experience a sharp increase as the baselines. Salsify is the second best owing to its frame skipping while H.265 must wait for retransmissions. In this experiment, both GRACE and Salsify use the same CC, leading to similar qualities on frames not skipped by Salsify. However, during congestion, GRACE's quality degrades only marginally without skipping any frames, limiting the drop of SSIM to less than 4 dB even when more than 10 consecutive frames encounter a packet loss of over 50%. With the assistance of state resync (§4.2), GRACE's quality resumes quickly (within 1 RTT) after packet losses.

User study: To validate GRACE's effectiveness, we conducted an IRB-approved user study, collecting 960 user ratings from 240 Amazon MTurk workers [1]. We first choose a few genres based on the real-time video streaming use cases, including cloud gaming, real-time sports events, daily human activities, and video conferencing. Then, we randomly selected 8 video clips from the UGC dataset [98]. These video clips were streamed using GRACE, Salsify codec, WebRTC with default FEC, and H.265 with Tambur. (A screenshot of each video clip is shown in Figure 26 in Appendix.) The sampled videos have a similar distribution of quality, realtimeness, and smoothness as seen in Figure 14. Following [22], when an MTurk user signs up for the user study, they are randomly assigned to rate their user experience on a scale of 1–5 for the videos delivered through different methods. Figure 17 displays the mean opinion score (MOS) for each video, confirming that the videos rendered by GRACE are consistently favored by real users.

5.4 Microbenchmarking

Encoding/decoding latency breakdown: Figure 18 shows a breakdown of the encoding and decoding delays of GRACE on an Nvidia A40 GPU (5× cheaper and 3× slower than Nvidia A100). GRACE encodes and decodes a 720p frame within 29.7 ms (33 fps) and 19.5 ms (51.2 fps), respectively. It can

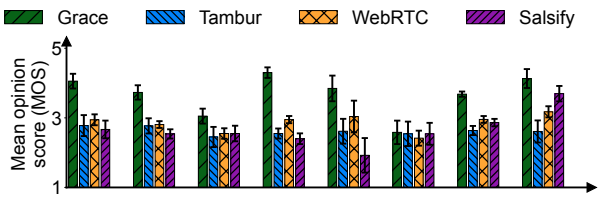


Figure 17: *User study experiment shows that videos streamed by* GRACE *are consistently favored by real users. The error bar shows the standard deviation of the mean.*

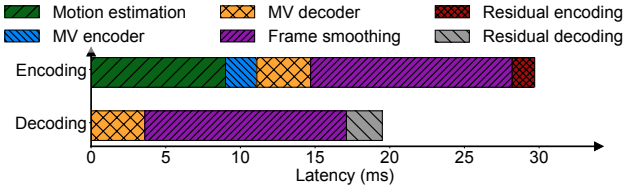


Figure 18: *Latency breakdown of GPU-based encoding and decoding of* GRACE *on a 720p frame.*

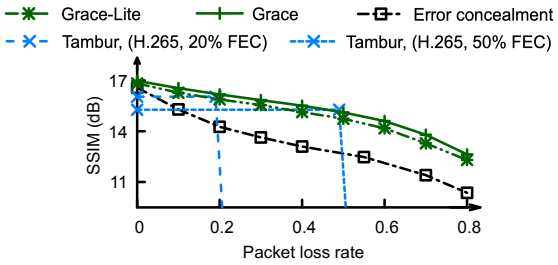


Figure 19: GRACE*-Lite realizes similar loss resilience to* GRACE *and outperforms other baselines.*

also encode/decode 480p video at 65.8 fps/104.1 fps.

This breakdown also carries several implications. First, the fast resync logic (§4.2) requires the encoder to run the MV decoder and residual decoder, which together only consume 6 ms on a 720p frame, allowing resync to complete with a minimal increase in encoding delay. Moreover, GRACE may need to encode a frame multiple times as explained in §4.3, but the extra overhead only involves residual encoding, which takes only 1.5 ms on a 720p frame.

Speed optimization in GRACE-Lite: With the optimizations described in §4.3, GRACE-Lite reduces the encoding delay of a 720 frame on iPhone 14 Pro from 314 ms to 38.1 ms, and the decoding delay from 239 ms to 14.4 ms. We also report GRACE-Lite's speed on CPUs with OpenVINO compilation in Appendix C.9. Figure 19 compares the loss resilience of GRACE-Lite and GRACE with the two most competitive baselines in §5.2—neural error concealment and Tambur. At the same packet loss, GRACE-Lite achieves slightly lower quality than GRACE, yet it still outperforms other baselines.

Impact of joint training: Figure 20 compares GRACE with its two variants: GRACE-P and GRACE-D, showing that both variants have lower levels of loss resilience than GRACE due to not jointly training the encoder and decoder. Appendix C.10 shows an example of frames decoded by the variants.

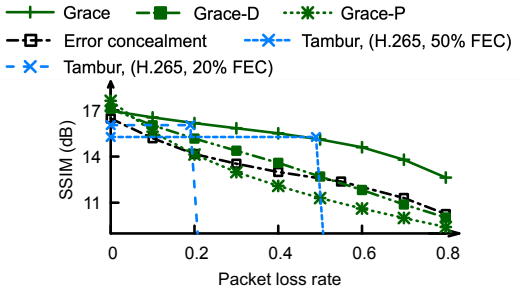


Figure 20: *Although* GRACE*-D and* GRACE*-P attain slightly better quality than* GRACE *in the absence of packet loss, they are much less resilient to loss than* GRACE*.*

6 Limitation

The current implementation of GRACE still has several limitations. First, it is not optimized enough to run at 30 fps on very resource-constrained devices that barely sustain a classic video codec. For instance, achieving real-time encoding and decoding on regular CPUs (e.g., Intel Xeon Silver 4216) still requires 32 cores (§C.9). Secondly, due to its use of NVC, GRACE may have lower compression efficiency than traditional handcrafted codecs on some video content that deviates a lot from the training data of NVC. For instance, its compression efficiency is worse than H.26x on videos with high spatial complexity (§5.2). In rare instances, GRACE is observed to fail to accurately reconstruct original frames under high packet losses. Third, our focus with GRACE is on unicast video communication rather than multiparty conferencing. We hope GRACE can inspire future work to address these limitations. Potential avenues include democratizing GRACE on more devices by embracing the recent advancements in hardware [23, 24], distilling more lightweight models suitable for less powerful devices. We acknowledge there is not a good solution to address GRACE's generalization issue, which is a problem not unique to GRACE but inherent in general NVCs. We hope that future measurement studies may shed light on the generalization of NVCs and contribute to their improvement.

7 Conclusion

This paper presents GRACE, a real-time video system designed for loss resilience, preserving quality of experience (QoE) for users across diverse packet losses. GRACE enhances loss resilience by jointly training a neural encoder and decoder under a spectrum of packet losses. It attains video quality on par with conventional codecs in the absence of packet loss, and exhibits a less pronounced quality degradation as packet loss escalates, outperforming existing loss-resilient methods.

8 Acknowledgement

We thank the anonymous reviewers and our shepherd Dongsu Han. This project is supported by NSF CNS 2146496, 2131826, 2313190, 1901466, and UChicago CERES Center.

References

[1] Amazon Mechanical Turk. https://www.mturk.com/.

[2] Aurora5 HEVC Test Results. https://www.visionular.com/en/putting-the-aurora5-hevc-encoder-to-the-test/.

[3] Bringing Zoom's end-to-end optimizations to WebRTC. https://blog.livekit.io/livekit-one-dot-zero/.

[4] Context-adaptive binary arithmetic coding. https://en.wikipedia.org/wiki/Context-adaptive_binary_arithmetic_coding.

[5] Core ML Documentation. https://developer.apple.com/documentation/coreml.

[6] Features of WebRTC VR Streaming. https://flashphoner.com/features-of-webrtc-vr-streaming/.

[7] FFmpeg streaming guide. http://trac.ffmpeg.org/wiki/StreamingGuide.

[8] Linear Congruential Generator. https://en.wikipedia.org/wiki/Linear_congruential_generator.

[9] Mamahi Cellular traces. https://github.com/ravinet/mahimahi/tree/master/traces.

[10] Measuring Broadband Raw Data Releases. https://www.fcc.gov/oet/mba/raw-data-releases.

[11] Open Source Cloud Gaming with WebRTC. https://webrtchacks.com/open-source-cloud-gaming-with-webrtc/.

[12] SI/TI calculation tools. https://github.com/VQEG/siti-tools.

[13] torchac: Fast Arithmetic Coding for PyTorch. https://github.com/fab-jul/torchac.

[14] Torch.compile tutorial . https://pytorch.org/tutorials/intermediate/torch_compile_tutorial.html.

[15] VP9 encoding/decoding performance vs. HEVC/H.264. https://blogs.gnome.org/rbultje/2015/09/28/vp9-encodingdecoding-performance-vs-hevch-264/.

[16] WebRTC and IoT Applications. https://rtcweb.in/webrtc-and-iot-applications/.

[17] WebRTC Cloud Gaming: Unboxing Stadia. https://webrtc.ventures/2021/02/webrtc-cloud-gaming-unboxing-stadia/.

[18] WebRTC: Enabling Collaboration Augmented Reality App. https://arvrjourney.com/webrtc-enabling-collaboration-cebdd4c9ce06?gi=e19b1c0f65c0.

[19] WebRTC in IoT: What is the Intersection Point? https://mobidev.biz/blog/webrtc-real-time-communication-for-the-internet-of-things.

[20] What powers Google Meet and Microsoft Teams? WebRTC Demystified. https://levelup.gitconnected.com/what-powers-google-meet-and-microsoft-teams-webrtc-demystified-step-by-step-tutorial-e0cb422010f7.

[21] Better Portable Graphics. https://bellard.org/bpg/, 2014.

[22] SENSEI: Aligning Video Streaming Quality with Dynamic User Sensitivity, author=Zhang, Xu and Ou, Yiyang and Sen, Siddhartha and Jiang, Junchen. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 303–320, 2021.

[23] Deploying Transformers on the Apple Neural Engine. https://machinelearning.apple.com/research/neural-engine-transformers, 2022.

[24] Harnessing the NVIDIA Ada Architecture for Frame-Rate Up-Conversion in the NVIDIA Optical Flow SDK. https://developer.nvidia.com/blog/harnessing-the-nvidia-ada-architecture-for-frame-rate-up-conversion-in-the-nvidia-optical-flow-sdk/, 2023.

[25] Asma Ben Abdallah, Amin Zribi, Ali Dziri, Fethi Tlili, and Michel Terré. H.264/AVC video transmission over UWB AV PHY IEEE 802.15. 3c using UEP and adaptive modulation techniques. In *2019 International Conference on Advanced Communication Technologies and Networking (CommNet)*, pages 1–6. IEEE, 2019.

[26] Doreid Ammar, Katrien De Moor, Min Xie, Markus Fiedler, and Poul Heegaard. Video QoE killer and performance statistics in WebRTC-based video communication. In *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, pages 429–436. IEEE, 2016.

[27] Anurag Arnab, Mostafa Dehghani, Georg Heigold, Chen Sun, Mario Lučić, and Cordelia Schmid. Vivit: A video vision transformer. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 6836–6846, 2021.

[28] Ahmed Badr, Ashish Khisti, Wai-tian Tan, Xiaoqing Zhu, and John Apostolopoulos. FEC for VoIP using dual-delay streaming codes. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[29] Niklas Blum, Serge Lachapelle, and Harald Alvestrand. WebRTC-Realtime Communication for the Open Web Platform: What was once a way to bring audio and video to the web has expanded into more use cases we could ever imagine. *Queue*, 19(1):77–93, 2021.

[30] Eirina Bourtsoulatze, David Burth Kurka, and Deniz Gündüz. Deep joint source-channel coding for wireless image transmission. *IEEE Transactions on Cognitive Communications and Networking*, 5(3):567–579, 2019.

[31] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Analysis and design of the google congestion control for web real-time communication (WebRTC). In *Proceedings of the 7th International Conference on Multimedia Systems*, pages 1–12, 2016.

[32] Fabrizio Carpi, Christian Häger, Marco Martalò, Riccardo Raheli, and Henry D. Pfister. Reinforcement Learning for Channel Coding: Learned Bit-Flipping Decoding. In *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 922–929, 2019.

[33] Jeff Castura and Yongyi Mao. Rateless coding over fading channels. *IEEE communications letters*, 10(1):46–48, 2006.

[34] Ya-Liang Chang, Zhe Yu Liu, Kuan-Ying Lee, and Winston Hsu. Free-form video inpainting with 3d gated convolution and temporal patchgan. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9066–9075, 2019.

[35] Sheng Cheng, Han Hu, and Xinggong Zhang. ABRF: Adaptive BitRate-FEC Joint Control for Real-Time Video Streaming. *IEEE Transactions on Circuits and Systems for Video Technology*, 2023.

[36] Sheng Cheng, Han Hu, Xinggong Zhang, and Zongming Guo. DeepRS: Deep-learning based network-adaptive FEC for real-time video communications. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2020.

[37] Kristy Choi, Kedar Tatwawadi, Aditya Grover, Tsachy Weissman, and Stefano Ermon. Neural joint source-channel coding. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 1182–1192. PMLR, 09–15 Jun 2019.

[38] Wen-Jeng Chu and Jin-Jang Leou. Detection and concealment of transmission errors in H.261 images. *IEEE Transactions on Circuits and Systems for Video Technology*, 8(1):74–84, 1998.

[39] Mauro Conti, Simone Milani, Ehsan Nowroozi, and Gabriele Orazi. Do Not Deceive Your Employer with a Virtual Background: A Video Conferencing Manipulation-Detection System. *arXiv preprint arXiv:2106.15130*, 2021.

[40] Mallesham Dasari, Kumara Kahatapitiya, Samir R. Das, Aruna Balasubramanian, and Dimitris Samaras. Swift: Adaptive video streaming with layered neural codecs. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 103–118, Renton, WA, April 2022. USENIX Association.

[41] Sandesh Dhawaskar Sathyanarayana, Kyunghan Lee, Dirk Grunwald, and Sangtae Ha. Converge: QoE-driven Multipath Video Conferencing over WebRTC. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 637–653, 2023.

[42] Yves Dhondt and Peter Lambert. Flexible Macroblock Ordering: an error resilience tool in H. 264/AVC. In *5th FTW PhD Symposium*. Ghent University. Faculty of Engineering, 2004.

[43] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[44] Patrick Esser, Robin Rombach, and Bjorn Ommer. Taming transformers for high-resolution image synthesis. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12873–12883, 2021.

[45] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. Salsify: Low-Latency network video through tighter integration between a video codec and a transport protocol. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 267–282, Renton, WA, April 2018. USENIX Association.

[46] Chen Gao, Ayush Saraf, Jia-Bin Huang, and Johannes Kopf. Flow-edge guided video completion. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XII 16*, pages 713–729. Springer, 2020.

[47] Boni García, Micael Gallego, Francisco Gortázar, and Antonia Bertolino. Understanding and estimating quality of experience in WebRTC applications. *Computing*, 101:1585–1607, 2019.

[48] Boni García, Francisco Gortázar, Micael Gallego, and Andrew Hines. Assessment of qoe for video and audio in webrtc applications using full-reference models. *Electronics*, 9(3):462, 2020.

[49] Dan Grois, Detlev Marpe, Amit Mulayoff, Benaya Itzhaky, and Ofer Hadar. Performance comparison of h. 265/mpeg-hevc, vp9, and h. 264/mpeg-avc encoders. In *2013 Picture Coding Symposium (PCS)*, pages 394–397. IEEE, 2013.

[50] Tobias Gruber, Sebastian Cammerer, Jakob Hoydis, and Stephan ten Brink. On deep learning-based channel decoding. In *2017 51st Annual Conference on Information Sciences and Systems (CISS)*, pages 1–6, 2017.

[51] Deniz Gündüz, Paul de Kerret, Nicholas D Sidiropoulos, David Gesbert, Chandra R Murthy, and Mihaela van der Schaar. Machine learning in the air. *IEEE Journal on Selected Areas in Communications*, 37(10):2184–2199, 2019.

[52] Zhaoyuan He, Yifan Yang, Shuozhe Li, Diyuan Dai, and Lili Qiu. Neural Video Recovery for Cloud Gaming. *arXiv preprint arXiv:2307.07847*, 2023.

[53] Zhaoyuan He, Yifan Yang, Lili Qiu, and Kyoungjun Park. Real-Time Neural Video Recovery and Enhancement on Mobile Devices. *arXiv preprint arXiv:2307.12152*, 2023.

[54] Stefan Holmer, Mikhal Shemer, and Marco Paniconi. Handling packet loss in WebRTC. In *2013 IEEE International Conference on Image Processing*, pages 1860–1864. IEEE, 2013.

[55] Zhihao Hu, Guo Lu, and Dong Xu. FVC: A new framework towards deep video compression in feature space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1502–1511, 2021.

[56] Ismaeil Ismaeil, Shahram Shirani, Faouzi Kossentini, and Rabab Ward. An efficient, similarity-based error concealment method for block-based coded images. In *Proceedings 2000 International Conference on Image Processing (Cat. No. 00CH37101)*, volume 3, pages 388–391. IEEE, 2000.

[57] ITU-T. Recommendation G.114, one-way transmission time. *Series G: Transmission Systems and Media, Digital Systems and Networks, Telecommunication Standardization Sector of ITU*, 2003.

[58] P ITU-T RECOMMENDATION. Subjective video quality assessment methods for multimedia applications. 1999.

[59] Jaeyeon Kang, Seoung Wug Oh, and Seon Joo Kim. Error compensation framework for flow-guided video inpainting. In *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XV*, pages 375–390. Springer, 2022.

[60] Amin Abdel Khalek, Constantine Caramanis, and Robert W Heath. A cross-layer design for perceptual optimization of H. 264/SVC with unequal error protection. *IEEE Journal on selected areas in Communications*, 30(7):1157–1171, 2012.

[61] Jaehong Kim, Youngmok Jung, Hyunho Yeo, Juncheol Ye, and Dongsu Han. Neural-enhanced live streaming: Improving live video ingest via online learning. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 107–125, 2020.

[62] Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[63] Vineeth Shetty Kolkeri. *Error concealment techniques in H. 264/AVC, for video transmission over wireless networks*. PhD thesis, The University of Texas at Arlington, 2009.

[64] Sunil Kumar, Liyang Xu, Mrinal K Mandal, and Sethuraman Panchanathan. Error resiliency schemes in H. 264/AVC standard. *Journal of Visual Communication and Image Representation*, 17(2):425–450, 2006.

[65] David Burth Kurka and Deniz Gündüz. Deepjscc-f: Deep joint source-channel coding of images with feedback. *IEEE Journal on Selected Areas in Information Theory*, 1(1):178–193, 2020.

[66] Peter Lambert, Wesley De Neve, Yves Dhondt, and Rik Van de Walle. Flexible macroblock ordering in H. 264/AVC. *Journal of Visual Communication and Image Representation*, 17(2):358–375, 2006.

[67] Tianhong Li, Vibhaalakshmi Sivaraman, Lijie Fan, Mohammad Alizadeh, and Dina Katabi. Reparo: Loss-Resilient Generative Codec for Video Conferencing. *arXiv preprint arXiv:2305.14135*, 2023.

[68] Weiping Li. Overview of fine granularity scalability in MPEG-4 video standard. *IEEE Transactions on circuits and systems for video technology*, 11(3):301–317, 2001.

[69] Zhen Li, Cheng-Ze Lu, Jianhua Qin, Chun-Le Guo, and Ming-Ming Cheng. Towards an end-to-end framework for flow-guided video inpainting. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 17562–17571, 2022.

[70] Jingyun Liang, Jiezhang Cao, Guolei Sun, Kai Zhang, Luc Van Gool, and Radu Timofte. Swinir: Image restoration using swin transformer. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1833–1844, 2021.

[71] Rui Liu, Hanming Deng, Yangyi Huang, Xiaoyu Shi, Lewei Lu, Wenxiu Sun, Xiaogang Wang, Jifeng Dai, and Hongsheng Li. FuseFormer: Fusing Fine-Grained Information in Transformers for Video Inpainting. In *ICCV*, 2021.

[72] Yunzhuo Liu, Bo Jiang, Tian Guo, Ramesh K. Sitaraman, Don Towsley, and Xinbing Wang. Grad: Learning for overhead-aware adaptive video streaming with scalable video coding. In *Proceedings of the 28th ACM International Conference on Multimedia*, MM '20, page 349–357, New York, NY, USA, 2020. Association for Computing Machinery.

[73] Guo Lu, Wanli Ouyang, Dong Xu, Xiaoyun Zhang, Chunlei Cai, and Zhiyong Gao. DVC: An end-to-end deep video compression framework. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10998–11007, 2019.

[74] Rong Luo and Bin Chen. A Hierarchical Scheme of Flexible Macroblock Ordering for ROI based H.264/AVC Video Coding. In *2008 10th International Conference on Advanced Communication Technology*, volume 3, pages 1579–1582, 2008.

[75] Yi Ma, Yongqi Zhai, and Ronggang Wang. DeepFGS: Fine-Grained Scalable Coding for Learned Image Compression. *arXiv preprint arXiv:2201.01173*, 2022.

[76] David JC MacKay. Fountain codes. *IEE Proceedings-Communications*, 152(6):1062–1068, 2005.

[77] David JC MacKay and Radford M Neal. Near Shannon limit performance of low density parity check codes. *Electronics letters*, 33(6):457–458, 1997.

[78] Kyle MacMillan, Tarun Mangla, James Saxon, and Nick Feamster. Measuring the performance and network utilization of popular video conferencing applications. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 229–244, 2021.

[79] Michael Mathieu, Camille Couprie, and Yann LeCun. Deep multi-scale video prediction beyond mean square error. *arXiv preprint arXiv:1511.05440*, 2015.

[80] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. Achieving consistent low latency for wireless real-time communications with the shortest control loop. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 193–206, 2022.

[81] Cholman Nam, Changgon Chu, Taeguk Kim, and Sokmin Han. A novel motion recovery using temporal and spatial correlation for a fast temporal error concealment over H. 264 video sequences. *Multimedia Tools and Applications*, 79:1221–1240, 2020.

[82] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, 2015.

[83] Mirko Palmer, Malte Appel, Kevin Spiteri, Balakrishnan Chandrasekaran, Anja Feldmann, and Ramesh K. Sitaraman. Voxel: Cross-layer optimization for video streaming with imperfect transmission. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '21, page 359–374, New York, NY, USA, 2021. Association for Computing Machinery.

[84] Jan Peters and Stefan Schaal. Reinforcement learning of motor skills with policy gradients. *Neural networks*, 21(4):682–697, 2008.

[85] Devdeep Ray, Connor Smith, Teng Wei, David Chu, and Srinivasan Seshan. SQP: Congestion Control for Low-Latency Interactive Video Streaming. *arXiv preprint arXiv:2207.11857*, 2022.

[86] Michael Rudow, Francis Y. Yan, Abhishek Kumar, Ganesh Ananthanarayanan, Martin Ellis, and KV Rashmi. Tambur: Efficient loss recovery for videoconferencing via streaming codes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 953–971, 2023.

[87] Arun Sankisa, Arjun Punjabi, and Aggelos K Katsaggelos. Video error concealment using deep neural networks. In *2018 25th IEEE International Conference on Image Processing (ICIP)*, pages 380–384. IEEE, 2018.

[88] Thomas Schierl, Thomas Stockhammer, and Thomas Wiegand. Mobile video transmission using scalable video coding. *IEEE transactions on circuits and systems for video technology*, 17(9):1204–1217, 2007.

[89] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the

H. 264/AVC standard. *IEEE Transactions on circuits and systems for video technology*, 17(9):1103–1120, 2007.

[90] Taveesh Sharma, Tarun Mangla, Arpit Gupta, Junchen Jiang, and Nick Feamster. Estimating WebRTC Video QoE Metrics Without Using Application Headers. *arXiv preprint arXiv:2306.01194*, 2023.

[91] Yibo Shi, Yunying Ge, Jing Wang, and Jue Mao. AlphaVC: High-Performance and Efficient Learned Video Compression. In *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XIX*, pages 616–631. Springer, 2022.

[92] Vibhaalakshmi Sivaraman, Pantea Karimi, Vedantha Venkatapathy, Mehrdad Khani, Sadjad Fouladi, Mohammad Alizadeh, Frédo Durand, and Vivienne Sze. Gemino: Practical and Robust Neural Compression for Video Conferencing. *arXiv preprint arXiv:2209.10507*, 2022.

[93] Keyu Tan and Alan Pearmain. A new error resilience scheme based on FMO and error concealment in H. 264/AVC. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1057–1060. IEEE, 2011.

[94] Wai-tian Tan and Avideh Zakhor. Multicast transmission of scalable video using receiver-driven hierarchical FEC. In *Packet Video Workshop*, volume 99, 1999.

[95] Wai-Tian Tan and Avideh Zakhor. Video multicast using layered fec and scalable compression. *IEEE Transactions on circuits and systems for video technology*, 11(3):373–386, 2001.

[96] Yao Wang and Qin-Fan Zhu. Error control and concealment for video communication: A review. *Proceedings of the IEEE*, 86(5):974–997, 1998.

[97] Yi Wang, Xiaoqiang Guo, Feng Ye, Aidong Men, and Bo Yang. A novel temporal error concealment framework in H. 264/AVC. In *2013 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6. IEEE, 2013.

[98] Yilin Wang, Sasi Inguva, and Balu Adsumilli. YouTube UGC dataset for video compression research. In *2019 IEEE 21st International Workshop on Multimedia Signal Processing (MMSP)*, pages 1–5. IEEE, 2019.

[99] Stephan Wenger and Michael Horowitz. Scattered slices: a new error resilience tool for H. 26L. *JVT-B027*, 2, 2002.

[100] Stephen B. Wicker and Vijay K. Bhargava. *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[101] Jiangkai Wu, Yu Guan, Qi Mao, Yong Cui, Zongming Guo, and Xinggong Zhang. ZGaming: Zero-Latency 3D Cloud Gaming by Image Prediction. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 710–723, 2023.

[102] Chongyang Xiang, Jiajun Xu, Chuan Yan, Qiang Peng, and Xiao Wu. Generative adversarial networks based error concealment for low resolution video. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1827–1831. IEEE, 2019.

[103] Tianfan Xue, Baian Chen, Jiajun Wu, Donglai Wei, and William T Freeman. Video enhancement with task-oriented flow. *International Journal of Computer Vision*, 127(8):1106–1125, 2019.

[104] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning *in situ*: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, Santa Clara, CA, February 2020. USENIX Association.

[105] Ren Yang, Fabian Mentzer, Luc Van Gool, and Radu Timofte. Learning for video compression with recurrent auto-encoder and recurrent probability model. *IEEE Journal of Selected Topics in Signal Processing*, 15(2):388–401, 2020.

[106] Hyunho Yeo, Hwijoon Lim, Jaehong Kim, Youngmok Jung, Juncheol Ye, and Dongsu Han. NeuroScaler: neural video enhancement at scale. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 795–811, 2022.

[107] Huanhuan Zhang, Anfu Zhou, Yuhan Hu, Chaoyue Li, Guangping Wang, Xinyu Zhang, Huadong Ma, Leilei Wu, Aiyun Chen, and Changhui Wu. Loki: improving long tail performance of learning-based real-time video adaptation by fusing rule-based models. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, MobiCom '21, page 775–788, New York, NY, USA, 2021. Association for Computing Machinery.

[108] Huanhuan Zhang, Anfu Zhou, Jiamin Lu, Ruoxuan Ma, Yuhan Hu, Cong Li, Xinyu Zhang, Huadong Ma, and Xiaojiang Chen. OnRL: improving mobile video telephony via online reinforcement learning. In *Proceedings of the 26th Annual International Conference*

on Mobile Computing and Networking, MobiCom '20, New York, NY, USA, 2020. Association for Computing Machinery.

[109] Junzi Zhang, Jongho Kim, Brendan O'Donoghue, and Stephen Boyd. Sample efficient reinforcement learning with REINFORCE. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 10887–10895, 2021.

[110] Kaidong Zhang, Jingjing Fu, and Dong Liu. Flow-guided transformer for video inpainting. In *European Conference on Computer Vision*, pages 74–90. Springer, 2022.

[111] Kaidong Zhang, Jingjing Fu, and Dong Liu. Inertia-guided flow completion and style fusion for video inpainting. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 5982–5991, 2022.

[112] Qing Zhang and Guizhong Liu. Error resilient coding of H. 264 using intact long-term reference frames. 2008.

[113] Zenghua Zhao and Shubing Long. RD-Based Adaptive UEP for H. 264 Video Transmission in Wireless Networks. In *2010 International Conference on Multimedia Information Networking and Security*, pages 72–76. IEEE, 2010.

[114] Anfu Zhou, Huanhuan Zhang, Guangyuan Su, Leilei Wu, Ruoxuan Ma, Zhen Meng, Xinyu Zhang, Xiufeng Xie, Huadong Ma, and Xiaojiang Chen. Learning to coordinate video codec with transport protocol for mobile video telephony. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[115] Jie Zhou, Bo Yan, and Hamid Gharavi. Efficient motion vector interpolation for error concealment of H. 264/AVC. *IEEE Transactions on Broadcasting*, 57(1):75–80, 2010.

[116] X Zhu, P Pan, M Ramalho, S Mena, P Jones, J Fu, S D'Aronco, and C Ganzhorn. Nada: A unified congestion control scheme for real-time media, draft-ietf-rmcat-nada-02. *Internet Engineering Task Force, IETF*, 2016.

[117] Xutong Zuo, Yong Cui, Xin Wang, and Jiayu Yang. Deadline-aware Multipath Transmission for Streaming Blocks. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 2178–2187. IEEE, 2022.

A Details of NVC architecture and training

A.1 More details on GRACE's NVC model

Grace uses the exact same model architecture as the original DVC model [73]. With an RGB input image of size $C \times H \times W$, where H, W are the height and width of the image, and $C = 3$ is number of channels in RGB images, the encoder neural network will encode the image into a compressed motion vector of size $128 \times (H/16) \times (W/16)$ and a compressed residual of size $96 \times (H/16) \times (W/16)$. Then those two compressed features will be quantized and converted into bytesteam using entropy encoding.

When we finetune the DVC model to get our GRACE's loss resilient model, we train on the 90k Vimeo Dataset, with batch size of 4, learning rate of 10^{-4} and learning rate decay of 0.1, and an Adam optimizer.

A.2 Making GRACE trainable

Since P is a non-differentiable random function, the gradient of the expectation of D in Eq. 2 cannot be directly calculated. To address this issue, we use the REINFORCE trick [62] for reparameterization. First, given the packet loss distribution $P(\mathbf{y})$, we can apply the differentiation property of logarithms to get

$$\nabla_\phi P(\mathbf{y}) = P(\mathbf{y}) \nabla_\phi \log P(\mathbf{y})$$

Therefore, our gradient of the expectation of $D(g_\theta(\mathbf{y}), \mathbf{x})$ becomes

$$\begin{aligned} &\nabla_\phi \mathbb{E}_{\mathbf{y}\sim P(\mathbf{y})}([D(g_\theta(\mathbf{y}), \mathbf{x})]) \\ &\qquad = \mathbb{E}_{\mathbf{y}\sim P(\mathbf{y})}([D(g_\theta(\mathbf{y}), \mathbf{x}) \nabla_\phi \log P(\mathbf{y})]) \end{aligned} \quad (3)$$

which can be estimated using Monte-Carlo sampling $\approx \frac{1}{N}\sum_{i=1}^{N} D(g_\theta(\mathbf{y}_i), \mathbf{x}) \nabla_\phi \log P(\mathbf{y}_i)$. Since in our application, the loss is an independent and identically distributed random variable, the gradient evaluates to either 0 or 1, hence we propagate the gradients for the encoder only for $D(g_\theta(\mathbf{y}_i), \mathbf{x})$ where $P(\mathbf{y}_i) = 1$.

B Realtime video framework for GRACE

B.1 Fast re-encoding and re-decoding under loss

In GRACE's NVC, the most time-consuming components are motion estimation NN and frame smoothing NN, taking 28% and 42% of the total encoding time respectively. Fortunately, we do not need to use them during resync (§4.2). When the packet loss feedback arrives at the encoder, it takes the following steps to generate a new reference frame to re-sync with the decoder (Assuming the loss feedback is for 6th frame and the encoder is about to encode 10th frame)

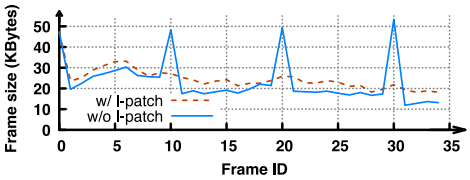


Figure 21: *Encoding each P-frame with a small I-patch leads to smoother frame sizes than naively inserting I-frames.*

- First, GRACE re-decodes the motion vector and residuals based on the packet loss feedback for 6th frame. This step needs to run the motion decoder NN and residual decoder NN, which only takes around 18% of the encoding time.
- Second, GRACE apply the cached motion vector and residuals of 7th frame on the "reconstructed" 6th frame to generate the "reconstructed" 7th frame. It applies the same logic on 8th and 9th frame and finally gets the "reconstructed" 9th frame. We do not run frame smoothing NN since the quality of the reference frame does not have a significant impact on compression efficiency. Therefore, this step does not involve any NN inference. It only needs to apply the motion and add the residuals, which takes 1% of the encoding time.
- Finally, GRACE uses the "reconstructed" 9th frame as the reference frame to encode the 10th frame. It is the same as encoding a frame when there are no packet losses. It will add an extra tag to the frame so that the receiver knows which reference frame to use.

To summarize, the encoder side's computational overhead is usually less than 10%. The logic requires the encoder to cache the motion vectors and residuals, but the cached value of frame x can be dropped after receiving the packet loss feedback of that frame.

At the receiver side, when receiving the frame with the extra tag, it will follow the same process as the second step above to generate the same "reconstructed" reference frame as the encoder. Again, the overhead is negligible since it does not require NN inference.

B.2 How GRACE handles I-frames

GRACE uses BPG [21] (also used in H.265) to encode and decode I-frames every 1000 frames. That said, in many NVCs (including DVC), the quality of P-frames will gradually degrade after an I-frame. By simply adding frequent I-frames (e.g., every 10 frames), we can achieve similar average compression efficiency with H.264 and H.265 when they use an optimal I-frame interval. However, since I-frames are larger than P-frames, adding too many I-frames causes frequent spikes in frame size. Instead, GRACE uses an *extra* small square-sized patch as a tiny I-frame, called *I-patch*, on *every P frame*. We split each frame into k patches, and for a window of k frames, each frame is sent with an extra I-patch at a different location, so I-patch "scan through" the whole frame every k frame. By default, $k = 30$ though we empirically found any value between 10 and 30 works well. With I-patch, GRACE does not need to send any I-frames (except the first frame). We use BPG [21] to encode/decode the I-patch. Figure 21 shows that when $k = 10$, I-patch mitigates the sudden size increase caused by I-frames.

It is worth noting that though I-patch encoding can also use a loss-resilient NVC, we do not protect their packet loss to simplify the system design. This is because if each patch will see an I-patch every k frames, so even if one patch is lost, its impact is confined to the next k frames, and empirically, even this impact is marginal since P-frames are still delivered.

B.3 Working with congestion control

GRACE can be integrated with any existing congestion control (CC) algorithms. When combined with GRACE, CC does *not* need to retransmit packets, unless no packets of a frame are received. CC determines the sending rate of packets and the target size of the next frame, while GRACE decides the content in each packet. Therefore, GRACE would not change the properties of the CC, such as fast convergence, oscillation avoidance, and TCP friendliness. In real-time video communication, traditional CC algorithms like GCC [31] typically mitigate packet losses by reducing bandwidth use, due to the non-loss-tolerant nature of conventional video codecs. These codecs necessitate retransmissions when packet loss happens, causing frame delays and video stalls. Conversely, GRACE is designed to handle packet losses by decoding the partially received frames with graceful quality. This capability allows GRACE to employ a more aggressive congestion control strategy, which, while resulting in occasional packet losses, enhances bandwidth utilization. An illustration of this approach can be found in Appendix C.7, where GRACE works with Salsify's congestion control (Sal-CC) [45] that yields a higher average sending rate albeit with increased packet loss.

B.4 Integration in WebRTC

GRACE is implemented with 3K lines of code, in both Python (mostly for NVC NNs) and C++ (for frame delivery and WebRTC integration). The code and trained model of GRACE will be made public upon the publication of this paper. The integration with WebRTC is logically straightforward since GRACE (including I-frame and P-frame encodings) exposes similar interface as the default codec in WebRTC.

We substitute the libvpx VP8 Encoder/Decoder in WebRTC with our GRACE implementation. When the sender encodes a frame, it parses the image data from the `VideoFrame` data structure (YUV format) into `torch.Tensor` (RGB format) and feed it into our GRACE encoder, which will return the encoded result as a byte array. Then the encoded bytes are stored into an `EncodedImage` (class in WebRTC) and sent through the network to the receiver as RTP packets. We modify the

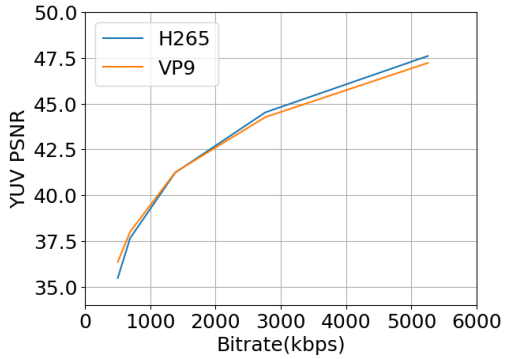


Figure 22: *H265 vs VP9 Encoding Efficiency on Kinetics*

built-in `RtpVideoStreamReceiver` (class in WebRTC) so that the receiver could flexibly decode the received packets even when not all the packets are received. When the receiver decides to decode the frame, it depacketizes the received packets into encoded data. Then it will use the GRACE decoder to decode the image into RGB format and then convert it back to YUV for displaying on the receiver side.

C Supportive details for GRACE's evaluation experiments

C.1 VP9 and H265 Comparison

In our paper we mainly compared with codecs in the H26x family. Since many prior work used VPx codec, we ran a simple experiment to show they have similar efficiency. We randomly chose 12 videos with resolution 1280x720 from the Kinetics dataset we used and compared encoding efficiency between VP9 and H265. We configured VP9 to use speed/quality tradeoff level 8 and set H265 to very-fast, zero-latency, and no B-frame. We confirm that they have similar performance as shown in Fig 22.

C.2 Baseline and testbed implementation details

We provide the extra implementation details of our baselines here:

- **Tambur**: To match the implementation in Tambur's paper [86], we force the codec to not encode any I-frames. Following recent work in real-time video coding [2, 7], we use the `zerolatency` option (no B-frames) and the `fast` preset of H.265. The command line we used to encode a video is `ffmpeg -y -i Video.y4m -c:v libx265 -preset fast -tune zerolatency -x265-params "crf=Q:keyint=3000" output.mp4` where Q controls the quality of the frame.
- **Error concealment**: We employ ECFVI [59], an NN-based error concealment pipeline, to mitigate errors from packet losses with H.265 encoding/decoding. When an incomplete frame is received, it starts a 3-step process to compensate for the errors. First, it uses a neural network to estimate the motion vector of the missing part from the previous N frames. Next, the missing pixel values are propagated from the reference frame using the estimated motion vector. Finally, an inpainting neural network is applied to enhance frame quality and minimize error propagation. We set $N = 5$ during our evaluation.

  ECFVI operates under the assumption that packet loss only corrupts portions of a frame, leaving the rest part (corresponding to the arrived packets) decodable. However, as discussed in §4.1, a single packet loss typically renders an entire frame undecodable in H.264/H.265. To reconcile this, we use flexible macroblock ordering (FMO) technique within the underlying H.265 video codec. This allows different parts of a frame to be encoded and packetized independently into distinct packets. In our baseline implementation, the frame is partitioned into 64×64-pixel blocks and randomly mapped to various packets during packetization. This method introduces a size overhead, as the codec cannot eliminate redundancy among packets. Based on prior works [64, 74, 99], we account for an additional 10% size overhead to ensure that each packet is individually decodable.

  ECFVI is chosen as the baseline for error concealment for two main reasons: *(i)* Its 3-step method is recognized as state-of-the-art within the computer vision research area. It surpasses the prior works that only do motion estimation [87] or inpainting [34]. *(ii)* Similar methods have been adopted by various recent works such as [46], [69], and [111], while ECFVI ranking as the most proficient among them. *(iii)* ECFVI's performance is also on par with or better than other recent error concealment techniques, including those utilizing transformers [71, 110].
- **Voxel** (selective frame skipping): We sort the video frames by the SSIM drop caused by skipping the frame (in real-time video communication, we usually cannot get the quality drop caused by skipping frames in advance. Thus, we are making an idealized assumption that improves the baseline). For 25% frames with the lowest SSIM drop, we use the default error concealment method in H.264/AVC [115] without any packet retransmission, and for the remaining frames (which cause more SSIM drops when skipped), we retransmit all the lost packets. We use a GoP (chunk length) of 4 seconds, which is also used by Voxel.
- **Salsify** (functional codec): We implement the Salsify codec based H.265 with the following two key features: firstly, the encoded frame size never surpasses the target bitrate determined by the underlying congestion control algorithm; secondly, upon packet loss, the encoder can dynamically select a reference frame, enabling subsequent frames to be decoded without resending any packets.

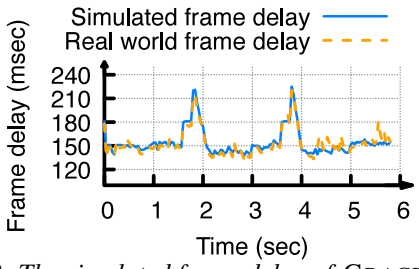


Figure 23: *The simulated frame delay of* GRACE *is close to the real world measured frame delay*

C.3 Simulator validation

Our simulator runs on an Ubuntu 18.04 server with 2 Intel Xeon 4210R CPU, and 256GB memory, with 2 Nvidia A40 GPUS. To validate that the frame delay measured in simulation matches the real-world numbers, we run a real-world emulation using GRACE. Being the same as simulation, we use 2 Nvidia A40 GPUs, one for encoding and one for decoding. The encoder process encodes the video using GRACE's encoder and send the encoded packets through an emulated network. The decoder process decodes the frame using the same logic as mentioned in §4. We compute the real-world frame delay by calculating the difference between the encoding time and the decoding time of a frame. Figure 23 compares the simulated frame delay and real-world measured frame. We use the bandwidth trace same as Figure 16. The result validates that our simulated frame delay is accurate. It is worth noting that we are running real encoding and decoding process in the simulation, hence the calculated frame quality should also be the same as using GRACE in the real world.

C.4 Distribution of video content complexity

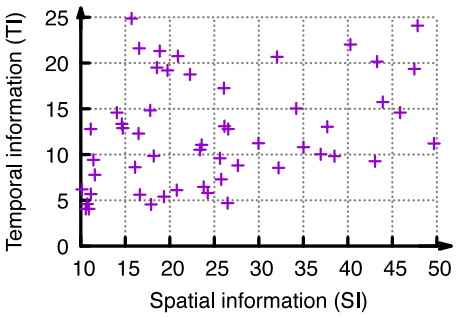


Figure 24: Spatial information (SI) and temporal information (TI) of test videos

To validate the test videos that we use cover different content complexities and movements, we calculate the spatiotemporal complexity of the video. We use Spatial Information (SI) and Temporal Information (TI) [58], which are frequently-used metrics to measure the spatiotemporal complexity and a larger SI/TI means that the video has a higher spatial/temporal complexity. The metrics are calculated by the tool [12] provided by Video Quality Experts Group (VQEG) and the result is shown in Figure 24.

The result validates that *(i)* the spatiotemporal complexity of the videos we used covers a wide range: SI is ranging from 15 to 85 and TI is ranging from 3 to 25. *(ii)* Our test videos covers all the following types: high spatial complexity and high temporal complexity, high spatial complexity but low temporal complexity, low spatial complexity but high temporal complexity, and low spatial complexity and low temporal complexity.

C.5 Illustration example where GRACE performs poorly

In some rare cases, GRACE may suffer from poor quality. Figure 25 visualizes an example of four consecutive frames when GRACE performs poorly. As shown in the yellow box, the frame decoded by GRACE has some notable artifacts around the moving object, which degrades the SSIM.

C.6 Screenshot of videos we used for user study

Figure 26 shows the screenshot of the videos we used for the user study (in §5.3)

C.7 Working with other congestion control

GRACE can also work with the congestion control algorithm proposed in Salsify (Sal-CC) [45], which is more aggressive than GCC. Sal-CC has a higher average sending rate, while paying the cost of potentially having more packet losses. Figure 27 show that changing from GCC to Sal-CC increases the average SSIM of 0.7-1.1dB for GRACE with a negligible increase in video stall ratio. In contrast, the video stall ratio for Salsify codec will increase a lot when using Sal-CC, because Salsify codec needs to keep skipping frames for more than one RTT when packet loss happens, which leads to frequent video stalls.

C.8 Working with super resolution

In line with the discussion in §2.2, Super-Resolution (SR) can supplement the receiver-side video quality. We employed SwinIR [70], a leading SR model, in our simulation to confirm that GRACE, like baselines, can also leverage SR benefits. Our experiments demonstrated that SR boosts receiver-side quality for all codecs, irrespective of the specific codec employed. For more details, refer to Appendix C.8.

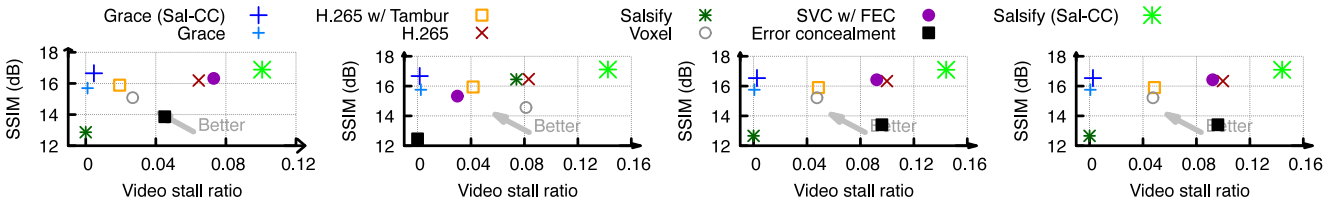
Figure 28 shows the tradeoff between quality and video stall ratio when using SR to enhance the quality at the receiver side. We run the simulation using LTE traces with a 100ms one-way delay and a 25-packet queue and then use a state-of-the-art SR model, SwinIR [70], to improve the quality of the decoded videos. When using SR, GRACE can still have on-par SSIM with Salsify codec and H.265 w/ Tambur: the

Figure 25: An example where GRACE performs poorly. It shows four consecutive decoded frames where the pink brush moves down quickly. Some artifacts in The yellow box degrade the frame quality and impact the SSIM.

(a) Sports: Football (b) Sports: Soccer (c) Gaming: Fortnite (d) Gaming: Genshin Impact

(e) Daily movement: Taekwondo (f) Daily movement: Baby in crib (g) Talking heads: indoor (h) Talking heads: outdoor

Figure 26: Summary of videos used in our user study. They span four categories: sports (a, b), gaming (c, d), daily movement (e, f), and talking heads (g, h).



(a) *One-way delay = 50ms* (b) *One-way delay = 75ms* (c) *One-way delay = 100ms* (d) *One-way delay = 150ms*

Figure 27: *End-to-end simulation result under different one-way delay. Network queue length = 25 packets*

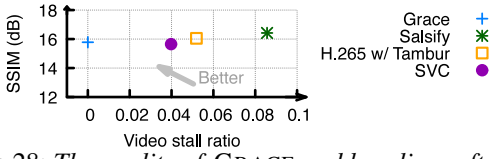


Figure 28: *The quality of* GRACE *and baselines after super-resolution*

	Encoding (ms)		Decoding (ms)	
	720p	480p	720p	480p
GRACE-Lite	35.1	17.2	40.9	21.6

Table 2: *Encoding/decoding time per frame for* GRACE*-Lite on Intel CPU*

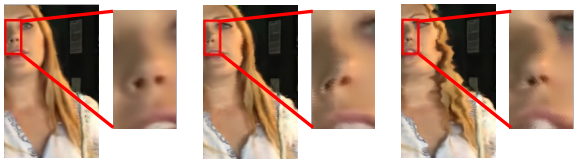
SSIMs are 15.8 dB, 16.4 dB, and 16.0 dB respectively. The SSIM of SVC (15.4 dB) is still lower than GRACE even with super-resolution. This is because packet loss can make higher layers of SVC undecodable, resulting in lower quality. This shows SR technique is complementary to our work, as it can improve the quality for any codecs at the receiver side.

C.9 Encoding/decoding time on CPU

We use OpenVINO library to run GRACE on a 32-core Intel(R) Xeon(R) Silver 4210R CPU. Table 2 shows the encoding/decoding time of a 720p/480p frame respectively. It can encode/decode a 720p frame at 28.5 fps and 24.4 fps respectively.

	SSIM (dB)	% of non rendered frames	Video stall ratio
GRACE	15.53	0.21	0.0011
GRACE-Lite	15.01	0.22	0.0012
GRACE-D	13.91	0.24	0.0014
GRACE-P	12.53	0.33	0.0023

Table 3: *End-to-end simulation shows* GRACE*-Lite has the same benefits in video realtimeness/smoothness compared to* GRACE *with marginal quality drop. Although* GRACE*-D and* GRACE*-P have similar video realtimeness/smoothness as* GRACE*, they suffer from low video quality.*



Grace
SSIM: 12.0 dB

Grace-D
SSIM: 10.7 dB

Grace-P
SSIM: 8.8 dB

Figure 29: *Comparing reconstructed image when the same packet loss is applied to the pre-trained NVC (*GRACE*-P), a variant with only decoder fine-tuned with loss (*GRACE*-D), and* GRACE *(both encoder and decoder jointly fine-tuned).*

C.10 Simulation results and visualization examples for GRACE-Lite, GRACE-P and GRACE-D

Table 3 shows the end-to-end simulation results comparing GRACE, GRACE-Lite, GRACE-D, and GRACE-P. We use the LTE traces, and set the one-way-delay to 100 ms and the network queue length to 25 packets. GRACE-Lite has both similar quality and realtimeness/smoothness as GRACE. Without jointly training the encoder and decoder with loss, GRACE-P and GRACE-D fail to achieve similar quality as GRACE.

Figure 29 visualizes the reconstructed frame of GRACE, GRACE-P, and GRACE-D when the same 50% packet loss is applied to the encoded tensor of the same image, demonstrating that by jointly training both the encoder and decoder under various packet losses, GRACE delivers the best reconstruction quality without any prominent artifacts, and achieves a high SSIM.

LiFteR: Unleash Learned Codecs in Video Streaming with Loose Frame Referencing

Bo Chen[1], Zhisheng Yan[2], Yinjie Zhang[1], Zhe Yang[1], Klara Nahrstedt[1]
[1]*University of Illinois at Urbana-Champaign,* [2]*George Mason University*

Abstract

Video codecs are essential for video streaming. While traditional codecs like AVC and HEVC are successful, learned codecs built on deep neural networks (DNNs) are gaining popularity due to their superior coding efficiency and quality of experience (QoE) in video streaming. However, using learned codecs built with sophisticated DNNs in video streaming leads to slow decoding and low frame rate, thereby degrading the QoE. The fundamental problem is the tight frame referencing design adopted by most codecs, which delays the processing of the current frame until its immediate predecessor frame is reconstructed. To overcome this limitation, we propose LiFteR, a novel video streaming system that operates a learned video codec with loose frame referencing (LFR). LFR is a unique frame referencing paradigm that redefines the reference relation between frames and allows parallelism in the learned video codec to boost the frame rate. LiFteR has three key designs: (i) the LFR video dispatcher that routes video data to the codec based on LFR, (ii) LFR learned codec that enhances coding efficiency in LFR with minimal impact on decoding speed, and (iii) streaming supports that enables adaptive bitrate streaming with learned codecs in existing infrastructures. In our evaluation, LiFteR consistently outperforms existing video streaming systems. Compared to the existing best-performing learned and traditional systems, LiFteR demonstrates up to 23.8% and 19.7% QoE gain, respectively. Furthermore, LiFteR achieves up to a 3.2× frame rate improvement through frame rate configuration.

1 Introduction

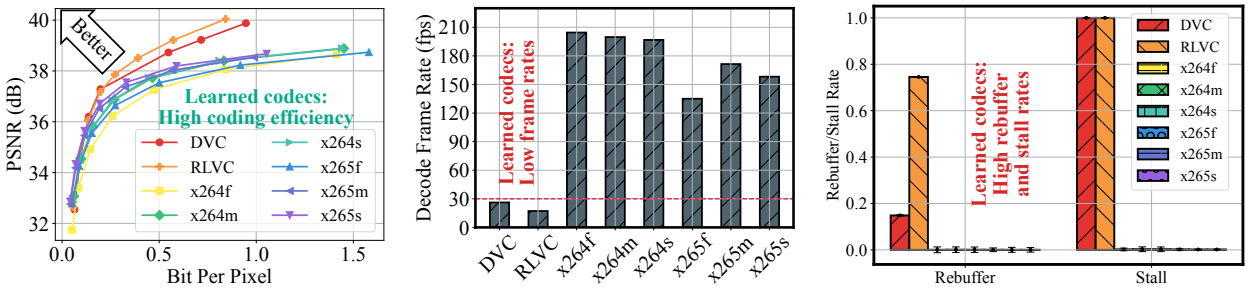
The video streaming industry has grown rapidly in recent years, with revenues of $72.2 billion in 2021 and expected to reach $115 billion by 2026 [20]. To achieve high-quality video streaming with minimal bandwidth usage, an essential component is the video codec, which compresses the video data while maintaining its visual fidelity.

Traditional codecs, such as AVC (x264) [57], HEVC (x265) [52], and MPEG-2 [33], are widely used and built on handcrafted modules such as block-based motion estimation, discrete cosine transform, and entropy coding. Recently, *learned codecs*, built entirely on deep neural network-based modules and end-to-end optimized, have been introduced and have demonstrated superior *coding efficiency*, i.e., the ability to encode videos with low bitrates while maintaining video quality [2, 25, 38, 39, 62, 63]. As a result, learned codecs have the potential to offer a better quality of experience (QoE) in video streaming.

Despite the potential, the use of existing learned codecs in video streaming faces major limitations. The complicated neural processing incurs a slow decode speed at the video player, causing a low frame rate and QoE during video playback. Our results (§2) show that, on a *GPU*, the frame rate of systems using learned codecs is still one magnitude lower than that of systems using traditional codecs on a *CPU*. On many hardware configurations, learned codecs cannot reach a real-time frame rate for smooth streaming. While the QoE in systems using traditional codecs is mainly affected by slow video downloads, we discover that those using learned codecs can suffer additionally from slow decode speed.

The root cause of this limitation is the *tight frame referencing* (TFR) principle that has been in use for decades in traditional video codecs. This principle encodes and decodes each frame by referencing the immediate predecessor frame. For example, frame#1 is referenced by frame#2, and frame#2 is referenced by frame#3. It minimizes the difference between the current frame and the reference frame and makes coding efficient. However, the processing of the current frame must be delayed until the processing of its reference frame is completed. As traditional codecs are highly optimized in computation complexity [52, 57], they can work with TFR with a reasonable frame rate. In contrast, TFR incurs an unacceptable frame rate in learned codecs with computation-intensive deep neural networks (DNNs). The frame dependency underlying TFR also impedes the possibility of parallel frame processing.

Aiming at improving the frame rate and achieving high QoE in video streaming, we propose to operate the learned



(a) Learned codecs has better coding efficiency. (b) Learned codecs are slow in decoding. (c) Learned codecs rebuffer and stall.

Figure 1: The potential and limitation of applying learned codecs in video streaming systems.

codec with *loose frame referencing* (LFR). In contrast to TFR, the reference of a frame in LFR only needs to be a temporally close frame, instead of the immediate predecessor. The rationale is that similarity exists between temporally close video frames, which are not necessarily adjacent. As such, frames with the same reference can be processed in parallel, which improves the frame rate of learned video codecs. More importantly, the similarity between a frame and its reference with LFR still preserves the coding efficiency.

As TFR has been the de facto design for both traditional and learned codecs that maximizes coding efficiency, implementing LFR in video streaming presents three challenges. First, optimally balancing the coding efficiency and decoding speed with LFR is non-trivial. Additionally, the reference frames in LFR could lead to significant memory consumption. Second, the existing architecture of the learned codec cannot handle the degradation in coding efficiency caused by LFR. Third, the existing streaming infrastructure is compatible with the learned codec regarding bitrate adaptation, frame rate configuration, and buffer level.

We design LiFteR, a novel on-demand video streaming system that leverages a learned codec with LFR, to address the above challenges. First, we introduce the *LFR video dispatcher* that routes frames to the learned codec iteratively. It strikes a balance between coding efficiency and decoding speed by defining frame dependency based on a binary tree and pre-order traversal. The memory consumption of it is constrained by processing frames in a GOP on a per-tree basis. Second, we design a unique LFR learned codec. It improves coding efficiency by exploiting the inter-frame correlation presented by LFR with self-attention. More importantly, it imposes minimal impact on the frame rate due to highly parallelized motion estimation. Finally, we adopt bitrate-adaptive training, frame rate configuration, and an enhanced adaptive bitrate (ABR) algorithm to bridge the gap between learned codecs and modern streaming infrastructure.

We compare LiFteR to video streaming systems built on different traditional codecs (x264 and x265) and learned codecs (DVC [39] and RLVC [63]). Results show that LiFteR consistently outperforms other streaming systems on different GPU capabilities and network conditions. Compared to the best-performing baselines using learned and traditional codecs, LiFteR achieves up to 23.8% and 19.7% QoE gain, respectively. We also demonstrate the capability of LiFteR to boost system frame rate by up to 3.2× via LFR on everyday GPUs. In summary, the contributions of this work are as follows.

- We identify the limitations of applying learned codecs in video streaming and propose the LFR paradigm.
- We build LiFteR to showcase a practical system through the design of the LFR video dispatcher, the LFR learned codec, and streaming supports for the learned codec.
- We evaluate LiFteR to demonstrate its multi-fold benefits in frame rate, video quality, and rebuffer rate.

2 Background and Motivation

Adaptive video streaming has been predominantly used. At the server, a video is segmented and encoded into different bitrates in advance. A client-side ABR algorithm [51, 61] adaptively downloads video segments of the appropriate bitrate level based on network conditions. The received segments are then decoded for playback. Studies have shown that the QoE of video streaming [51] is primarily determined by video quality Q, measured by PSNR, and playback smoothness, measured by the rebuffer rate R (the rebuffer duration divided by the video session duration). Therefore, the QoE metric can be formally defined as in Equation 1.

$$QoE = Q - \gamma R, \tag{1}$$

where γ is a parameter balancing Q and R. The value of γ in this paper is chosen according to BOLA [51].

2.1 Advantages of Learned Codecs

Video codecs play a crucial role in ensuring QoE in video streaming by improving the trade-off between video quality and bitrate. Though, traditional codecs like x264, x265, and MPEG [33, 52, 57] have been widely utilized in video streaming, the advancement in deep learning allows learned codecs [2, 38, 39, 63] to challenge the dominance of them. Unlike traditional codecs, learned codecs implement the codec

pipeline with DNN modules instead of handcrafted modules. These DNN modules are trained end-to-end to optimize video quality and bitrate.

In Figure 1(a), the coding efficiency of two learned codecs (DVC [39] and RLVC [63]) is compared to three *presets*, i.e., veryslow (s), medium (m), and veryfast (f), of traditional codecs, i.e., x264 and x265. For instance, x264f represents the codec x264 using the veryfast preset. The comparison is performed on the UVG dataset [42] with the same group of pictures (GOP) size as specified in §4. Each point on the figure represents a *rate-distortion* trade-off for a codec, where bits per pixel (*bpp*) is the rate metric and PSNR is the distortion metric. A rate-distortion curve closer to the top-left side of the figure indicates higher coding efficiency, thus demonstrating the superiority of learned codecs. For example, DVC achieves a similar PSNR (38.73 dB) as x265s (38.68 dB), the best-performing traditional codec, while reducing the *bpp* by almost half (0.55 vs. 1.05 *bpp*). Similarly, RLVC improves the PSNR by 1 dB compared to x265s (39.22 dB vs. 38.20 dB) with a similar *bpp* (0.57 vs. 0.59 *bpp*).

2.2 Learned Codecs Are Slow In Decoding

Despite the benefits of learned codecs, they exhibit a relatively low decode frame rate due to the computation overhead of DNNs, which may lead to rebuffering or stall and affect QoE in video streaming. For simplicity, we will use the term "frame rate" whenever referring to the decode frame rate throughout the remainder of this paper. To assess such impacts, we developed a video streaming prototype with various traditional and learned codecs (see §5 for a detailed setup). Our rate adaptation algorithm was BOLA [51], a widely used industrial-level ABR algorithm. We used the UVG [42] and MCL-JCV [56] video datasets, along with 1,000 traces from the FCC broadband network data [23] to emulate the streaming environment. The video client utilizing traditional codecs and learned codecs was run on an Intel Core i9-8950HK CPU and an NVIDIA GTX 1080 Ti GPU, respectively.

Figure 1(b) shows that, despite the advantage of the hardware, the frame rates of systems using learned codecs remain one order of magnitude lower than traditional codecs. It is worth noting that the "veryfast" preset in x265 typically yields a faster frame rate compared to the "medium" and "veryslow" presets. However, the impact of presets may vary based on distinct video contents and configurations, such as quantization parameters and resolution. Consequently, scenarios may arise where x265f does not achieve a faster frame rate than x265m and x265s, as depicted in Figure 1(b). Figure 1(c) compares the QoE of systems via rebuffer rate and stall rate (the number of segments that cannot be played immediately after its previous segment, divided by the total number of segments). The rebuffer rates of DVC (0.15) and RLVC (0.75) are drastically higher than those of traditional codecs, indicating the videos are freezing 15% and 75% of the time.

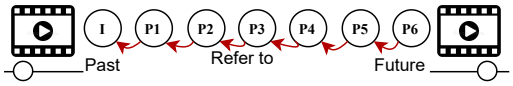


Figure 2: The frame processing dependency in today's codecs.

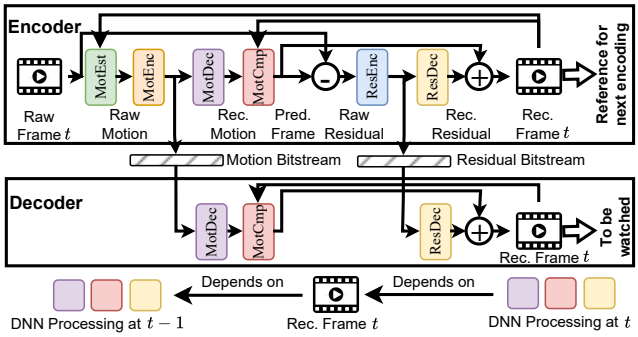


Figure 3: The canonical pipeline of learned codecs. This iterative pipeline means DNN processing at timestamp t is dependent on that at timestamp $t-1$.

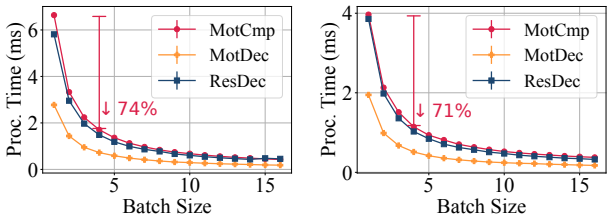
The stall rate of 1.0 for both learned codecs implies that video playback freezes for every segment, as the decoding frame rate is lower than the target frame rate of the source of 30 fps.

2.3 Preliminary Analysis of Learned Codecs

Background. Similar to traditional codecs, a video is divided into GOPs in learned codecs for encoding/decoding. Typically, the processing of a GOP starts from an I frame (intra-coded) that can be processed by an image codec. As depicted in Figure 2, all other frames in the GOP are P frames (predicted) and undergo predictive coding with the immediate predecessor frame as the reference frame, i.e., TFR. An alternate implementation inserts additional B frames (bi-directional) between I frame and P frames, referencing past and future frames [62]. Still, the fundamental frame referencing of I and P frames of such an implementation follows TFR.

The canonical pipeline for encoding a P frame at timestamp t, $t = 1, 2, ...$, with learned codecs, is illustrated in Figure 3 (top), involving four DNN-based modules: motion estimation (MotEst), motion encoder/decoder (MotEnc/MotDec), motion compensation (MotCmp), and residual encoder/decoder (ResEnc/ResDec). Initially, MotEst estimates the motion, a vector representing the displacement of each pixel, based on the current frame (Raw Frame t) and the reference frame (Reconstructed Frame $(t-1)$), the reconstructed immediate predecessor frame. MotEnc compresses the raw motion data into bitstreams and MotDec decompresses bitstreams into the reconstructed motion, representing the motion received by the video decoder. MotCmp then generates the predicted frame based on the reconstructed motion and the reference frame. The raw residual data between the predicted and raw current frame is then calculated via subtraction and processed by ResEnc and ResDec into the reconstructed residual. Finally, the reconstructed residual and the predicted frame are added to

produce the reconstructed current frame (Rec. Frame t). The decoding process partially follows the encoder pipeline in Figure 3 (top) that produces the current reconstructed frame (Rec. Frame t) from the bitstreams of the motion and residual.



(a) NVIDIA GTX 1080 Ti (b) NVIDIA RTX 2080 Ti

Figure 4: DNNs for decoding save time via parallelism.

The need and potential of parallelism. Unlike heavily-engineered modules in traditional codecs that can run fast enough to support real-time encoding/decoding, the sophisticated DNN modules in learned codecs are inherently slow. As a result, even a single iteration in Figure 3 pipeline can be time-consuming, resulting in a low frame rate.

Parallelism, which accelerates DNN by processing multiple instances of data simultaneously in a batch, is a potential solution for improving the frame rate of learned codecs, by processing frames in parallel. To demonstrate the potential of parallelism, we isolate DNN modules for decoding, i.e., MotCmp, MotDec, and ResDec, from learned codecs. Then, we separately measure the mean processing time per frame of these DNN modules for different batch sizes, ranging from 1 to 16 frames, on NVIDIA GTX 1080 Ti and RTX 2080 Ti. Figure 4 demonstrates the average processing time per frame when the batch size is one, the case with TFR, can be significantly improved by adopting a larger batch size. When the batch size increases from one to four, the speed of the motion compensation modules can be boosted by 74% and 71% on NVIDIA GTX 1080 Ti and RTX 2080Ti, respectively.

Parallelism is infeasible with TFR. Parallelism requires that the processing of different frames $t = 1, 2, \ldots$ is independent. For existing learned codecs with TFR, DNN processing of modules like MotDec, MotCmp, and ResDec at timestamp t happens after the reconstructed frame $t-1$. The reconstructed frame $t-1$, produced by DNN modules MotDec, MotCmp, and ResDec, happens after DNN processing of MotDec, MotCmp, and ResDec at timestamp $t-1$. As a result, the processing of MotDec, MotCmp, and ResDec at timestamp t depends on the processing at timestamp $t-1$, as illustrated in Figure 3 (bottom). Such a dependency contradicts parallelism.

2.4 Intuition and Challenges

Intuition. Our intuition is that frame similarity exists not only between adjacent frames but also between non-adjacent, temporally close frames. By leveraging the temporally close frames for reference in video coding, i.e., loose frame referencing (LFR), multiple frames can share the same reference frame. Therefore, these frames with the same reference can be processed in parallel, improving the frame rate. Meanwhile, the similarity between a frame and its reference, a temporally close frame preserves coding efficiency.

Challenges. Designing a learned video streaming system with LFR presents three main challenges:

1. **Processing pipeline:** It remains a question of how to configure LFR that 1) optimally balances the coding efficiency and decoding speed and 2) minimizes the memory usage due to buffering multiple reference frames.

2. **Learned codec:** LFR, by its design, introduces a larger difference between the raw and reference frames than TFR. The existing learned codec design cannot handle such discrepancies.

3. **Streaming infrastructure:** The existing video streaming infrastructure is incompatible with learned codecs, which exhibit different behaviors from traditional ones.

3 LiFteR Overview

The overview of LiFteR is illustrated in Figure 5, which comprises offline and online stages. In the offline stage, we construct the *LFR learned codec* (§3.2), a unique codec design that mitigates the impact of LFR by using an *elastic compression component* (§3.2.2) and maintains the decoding speed through *highly-parallelized motion estimation* (§3.2.1). Then, the codec is trained and configured before being deployed on the media server and the client.

After deployment, the server utilizes the *LFR video dispatcher* (§3.1) to route the raw video to the video encoder iteratively based on LFR. The LFR video dispatcher balances coding efficiency and decoding speed with a *dependency graph* (§3.1.1) and constrains memory usage via a *frame iterator* (§3.1.2). The video encoder compresses the raw video dispatched by the LFR video dispatcher into video segments of different bitrates, ready for Dynamic Adaptive Streaming over HTTP (DASH).

During the online stage, the client employs an ABR algorithm to download video segments with appropriate bitrates from the media server. These downloaded segments are routed to the decoder by the LFR video dispatcher and reconstructed into video frames. To integrate LiFteR into existing ABR streaming infrastructures, we build streaming supports that enable *adaptive bitrate* in training (§3.3.1), *frame rate configuration* (§3.3.2), and the *enhanced ABR algorithm* (§3.3.3).

3.1 LFR Video Dispatcher

LFR video dispatcher models frame processing dependency with the *dependency graph* (§3.1.1). Driven by the depen-

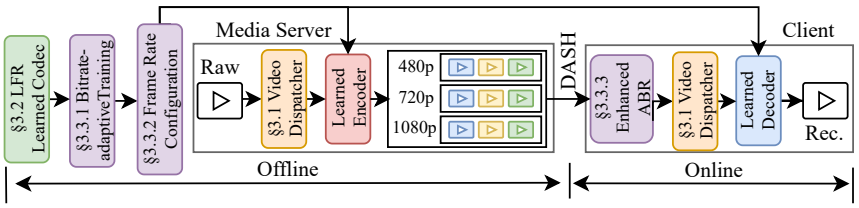


Figure 5: Overview of the video streaming pipeline in LiFteR.

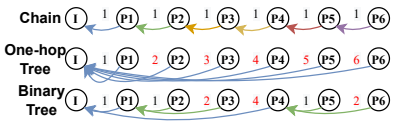


Figure 6: Frame reference structures. A binary tree balances reference cost and depth.

dency graph, it iteratively feeds frames to codec (§3.1.2).

3.1.1 Dependency Graph

In the dependency graph, each vertex with a unique positive integer label corresponds to a video frame processed at a particular timestamp, and the directed edge represents the dependency, pointing from a reference frame to the to-be-processed frame. Traversing the dependency graph is equivalent to processing video frames, which starts from a visited vertex representing the I frame. Then, graph traversal proceeds in iterations until all vertices are visited. In each iteration, we can visit vertices pointed by directed edges starting from visited vertices in previous iterations. The visited vertices in the same iteration represent frames that can be processed in parallel. As this paper focuses on the dependency of I and P frames (Figure 2) where a frame with a smaller timestamp is typically processed earlier, we constrain this dependency graph such that the end of a directed edge has a larger index than the start. We aim to minimize two metrics when deciding the shape and labeling of indices in the dependency graph.

1) Reference cost: The sum of the difference in indices between the end and start of directed edges, which indicates the difference between the frames and their reference in video coding and relates to coding efficiency.

2) Reference depth: The maximum depth of a vertex, which indicates the number of iterations needed to process all frames and relates to parallelism.

Shape: binary tree. It is challenging to balance the reference cost and depth. The dependency graph representing TFR has the shape of a chain. In Figure 6 (top), we provide a visual representation of the chain while also labeling index differences between frames and their references. Iterations are colored differently for clarity. The chain displays a low reference cost of 6 and a high reference depth of 6. Alternatively, we construct the dependency graph as a one-hop tree in Figure 6 (middle), which represents a fully parallelized way where all P frames reference the same I frame. This tree boasts a small reference depth (1) but has a high reference cost (21).

Our insight is that the binary tree as the dependency graph effectively balances the reference cost and depth. First, unlike the chain, the reference depth in a binary tree increases slowly with the number of vertices in the tree in a logarithmic manner. Second, in contrast to the one-hop tree, the reference cost is more constrained as one frame is used at most twice for frame referencing. While N-ary trees ($N = 2, 3, \ldots$), e.g., the 3-ary tree, may have similar properties as the binary tree, as the first effort to explore frame dependency with a tree, we focus on the binary tree for simplicity.

Labeling: pre-order traversal. The derivation of the dependency graph is essentially finding the Minimum Spanning Tree (MST) in the particular shape of the binary tree, termed Minimum Spanning Binary Tree (MSBT), within a dense graph. Assuming there are N frames, the dense graph consists of N vertices indexed by $0, 1, \ldots, N-1$. In this dense graph, there exists an edge between every pair of vertices, whose absolute difference in indices is the edge weight. It is important to minimize the reference cost in labeling.

Naively, we can construct the MST by leveraging the well-known Prim's algorithm to add vertices and edges in the dense graph into a tree starting with the vertex 0. However, the derived MST would have the shape of the chain like Figure 6 (top) instead of the binary tree. To tackle this problem, we modify the Prim's algorithm by skipping the addition of vertices and edges that 1) cause the MST to have a depth that is higher than a complete binary tree of N vertices, i.e., $\lceil \log_2(N+1) \rceil$ and 2) cause one vertex to have more than two children. The resulting algorithm is equivalent to assigning frame indices (from small to large) to vertices of a binary tree via pre-order traversal. Intuitively, this algorithm greedily minimizes the reference costs of both children of any vertex.

Theoretical analysis. Figure 6 (bottom) illustrates the dependency graph with a binary tree and pre-order traversal. Such a tree results in a reference depth (2) that is considerably smaller than that of the chain while using only half the reference cost of the one-hop tree, i.e., 6. In Figure 7, we delve further into how the reference cost and depth change as the number of frames in a GOP increases. The number of frames is expressed as a function of the full binary tree depth D. Our analysis reveals that the binary tree strikes a better balance between the frame rate (reference depth) and the coding efficiency (reference cost) and displays a more pronounced advantage when processing more frames.

3.1.2 Frame Iterator

With the dependency graph, a straightforward way is to include all frames of a GOP in it. However, a GOP can have a large number of frames, which requires processing more frames in parallel than a system can afford. To scale to arbi-

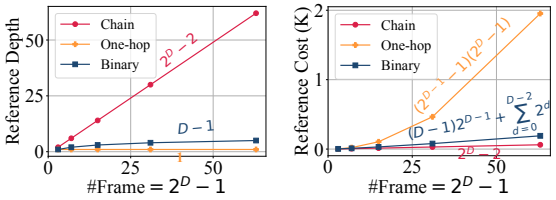


Figure 7: The binary tree achieves a better trade-off between reference cost and depth.

trary GOP sizes, we process frames in a GOP on a per-tree basis, as illustrated in Figure 8, where each tree is processed with three steps: slice, map, and iterate.

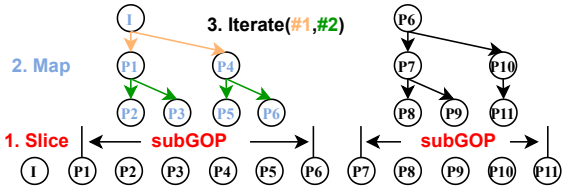


Figure 8: Video processing on a per-tree basis.

Slice: Frames within a GOP (excluding the I frame) are sliced into non-overlapping, consecutive *subGOPs*, each containing N sequential frames. Here, N is the subGOP size, which equals the number of non-root vertices in a full binary tree. For instance, full binary trees of depth $D = 2, 3, 4$ correspond to subGOP sizes $N = 2^D - 2 = 2, 6, 14$.

Map: Frames in each subGOP are mapped to the binary tree via pre-order traversal as shown in Figure 8. The root of a tree is the latest I or P frame preceding a subGOP. The reference of each frame is determined as the frame at its parent vertex. It is worth noting that the last subGOP might not have enough frames to fill all vertices in the full binary tree, which marginally affects the functionality of our approach.

Iterate: To achieve parallelism in the video encoding/decoding, the frames are processed level by level iteratively in the tree. The processing of frames at the same level of the tree is parallelized. For example, the codec first processes frames that reference the tree root (I frame), such as P1 and P4 in Figure 8. Then, it processes frames (P2, P3, P5, and P6) that reference the previously processed frames (P1 and P4).

3.2 LFR Learned Codec

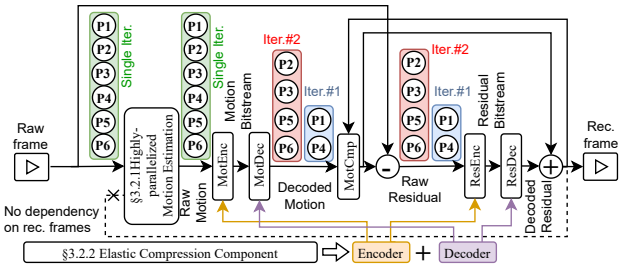


Figure 9: Design of LFR Learned Codec.

LFR learned codec (Figure 9) replaces motion estimation in canonical learned codec (Figure 3) with *highly-parallelized motion estimation* (§3.2.1) and encodes/decodes motion/residual of one or multiple frames via *elastic compression component* (§3.2.2).

3.2.1 Highly-parallelized Motion Estimation

In Figure 8, the processing of frames at the lower level (e.g., two frames at level 1 of the tree) are less parallelized than those at the higher level (e.g., four frames at level 2), which hinders the decoding speed. The fundamental problem is that DNN modules in the learned codec, i.e., motion estimation and compensation, rely on the reconstructed version of the reference frame(s).

Frame approximation. The key intuitions are that 1) the raw and reconstructed frames are similar and 2) motion estimation is performed only on the encoder. Therefore, it is possible to approximate (replace) the reconstructed reference frames with the raw reference frames with the same index in motion estimation. As such, the processing of all frames (e.g., frames P1, P2, ..., and P6 in Figure 8), relying on only raw frames, are parallelized. Such approximation allows us to speed up motion estimation, motion encoder, and motion decoder. More importantly, the coding efficiency is negligibly affected due to the frame similarity.

3.2.2 Elastic Compression Component

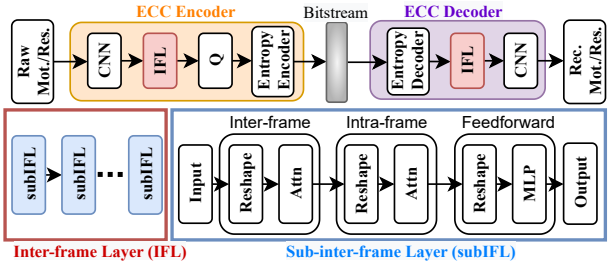


Figure 10: Elastic compression component design

As shown in Figure 10, the elastic compression component (ECC) comprises an encoder and a decoder. Following existing efforts on learned codecs [39, 63], the encoder utilizes a convolutional neural network (CNN) to transform the raw motions or residuals, into high-level features. Then, quantization and entropy encoding [58] are performed to compress quantized features into more compact bitstreams. In reverse, the decoder adopts entropy decoding to convert the bitstreams to quantized features, and a CNN to transform the features to the same shape as the raw motions or residuals. However, as the difference between a frame and its reference is increased by LFR, the coding efficiency could be compromised.

Inter-frame layer. We make two key observations. First, LFR expands the frame dimension of the input to DNN modules from one to multiple. The expanded dimension presents

inter-frame dependency in video coding that does not exist in existing learned codecs [38, 39, 63]. Such dependency presents an opportunity to enhance coding efficiency. Second, the self-attention module [55] is a generic learner of dependency adopted in numerous video-related tasks [9, 26, 44, 46] for spatial and temporal dependency.

Based on these observations, we design the *inter-frame layer* (IFL) that leverages inter-frame dependency in LFR with self-attention. The IFL involves sequentially connected sub-inter-frame layers (subIFL). Each subIFL is constructed with inter-frame, intra-frame, and feedforward blocks. Given common practice, the IFL is placed after CNN in the encoder and before CNN in the decoder.

The input dimension of the subIFL is $F \times C \times H \times W$, where F, C, H, and W represent the number of input frames, the number of channels, and the height and width of the feature, respectively. The output of the subIFL has the same dimension as its input. The inter-frame and intra-frame blocks exploit inter-frame and intra-frame dependency with a reshape function and the multi-head Attention (Attn) [55]. The input of Attn has three dimensions ($batch \times sequence \times feature$), which correlates data in the *sequence* dimension, with no correlation along the *batch* dimension. We reshape the input to shapes $HW \times F \times C$ and $F \times HW \times C$ for inter-frame (F) and intra-frame (HW) dependency, respectively. The last block converts the input shape to $BHW \times C$ and applies a multi-layer perceptron (MLP) consisting of two linear neural network layers to refine features.

3.3 Streaming Supports

We integrate our system into existing streaming infrastructure through *bitrate-adaptive training* (§3.3.1), *frame rate configuration* (§3.3.2), and *enhanced ABR algorithm* (§3.3.3).

3.3.1 Bitrate-adaptative Training

To leverage ABR algorithms that require multiple bitrates of the same segment, a learned codec must be configurable to encode and decode videos at different compression levels, which correspond to different trade-offs of bitrates and distortion. However, a typical learned codec cannot adjust the trade-offs of bitrates and distortion as easily as traditional codecs with the quantization parameter (QP) [52, 57]. A naive approach that prepares different versions of the codec of different compression levels is memory-consuming.

Compression-level embedding. To attain similar rate adaptability with learned codecs, we adopt a *compression-level embedding* approach [48]. In this approach, the compression level is treated as an input, in addition to video frames, to the learned codec. Specifically, multiple compression levels $l = 1$, 2, ..., are converted into one-hot vectors, spatially tiled, and concatenated to the input of DNN modules. This approach allows the trade-off of bitrates and distortion to be conveniently adjusted by changing the input, without storing multiple versions of the codec. Meanwhile, this approach minimally degrades the coding efficiency compared to a canonically trained learned codec.

3.3.2 Frame Rate Configuration

When working with a specific hardware platform, it is critical to attain a targeted frame rate. For canonical learned codecs, adjusting the frame rate may require compressing and retraining DNNs in the learned codec, which is time-consuming.

SubGOP probing. In LiFteR, this can be easily achieved by configuring the subGOP size. Specifically, we probe the target hardware with the LiFteR decoder using different subGOP sizes. Based on the frame rates achieved at various subGOP sizes, we select the smallest one that surpasses the targeted frame rate, e.g., 30 fps. Choosing a subGOP size than what is needed is not recommended, as it would consume more GPU resources while offering only marginal enhancements to coding efficiency (Figure 18).

3.3.3 Enhanced ABR Algorithm

The buffer level is a critical factor in the decision-making process of adaptive bitrate (ABR) algorithms, such as BOLA [51], which adjusts video quality according to network conditions. In video streaming systems, the buffer level represents the duration of video frames in the playback buffer. A low buffer level indicates low network bandwidth, causing the ABR algorithm to download segments of lower bitrates to avoid rebuffering. Conversely, a higher buffer level triggers the ABR algorithm to download segments of higher bitrates. However, for learned codecs, the buffer level does not accurately reflect network conditions and may mislead ABR algorithms for learned codecs. For instance, when the decoding rate of frames equals the consumption rate, the playback buffer remains almost empty, as each decoded frame is immediately consumed. This is likely to happen with learned codecs whose decoding rate is more comparable to the consumption rate than traditional codecs. As the buffer level is low, the ABR algorithm may download segments of low bitrates, even though the network bandwidth is abundant, leading to wastage.

Virtual buffer. To tackle this issue, we introduce the *virtual buffer*. It captures the length of received but unwatched segments, irrespective of whether they are stored in the replay buffer. By utilizing the level of the virtual buffer instead of the actual buffer level in the ABR algorithm, we avoid potential conservative decision-making by the ABR algorithm.

4 Implementation

Model choices and streaming configuration. We implement motion estimation using convolutional neural networks [47] and motion compensation using a warping function and an

interpolation network [39]. As per [9], we configure the attention mechanism's number of heads and each head's channel number to 8 and 64, respectively, and the number of sub-inter-frame dependencies (subIFL) to 12. During training, we set the subGOP size to six, which we empirically find to be sufficient for the high coding efficiency and frame rate of LiFteR. We utilize BOLA [51], an industrial-level ABR algorithm [19], as the ABR algorithm in LiFteR, which decides the bitrate every time the system tries to download a segment. The segment and GOP size are set to 5s following [21].

Training. We jointly optimize our model using the Vimeo-90k dataset [60]. In this dataset, every training sample consists of seven images, with the first image being the I frame ($k=0$) and the remaining images being P frames ($k=1,...K$). K corresponds to the number of P frames in a GOP during training. The I frame is encoded and decoded using an image codec, Better Portable Graphics (BPG) [7], while the learned codec compresses the P frames on a per-tree basis. We scale the training images' resolution to 256×256. We use the Adam optimizer [31] with a learning rate of 10^{-4}, which is reduced by a factor of 10 after convergence until 10^{-6}. Our loss function $\mathcal{L}$ is defined as follows.

$$\mathcal{L}=\sum_{k=1}^{K}\mathop{\mathbb{E}}_{l}[\lambda^{l}\mathcal{D}(x_k,\hat{x_k}^{l})+\hat{v_k}^{l}+\hat{r_k}^{l}]. \tag{2}$$

We set $\lambda^{l}=256,512,1024,2048,4096,8192,16384$ at different compression levels $l=0,1,...,6$, which cover bitrates from 1 Mbps to 16 Mbps. The estimated bits per pixel (bpp) of the motion or residual for the compression level l is denoted by $\hat{v_k}^{l}$ or $\hat{r_k}^{l}$. The distortion, measured by mean square error (MSE), between the raw frame x_k and the reconstructed frame at compression level l, $\hat{x_k}^{l}$, is denoted by $\mathcal{D}(x_k,\hat{x_k}^{l})$. In training, the compression level l in each sample is randomly and iteratively generated, following ELFVC [48].

5 Evaluation

We evaluate LiFteR regarding streaming performance, coding efficiency, and adaptability. Our highlights are

1. LiFteR achieves superior QoE compared with systems using learned and traditional codecs across different GPUs and network conditions (Figures 11-14).

2. LiFteR improves the rate-distortion trade-off of learned and traditional codecs (Figure 16).

3. LiFteR maintains a real-time frame rate across different GPUs (Figure 15) and improves the frame rate by adapting the subGOP size (Figure 17).

4. The component designs of LiFteR show effectiveness in contributing to the overall gain (Figures 18-22).

5.1 Methodology

Hardware setup. We conducted our experiments on Linux desktops featuring various NVIDIA GeForce GPUs and CPUs, which are listed in Table 1. The learned codecs are executed on all three Linux desktops, "1080", "2080", and "3090". Although the traditional codecs always run on "2080", the choice of the three hardware platforms has a minimal impact on the QoE for them.

Table 1: Hardware setup.

Name	GPU	CPU
1080	GTX 1080 Ti	Intel Core i9 @ 2.90GHz
2080	RTX 2080 Ti	Intel Core i7 @ 3.60GHz
3090	RTX 3090 Ti	AMD Ryzen 9 @ 4.95GHz

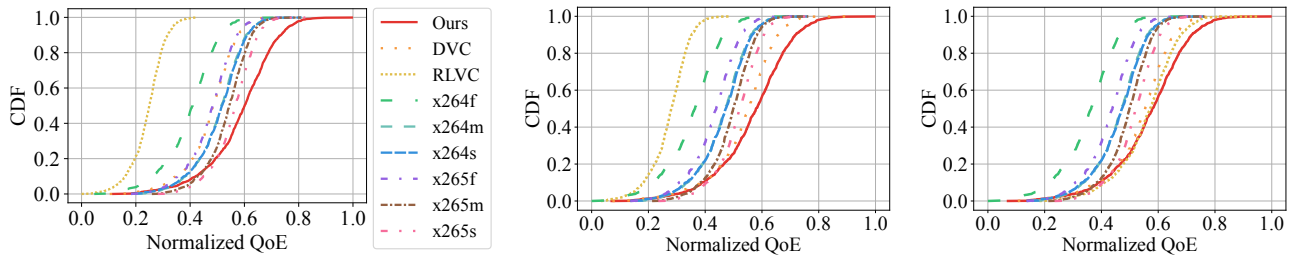
Network traces. We use network traces from FCC [23], randomly selecting 1,000 traces from two tests conducted for "video streaming" and "http get". These traces represent a range of diverse network scenarios. The "video streaming" and "http get" traces have an average bandwidth of 3.9 Mbps and 15.8 Mbps, respectively, indicating limited and adequate bandwidth.

Video datasets. We merged two video datasets, UVG [42] and MCL-JCV [56]. This unified dataset consists of 37 videos with a resolution of 2K, operating at 30 frames per second with a total runtime of around 5 minutes.

Baselines. In our evaluation, we compare LiFteR with video streaming systems that use state-of-the-art learned codecs (DVC [39] and RLVC [63]) and traditional codecs (x264 [57] and x265 [52]), all employing BOLA for rate adaptation with the same GOP and segment size as LiFteR. To test the traditional codecs, we use the FFmpeg [54] implementation and configure each codec into three modes: veryfast, medium, and veryslow. These modes are denoted by x264-veryfast (x264f), x264-medium (x264m), x264-veryslow (x264s), x265-veryfast (x265f), x265-medium (x265m), and x265-veryslow (x265s). Commands for the different modes can be found in Appendix A. To optimize PSNR, DVC and RLVC are configured in the same way as LiFteR. To ensure a fair evaluation, we encode videos for all baseline systems into seven bitrates, covering a range that is comparable to LiFteR's videos. Note that there are learned codecs [2, 48] that adopt alternate designs of motion estimation and compensation instead of that of DVC, RLVC, and ours, which are not included in the evaluation for fairness. However, as they still rely on TFR, we claim the performance gain achieved with LFR is applicable to them.

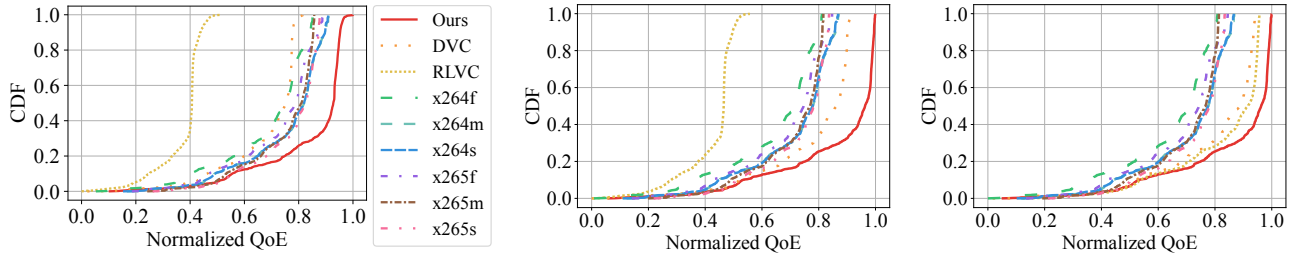
5.2 End-to-end performance

End-to-end QoE. Figures 11 and 12 illustrate that the overall QoE of LiFteR outperforms other baselines on network traces with limited ("video stream") and adequate ("http get") bandwidth, respectively. The QoE metric is calculated



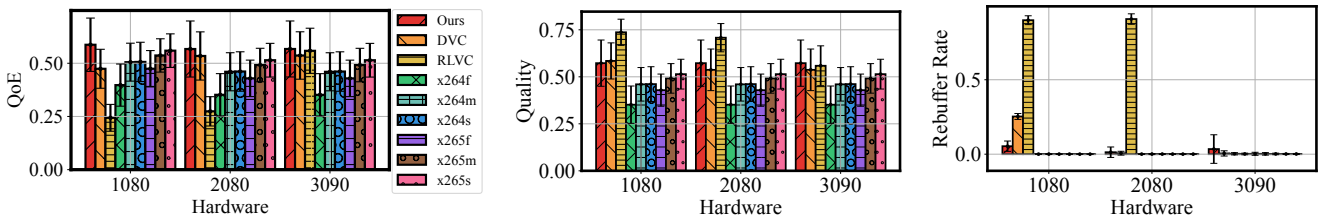
(a) NVIDIA GTX 1080 Ti (b) NVIDIA RTX 2080 Ti (c) NVIDIA RTX 3090 Ti

Figure 11: LiFteR shows consistent advantages in QoE across different hardware on the "video stream" traces.



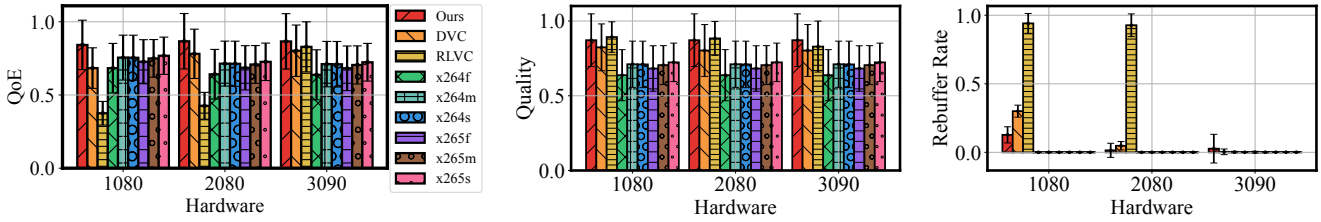
(a) NVIDIA GTX 1080 Ti (b) NVIDIA RTX 2080 Ti (c) NVIDIA RTX 3090 Ti

Figure 12: LiFteR demonstrates consistent advantages in QoE across different hardware on the "http get" traces.



(a) Norm. QoE (b) Norm. Video Quality (c) Norm. Rebuffer Rate

Figure 13: LiFteR's normalized QoE is 1.5%-23.8% and 5%-10.3% higher than the best-performing learned and traditional approaches, respectively, with limited bandwidth ("video stream" traces).



(a) Norm. QoE (b) Norm. Video Quality (c) Norm. Rebuffer Rate

Figure 14: LiFteR demonstrates 4.5%-23.2% and 9.6%-19.7% improvements in normalized QoE than the best-performing learned and traditional approaches, respectively, with adequate bandwidth (on "http get" traces).

based on Equation 1, normalized per hardware and trace, with the lowest and highest values mapped to 0 and 1. On the "video stream" traces with limited bandwidth, LiFteR's normalized QoE is 1.5%-23.8% and 5%-10.3% higher than the best-performing learned and traditional streaming systems, respectively (Figure 13(a)). On the "http get" traces with adequate bandwidth, the advantages in normalized QoE become 4.5%-23.2% and 9.6%-19.7%, respectively (Figure 14(a)), indicating that LiFteR better utilizes adequate bandwidth than other approaches. The performance of DVC and RLVC is significantly impacted by the hardware platform. DVC has the second-best performance on "2080" but one of the worst performances on "1080", whereas RLVC has the second-best performance on "3090" but the worst performance on "1080" and "2080". In contrast, LiFteR performs well on different hardware consistently.

QoE breakdown. To improve visualization, the metrics of video quality and rebuffer rate were normalized per hardware and trace, with the lowest and highest values being mapped to 0 and 1, respectively. In Figure 13(b) and Figure 14(b), the

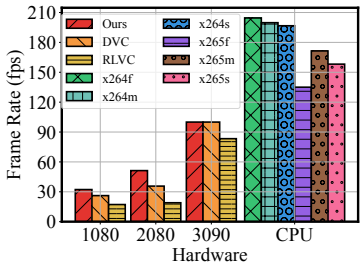


Figure 15: LiFteR decodes faster than other learned codecs.

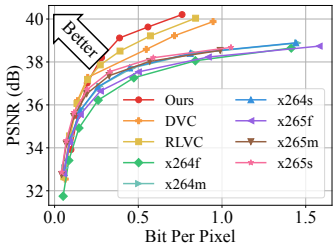


Figure 16: LiFteR exhibits the best coding efficiency.

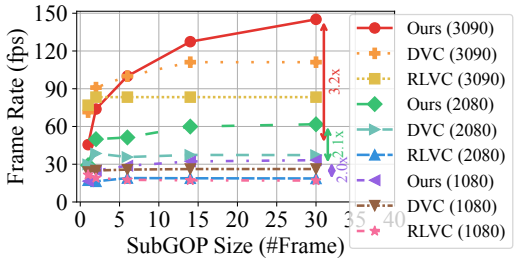


Figure 17: LiFteR can boost the frame rate by 3.2× via adaptation.

video quality of DVC and RLVC is mostly better than others because their virtual buffers are consumed slower or increasing, resulting in the ABR algorithm downloading segments with higher bitrates. Regarding the rebuffer rate (Figure 13(c) and Figure 14(c)), LiFteR significantly outperforms DVC and RLVC due to its real-time frame rate (Figure 15). For high-end hardware like NVIDIA RTX 3090 Ti, the rebuffer rate of LiFteR is lightly above others because it tends to wait for bitstreams of several frames before parallel processing. It causes rebuffering more easily than learned codecs with TFR (DVC and RLVC), which plays frames immediately after receiving them. Although this downside is outweighed by the benefits of LiFteR's superior frame rates on low-end hardware, it becomes noticeable for high-end hardware. Nevertheless, LiFteR's overall QoE is better than others due to its superior quality. We also notice a gap in rebuffer rate between our system and traditional systems, particularly on less powerful hardware. The reason is primarily the longer decoding latency of LiFteR than traditional codecs. It causes LiFteR to take more time on average to display the first frame after rebuffering. Nevertheless, the subsequent frames in LiFteR do not experience any additional delays. As a result, the QoE of LiFteR is not significantly affected by this fact.

Frame rate. In Figure 15, we compare the frame rates of systems using learned codecs on different GPUs and systems using traditional codecs on a CPU. LiFteR achieves consistent real-time frame rates across different hardware platforms. However, DVC and RLVC demonstrate worse frame rates than LiFteR on low-end hardware, such as "1080" and "2080". Despite optimizations in LiFteR, it does not outperform traditional video codecs, which are heavily engineered in speed.

Compression performance. In Figure 16, it is evident that LiFteR outperforms other baselines in terms of coding efficiency. Among the traditional systems, x265-veryslow performs the best, achieving a PSNR of 38.20 dB at 0.59 *bpp*. However, LiFteR achieves the same PSNR with less than half the bandwidth usage, requiring only 0.28 bpp. In comparison, the best-performing learned approach, RLVC, needs 0.39 *bpp* to achieve a PSNR of 38.50 dB, while LiFteR achieves a superior PSNR of 39.12 dB at the same *bpp*. LiFteR's superior coding efficiency stems from (i) the inter-frame dependency presented by LFR at the cost of increased GPU utilization, and (ii) ECC (§3.2.2), which effectively captures such dependency via the self-attention mechanism. These factors mitigate and outweigh the negative impact of the LFR video dispatcher (§3.1) and highly-parallelized motion estimation (§3.2.1), which potentially widen the gap between a frame and its reference.

Adaptability of LiFteR. LiFteR has the unique feature of adapting the subGOP size to achieve varying speeds. We evaluated LiFteR's performance across different subGOP sizes, including 2, 6, 14, and 30, corresponding to the full binary tree depths of 2, 3, 4, and 5, respectively. We also included a special case, "subGOP=1", representing LiFteR processing with TFR, similar to DVC and RLVC. In Figure 17, we can observe the frame rate of LiFteR as a function of the subGOP size and compare it with other learned approaches processing the same number of frames. The results indicate that LiFteR improves its speed by 2-3.2× as the subGOP size increases, while the other learned approaches' speed remains almost unchanged. It is worth noting that there is a slight increase in frame rates of other learned approaches. The reason is that the GPU is relatively slower when performing the first forward propagation. Then, it improves and gradually stabilizes for subsequent forward propagations. As a result, even if these approaches process frames one by one with TFR, their speed is slightly increased when the number of frames increases.

Figure 18 compares LiFteR's coding efficiency at different subGOP sizes. Initially, LiFteR is suboptimal when the subGOP size is 1 or 2. We speculate the time dimension of the input is not large enough to benefit from ECC for these small subGOP sizes. As the subGOP size increases, the coding efficiency improves and stabilizes. This result also indicates LiFteR's ability to adapt to larger subGOP sizes (14 and 30) than what it is trained with (6) with noticeable degradation.

5.3 Design Analysis

Impact of dependency graph and ECC. The dependency graph and ECC are the two key designs in LiFteR. To demonstrate their significance via comparison, we introduced three alternative designs, in addition to our original design ("Default"): 1) "w/o IFL": eliminating inter-frame dependency from the learned codec, i.e., removing IFL, 2) "Chain": substituting the binary tree in the dependency graph with a chain, and 3) "One-hop": substituting the binary tree in the depen-

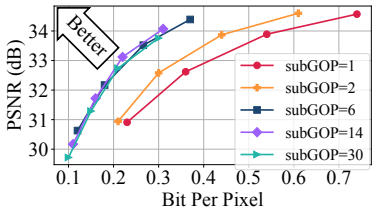


Figure 18: Coding efficiency of LiFteR initially improves with a larger subGOP size.

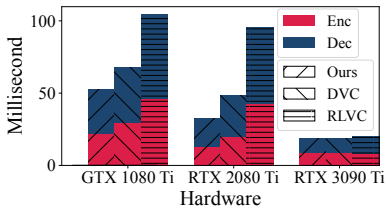


Figure 19: Time resource in learned codecs is generally allocated more to decoding than encoding.

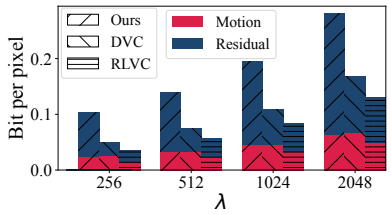
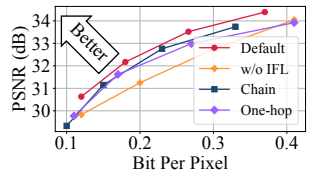
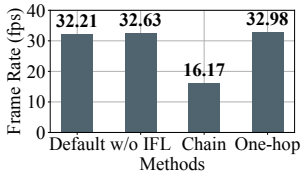


Figure 20: Bit resource in learned codecs is distributed more to residual than motion.

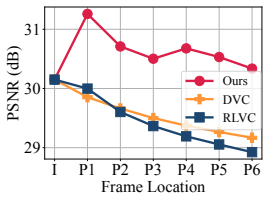


(a) Coding efficiency

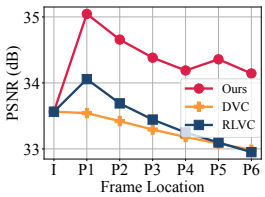


(b) Frame rate

Figure 21: Binary tree and inter-frame dependency optimally balance coding efficiency and frame rate.



(a) $\lambda = 256$



(b) $\lambda = 2048$

Figure 22: LiFteR's quality is more stable than others.

dency graph with a one-hop tree. We examine the impact of each design on coding efficiency and frame rate. For coding efficiency, Figure 21(a) shows both "Chain" and "One-hop" modes cause a PSNR drop of approximately 0.5 dB at the same *bpp*, compared to "Default". Additionally, the "w/o IFL" mode experiences a more significant loss of coding efficiency than the "Default" mode, as its PSNR drops by around one dB at the same *bpp*. These results highlight the importance of the combination of binary tree and inter-frame dependency in achieving satisfactory coding efficiency. Regarding frame rate, the "Default" mode has roughly twice the frame rate of the "Chain" mode, demonstrating a significant advantage of the binary tree. The "One-hop" mode exhibits a similar frame rate as the "Default" mode, indicating that is it not necessary to parallelize the processing of all frames to achieve substantial decode speed improvement. It is also found that the "w/o IFL" and "Default" modes have similar frame rates, which suggests that the inclusion of the inter-frame dependency has negligible impact on the frame rate. Overall, the binary tree and the leverage of inter-frame dependency strike the optimal balance of coding efficiency and frame rate for LiFteR.

Impact of the virtual buffer. To illustrate the significance of the virtual buffer, we remove it from LiFteR and perform streaming experiments with the "1080" hardware on two network traces. Figure 23 reports the average normalized QoE, video quality, and rebuffer rate. We observe a decline in both the learned approaches' QoE and video quality. Even when the bandwidth condition changes from limited to adequate, the quality and QoE of the learned approaches do not improve as significantly as the traditional approaches. This is because, without the virtual buffer, the real buffer in learned approaches remains consistently low, causing the ABR algorithm to download low-quality segments, irrespective of the bandwidth condition. Additionally, the downloaded segments have low bitrates, so the contrast in rebuffer rates between traditional and learned approaches is less pronounced than in Figure 13(c) and Figure 14(c).

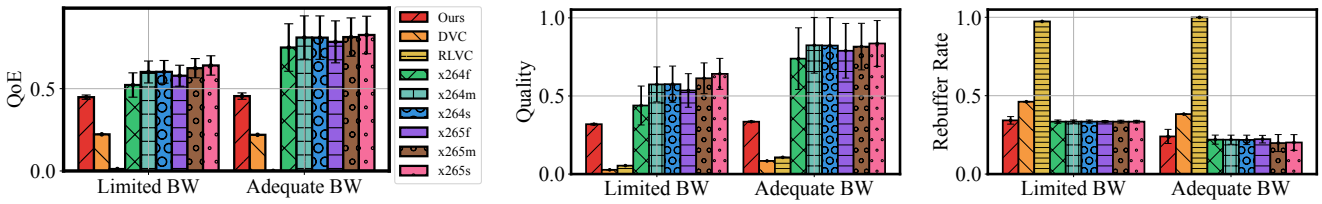
5.4 Micro Benchmark

Resource allocation. Time (computation time) and space (bitstream size) are two critical resources for learned codecs. We analyze the allocation of these resources in our approach, DVC, and RLVC using the UVG dataset. Figure 19 displays the time spent on encoding and decoding across various hardware. All approaches require more time for encoding than decoding, as the decoder modules are a subset of the encoder modules. Figure 20 illustrates the allocation of bits for motions and residuals over λ values of 256, 512, 1024, and 2048. We observe that our approach allocates a higher percentage of bits to the residual bitstream than the other approaches. We speculate that this is because LFR increases errors in ME and MC, thereby necessitating more bits in the residual to compensate for those errors.

Error propagation. In Figure 22, we visualize the quality of frames reconstructed by our codec, DVC, and RLVC, based on their position in a seven-frame GOP using the UVG dataset ($\lambda = 256$ and 2048). For our codec, we set the subGOP size to 6. As the temporal distance between P and I frames increases, the PSNR of the P frame monotonically decreases for DVC and RLVC, which could lead to degradation of QoE when viewing frames distant from the I frame, particularly with large GOP sizes. In contrast, our approach with LFR can increase the PSNR of P frames even when their temporal distance to the I frame increases, potentially allowing for a smoother viewing experience.

6 Related Work

Video codecs. Video codec standards, such as MPEG-2 [33],



(a) Norm. QoE (b) Norm. Video Quality (c) Norm. Rebuffer Rate

Figure 23: Learned codecs do not outperform traditional codecs without the virtual buffer.

H.264 [57], and H.265 [52], rely on traditional handcrafted methods. Due to recent advancements in deep learning, researchers have started to replace operations in traditional video codecs with DNNs [1, 6, 12, 16, 18, 37, 40, 47, 59]. DNNs can also serve as hints to assist codecs in video analytics [13,14,22,36]. Besides, researchers have proposed learned codecs built purely from DNNs [2,25,38,39,48,62,63]. Another category of learned approaches integrates DNN and progressive coding [21]. However, all existing learned codecs employ a TFR principle, suffering from slow decoding and low QoE problems. LiFteR addresses the low frame rate problem in video streaming systems with learned codecs with LFR. Further, LiFteR can be easily integrated into these TFR-based learned codecs for improved decoding speed. Although learned codecs with speed optimization like ELFVC [48] may achieve near real-time performance, e.g., 30 fps, on certain hardware, they cannot meet the real-time requirements when the hardware degrades or the frame rate requirements increase to 60 fps or 120 fps. In contrast, LiFteR flexibly handles these variations by adapting the subGOP size.

Video streaming. Enhancing the quality of video streaming has been the focus of numerous techniques, typically falling into one of three categories: push-based, pull-based, and video super-resolution (VSR). Push-based strategies analyze playback statistics from clients and push the appropriate bitrate of videos to each client from a central server. Several studies [8,24,28,35] have investigated the effectiveness of these approaches. In contrast, pull-based strategies guide clients to download videos of appropriate bitrates from the server based on the predicted bandwidth or buffer level. Several studies, such as [41,51,61,65,67,68], have explored the efficacy of these strategies. VSR techniques can also be applied to improve video streaming quality, where super-resolution models are used to increase the video resolution of downloaded segments, enhancing QoE [30,64,66]. Our approach innovates the video codecs of video streaming systems and can be integrated into these orthogonal designs without requiring any changes to the network protocol or application details.

Parallelism in video codec. Parallel processing techniques have been widely exploited in video compression to circumvent the speed limit of processors. There have been sophisticated hardware designs that parallelize vector quantization [43,45], discrete cosine transform [17,29,53], variable length coding [11], and motion estimation [15, 27, 49]. Software-based techniques are categorized into spatial and temporal parallelism. Spatial parallelism [4,5,32] processes different regions in a frame concurrently. Temporal parallelism [3,50] allocates several video frames to each processor. In contrast to these works focusing on parallelism at the GOP level, we allow parallelism at the frame level.

7 Discussion

Hardware requirements. As LiFteR trades the GPU utilization for its frame rate via LFR, the GPU memory might limit its decode speed. However, our experiments have demonstrated that everyday GPUs are sufficient for LiFteR to achieve a frame rate over 30 fps (Figure 15). It is also feasible to further reduce the hardware requirements via neural network compression [10,34], which is orthogonal to our approach.
Applicable scenarios. As shown in Figure 19, the encoding time of LiFteR, like other learned codecs, is higher than that of decoding. However, in video-on-demand (VoD) streaming, the encoding speed is not critical since video segments are encoded before streaming. Therefore, LiFteR is highly suitable in the VoD streaming scenario.

8 Conclusion

Tight frame referencing has a long history of being adopted in video codecs. However, it is proven ineffective in video streaming with learned codecs, causing a low frame rate. To overcome its limitation, we design LiFteR, a video streaming system that employs a learned codec with loose frame referencing. Our experiments show that LiFteR delivers superior QoE compared to systems using existing traditional and learned codecs consistently.

Acknowledgments

We thank our shepherd Francis Yan and the anonymous reviewers for their valuable feedback, which greatly improves this paper. This work was supported by NSF under the award number NSF CNS 1900875, NSF CNS 2106592, NSF IIS 2140620, NSF IIS 2140645, NSF OAC 1835834, and NSF OAC 2144764 and Army Research Lab under the award number ARMY W911NF-17-2-0196.

References

[1] Eirikur Agustsson, Fabian Mentzer, Michael Tschannen, Lukas Cavigelli, Radu Timofte, Luca Benini, and Luc Van Gool. Soft-to-hard vector quantization for end-to-end learning compressible representations. *arXiv preprint arXiv:1704.00648*, 2017.

[2] Eirikur Agustsson, David Minnen, Nick Johnston, Johannes Balle, Sung Jin Hwang, and George Toderici. Scale-space flow for end-to-end optimized video compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8503–8512, 2020.

[3] I Ahmad, SM Akramullah, ML Liou, and M Kafeel. A scalable off-line mpeg-2 encoder using a multiprocessor machine. *Parallel Computing*, 2001.

[4] Shahriar M Akramullah, Ishfaq Ahmad, and Ming L Liou. A data-parallel approach for real-time mpeg-2 video encoding. *Journal of parallel and distributed computing*, 30(2):129–146, 1995.

[5] Shahriar M Akramullah, Ishfaq Ahmad, and Ming L Liou. Performance of software-based mpeg-2 video encoder on parallel and distributed systems. *IEEE Transactions on Circuits and Systems for Video Technology*, 7(4):687–695, 1997.

[6] Johannes Ballé, Valero Laparra, and Eero P Simoncelli. End-to-end optimized image compression. *arXiv preprint arXiv:1611.01704*, 2016.

[7] Fabrice Bellard. Bpg image format, 2018.

[8] Abdelhak Bentaleb, Ali C Begen, and Roger Zimmermann. Sdndash: Improving qoe of http adaptive streaming using software defined networking. In *Proceedings of the 24th ACM international conference on Multimedia*, pages 1296–1305, 2016.

[9] Gedas Bertasius, Heng Wang, and Lorenzo Torresani. Is space-time attention all you need for video understanding? *arXiv preprint arXiv:2102.05095*, 2021.

[10] Han Cai et al. Once-for-all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791*, 2019.

[11] Hao-Chieh Chang, Liang-Gee Chen, Yung-Chi Chang, and Sheng-Chieh Huang. A vlsi architecture design of vlc encoder for high data rate video/image coding. In *1999 IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 398–401. IEEE, 1999.

[12] Bo Chen, Zhisheng Yan, Hongpeng Guo, Zhe Yang, Ahmed Ali-Eldin, Prashant Shenoy, and Klara Nahrstedt. Deep contextualized compressive offloading for images. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 467–473, 2021.

[13] Bo Chen, Zhisheng Yan, and Klara Nahrstedt. Context-aware image compression optimization for visual analytics offloading. In *Proceedings of the 13th ACM Multimedia Systems Conference (MMSys)*, pages 27–38, 2022.

[14] Bo Chen, Zhisheng Yan, and Klara Nahrstedt. Context-aware optimization for bandwidth-efficient image analytics offloading. *ACM Transactions on Multimedia Computing, Communications and Applications*, 2023.

[15] Jie Chen and KJ Ray Liu. A complete pipelined parallel cordic architecture for motion estimation. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 45(5):653–660, 1998.

[16] Tong Chen, Haojie Liu, Qiu Shen, Tao Yue, Xun Cao, and Zhan Ma. Deepcoder: A deep neural network based video compression. In *2017 IEEE Visual Communications and Image Processing (VCIP)*, pages 1–4. IEEE, 2017.

[17] Ching-Te Chiu and KJ Liu. Real-time parallel and fully-pinelined two-dimensional dct lattice structures with application to hdtv systems. Technical report, 1991.

[18] Hyomin Choi and Ivan V Bajić. Deep frame prediction for video coding. *IEEE Transactions on Circuits and Systems for Video Technology*, 30(7):1843–1855, 2019.

[19] DASH Reference Client. Dash reference client, 2023.

[20] David Curry. Video streaming app revenue and usage statistics (2023), 2023.

[21] Mallesham Dasari, Kumara Kahatapitiya, Samir R Das, Aruna Balasubramanian, and Dimitris Samaras. Swift: Adaptive video streaming with layered neural codecs. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 103–118, 2022.

[22] Kuntai Du, Ahsan Pervaiz, Xin Yuan, Aakanksha Chowdhery, Qizheng Zhang, Henry Hoffmann, and Junchen Jiang. Server-driven video streaming for deep learning inference. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 557–570, 2020.

[23] FCC. Measuring broadband america - july 2012, 2012.

[24] Aditya Ganjam, Faisal Siddiqui, Jibin Zhan, Xi Liu, Ion Stoica, Junchen Jiang, Vyas Sekar, and Hui Zhang. C3: Internet-scale control plane for video quality optimization. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 131–144, 2015.

[25] Amirhossein Habibian, Ties van Rozendaal, Jakub M Tomczak, and Taco S Cohen. Video compression with rate-distortion autoencoders. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7033–7042, 2019.

[26] Yanbin Hao, Shuo Wang, Pei Cao, Xinjian Gao, Tong Xu, Jinmeng Wu, and Xiangnan He. Attention in attention: Modeling context correlation for efficient video classification. *IEEE Transactions on Circuits and Systems for Video Technology*, 32(10):7120–7132, 2022.

[27] Zhong-Li He, Chi-Ying Tsui, Kai-Keung Chan, and Ming L Liou. Low-power vlsi design for motion estimation using adaptive pixel truncation. *IEEE Transactions on circuits and systems for video technology*, 10(5):669–678, 2000.

[28] Junchen Jiang, Shijie Sun, Vyas Sekar, and Hui Zhang. Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 393–406, 2017.

[29] Ig-kyun Kim, Jin-jong Cha, and Han-jin Cho. A design of 2-d dct/idct for real-time video applications. In *ICVC'99. 6th International Conference on VLSI and CAD (Cat. No. 99EX361)*, pages 557–559. IEEE, 1999.

[30] Jaehong Kim, Youngmok Jung, Hyunho Yeo, Juncheol Ye, and Dongsu Han. Neural-enhanced live streaming: Improving live video ingest via online learning. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 107–125, 2020.

[31] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[32] Pasi Kolinummi, Juha Sarkijarvi, T Hamalainen, and Jukka Saarinen. Scalable implementation of h. 263 video encoder on a parallel dsp system. In *2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 1, pages 551–554. IEEE, 2000.

[33] Didier J Le Gall. The mpeg video compression algorithm. *Signal Processing: Image Communication*, 4(2):129–140, 1992.

[34] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.

[35] Xianshang Lin, Yunfei Ma, Junshao Zhang, Yao Cui, Jing Li, Shi Bai, Ziyue Zhang, Dennis Cai, Hongqiang Harry Liu, and Ming Zhang. Gso-simulcast: global stream orchestration in simulcast video conferencing systems. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 826–839, 2022.

[36] Luyang Liu, Hongyu Li, and Marco Gruteser. Edge assisted real-time object detection for mobile augmented reality. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[37] Zhenyu Liu, Xianyu Yu, Yuan Gao, Shaolin Chen, Xiangyang Ji, and Dongsheng Wang. Cu partition mode decision for hevc hardwired intra encoder using convolution neural network. *IEEE Transactions on Image Processing*, 25(11):5088–5103, 2016.

[38] Guo Lu, Chunlei Cai, Xiaoyun Zhang, Li Chen, Wanli Ouyang, Dong Xu, and Zhiyong Gao. Content adaptive and error propagation aware deep video compression. In *European Conference on Computer Vision*, pages 456–472. Springer, 2020.

[39] Guo Lu, Wanli Ouyang, Dong Xu, Xiaoyun Zhang, Chunlei Cai, and Zhiyong Gao. Dvc: An end-to-end deep video compression framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11006–11015, 2019.

[40] Wei Luo and Bo Chen. Neural image compression with quantization rectifier. In *ICML 2023 Workshop Neural Compression: From Information Theory to Applications*, 2023.

[41] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210. ACM, 2017.

[42] Alexandre Mercat, Marko Viitanen, and Jarno Vanne. Uvg dataset: 50/120fps 4k sequences for video codec analysis and development. In *Proceedings of the 11th ACM Multimedia Systems Conference*, pages 297–302, 2020.

[43] Toshiyuki Nozawa, Masahiro Konda, Masanori Fujibayashi, Makoto Imai, Koji Kotani, Shigetoshi Sugawa, and Tadahiro Ohmi. A parallel vector-quantization processor eliminating redundant calculations for real-time motion picture compression. *IEEE Journal of Solid-State Circuits*, 35(11):1744–1750, 2000.

[44] Hughes Perreault, Guillaume-Alexandre Bilodeau, Nicolas Saunier, and Maguelonne Héritier. Spotnet: Self-attention multi-task network for object detection. In *2020 17th Conference on Computer and Robot Vision (CRV)*, pages 230–237. IEEE, 2020.

[45] KS Prashant and V John Mathews. A massively parallel algorithm for vector quantization. In *Proceedings of the 1995 NASA Space and Earth Sciences Workshop*. Citeseer, 1995.

[46] Prajit Ramachandran, Niki Parmar, Ashish Vaswani, Irwan Bello, Anselm Levskaya, and Jon Shlens. Stand-alone self-attention in vision models. *Advances in neural information processing systems*, 32, 2019.

[47] Anurag Ranjan and Michael J Black. Optical flow estimation using a spatial pyramid network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4161–4170, 2017.

[48] Oren Rippel, Alexander G Anderson, Kedar Tatwawadi, Sanjay Nair, Craig Lytle, and Lubomir Bourdev. Elf-vc: Efficient learned flexible-rate video coding. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 14479–14488, 2021.

[49] Alexander Roach and Alireza Moini. Vlsi architecture for motion estimation on a single-chip video camera. In *Visual Communications and Image Processing 2000*, volume 4067, pages 1441–1450. SPIE, 2000.

[50] Ke Shen and Edward J Delp. A spatial-temporal parallel approach for real-time mpeg video compression. In *Proceedings of the 1996 ICPP Workshop on Challenges for Parallel Processing*, volume 2, pages 100–107. IEEE, 1996.

[51] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. *IEEE/ACM Transactions on Networking*, 28(4):1698–1711, 2020.

[52] Gary J Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (hevc) standard. *IEEE Transactions on circuits and systems for video technology*, 22(12):1649–1668, 2012.

[53] M-T Sun, T-C Chen, and Albert M Gottlieb. Vlsi implementation of a 16* 16 discrete cosine transform. *IEEE transactions on circuits and systems*, 36(4):610–617, 1989.

[54] Suramya Tomar. Converting video formats with ffmpeg. *Linux Journal*, 2006(146):10, 2006.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[56] Haiqiang Wang, Weihao Gan, Sudeng Hu, Joe Yuchieh Lin, Lina Jin, Longguang Song, Ping Wang, Ioannis Katsavounidis, Anne Aaron, and C-C Jay Kuo. Mcl-jcv: a jnd-based h. 264/avc video quality assessment dataset. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 1509–1513. IEEE, 2016.

[57] Thomas Wiegand, Gary J Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the h. 264/avc video coding standard. *IEEE Transactions on circuits and systems for video technology*, 13(7):560–576, 2003.

[58] Ian H Witten, Radford M Neal, and John G Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

[59] Chao-Yuan Wu, Nayan Singhal, and Philipp Krahenbuhl. Video compression through image interpolation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 416–431, 2018.

[60] Tianfan Xue, Baian Chen, Jiajun Wu, Donglai Wei, and William T Freeman. Video enhancement with task-oriented flow. *International Journal of Computer Vision*, 127(8):1106–1125, 2019.

[61] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Alexander Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *NSDI*, volume 20, pages 495–511, 2020.

[62] Ren Yang, Fabian Mentzer, Luc Van Gool, and Radu Timofte. Learning for video compression with hierarchical quality and recurrent enhancement. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6628–6637, 2020.

[63] Ren Yang, Fabian Mentzer, Luc Van Gool, and Radu Timofte. Learning for video compression with recurrent auto-encoder and recurrent probability model. *IEEE Journal of Selected Topics in Signal Processing*, 15(2):388–401, 2020.

[64] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. Neural adaptive content-aware internet video delivery. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 645–661, 2018.

[65] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 325–338, 2015.

[66] Anlan Zhang, Chendong Wang, Bo Han, and Feng Qian. Yuzu:neural-enhanced volumetric video streaming. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 137–154, 2022.

[67] Rui-Xiao Zhang, Tianchi Huang, Ming Ma, Haitian Pang, Xin Yao, Chenglei Wu, and Lifeng Sun. Enhancing the crowdsourced live streaming: a deep reinforcement learning approach. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 55–60, 2019.

[68] Yuanxing Zhang, Pengyu Zhao, Kaigui Bian, Yunxin Liu, Lingyang Song, and Xiaoming Li. Drl360: 360-degree video streaming with deep reinforcement learning. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1252–1260. IEEE, 2019.

A H.264 and H.265 Commands

We denote the frame width, the frame height, the frame rate, the GOP size, the compression quality, and the output filename as w, h, fps, GOP, Q, and $output$. The command for compressing a video from Pipe using 'veryfast', 'medium' and 'veryslow' modes of x264 are listed as follows, respectively.

ffmpeg -y -s wxh -pixel_format bgr24 -f rawvideo -r fps -i pipe: -vcodec libx264 -pix_fmt yuv420p -preset veryfast -tune zerolatency -crf Q -g GOP -bf 2 -b_strategy 0 -sc_threshold 0 -loglevel debug output

ffmpeg -y -s wxh -pixel_format bgr24 -f rawvideo -r fps -i pipe: -vcodec libx264 -pix_fmt yuv420p -preset medium -crf Q -g GOP -bf 2 -b_strategy 0 -sc_threshold 0 -loglevel debug output

ffmpeg -y -s wxh -pixel_format bgr24 -f rawvideo -r fps -i pipe: -vcodec libx264 -pix_fmt yuv420p -preset veryslow -crf Q -g GOP -bf 2 -b_strategy 0 -sc_threshold 0 -loglevel debug output

The command for compressing a video from Pipe using 'veryfast', 'medium' and 'veryslow' modes of x265 are listed as follows, respectively.

ffmpeg -y -s wxh -pixel_format bgr24 -f rawvideo -r fps -i pipe: -vcodec libx265 -pix_fmt yuv420p -preset veryfast -tune zerolatency -x265-params "crf=Q:keyint=GOP:verbose=1" output

ffmpeg -y -s wxh -pixel_format bgr24 -f rawvideo -r fps -i pipe: -vcodec libx265 -pix_fmt yuv420p -preset medium -x265-params "crf=Q:keyint=GOP:verbose=1" output

ffmpeg -y -s wxh -pixel_format bgr24 -f rawvideo -r fps -i pipe: -vcodec libx265 -pix_fmt yuv420p -preset veryslow -x265-params "crf=Q:keyint=GOP:verbose=1" output

MadEye: Boosting Live Video Analytics Accuracy with Adaptive Camera Configurations

Mike Wong[♠] Murali Ramanujam[♠] Guha Balakrishnan[◇] Ravi Netravali[♠]

[♠]*Princeton University* [◇]*Rice University*

Abstract

Camera orientations (i.e., rotation and zoom) govern the content that a camera captures in a given scene, which in turn heavily influences the accuracy of live video analytics pipelines. However, existing analytics approaches leave this crucial adaptation knob untouched, instead opting to only alter the way that captured images from fixed orientations are encoded, streamed, and analyzed. We present MadEye, a camera-server system that automatically and continually adapts orientations to maximize accuracy for the workload and resource constraints at hand. To realize this using commodity pan-tilt-zoom (PTZ) cameras, MadEye embeds (1) a search algorithm that rapidly explores the massive space of orientations to identify a fruitful subset at each time, and (2) a novel knowledge distillation strategy to efficiently (with only camera resources) select the ones that maximize workload accuracy. Experiments on diverse workloads show that MadEye boosts accuracy by 2.9-25.7% for the same resource usage, or achieves the same accuracy with 2-3.7× lower resource costs.

1 Introduction

Building on the steady growth in camera deployments and advances in deep neural networks (DNNs) for vision tasks (e.g., classification or detection) [8, 21, 46, 66, 70], live video analytics pipelines have become prevalent. These pipelines operate by continually streaming live video feeds from cameras to processing servers (either edge [9, 11, 76, 82, 109] or cloud [33, 56, 67, 119]), where DNNs are run on incoming frames to produce low latency and highly accurate results for different application queries, i.e., combinations of task, DNN, and object(s) of interest. Key use cases include autonomous driving, footfall tracking, traffic coordination, business analytics, among others [6, 10, 12, 25, 27, 31, 41, 42, 90, 91].

Given their practical importance, much research has been devoted to improving both the resource efficiency and accuracy of live video analytics pipelines. Existing solutions include accuracy-aware tuning of inference configuration, encoding, or appearance knobs [34, 40, 57, 85, 119], filtering out redundant content [26, 33, 53, 67], using cheaper model variants [9, 92], improving job scheduling [82, 96, 119], and so on. However, all of these works assume that the content observable by cameras is unchangeable, and instead can only be encoded, streamed, or analyzed differently. In essence, they focus on optimizing *fixed, preset* camera deployments.

Unfortunately, the deployment of cameras for analytics is itself a daunting task for operators. Subject to practical constraints (e.g., mounts, power sources), for a scene of interest, operators must determine the number of cameras to deploy and the orientation (i.e., combination of rotation and zoom factor) to use for each. There exist many possible orientations, and altering these decisions requires manual intervention. Yet we find that doing so can be highly fruitful: across different workloads and scenes, dynamically adapting orientations over time can yield accuracy improvements of 21.3-35.3% (without inflating resource usage) compared to even the *best* fixed-orientation scheme. These wins cannot be reaped by simply deploying more fixed cameras to simultaneously cover more orientations: most orientations are 'best' for short total times (median of 6 sec per 10-min video), drastically hindering the efficiency of such an approach, especially in resource-constrained settings where video analytics run [9, 68, 76, 97].

An alternative strategy is to leverage PTZ (pan-tilt-zoom) cameras that offer software libraries for tuning orientations, thereby providing a logical approach to capturing the above wins. Indeed, despite existing for nearly two decades, PTZ camera popularity has surged in recent years (global market value of $1.5 billion in 2021 that is expected to reach $3.6 billion by 2028 [20]) largely due to declining price points that can rival fixed-camera costs [1–4, 94, 102]. However, multiple challenges complicate their use for live analytics (§2.3). First, queries are highly sensitive, in different ways, to orientation knobs due to their diverse goals (e.g., tasks), inherent model biases (how models perceive scenes and objects), and scene dynamism (where objects are located) – optimizing orientation tuning for one workload can forego up to 25.1% of the potential median accuracy wins for another. Second, the 'grid' of orientations is large, but the selection space is sparse, with steep accuracy drops from the best orientation(s) to others at any time. Third, the best orientation changes rapidly, e.g., 85% of changes occur in ≤1 sec since the last change.

To overcome these issues, we present **MadEye**, a camera-server system that automatically and continually adapts PTZ camera orientations to maximize analytics accuracy for the scene and workload at hand. The key insight behind MadEye is that the speed at which commodity PTZ cameras can

change orientations (i.e., upwards of 600° per sec with concurrent zoom) far outpaces the rate at which applications require analytics results (typically 1-30 frames per second (fps), i.e., every 33-1000 ms). This, in turn, allows MadEye to eschew typical non-stationary multi-armed bandit strategies [64, 81, 107] that rely purely on previous explorations to determine orientation importance, in favor of a more informed strategy based on *current* scene content. Concretely, in each timestep (33 ms for 30 fps) and subject to network/compute resource availability, MadEye cameras explore multiple orientations and quickly determine which will maximize workload accuracy and warrant transmission to backend servers for full inference. Though intuitive, realizing this strategy in practice involves addressing several core challenges.

First, to enable fast camera-side evaluation of the importance of different orientations, MadEye adopts a custom knowledge distillation [49] strategy with edge-grade, ultra-compressed NN models. To cope with their potentially limited predictive power, we task them with modeling query sensitivities only to the point of accurately *ranking* orientations in terms of impact on workload accuracy – precise results are left to backend servers. Even with this relaxed framing, MadEye must employ several optimizations to achieve sufficient rank accuracy. Most notably, MadEye trains edge models using a common abstraction – detection for objects of interest – that reflects the minimum information needed to capture sensitivities and biases for popular tasks. Task-specific semantics need not be baked into edge models, and instead can be incorporated by post processing the generated detections.

Edge models are *continually* trained on MadEye's backend using both the latest and historical workload results to mitigate data skew towards recently-selected orientations. Importantly, to balance resource costs and accuracy, each edge model covers only a single query but all orientations. The intuition is that, while results from different query models can substantially divergen [14, 35, 63, 77], feature-level variance between orientations for the same scene is considerably narrower, often smaller than that in typical pre-training datasets [69]. Accordingly, MadEye freezes pre-trained feature extraction layers across queries, caching those weights on cameras to lower retraining and (downlink) model update overheads.

Second, we devise a novel, on-camera search strategy to explore orientations with the goal of capturing the best one (accuracy-wise) at each timestep. Three key empirical observations guide our search: (1) despite rapid temporal shifts, transitions between best orientations move slowly in the spatial dimension, (2) the best orientations at any time are usually clustered spatially, and (3) neighboring orientations (with overlapping views) have highly correlated trends in efficacy.

Building on these observations, MadEye explores a flexible shape of contiguous orientations at each timestep, and considers shifting only towards neighboring orientations whose efficacy can be robustly predicted. Decisions to keep/remove orientations are governed by both response rates (and the corresponding time budgets) and *relative* comparisons of recent edge model results. For the former, MadEye uses an efficient heuristic to determine path feasibility in the time budget (a variant of the NP-Hard Traveling Salesman Problem [48]). For the latter, MadEye gracefully trades off exploration (i.e., shape size) for network usage (i.e., sending more orientations for backend inference) to bound the effects of edge model errors and maximize accuracy for the required response rate.

To evaluate MadEye, we developed the first (to our knowledge) dataset that supports tuning rotation and zoom at each time instant by splicing out scenes of interest from publicly available 360° videos. Using this dataset, we evaluated MadEye on a variety of network conditions, commodity PTZ camera hardware, and workloads that incorporate multiple vision DNNs and query tasks: classification, counting (per-frame and aggregate), and detection. Across these settings, MadEye boosts accuracy by 2.9-25.7% compared to an oracle fixed-orientation strategy without inflating resource usage; these wins are within 1.8-13.9% of the oracle dynamic strategy. Framed differently, MadEye achieves those accuracy boosts with 2-3.7× lower resource footprints than the best strategy of using (multiple) fixed-orientation cameras. Moreover, MadEye outperforms recent PTZ tracking algorithms [93, 98] (by 2.0-3.8×) and multi-armed bandit solutions [106] (by 5.8×). The source code and datasets for MadEye are available at `https://github.com/michaeldwong/madeye`.

2 Background and Motivation

We start with an overview of live video analytics deployments (§2.1). We then show measurements highlighting the importance of dynamically adapting camera orientations to workloads and scenes (§2.2), and the challenges associated with realizing those benefits in practice (§2.3).

2.1 Overview of Live Video Analytics

In a live video analytics deployment, one or more cameras continually stream their video frames to servers for processing. Servers can range from distant (but powerful) cloud machines [96, 119] to nearby (but weaker) edge boxes [9, 76, 82], and are tasked with running queries on the incoming frames to support different applications. Queries most often involve running deep neural network (DNN) inference on individual frames, with the goals of locating and characterizing various objects in the scene, e.g., an intersection. Moreover, the queries for different applications can vary in terms of the tasks they perform, the objects they consider, the DNNs they use (different architectures and weights), and the response rates they require. For instance, footfall tracking for business analytics will count people passing through an area, with response rates at 1 fps or less [12]. In contrast, smart driving or sports analytics applications will detect the specific locations of cars or people, with response rates upwards of 30 fps [91].

In this paper, we focus on the following four query tasks (and their corresponding accuracy metrics) that have been prevalent in recent literature [23, 33, 58, 59, 67] and real-world

deployments [74, 82]; §3.4 details our target queries. We note that these tasks also serve as the building blocks for more complex queries, e.g., tracking relies on detections.

- **Binary classification**: asks if any objects of interest are present in a frame. Accuracy across the video is measured as the fraction of frames with the correct binary decision.
- **Counting**: counts the number of objects of interest in each frame. Accuracy for each frame is measured as the percent difference between the returned and ground truth counts.
- **Detection**: finds the precise bounding box coordinates for objects of interest in a frame. Accuracy per frame is measured using mAP [38], which evaluates the overlap between each returned box and its ground truth counterpart.
- **Aggregate counting**: counts the *unique* objects of interest that appear in a scene. Accuracy per video is the percent difference between the returned and ground truth counts.

Over time, an analytics deployment will face diverse workloads to run on the feeds it manages, each varying in query composition and size [77, 82]. Yet, the overarching goals persist: subject to resource constraints, deliver low-latency results (at the desired response rate) with maximal accuracy.

2.2 Opportunities with Tuning Camera Orientations

Existing video analytics systems (§6) assume that a stationary camera's orientation (rotation and zoom), and thus what it ingests from the target scene, is fixed and incapable of being adapted. To quantify the significance of this restriction, we run experiments on our 50-video dataset and workloads that incorporate 4 model architectures, the tasks from §2.1, and people/cars; §5.1 details our setup. Each video supports tuning of rotations (150° horizontally by 30°, 75° vertically by 15°) and zoom (1-3×); we use other granularities in §5.4.

For each video, we obtained per-frame (15 fps here) results for each workload by running its queries on all 75 orientations. We then define accuracy relative to the *best* orientation for each frame, i.e., the orientation that maximized per-frame accuracy for the workload. In other words, best orientations represent an upper bound on the accuracy that could be achieved by using a single orientation at a given timestep. For instance, for counting, an orientation's accuracy at any time is its object of interest count divided by the max count across all orientations at that time. Using this methodology, we compare three schemes: (1) *one time fixed* which selects the best orientation at time=0 and keeps it throughout the video, (2) *best fixed* which uses oracle knowledge to pick the best single orientation that maximizes average workload accuracy for the video, and (3) *best dynamic* which selects the best orientation per frame in the video.

As shown in Figure 1, adapting camera orientations brings substantial accuracy improvements without inflating resource usage, i.e., the same number of frames are transmitted and processed: median boosts with *best dynamic* are 30.4-46.3% over *one time fixed* and 21.3-35.3% over the *best fixed* scheme that

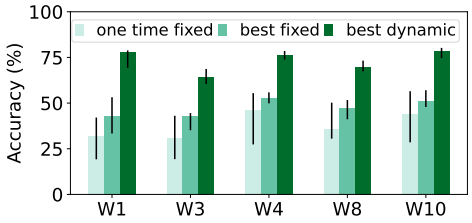


Figure 1: Accuracy for 5 representative workloads when using varying degrees of orientation adaptation. Bars list results for the median video, with error bars spanning 25-75th percentiles.

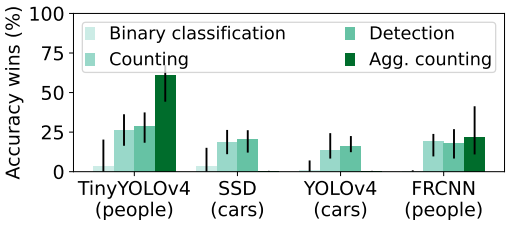


Figure 2: Accuracy wins from adapting orientations (compared to *best fixed*) grow as query specificity grows. Bars list median videos, with error bars for 25-75th percentiles. We exclude agg. counting+cars due to limits of multi-object trackers (§5.1).

is an upper bound for any fixed-orientation approach. Figure 2 breaks down these results by query task. Notably, the importance of adapting orientations grows as query types become more specific. For instance, for YOLOv4 and cars, median accuracy improvements over *best fixed* are 1.2%, 13.4%, and 16.4% for binary classification, counting, and detection. The reason is that coarser queries mask certain differences across orientations, e.g., if many objects of interest are present in the scene, any orientation that catches a single object will deliver max accuracy for binary classification; counting, on the other hand, will favor the orientation with the most objects.

Primer on PTZ cameras. Pan-tilt-zoom (PTZ) cameras present an intuitive mechanism to realize such adaptation. PTZ cameras come in two forms, traditional [36, 86] and electronic (ePTZ) [51, 87], both of which support software tuning of pan (horizontal rotation), tilt (vertical rotation), and zoom. The key difference between the two variants is in their tuning mechanisms. Traditional PTZ cameras embed physical motors to rotate at up to 600°-per-second and concurrently optically zoom (i.e., without reducing resolutions). In contrast, ePTZ cameras capture wide field-of-views and employ near-instantaneous digital rotation and zoom to focus on specific parts of the scene. ePTZ cameras change orientations faster and are cheaper, but also cover smaller rotation areas (150° vs. 360°) and degrade image quality by using digital zoom. PTZ cameras rival traditional ones in on-board compute resources, with recent offerings housing edge-grade GPUs [79].

2.3 Challenges

Despite the potential benefits of adapting camera orientations using PTZ cameras, three fundamental challenges complicate this approach in practice. We describe them in turn.

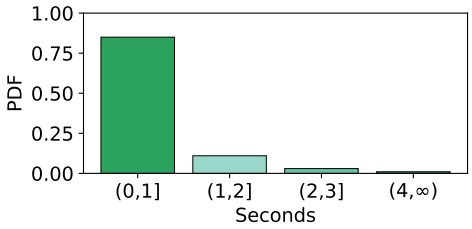


Figure 3: Shifts in the best orientation are frequent. Results list a PDF (binned by 1 sec) of time between switches in best orientation across all videos and workloads.

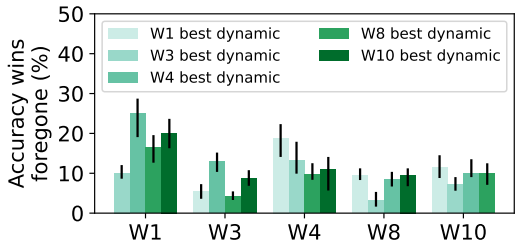


Figure 4: Workloads exhibit different sensitivity to orientations. Results apply the best orientations for workload X (legend) to workload Y (x axis), and plot the accuracy wins (over best fixed for Y) that are lost from not using the best orientations for workload Y. Bars list medians; error bars for 25-75th percentiles.

C1: rapid changes in best orientation over time. As shown in Figure 3, switches in best orientation are frequent: 85% of switches occur in ≤1 sec since the last switch. The reason for this high flux is threefold: (1) the typical motion for certain objects (e.g., moving car or walking person) is sufficient to frequently cross orientation boundaries, especially since orientations exhibit a degree of content overlap, e.g., an object may be visible in two orientations and quickly move to change the one it is most prominent in, (2) models can provide inconsistent results across even back-to-back frames that are seemingly unchanged [60, 61], so best orientation swaps can occur even with static objects, and (3) aggregate queries seek previously unseen objects and thus intentionally move to new orientations, e.g., the percentage of sub-second switches mildly drops to 70% when aggregate queries are excluded.

C2: diverse workload sensitivities to zoom and rotation. At any point in time, the best orientation can vary across individual queries and workloads. Figure 4 illustrates this, showing that adapting orientations to maximize accuracy for one workload can result in foregoing 3.2-25.1% of the potential (median) accuracy wins for other workloads.

Figure 5 highlights this at a query level, showing that different models, objects, and tasks can all influence orientation selections. Model discrepancies influence what can be discerned in the scene during inference and under what orientations. For instance, with people counting, selecting best orientations for a YOLOv4 query will miss out on 26.3% median accuracy wins for the same task using SSD (even when trained on the same dataset). In contrast, tasks dictate the specificity needed in the collected results, e.g., optimizing for counting people

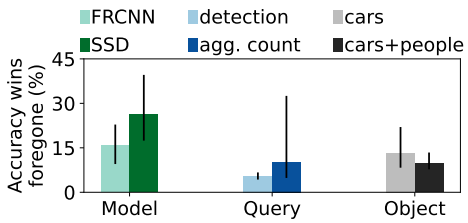


Figure 5: Applying the best orientations for a base query of {YOLOv4, counting, people} to a query Y that modifies a single element in the base query; we compare the accuracy wins (over best fixed) to those when using the best orientations for Y. Bars list medians; error bars for 25-75th percentiles.

rather than aggregate people counting with the same model foregoes 10.2% of potential wins. Lastly, object type governs the importance of regions based on object densities, as well as the features used for and difficulty in detecting relevant objects (smaller objects are tougher to discern [88]). Thus, optimizing for a YOLOv4 people counting query would forego 13.3% of wins if the query considered cars instead.

Figure 6 provides example screenshots to illustrate the benefits and harm of changing orientations. Importantly, tuning orientations does not simply bring new objects into field of view, and instead *plays a large role in a model's ability to detect objects that were already visible*. Indeed, simply using the lowest zoom factor will capture the largest portion of the scene, but can result in models missing in-view objects due to degradations in object size or resolution.

C3: massive (but sparse) search space. The orientation space exhibits substantial sparsity in the spatial and temporal dimensions. For the former, among the 75 orientations at any time, only 1 (or several, with ties) is best, with steady dropoff in accuracy to the others, e.g., median dips of 4.8% and 20.7% from the best to 2nd and 5th best. For the latter, most orientations are best for short total times in each video, with median durations of 5-6 sec across workloads (Figure 7).

3 Design

Figure 8 shows the end-to-end operation of MadEye. The main insight behind MadEye is to leverage fast PTZ rotation speeds to explore many orientations in each timestep (i.e., between when results are needed for an fps), and then select, based on their *current* content, the one(s) that maximize workload accuracy under resource constraints. The idea is to limit the "guess work" compared to prior search algorithms that rely only on past orientation efficacy (§5.3).

As in other video analytics systems [33, 57, 67, 82], users register queries with a backend agent (on an edge or cloud server), specifying a target scene, as well as a model to use, object(s) of interest, and a task. To operate under camera compute constraints, MadEye then trains edge-compatible (i.e., highly compressed) models (§3.1), not to replace the original (more accurate) query models (as in typical knowledge distillation [49]), but instead to *approximately* extract information of importance in a frame for each query. In other words, ap-

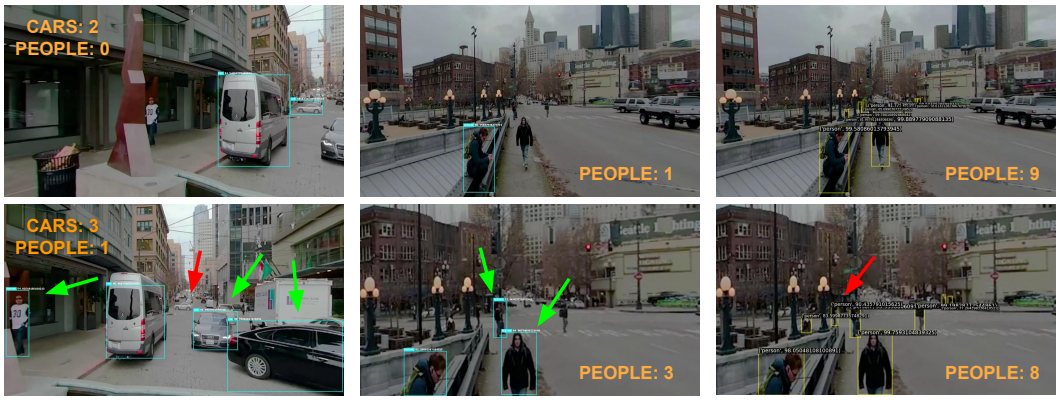


Figure 6: Screenshots showing the (diverse) impact of rotation and zoom for different queries. Each column shows two images from the same time instant that use either different rotation or zoom. On the bottom row, green arrows show newly captured objects, while red arrows show objects that are newly missed after the orientation change. Left: rotation brings a new object into the scene, helps detect 2 previously-visible objects, but loses a previously-detected object. Middle: zooming in helps detect new people. Right: after switching models, the same zoom from the middle column actually reduces the number of detected people.

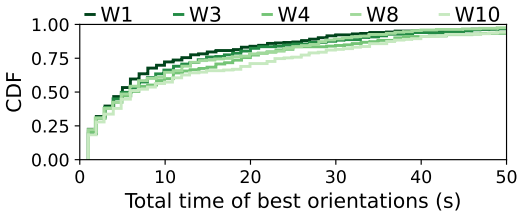


Figure 7: Most orientations are best for short total times in each video. Results consider all orientation-video pairs per workload.

proximation models are explicitly designed to estimate the inherent sensitivities of each query (C2 from §2.3).

To cope with the large space of orientations and rapid shifts in best ones (C1 and C3 from §2.3), MadEye employs an efficient on-camera search strategy (§3.3) that explores as many potentially fruitful orientations as possible while avoiding fps violations for results. The camera then runs approximation models on all captured orientations in each timestep and uses the results to (1) *rank* the orientations in terms of their likelihood to maximize workload accuracy, and (2) determine the orientations to consider in the next timestep. The highest ranked orientations that the network can support are sent to the backend for full workload inference, with results used to continually adapt approximation models to the scene (§3.2).

3.1 Designing Approximation Models

The primary objective of MadEye's approximation models is to quantify the *relative* importance of orientations for the queries in a workload. However, this requires capturing the sensitivity of each query to different orientation and scene dynamics, subject to camera compute constraints. Given the potential complexity of workload queries, we eschew noisy (and limited) vision features based on local gradients [30, 73] in favor of knowledge distillation with compressed models [49]. However, we alter this approach in several ways to favorably balance ranking accuracy and resource efficiency.

We design approximation models using a common abstraction that reflects the minimum amount of information needed to sufficiently rank orientations. The key idea is that the core elements of query sensitivity pertain to how models find and characterize objects, rather than how tasks post-process those results. Thus, MadEye's approximation models are structured purely as ultra-lightweight detectors for objects of interest; this strategy also avoids tricky development of compressed models per task. Concretely, we use the smallest variant of the edge EfficientDet family [103], EfficientDet-D0 (3.9M parameters, >150 fps on a Jetson edge GPU), which enables MadEye to scale to multi-query workloads. More complex detectors could be used, but cameras possess limited GPU memory [67, 82], and inference delays negatively influence the degree to which MadEye can explore orientations (§3.3).

Why a detector? Two alternatives we considered for the approximation models are to directly estimate object counts in an image, and to directly output rank orderings across multiple images. However, we empirically observed high error rates with both. This is largely because such approaches can only relate the presence of features to objects via a global regression over an entire image (or multiple images), failing to leverage local regressions via bounding box predictions to boost precision. While image-level DNN object counters do exist [99, 113, 118, 122], they focus on large crowds of people. In contrast, there are often few objects of interest in an orientation at any time (§2.3), making rank orderings extremely sensitive to small errors in count prediction.

MadEye uses one approximation model per query, rather than per workload or object. Though more efficient, we avoid per-workload and per-object approximation models as we (like others [77]) find that different DNNs can exhibit wildly varying response profiles to even the same object classes due to object-independent factors like scale and resolution [50].

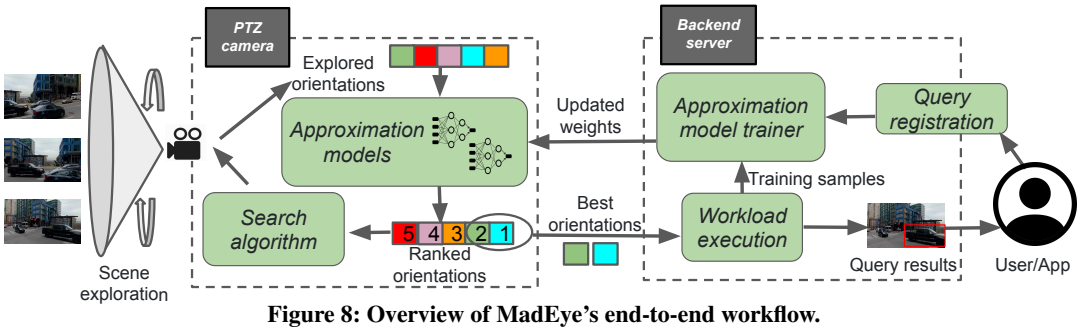


Figure 8: Overview of MadEye's end-to-end workflow.

Moreover, DNNs trained on very different datasets are known to inherit different algorithmic biases [14, 35, 63, 80, 100, 111].

However, each approximation model is configured to support *all* orientations for two reasons. First, the number of orientations is large (§2.3), making per-orientation approximation models impractical with on-camera GPUs. Second, neighboring orientations exhibit substantial overlap, and since we only consider orientations for a given scene, divergence in background content, lighting, shadows, etc. are minimal. Indeed, we measured the perceptual distance [89] of images (LPIPS) from different orientations in the same scene to be 0.30. For context, the same value for the popular MS-COCO and Pascal VOC datasets used to successfully pre-train many vision models (including EfficientDet) are 0.46 and 0.41.

Estimating workload accuracies. MadEye post-processes the generated bounding boxes from all approximation models to compute *predicted workload accuracies* for orientation ranking. To do this, MadEye follows the per-task accuracy metrics from §2.1, but computes per-orientation predicted accuracy in a relative manner compared to the other orientations under test. For instance, counting computes the ratio of object counts between each orientation and the max among the set of explored orientations at that timestep, while detection expands this to incorporate object area sizes (as per mAP score). Lastly, aggregate counting modulates count scores to favor less explored orientations (that may have unseen objects).

3.2 Continually Training Approximation Models

MadEye servers train a new approximation model for each new query, with the goals of being fast and accurate (in ranking orientations). Initial training uses a small set of 1000 historical images from the target scene that is then labeled (online) using the DNN in the registered query; label generation takes 7-90 sec depending on the DNN. However, to accelerate this process, MadEye begins with a version of EfficientDet that is pre-trained on Pascal VOC, and freezes both the backbone and BiFPN layers responsible for feature extraction and fusion. Only weights for the final 3 bounding box and class prediction layers are fine-tuned to mimic the target query's behavior. The rationale is that model features progressively move from general (e.g., textures, gradients) to task-specific (e.g., object labels) from model start to end [16, 116, 117]. Initial fine-tuning lasts 40 epochs (≈25 mins).

Even after initial fine-tuning, approximation models may fail to generalize to changing scene dynamics [101], leading to degrading accuracy. To cope with such data drift, MadEye employs continual learning (every 120 sec) to update the model's weights using the latest query results on orientations sent to the server for full workload inference. While continual learning has been applied to edge video analytics [9, 75], MadEye requires several alterations from prior efforts. The main challenge is that within each retraining window, samples are only available for the orientations that MadEye's camera-side component recently visited and deemed worthy of backend inference. Since orientations are typically best for short total times (§2.3), there is often severe imbalance in the orientations covered by new training samples. For instance, with perfect rankings, the average 2-minute window sees only 9.3% of orientations get sent to the backend. This can result in overfitting to certain orientations, and catastrophic forgetting [62] for others that may soon be ranked highly.

To deal with this, MadEye retrieves the most recent historical training samples from each orientation and uses this to balance the dataset. As we will discuss in §3.3, we find that orientation shifts are often spatially localized, with changes to distant orientations happening over longer timescales. Thus, MadEye pads the data samples for neighboring orientations (up to 3 away from the latest one) to match the count for the most popular orientation in the retraining window. The remaining orientations use an exponentially declining number of samples based on their distance from the latest orientation.

Each continual learning round considers the last 120 best orientations selected by MadEye (1 sample per second since the last retraining round), as well as 200 historical images (on average) to account for neighboring orientations; a random 30% of this is reserved for validation. Retraining runs for 5 epochs and takes an average of 32 seconds. Note that all continual learning runs asynchronously on backend servers; §5.4 profiles network usage for shipping model updates which are small due to the use of ultra-lightweight (compressed) detectors and the aforementioned backbone freezing.

3.3 Exploring and Ranking Orientations

The primary goal of MadEye's on-camera component is to efficiently explore (a subset of) the large orientation space to capture the best orientation for each timestep. Realizing

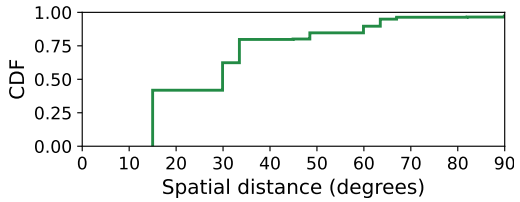


Figure 9: Spatial distance between successive best orientations is small, with most transitions between neighboring orientations. Results aggregate across all videos and workloads for 15 fps.

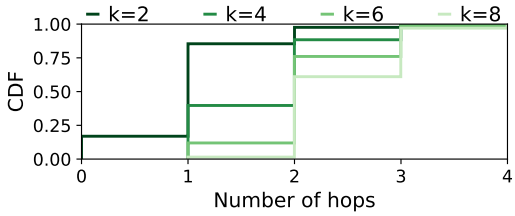


Figure 10: Top ranked orientations are often spatially clustered. Results use 15 fps, are aggregated across all workloads and videos, and show the max distance between orientations in the top k ranked orientations at each timestep.

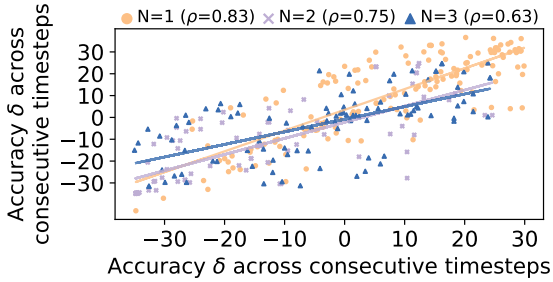


Figure 11: Correlation in accuracy changes across orientations separated by N hops. Results list Pearson Correlation Coefficients and cover 3 representative videos and workloads (15 fps).

this is challenging for three reasons. First, MadEye only has visibility into the orientations that it has recently explored, but other orientations can change in content and importance at any time. Second, even among recently explored orientations, MadEye only has access to coarse results from approximation models (i.e., that accurately capture only relative importance) for most. Third, each timestep is not only dedicated to exploration, but also (1) running approximation models on explored orientations, (2) encoding and shipping select orientations to the server, and (3) running the workload on shipped images.

Rather than relying on previous (and potentially stale) observations at each orientation (§5.3), MadEye opts for a more informed strategy guided by 3 empirical observations.

- Although best orientations change rapidly over time (§2.3), those changes are far slower in the spatial dimension. Figure 9 illustrates this, showing that the median and 90th percentile spatial distance between successive best orientations are 30° and 63.5°, which pertains to shifts spanning only 1 or 2 orientations in our default grid (§5.1).
- The best performing orientations (accuracy-wise) at any time are often spatially clustered (Figure 10). Concretely, across our dataset, the 75th percentile distance separating orientations in the top k at each timestep is 1 and 2 orientations for k values of 2 and 6.
- Accuracy for neighboring orientations often shift in tandem. Indeed, as shown in Figure 11, the correlation coefficient for accuracy changes in direct neighbors is 0.83; intuitively, this value shrinks to 0.75 when considering neighbors 2-hops away (that exhibit less content overlap).

Taken together, these findings motivate a search strategy that considers a flexible shape of contiguous orientations at each timestep, and swaps out underperforming orientations in the previous shape only for neighboring ones whose trends we can robustly predict for the next timestep. We start with a description of the algorithm that does not account for zoom or resource constraints and later incorporate those elements. Common themes are: only relative comparisons of approximation model results are used, we leverage all outputs from those models (including bounding boxes), and search decisions are entirely local (i.e., on cameras) to remain rapid.

MadEye begins with a rectangular seed shape that reflects the largest coverable area in the time budget, thereby maximizing early exploration; we reset to this shape any time 0 objects of interest are found in a shape. The corresponding orientations are captured and analyzed with approximation models to compute a predicted workload accuracy for each (§3.1). After sending the top k orientations to the server for workload inference, MadEye must use these prior results to determine the set of orientations to explore in the next timestep.

To do this, MadEye labels each orientation from the last timestep with a value that indicates the likelihood of being fruitful in the *next* timestep. Concretely, we combine the exponentially weighted moving averages from recent (10) timesteps for (1) any computed predicted accuracy values, and (2) the deltas between those values. Weighted averages are used to remain robust to inconsistencies in DNN results across consecutive frames [77, 84], which is especially pronounced with MadEye's compressed approximation models.

Using those labels, MadEye must now determine which orientations to remove and add for the upcoming timestep. For this, MadEye sorts orientations into an ordered list based on their label values. Using pointers at the head H (largest label) and tail T (smallest label) of the list, MadEye iteratively compares orientations by asking: should we remove the orientation at T in favor of adding a neighbor to H? Concretely, MadEye computes the ratio of label values for H/T. If (1) that ratio exceeds a threshold (indicating a substantial disparity in the potential of H and T), (2) H has neighbors not already in the shape, and (3) removing T would not break contiguity, we remove the orientation at T and increment the pointer. The process repeats by considering the addition of another neighbor for H, this time using a larger threshold to account for the additional uncertainty of adding more neighbors. H is decremented when a neighbor cannot be added, and the process ends when even one neighbor for H cannot be added.

For each iteration that results in a neighbor addition for H, MadEye selects among H's neighbors by analyzing the bounding boxes that its approximation models generated in the last timestep. For each candidate neighbor, we compute the ratio of two values: normal distances to the center of H and to the centroid of all bounding boxes in H. Values <1 indicate lower chances of H's objects moving to the candidate in the next timestep. We repeat this process for all other orientations in the last shape that the candidate exhibits any non-zero overlap with. Candidate neighbor scores are computed as the weighted sum of these ratios (weights according to degree of overlap), and the candidate with the max score is selected.

Reachability and path selection. The search algorithm thus far ignores whether a PTZ camera can sufficiently cover the selected shape in a given time budget. Formally, the shape of orientations can be represented as a fully-connected undirected graph with edge weights pertaining to the time taken to move between two adjacent orientations (given a rotation speed). Our goal is to determine whether the shape is coverable in a given time budget, and if so, what is the shortest path. The paths between orientations satisfy the triangle inequality property [104], so this can be modeled as a variant of the NP-Hard Traveling Salesman Problem (TSP) [18]. Given our tight time budgets, MadEye employs the Minimum Spanning Tree (MST) heuristic [48], but optimizes it to minimize online delays. In particular, since our orientation grid is static, we precompute pairwise distances and the entire MST ahead of time. Online, for a given shape, we quickly extract and perform a preorder walk on the corresponding subgraph to get the shortest path. This reduces the heuristic to linear complexity (in orientations); each path computation takes 14 μs, and the resultant paths are within 92% of optimal. Upon failure, MadEye greedily removes the orientation with the lowest potential (that does not break contiguity) and rechecks reachability.

Balancing search size and network/compute delays. MadEye pipelines its exploration through orientations with the running of approximation models on each one. However, network transmission to and workload inference on the backend do not overlap with orientation exploration. The reason is that transmissions are governed by global ranks across *all* orientations explored in each timestep. Thus, in each timestep, we face a tradeoff between exploring more orientations and sending more orientations to the backend.

MadEye resolves this tension based on the expected difficulty for its approximation models to accurately rank the considered orientations, which in turn governs the risk associated with exploring more orientations (and sending fewer to get ground truth results). Intuitively, scenarios where the considered orientations are projected to contribute similar accuracies pose the biggest difficulty for approximation models (as the gaps between ranks shrinks). MadEye determines the right balance by first selecting a target number of frames to send according to the training accuracy for approximation models (provided by the backend) and the variance in predicted accuracy values in the last timestep, e.g., with 85% training accuracy, any frames within 15% accuracy of the top ranked frame are sent. MadEye then computes a target shape size for exploration, accounting for network delays (harmonic mean of past 5 transfers [115]), backend compute delays, rotation speeds, and approximation model inference delays.

Handling zoom. After selecting the set of orientations to visit, the search algorithm must determine the zoom factor to use for each one. The challenge is that past accuracies are insufficient for determining zoom fidelity as MadEye cannot know what objects are being missed by not zooming in/out. Instead, we rely on bounding boxes from approximation models to determine the risk of zooming in. When an orientation is added to the shape, we start at the lowest zoom factor to gain visibility into its whole content. At each timestep, we compute the average distance between each bounding box and the centroid of all boxes; smaller distances indicate more clustering and less risk of zooming in. These values are compared with the area covered by each zoom factor to select one, and MadEye automatically zooms out after 3 seconds to avoid missing newly entering objects in the orientation.

Transmitting images. At the end of each timestep, MadEye must transmit select images to the server for workload inference. Unlike standard streaming, MadEye sends disjoint sets of images from each orientation's video stream. To keep bandwidth costs low, MadEye maintains a list of the last image shared for each orientation, and employs a functional encoder [39] that computes deltas relative to that image.

3.4 Query Support and Deployment Discussion

As noted in §2.1, MadEye targets the large class of object-centric queries (i.e., those that find and label people or cars) which have dominated reported industrial workloads [74, 82] and prior work [23, 33, 58, 59, 67, 75, 77]. Further, we consider tasks that return per-frame (e.g., counting, detection) and per-video (e.g., aggregate counting) results.

Supporting other object-centric queries with new objects of interest or tasks involves two main steps: training a new approximation model, and (potentially) developing new logic to rank orientations using approximation model results. The former is automated; MadEye does not make any assumptions about object type or task, and instead trains its approximation models as detectors for the objects of interest identified by results of provided query models. The latter is only required for new tasks and can be arbitrarily complex, e.g., a ranker can consider keypoints of detected objects for activity recognition queries. §A.1 presents results for other object types and tasks.

MadEye currently operates on videos from cameras that are *statically mounted*. For changing mounting locations (e.g., dashcams), MadEye will face similar challenges to any multi-armed bandit strategy: past content becomes a weaker indicator of orientation efficacy since orientations continually point

to new scenes. MadEye may still outperform prior approaches by comparing explored orientations in terms of current content, but we leave a detailed exploration to future work.

4 Implementation

MadEye's core components are written in 9.1k lines of Python code, with all training and inference tasks across the backend and camera run in PyTorch. We use TensorRT [7] to accelerate inference on the backend, and a variant of Nexus [96] as a round-robin scheduler for approximation model inference. Orientations are first represented as rotational values, projected onto a 360° space, and then converted using an in-house equirectangular-to-rectilinear image converter (in C++) to match the APIs offered by PTZ cameras [17]. For ground truth accuracy computations (§5.1) that need a global perspective on object locations and uniqueness, atop the ByteTrack multi-object tracker [114] that links objects across an orientation's video, we use `cv2` and `scikit-image` to extract image features (e.g., SIFT) that link objects across orientations.

5 Evaluation

We evaluated MadEye across diverse workloads, network settings, and videos. Our key findings are:

- MadEye increases median workload accuracies by 2.9-25.7% compared to an oracle fixed-orientation strategy (while using the same amount of resources); wins are within 1.8-13.9% of the oracle dynamic strategy.
- Achieving MadEye's accuracy wins with 1 PTZ camera would require the best 4-6 fixed-orientation cameras, which comes with a 2-3.7× inflation in resource costs.
- MadEye outperforms prior PTZ algorithms by 2.0-5.8×, providing 31.1-52.7% higher accuracy than Panoptes [98], tracking [93], and multi-armed bandits [106].
- MadEye gracefully balances on-camera exploration and transmission of orientations to maximize accuracy even as resources shrink and response rates rise.

5.1 Methodology

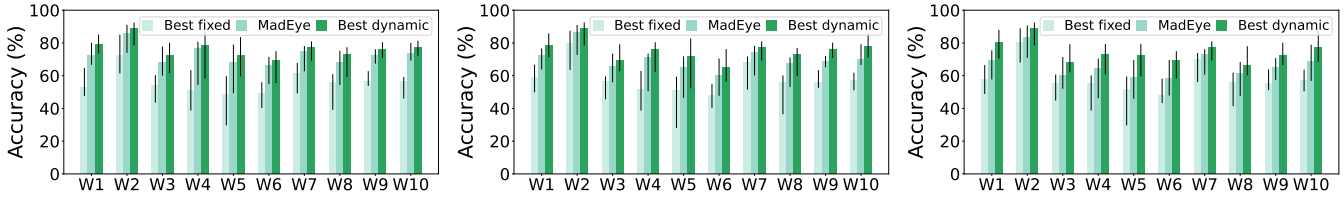
Video dataset for PTZ analysis. To the best of our knowledge, there does not exist a public video dataset for PTZ cameras that enables users to tune rotation and zoom knobs; instead, existing PTZ datasets reflect pre-determined knob decisions. Thus, to evaluate MadEye, we generate our own dataset. To construct our dataset, we begin with the abundance of 360° datasets. Concretely, we use 50 360-degree videos (5-10 mins each) from YouTube that incorporate scenes of interest resembling those from prior video analytics work [9, 33, 67], e.g., searching for people and/or cars in traffic intersections, walkways, shopping centers. From each video, we carve out the scenes of interest as regions that each span 150° horizontally and 75° vertically. We then subdivide each scene into grids of orientations to mimic recent PTZ offerings [51] (30° and 15° granularities for pan and tilt; we explore other grids in §5.4), and extract a full video per orientation. For zoom, since we operate on pre-captured videos, we employ digital zoom (1-3×) by cropping images and scaling back the dimensions to match the original image.

Models and workloads. We consider 4 popular architectures for vision tasks: SSD [71] and Faster RCNN [95] with ResNet-50 backbones, YOLOv4 and Tiny-YOLOv4 [108] with CSPDarknet53 backbones. We consider two versions of each model trained on Pascal VOC and MS-COCO, but show results for the latter as the trends were similar. To construct queries, we follow the same methodology from production analyses [82]. Each model can perform any of the 4 tasks from §2.1 with a focus on people or cars. We enumerate all possible workloads sized between 2-20 queries and pick 10 randomly. §A.2 details each workload. As in prior work [23, 82], we run each workload on all videos in our corpus that contain the objects of interest for the workload's queries. Since MadEye does not make any assumptions about frame arrival rates, we consider response rates between 1-30 fps.

Hardware and networks. On-camera computations run on an edge-grade Jetson Nano [79] equipped with a 128-core Maxwell GPU, quad-core ARM CPU with 1.43 GHz clock speed, and 4 GB of memory. We consider default camera rotation speeds of 400° per second and study this parameter in §5.4. Workload inference and training of approximation models run on a server with an NVIDIA RTX 2080 Ti GPU (8 GB RAM) and 18-core Intel Xeon 5220 CPU (2.2 GHz; 125 GB RAM). Camera and server components are connected with emulated Mahimahi networks [78] using fixed-capacity (24-60 Mbps; 5-20 ms) and real-world mobile traces.
Note. Although we run on real edge hardware and emulated networks, we do not use a deployed PTZ camera for our main evaluations as it would preclude practical consideration of diverse scenes and camera parameters, e.g., different rotation speeds. We present results with a real PTZ camera in §5.5.

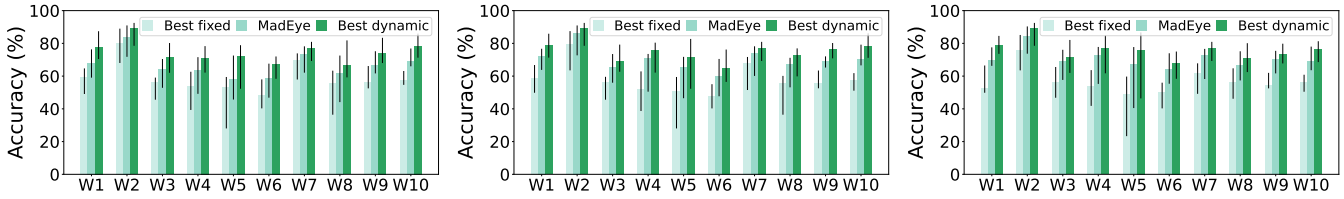
Metrics. Our primary evaluation metric is average workload accuracy per video. For each frame, following the accuracy definitions from §2.1, we compute per-orientation accuracy for each query relative to the orientation that delivers the max accuracy at that time. Per-query accuracies at each time are averaged to compute per-frame workload accuracies, which in turn are averaged to compute workload accuracy for a video.

While computing these values for binary classification and counting are straightforward, detections and aggregate counting need slight alterations. For detections, mAP scores depend on bounding box coordinates for specific objects and thus cannot be measured by comparing results across orientations. Thus, we consolidate the bounding boxes across orientations into a global view, and employ de-duplication [83] to eliminate redundant objects in overlapping regions. We then compute each orientation's mAP score relative to the global scene, and assign per-orientation accuracies as the ratio of its mAP



(a) 1 fps (b) 15 fps (c) 30 fps

Figure 12: Comparing MadEye with the best possible fixed- and adaptive-orientation schemes across all videos and workloads with a {24 Mbps, 20 ms} network and varying fps. Bars list medians with errors bars spanning 25-75th percentiles.



(a) Verizon LTE (b) 24 Mbps; 20 ms (c) 60 Mbps; 5 ms

Figure 13: Comparing MadEye with the best possible fixed- and adaptive-orientation schemes across all videos and workloads with fixed fps (15) and varying networks (improving from left to right). Bars list medians with errors bars spanning 25-75th percentiles.

score to the max one across orientations. For aggregate counting which is evaluated across an entire video, we compute the ratio of unique objects across the orientations that a system selects with the total number of unique objects in the video. Note that ByteTrack (§4) was unable to robustly support car tracking, so we exclude aggregate counting for cars.

5.2 Overall Results

We first compare MadEye with the two baselines from §2.2, *best fixed* and *best dynamic*, on different network and fps settings. Both baselines impractically rely on oracle knowledge of video content and workload accuracy, i.e., to pick the best orientation per video or per timestep, respectively, that maximizes accuracy for the target workload-video. Nonetheless, they serve as useful context for MadEye's performance. Note that MadEye automatically adapts the number of frames it explores and transmits based on network delays and response rates (§3.3). For *best fixed*, we leverage increasing network speeds by adding more fixed cameras (i.e., best, 2nd best, etc.), rather than simply capturing more (redundant) frames from 1 camera. *Best dynamic* does not change for any query other than aggregate counting, for which we send the largest number of fruitful orientations that the network can support.

Our results are captured in Figures 12-13. Across these settings, MadEye delivers median and 75th percentile accuracies that are 2.9-25.7% and 1.6-20.7% higher than *best fixed*, and within 1.8-13.9% and 1.3-12.5% of *best dynamic*. Digging deeper, our results show two key trends. First, as frame rates decrease (for a fixed network), MadEye's accuracies and wins over *best fixed* grow, e.g., for a {24 Mbps, 20 ms} network, median wins improve from 5.8-13.3% to 12.4-25.7% as fps drops from 15 to 1. The reason is that lower fps yields larger timesteps (e.g., 1 sec for 1 fps, 66.7 ms for 15 fps), enabling more exploration and/or transmission. Second, as network speeds grow (for fixed fps), the same trends persist (since each network transfer is faster) but to a lesser extent, e.g., median 15 fps wins grow to 8.6-18.4% for {60 Mbps, 5 ms}.

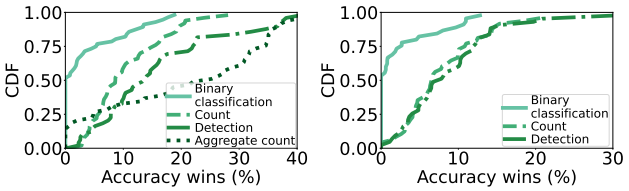


Figure 14: MadEye's accuracy wins (over best fixed) for different tasks and objects (left: people, right: cars). Results consider all videos and models, and use 15 fps and {24 Mbps; 20 ms}.

MadEye Variant	Median Accuracy (%)	# Fixed Cameras
MadEye-1	63.1	3.7
MadEye-2	66.3	5.5
MadEye-3	66.8	6.1

Table 1: Number of optimally-configured fixed cameras needed to match the accuracy of MadEye. MadEye-*k* refers to a version of MadEye that is restricted to sending the top *k* frames to the server for workload inference. Results consider a {24 Mbps; 20 ms} network, 15 fps response rate, and all video-workload pairs.

Figure 14 breaks down MadEye's wins over *best fixed* by task and object. Following the rationale from §2.2, accuracy boosts with MadEye grow as task specificity grows: median wins grow from 8.6% to 13.3% to 22.1% as we move from counting to detections to aggregate counting for people. We also observe consistently larger accuracy wins for people queries (rather than cars) due to their less structured motion patterns (more frequent and scattered orientation switches), e.g., for detections, wins for cars shrink to 6.7%.

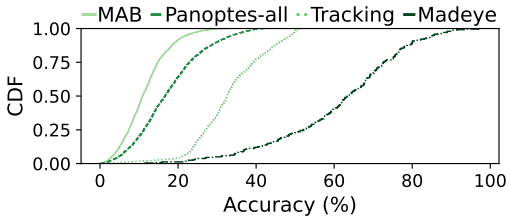


Figure 15: MadEye vs. 3 camera tuning strategies. Results are for all workloads and videos, 15 fps, and {24 Mbps; 20 ms}.

Results thus far focus on accuracy improvements. However, a key goal with MadEye is to maximize accuracy for a given resource cost, i.e., network and backend inference overheads. Table 1 lists the smallest number of optimally configured fixed cameras that would be required to match the accuracies that different versions of MadEye deliver, each of which sends a different number of frames per timestep. As shown, it would take 3.7 fixed cameras to realize the 63.1% accuracy that MadEye-1 achieves, implying a 3.7× reduction in network and backend compute usage. MadEye-2 is matched by 5.5 fixed cameras; here, however, the resource reduction factor is 2.8× since MadEye also sends 2 frames per timestep.

5.3 Comparisons with State-of-the-Art

We compare MadEye with 3 alternate approaches for adaptive camera orientations. Figure 15 shows results for a {24 Mbps; 20 ms} network and 15 fps; trends hold for all other scenarios.

First, we consider Panoptes [98], a recent PTZ system that configures orientations for workloads of applications, each explicitly concerned with specific orientation(s). For orientations of relevance, Panoptes generates a static round-robin schedule that is weighted according to how many queries an orientation is of interest to and how much motion has been detected historically in that orientation; higher weights indicate staying in an orientation for longer. Panoptes then switches between orientations according to this schedule with one exception: if motion gradients in the direction of any overlapping orientation of interest exceed a threshold, Panoptes switches there for several sec before resuming the round robin. Panoptes does not specify a zoom strategy, so we consider the best zoom (accuracy-wise) for any orientation it visits.

We consider two versions of Panoptes, *Panoptes-all* and *Panoptes-few*, in which each workload query is interested in all orientations or only its best fixed orientation, respectively. Max accuracy in both cases is defined relative to the best orientation among only the set of considered ones. As shown in Figure 15, MadEye outperforms Panoptes-all by 3.8×, with 46.8% higher accuracy at the median. The reason is that Panoptes cycles through orientations based on a pre-determined schedule and motion gradients *in the current orientation*, neither of which are sufficient indicators of importance of other orientations at the current time, e.g., orientations are suboptimal most of the time (§2.3). In contrast, MadEye considers many orientations per timestep, ranking them based on current content. The wins persist compared to Panoptes-few (not shown due to the different accuracy metric), but are less pronounced (median of 40.5%) as there are fewer unfruitful orientations for Panoptes to consider.

System	Resource reduction	Median accuracy
Chameleon [57]	2.4×	46.3%
Chameleon + MadEye	2.4×	56.1%

Table 2: MadEye preserves resource savings of recent systems, while improving accuracy. Results use 15 fps, {24 Mbps; 20 ms}.

Next, we consider tracking algorithms that most PTZ cameras come equipped with today [93]. This algorithm starts in a home region (best fixed in our experiment), selects the largest object it finds, and tracks that object continually across orientations aiming to keep it as centered as possible. The algorithm resets to the home region upon losing the tracked object. We consider a favorable variant in which all orientations explored in a timestep are shared with the backend, which uses the one with the highest accuracy. As shown, MadEye delivers 2.0× higher workload accuracies (31.1% more at the median) compared to this tracking scheme. The main reason again is that the presence of a large object is a poor indicator of accuracy importance as it fails to capture more general scene properties and specific query sensitivies.

Finally, we consider the common UCB1 multi-armed bandit (MAB) algorithm [106]. Each orientation is considered a lever with a weight set to the average observed accuracy across all past visits (we seed this with historical data). The algorithm continually selects an orientation to visit as the one with the highest sum of weighted average and upper confidence bound (which favors less-visited orientations). As with tracking, we send all visited orientations to the backend, which selects the best one per timestep. MadEye delivers 52.7% higher median accuracies than this scheme, i.e., a 5.8× win. Unlike the schemes above, MAB does factor in workload accuracies in selecting orientations. However, its adaptation considers only historical efficacy (not current content), and scene dynamics have shifted by the time it updates its patterns.

Compatibility with other optimizations. By focusing on previously un-tuned knobs (rotation and zoom) to boost accuracy, MadEye is largely compatible with prior efforts that optimize resource overheads. To illustrate this, we consider a variant of Chameleon [57] that dynamically tunes pipeline knobs (resolution and frame rate) to lower network and backend inference resource costs without harming accuracy; we brute force selections per frame focused on the best fixed orientation. We then run MadEye atop the fps and resolution selections that Chameleon makes, sending the same amount of network data. As shown in Table 2, Chameleon lowers resource costs by 2.4× compared to the naive scheme that sends all frames at the highest resolution; MadEye preserves these efficiency wins, while increasing accuracy by 9.8%.

5.4 Deep Dive Results

Rotation speeds. We evaluated the impact of camera rotation speed on MadEye's performance by considering values of

{200, 400, 500, infinite}° per second, a fixed network ({24 Mbps; 20 ms}), and 15 fps. Intuitively, accuracy grows as rotation speeds increase, e.g., jumping from 54.2% to 64.9% as rotation speed grows from 200 to 500° per second. The reason is that faster rotations enable the exploration of additional orientations or, in rarer instances, additional transmissions. Importantly, benefits plateau since most queries (other than aggregate counting) are fully satisfied accuracy-wise as long as MadEye finds the best orientation at each timestep.

Grid granularity. To understand the effect of grid granularity (with other settings fixed), we focus on the pan dimension (since it is wider) and consider steps of {15, 30, 45, 60}°. Overall, MadEye's accuracy benefits shrink as grids become more fine-grained (with more orientations), e.g., median accuracies drop from 67.5% to 51.8% when pan steps drop from 45 to 15. This is because, although exploration in a time budget is governed by rotation speeds rather than grid granularity, the same distance (in °) of exploration will warrant more approximation model inference on more orientations, thereby shrinking each timestep's exploration budget.

Overheads. On MadEye's backend, the primary overheads are in initializing approximation models and continually sharing model updates with the camera. Across our workloads, we find median bootstrapping delays to be 27 mins (including labeling and initial fine-tuning). Downlink streaming consumes 3.2 Mbps for the median experiment. Recall that both overheads are mitigated by MadEye's fine-tuning strategy (§3.2). On cameras, the main overheads are in selecting orientations to explore and running approximation models; for the median workload-video pair, per-timestep delays for each task were 17 μs and 6.7 ms for 15 fps and {24 Mbps; 20 ms}. The former benefits from pre-computed reachability analysis (§3.3).

Microbenchmarks. MadEye's performance is governed by two main tasks: (1) ranking orientations with approximation models, and (2) selecting orientations to explore to find the best one(s) per timestep. For the former, Figure 16 (in §A) show that MadEye's approximation models assign median ranks of 1.1-1.3 to the best explored orientation at each timestep, significantly outperforming the variant that relies on counting directly on images. For the latter, for the median workload-video pair on {24 Mbps; 20 ms} and 15 fps, MadEye explores best orientation 89.3% of the time, with 6.8% of errors coming from our conservative zoom strategy (§3.3).

Downlink network speeds. MadEye servers periodically ship updated weights for approximation models to cameras for orientation ranking. To understand the impact that downlink network speeds have on MadEye, we augmented the set of networks used in our main experiments with two network scenarios that deliver far slower transmissions: Narrowband-IoT ({10 Mbps; 50 ms} on average) and AT&T 3G network ({2 Mbps; 100 ms} on average). Across five representative videos and workloads (at 15 fps) from our corpus, we observed increases in weight transmission times from {11, 5, 2} seconds for Verizon LTE, {24 Mbps; 20 ms}, and {60 Mbps; 5 ms} networks to {13, 66} seconds for the two new networks. These delays, in turn, resulted in mild accuracy degradations of up to 0.9% and 2.1% relative to MadEye running on the {24 Mbps; 20 ms} network. The reason is that on such short timescales, previous approximation models deliver only moderate errors in orientation ranking as they were trained just several minutes in the past. Those errors are further bound by the fact that MadEye's search strategy moves slowly spatially to ensure that the *several* top-ranked orientations are all considered for ranking at each timestep (§3.3).

5.5 On-Camera Evaluation

To demonstrate that commodity PTZ offerings (and their corresponding tuning APIs and rotational motors) can support the computational and exploratory requirements that MadEye imposes, we ran experiments using the PTZOptics PT12X-USB [5]. We selected a random set of 6 videos and 4 workloads from our main corpus, and considered response rates of 1-30 fps and the networks from §5.1. For each scenario, we fed the video into the PTZ camera at 30 fps. MadEye's camera-side processing ran on a Jetson Nano and interacted directly with the camera through its native HTTP interface for tuning orientations; the server was the same as in §5.1.

Overall, the camera was able to support the required exploration and computation load. However, we observed several rare artifacts that deviate from our main setup: (1) seemingly random, though minor, delays in API responsiveness for tuning orientations, and (2) small delays to reach max rotation speed, which our setup failed to capture by not modeling the physical mechanics of camera motors. These discrepancies mildly reduced the number of orientations that MadEye could explore in especially high-exploration windows with scattered orientations, but the effect on accuracy was minimal: across the experiments, wins over *best fixed* dropped by <1%.

Note that these results do not evaluate the long-term wearing effect of high exploration on PTZ camera components, which falls outside the scope of this paper and our setup. However, the results in §5.4 show that MadEye can still provide substantial accuracy benefits even with highly restricted rotation speeds (and thus, exploration rates).

6 Related Work

Adapting video analytics knobs. VideoStorm [120] selects an input knob configuration (e.g., frame rate, resolution) per workload to lower resource costs and facilitate job scheduling on servers. Chameleon [57] extends such tuning to adapt to dynamic scenes, while Llama [40] further tunes knobs across heterogeneous hardware to meet latency targets. By focusing on tuning camera orientations, MadEye provides complementary benefits to these efforts, boosting accuracies while preserving their resource efficiency wins (§5.3). Other efforts focus on camera-side knobs as MadEye does. For example, CamTuner [85] learns to boost accuracy by automatically

tuning capture knobs that cameras do not usually auto-adjust, e.g., brightness and contrast. AccMPEG [34] predicts the effects of macroblock encoding settings on DNNs, and tunes encoding to maximize accuracy. MadEye shares the goal of these efforts – tune camera knobs to boost workload accuracy – but focuses on complementary knobs, i.e., orientations.

Frame filtering and result reuse. Many prior efforts exploit temporal redundancies in video data by filtering out frames for network transfer and processing, and reusing results accordingly [13, 22, 26, 29, 32, 43, 44, 65, 67, 84, 112, 121, 124]. Spatula [53] extends this to multi-camera settings, selecting among cameras in a network. These optimizations are logically similar to MadEye, which also aims to maximize accuracy per network usage. However, the techniques are largely complementary: filtering decisions could be made among explored orientations to maximize new content in transfers.

Computation and network optimizations. Several efforts lower compute footprints either by identifying lightweight model variants [19, 28, 45, 49, 52, 72, 92, 123], sharing model layers during inference [55, 82], or using smarter job scheduling [96, 120]. Other systems lower network overheads by compressing transmitted frames in a manner that is recoverable on the server or does not negatively impact accuracy [33, 37, 110]. MadEye is entirely complementary to both directions in that it solely focuses on judiciously selecting images (i.e., orientations) to process at any time for an application-provided model (which can be compressed).

Drone coordination. Many efforts adapt drone flight plans (and thus what on-board cameras see) to maximize analytics accuracy or scene coverage [15, 47, 54, 105]. However, these systems focus on identifying events of interest (e.g., wildfires) in a geographically dispersed area for a preset application. In contrast, MadEye tunes camera orientations for a single scene to cope with workload nuances and maximize accuracy.

7 Conclusion

MadEye continually tunes PTZ camera orientations to maximize accuracy for a given analytics workload and resource setting. Key to MadEye are a rapid algorithm that searches through the large space of orientations at each time, and a new, approximate knowledge distillation strategy that efficiently selects the most fruitful (accuracy-wise) orientations from those explored. Across diverse workloads and settings, MadEye improves accuracy by 2.9-25.7% or resource costs by 2.0-3.7× without affecting the other.

Acknowledgements. We thank Wyatt Lloyd, Huacheng Yu, Kevin Hsieh, Olga Russakovsky, the NSDI reviewers, and our shepherd, Danyang Zhuo, for their helpful and constructive comments. This work was supported by a Sloan Research Fellowship and NSF CNS grants 2147909, 2151630, 2140552, 2153449, and 2152313.

References

[1] C289 2MP WiFi PTZ Security Camera + Person/Vehicle Detection + 128GB Local Storage. https://www.zositech.com/products/wifi-security-camera-pan-tilt-zoom-works-with-alexa-c289?variant=44502418063602.

[2] Logitech PTZ PRO 2 . https://www.logitech.com/en-us/products/video-conferencing/conference-cameras/ptz-pro2-conferencecam.960-001184.html?utm_source=google&srsltid=AfmBOorJvnI86m27PDAg-lycP4cFBE732NYvfwAHdZ4bGdnxovP1y6VAN5s.

[3] OTTICA NDI®|HX PTZ Video Camera 20x Optical Zoom POE 1080/60p (White) . https://ikancorp.com/shop/cameras/ptz-cameras/ottica-ndihx-ptz-video-camera-20x-optical-zoom-poe-1080-60p-white/.

[4] ReoLink 5MP Smart PTZ WiFi Indoor Camera. https://reolink.com/us/product/e1-zoom/.

[5] USB PTZ Cameras. https://ptzoptics.com/usb/.

[6] Video analytics traffic study creates baseline for change. https://www.govtech.com/analytics/Video-Analytics-Traffic-Study-Creates-Baseline-for-Change.html, 2020.

[7] NVIDIA TensorRT. https://developer.nvidia.com/tensorrt, April 2021.

[8] Pytorch-yolov3. https://github.com/eriklindernoren/PyTorch-YOLOv3, 2021.

[9] Ekya: Continuous learning of video analytics models on edge compute servers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 119–135, Renton, WA, April 2022. USENIX Association.

[10] Video analytics market. https://www.fortunebusinessinsights.com/industry-reports/video-analytics-market-101114, 2022.

[11] Ganesh Ananthanarayanan, Victor Bahl, Landon Cox, Alex Crown, Shadi Nogbahi, and Yuanchao Shu. Video analytics - killer app for edge computing. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, pages 695–696, New York, NY, USA, 2019. Association for Computing Machinery.

[12] Larry Anderson. Video analytics applications in retail - beyond security. https://www.securityinformed.com/insights/co-2603-ga-co-2214-ga-co-1880-ga.16620.html/.

[13] Kittipat Apicharttrisorn, Xukan Ran, Jiasi Chen, Srikanth V. Krishnamurthy, and Amit K. Roy-Chowdhury. Frugal following: Power thrifty object detection and tracking for mobile augmented reality. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, SenSys '19, page 96–109, New York, NY, USA, 2019. Association for Computing Machinery.

[14] Guha Balakrishnan, Yuanjun Xiong, Wei Xia, and Pietro Perona. Towards causal benchmarking of biasin face analysis algorithms. In *Deep Learning-Based Face Analytics*, pages 327–359. Springer, 2021.

[15] Favyen Bastani, Songtao He, Ziwen Jiang, Osbert Bastani, and Sam Madden. Skyquery: An aerial drone video sensing platform. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2021, page 56–67, New York, NY, USA, 2021. Association for Computing Machinery.

[16] David Bau, Bolei Zhou, Aditya Khosla, Aude Oliva, and Antonio Torralba. Network dissection: Quantifying interpretability of deep visual representations. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6541–6549, 2017.

[17] Francois Beaufort. Control camera pan, tilt, and zoom – Pan, tilt, and zoom features on cameras are finally controllable on the web. https://web.dev/camera-pan-tilt-zoom/.

[18] Michael A Bender and Chandra Chekuri. Performance guarantees for the tsp with a parameterized triangle inequality. In *Algorithms and Data Structures: 6th International Workshop, WADS'99 Vancouver, Canada, August 11–14, 1999 Proceedings 6*, pages 80–85. Springer, 1999.

[19] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? *arXiv preprint arXiv:2003.03033*, 2020.

[20] Business Research Insights. Global ptz camera market research report 2020. https://www.businessresearchinsights.com/market-reports/ptz-cameras-market-100130.

[21] Zhaowei Cai, Mohammad Saberian, and Nuno Vasconcelos. Learning complexity-aware cascades for deep pedestrian detection. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, pages 3361–3369, Washington, DC, USA, 2015. IEEE Computer Society.

[22] Christopher Canel, Thomas Kim, Giulio Zhou, Conglong Li, Hyeontaek Lim, David G. Andersen, Michael Kaminsky, and Subramanya R. Dulloor. Scaling video analytics on constrained edge nodes. In *2nd SysML Conference*, 2019.

[23] Frank Cangialosi, Neil Agarwal, Venkat Arun, Junchen Jiang, Srinivas Narayana, Anand Sarwate, and Ravi Netravali. Privid: Practical, privacy-preserving video analytics queries. In *Proceedings of the 19th USENIX Conference on Networked Systems Design and Implementation*, NSDI'22, Berkeley, CA, USA, 2022. USENIX Association.

[24] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Realtime multi-person 2d pose estimation using part affinity fields. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7291–7299, 2017.

[25] David Cassel. Are we ready for ai-powered security cameras? https://thenewstack.io/are-we-ready-for-ai-powered-security-cameras/.

[26] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 155–168, 2015.

[27] Mary Collins. The Hudl Algorithm: Turning Video into Player Tracking Data. https://www.maryecollins.com/hudl-tracking.

[28] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[29] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association.

[30] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, volume 1, pages 886–893. Ieee, 2005.

[31] Sokemi Rene Emmanuel Datondji, Yohan Dupuis, Peggy Subirats, and Pascal Vasseur. A survey of vision-based traffic monitoring of road intersections. *Trans. Intell. Transport. Sys.*, 17(10):2681–2698, October 2016.

[32] Utsav Drolia, Katherine Guo, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Cachier: Edge-caching for recognition applications. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 276–286, 2017.

[33] Kuntai Du, Ahsan Pervaiz, Xin Yuan, Aakanksha Chowdhery, Qizheng Zhang, Henry Hoffmann, and Junchen Jiang. Server-driven video streaming for deep learning inference. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 557–570, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Kuntai Du, Qizheng Zhang, Anton Arapin, Haodong Wang, Zhengxu Xia, and Junchen Jiang. Accmpeg: Optimizing video encoding for accurate video analytics. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 450–466, 2022.

[35] Mengnan Du, Fan Yang, Na Zou, and Xia Hu. Fairness in deep learning: A computational perspective. *IEEE Intelligent Systems*, 36(4):25–34, 2020.

[36] Jinlong E, Lin He, Zhenhua Li, and Yunhao Liu. Wisecam: Wisely tuning wireless pan-tilt cameras for cost-effective moving object tracking. In *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*. IEEE, 2023.

[37] John Emmons, Sadjad Fouladi, Ganesh Ananthanarayanan, Shivaram Venkataraman, Silvio Savarese, and Keith Winstein. Cracking open the dnn black-box: Video analytics with dnns across the camera-cloud boundary. In *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, HotEdgeVideo'19, page 27–32, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Mark Everingham, Luc Gool, Christopher K. Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *Int. J. Comput. Vision*, 88(2):303–338, June 2010.

[39] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 267–282, USA, 2018. USENIX Association.

[40] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, Christos Kozyrakis. Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines. SoCC '21. ACM, 2021.

[41] Isha Ghodgaonkar, Subhankar Chakraborty, Vishnu Banna, Shane Allcroft, Mohammed Metwaly, Fischer Bordwell, Kohsuke Kimura, Xinxin Zhao, Abhinav Goel, Caleb Tung, et al. Analyzing worldwide social distancing through large-scale computer vision. *arXiv preprint arXiv:2008.12363*, 2020.

[42] Grand View Research. Global Sports Analytics Market Size Report, 2021-2028. https://www.grandviewresearch.com/industry-analysis/sports-analytics-market.

[43] Peizhen Guo, Bo Hu, Rui Li, and Wenjun Hu. Foggycache: Cross-device approximate computation reuse. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, page 19–34, New York, NY, USA, 2018. Association for Computing Machinery.

[44] Peizhen Guo and Wenjun Hu. Potluck: Cross-application approximate deduplication for computation-intensive mobile applications. *SIGPLAN Not.*, 53(2):271–284, mar 2018.

[45] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, page 123–136, New York, NY, USA, 2016. Association for Computing Machinery.

[46] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. Mask R-CNN. *CoRR*, abs/1703.06870, 2017.

[47] Songtao He, Favyen Bastani, Arjun Balasingam, Karthik Gopalakrishna, Ziwen Jiang, Mohammad Alizadeh, Hari Balakrishnan, Michael Cafarella, Tim Kraska, and Sam Madden. Beecluster: Drone orchestration via predictive optimization. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, MobiSys '20, page 299–311, New York, NY, USA, 2020. Association for Computing Machinery.

[48] Michael Held and Richard M Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970.

[49] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[50] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, et al. Speed/accuracy trade-offs for modern convolutional object detectors. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7310–7311, 2017.

[51] HuddleCamHD. Understanding the difference between EPTZ and PTZ . https://huddlecamhd.com/eptz-and-ptz/.

[52] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[53] Samvit Jain, Xun Zhang, Yuhao Zhou, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Victor Bahl, and Joseph Gonzalez. Spatula: Efficient cross-camera video analytics on large camera networks. In *ACM/IEEE Symposium on Edge Computing (SEC 2020)*, November 2020.

[54] Sagar Jha, Youjie Li, Shadi Noghabi, Vaishnavi Ranganathan, Peeyush Kumar, Andrew Nelson, Michael Toelle, Sudipta Sinha, Ranveer Chandra, and Anirudh Badam. Visage: Enabling timely analytics for drone imagery. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, MobiCom '21, page 789–803, New York, NY, USA, 2021. Association for Computing Machinery.

[55] Angela H. Jiang, Daniel L.-K. Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A. Kozuch, Padmanabhan Pillai, David G. Andersen, and Gregory R. Ganger. Mainstream: Dynamic stem-sharing for multi-tenant video processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 29–42, Boston, MA, July 2018. USENIX Association.

[56] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: Scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 253–266, New York, NY, USA, 2018. ACM.

[57] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: Scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 253–266, New York, NY, USA, 2018. Association for Computing Machinery.

[58] Daniel Kang, Peter Bailis, and Matei Zaharia. Blazeit: Optimizing declarative aggregation and limit queries for neural network-based video analytics. *Proc. VLDB Endow.*, 13(4):533–546, December 2019.

[59] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: Optimizing neural network queries over video at scale. *Proc. VLDB Endow.*, 10(11):1586–1597, August 2017.

[60] Kai Kang, Hongsheng Li, Junjie Yan, Xingyu Zeng, Bin Yang, Tong Xiao, Cong Zhang, Zhe Wang, Ruohui Wang, Xiaogang Wang, and Wanli Ouyang. T-CNN: Tubelets With Convolutional Neural Networks for Object Detection From Videos. *IEEE Trans. Cir. and Sys. for Video Technol.*, 28(10):2896–2907, October 2018.

[61] Kai Kang, Wanli Ouyang, Hongsheng Li, and Xiaogang Wang. Object detection from video tubelets with convolutional neural networks. In *CVPR*, 2016.

[62] Ronald Kemker, Marc McClure, Angelina Abitino, Tyler Hayes, and Christopher Kanan. Measuring catastrophic forgetting in neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018.

[63] Aditya Khosla, Tinghui Zhou, Tomasz Malisiewicz, Alexei A Efros, and Antonio Torralba. Undoing the damage of dataset bias. In *Computer Vision–ECCV 2012: 12th European Conference on Computer Vision, Florence, Italy, October 7-13, 2012, Proceedings, Part I 12*, pages 158–171. Springer, 2012.

[64] Judy Goldsmith Kshitija Taywade, Brent Harrison. Using non-stationary bandits for learning in repeated cournot games with non-stationary demand. *arXiv preprint arXiv:2201.00486*, 2022.

[65] Adarsh Kumar, Arjun Balasubramanian, Shivaram Venkataraman, and Aditya Akella. Accelerating deep learning inference via freezing. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[66] H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua. A convolutional neural network cascade for face detection. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5325–5334, June 2015.

[67] Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 359–376, New York, NY, USA, 2020. Association for Computing Machinery.

[68] Zhuqi Li, Yuanchao Shu, Ganesh Ananthanarayanan, Longfei Shangguan, Kyle Jamieson, and Victor Bahl. Spider: A multi-hop millimeter-wave network for live video analytics. In *ACM/IEEE Symposium on Edge Computing*. ACM/IEEE, December 2021.

[69] Michael Lin, Tsung-Yiand Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: Common objects in context. In *Computer Vision – ECCV 2014*, pages 740–755, 2014.

[70] T. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 936–944, July 2017.

[71] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. In *Computer Vision – ECCV 2016*, pages 21–37. Springer International Publishing, 2016.

[72] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2736–2744, 2017.

[73] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60, 2004.

[74] Yao Lu, Aakanksha Chowdhery, and Srikanth Kandula. Optasia: A relational platform for efficient large-scale video analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 57–70, New York, NY, USA, 2016. ACM.

[75] Mehrdad Khani, Ganesh Ananthanarayanan, Kevin Hsieh, Junchen Jiang, Ravi Netravali , Yuanchao Shu, Mohammad Alizadeh , Victor Bahl. Recl: Responsive resource-efficient continuous learning for video analytics. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, Boston, MA, April 2023. USENIX Association.

[76] Microsoft Azure. Azure stack edge, May 2021.

[77] Neil Agarwal, Ravi Netravali. Boggart: Towards General-Purpose Acceleration of Retrospective Video Analytics. NSDI '23. USENIX, 2023.

[78] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. Proceedings of ATC '15. USENIX, 2015.

[79] NVIDIA. Nvidia jetsonnano, April 2021.

[80] Kemal Oksuz, Baris Can Cam, Sinan Kalkan, and Emre Akbas. Imbalance problems in object detection: A review. *IEEE transactions on pattern analysis and machine intelligence*, 43(10), 2020.

[81] Omar Besbes, Yonatan Gur, Assaf Zeevi. Stochastic Multi-Armed-Bandit Problem with Non-stationary Rewards . In *NeurIPS*, 2014.

[82] Arthi Padmanabhan, Neil Agarwal, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Guoqing Harry Xu, and Ravi Netravali. Gemel: Model merging for memory-efficient, real-time video analytics at the edge, 2022.

[83] Xunyu Pan and Siwei Lyu. Detecting image region duplication using sift features. In *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 1706–1709. IEEE, 2010.

[84] Sibendu Paul, Utsav Drolia, Y. Charlie Hu, and Srimat T. Chakradhar. Aqua: Analytical quality assessment for optimizing video analytics systems, 2021.

[85] Sibendu Paul, Kunal Rao, Giuseppe Coviello, Murugan Sankaradas, Oliver Po, Y. Charlie Hu, and Srimat Chakradhar. Enhancing video analytics accuracy via real-time automated camera parameter tuning. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, SenSys '22, page 291–304, New York, NY, USA, 2023. Association for Computing Machinery.

[86] PTZ Optics. PTZ Optics PTZ cameras. https://ptzoptics.com/products/.

[87] PTZ Optics. What is ePTZ and how does it compare with true PTZ? https://ptzoptics.com/what-is-eptz/.

[88] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[89] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, Oliver Wang. The Unreasonable Effectiveness of Deep Features as a Perceptual Metric. In *CVPR*, 2018.

[90] Mohammed Rijas. Powering the edge with ai in an iot world. https://www.forbes.com/sites/forbestechcouncil/2020/04/06/powering-the-edge-with-ai-in-an-iot-world/.

[91] Alberto Rizzoli. 7 Game-Changing AI Applications in the Sports Industry. https://www.v7labs.com/blog/ai-in-sports, 2022.

[92] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411. USENIX Association, July 2021.

[93] SCW. PTZ Auto-Tracking Explained. https://www.getscw.com/knowledge-base/auto-tracking-explained.

[94] SecurityBros. Which Cheap Outdoor WiFi PTZ IP Camera is Best? Boavision vs Inqmega. https://securitybros.com/which-cheap-outdoor-wifi-ptz-ip-camera-is-best-boavision-vs-inqmega/.

[95] Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In *NeurIPS*, 2015.

[96] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 322–337, New York, NY, USA, 2019. Association for Computing Machinery.

[97] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[98] Shubham Jain, Viet Nguyen, Marco Gruteser, Paramvir Bahl. Panoptes: Servicing Multiple Applications Simultaneously using Steerable Cameras. In *IPSN*, 2017.

[99] Vishwanath A Sindagi and Vishal M Patel. Generating high-quality crowd density maps using contextual pyramid cnns. In *Proceedings of the IEEE international conference on computer vision*, pages 1861–1870, 2017.

[100] Ryan Steed and Aylin Caliskan. Image representations learned with unsupervised pre-training contain human-like biases. In *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*, pages 701–713, 2021.

[101] Abhijit Suprem, Joy Arulraj, Calton Pu, and Joao Ferreira. Odin: Automated drift detection and recovery in video analytics. *Proc. VLDB Endow.*, 13(12):2453–2465, July 2020.

[102] SV3C. SV3C Security Camera Outdoor . https://www.sv3c.com/.

[103] Mingxing Tan, Ruoming Pang, and Quoc V. Le. Efficientdet: Scalable and efficient object detection. *CoRR*, abs/1911.09070, 2019.

[104] Amos Tversky and Itamar Gati. Similarity, separability, and the triangle inequality. *Psychological review*, 89(2):123, 1982.

[105] Deepak Vasisht, Zerina Kapetanovic, Jong-ho Won, Xinxin Jin, Ranveer Chandra, Ashish Kapoor, Sudipta N. Sinha, Madhusudhan Sudarshan, and Sean Stratman. Farmbeats: An iot platform for data-driven agriculture. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 515–528, USA, 2017. USENIX Association.

[106] Lluís Martínez; Margarita Cabrera-Bean; Josep Vidal. A Multi-Armed Bandit Model for Non-Stationary Wireless Network Selection. In *2021 IEEE Globecom Workshops*. Ieee, 2021.

[107] Vidit Saxena, Joakim Jaldén, Joseph E. Gonzalez, Mats Bengtsson, Hugo Tullberg, Ion Stoica. Contextual Multi-Armed Bandits for Link Adaptation in Cellular Networks. In *NetAI*, 2019.

[108] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Scaled-yolov4: Scaling cross stage partial network, 2021.

[109] Junjue Wang, Ziqiang Feng, Shilpa George, Roger Iyengar, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards scalable edge-native applications. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, SEC '19, page 152–165, New York, NY, USA, 2019. Association for Computing Machinery.

[110] Yiding Wang, Weiyan Wang, Junxue Zhang, Junchen Jiang, and Kai Chen. Bridging the edge-cloud barrier for real-time advanced vision analytics. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[111] Zeyu Wang, Klint Qinami, Ioannis Christos Karakozis, Kyle Genova, Prem Nair, Kenji Hata, and Olga Russakovsky. Towards fairness in visual recognition: Effective strategies for bias mitigation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 8919–8928, 2020.

[112] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. Deepcache: Principled cache for mobile deep vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, page 129–144, New York, NY, USA, 2018. Association for Computing Machinery.

[113] Yifan Yang, Guorong Li, Dawei Du, Qingming Huang, and Nicu Sebe. Embedding perspective analysis into multi-column convolutional neural network for crowd counting. *IEEE Transactions on Image Processing*, 30:1395–1407, 2020.

[114] Yifu Zhang, Peize Sun, Yi Jiang, Dongdong Yu, Zehuan Yuan, Ping Luo, Wenyu Liu, Xinggang Wang. Bytetrack: Multi-object tracking by associating every detection box, 2021.

[115] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. *SIGCOMM Comput. Commun. Rev.*, 45(4):325–338, aug 2015.

[116] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[117] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13*, pages 818–833. Springer, 2014.

[118] Anran Zhang, Lei Yue, Jiayi Shen, Fan Zhu, Xiantong Zhen, Xianbin Cao, and Ling Shao. Attentional neural fields for crowd counting. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 5714–5723, 2019.

[119] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 377–392, Berkeley, CA, USA, 2017. USENIX Association.

[120] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 377–392, Boston, MA, March 2017. USENIX Association.

[121] Tan Zhang, Aakanksha Chowdhery, Paramvir Bahl, Kyle Jamieson, and Suman Banerjee. The design and implementation of a wireless video surveillance system. pages 426–438, 09 2015.

[122] Kun Zhao, Bin Liu, Luchuan Song, Weihai Li, and Nenghai Yu. Cascaded residual density network for crowd counting. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 2199–2203. IEEE, 2019.

[123] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

[124] Yuhao Zhu, Anand Samajdar, Matthew Mattina, and Paul Whatmough. Euphrates: Algorithm-soc co-design for low-power mobile continuous vision. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 547–560. IEEE Press, 2018.

A Appendix

A.1 Additional objects and tasks

To evaluate MadEye's generality, we ran experiments that target different object types (counting lions and elephants in safari videos using FasterRCNN and SSD) or tasks (find sitting people using the OpenPose [24] pose estimation model). Importantly, MadEye did not require any special tuning, and instead learned the (new) objects and tasks of interest via training on the original models' results. In line with our main evaluation results, MadEye improved accuracies over *best fixed* by 4.6 - 14.5% and 2.8 - 10.9% for lions and elephants respectively and by 9.5 - 17.1% for pose estimation when using 400° per second rotation speeds and a {24 Mbps, 20 ms} network. The differences in improvements can be explained by the dynamics of the videos – elephants in the videos were largely static so *best fixed* performed well; in contrast, the videos used for identifying lions and sitting people had many moving objects which necessitated frequent orientation switches.

A.2 Workloads

Model	Object	Type
SSD	people	aggregate count
Faster RCNN	cars	binary classification
SSD	people	count
YOLOv4	people	detection
Faster RCNN	people	detection

Table 3: Workload 1 (W1)

Model	Object	Type
YOLOv4	people	aggregate count
Tiny YOLOv4	people	aggregate count
Tiny YOLOv4	people	detection
YOLOv4	people	binary classification
Tiny YOLOv4	people	aggregate count
Faster RCNN	people	count
Faster RCNN	people	detection
Faster RCNN	car	count
YOLOv4	people	aggregate count
YOLOv4	people	detection
YOLOv4	people	count
Tiny YOLOv4	people	aggregate count
YOLOv4	car	count
YOLOv4	car	detection
Tiny YOLOv4	car	count
SSD	person	binary classification
Faster RCNN	car	count
SSD	car	count

Table 4: Workload 2 (W2)

Model	Object	Type
SSD	car	binary classification
Faster RCNN	people	aggregate count
Faster RCNN	people	count
Tiny YOLOv4	people	binary classification
Tiny YOLOv4	people	binary classification
Tiny YOLOv4	people	aggregate count
YOLOv4	people	count
Faster RCNN	people	aggregate count
SSD	people	binary classification
Faster RCNN	car	count
SSD	car	count

Table 5: Workload 3 (W3)

Model	Object	Type
Tiny YOLOv4	car	count
Faster RCNN	car	detection
Faster RCNN	people	aggregate count

Table 6: Workload 4 (W4)

Model	Object	Type
Tiny YOLOv4	car	count
SSD	car	count
Faster RCNN	people	aggregate count

Table 7: Workload 5 (W5)

Model	Object	Type
Tiny YOLOv4	people	aggregate count
Tiny YOLOv4	people	binary classification
SSD	car	count
YOLOv4	people	aggregate count
Tiny YOLOv4	people	count
Faster RCNN	car	binary classification
SSD	people	detection
Faster RCNN	car	detection
Faster RCNN	people	aggregate count
YOLOv4	car	count
Tiny YOLOv4	people	aggregate count
Faster RCNN	people	detection
SSD	people	aggregate count
YOLOv4	car	detection

Table 8: Workload 6 (W6)

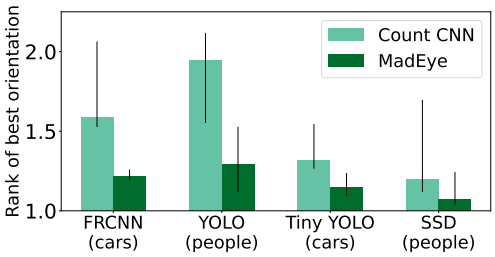


Figure 16: Comparing different approximation model designs: MadEye's lightweight detection models and compressed counting models (Count CNN). Results use all videos, {24 Mbps; 20 ms}, 15 fps, and list median rank assigned to the best explored orientation at each timestep (error bars for 25-75th percentiles).

Model	Object	Type
YOLOv4	people	binary classification
SSD	people	detection
Tiny YOLOv4	car	binary classification
Tiny YOLOv4	people	detection
SSD	people	binary classification
SSD	people	aggregate count
Tiny YOLOv4	people	detection
SSD	car	count
SSD	people	count
Faster RCNN	people	count
YOLOv4	people	count
Faster RCNN	people	binary classification
Tiny YOLOv4	people	aggregate count
Faster RCNN	people	aggregate count
Faster RCNN	car	count
YOLOv4	car	binary classification

Table 9: Workload 7 (W7)

Model	Object	Type
Faster RCNN	car	count
Tiny YOLOv4	people	binary classification
YOLOv4	people	aggregate count
YOLOv4	car	count
Tiny YOLOv4	people	aggregate count
Faster RCNN	people	aggregate count
YOLOv4	people	aggregate count
Faster RCNN	car	count
SSD	car	count
Faster RCNN	car	count
SSD	car	binary classification
YOLOv4	car	binary classification
SSD	car	binary classification
SSD	people	count
YOLOv4	people	count
YOLOv4	car	binary classification
Faster RCNN	person	aggregate count
SSD	car	detection

Table 10: Workload 8 (W8)

Model	Object	Type
Tiny YOLOv4	people	aggregate count
Faster RCNN	people	count
Faster RCNN	people	count
Tiny YOLOv4	car	detection
Tiny YOLOv4	people	binary clasification
YOLOv4	people	detection
Faster RCNN	people	count
YOLOv4	people	aggregate count
SSD	people	aggregate count

Table 11: Workload 9 (W9)

Model	Object	Type
Faster RCNN	people	aggregate count
Faster RCNN	car	count
Faster RCNN	people	count

Table 12: Workload 10 (W10)

Gemino: Practical and Robust Neural Compression for Video Conferencing

Vibhaalakshmi Sivaraman, Pantea Karimi, Vedantha Venkatapathy, Mehrdad Khani
Sadjad Fouladi ⊞, Mohammad Alizadeh, Frédo Durand, Vivienne Sze

Massachusetts Institute of Technology, ⊞ *Microsoft Research*

Abstract

Video conferencing systems suffer from poor user experience when network conditions deteriorate because current video codecs simply cannot operate at extremely low bitrates. Recently, several neural alternatives have been proposed that reconstruct talking head videos at very low bitrates using sparse representations of each frame such as facial landmark information. However, these approaches produce poor reconstructions in scenarios with major movement or occlusions over the course of a call, and do not scale to higher resolutions. We design Gemino, a new neural compression system for video conferencing based on a novel *high-frequency-conditional super-resolution* pipeline. Gemino upsamples a very low-resolution version of each target frame while enhancing high-frequency details (e.g., skin texture, hair, etc.) based on information extracted from a single high-resolution reference image. We use a multi-scale architecture that runs different components of the model at different resolutions, allowing it to scale to resolutions comparable to 720p, and we personalize the model to learn specific details of each person, achieving much better fidelity at low bitrates. We implement Gemino atop *aiortc*, an open-source Python implementation of WebRTC, and show that it operates on 1024×1024 videos in real-time on a Titan X GPU, and achieves 2.2–5× lower bitrate than traditional video codecs for the same perceptual quality.

1 Introduction

Video conferencing applications have become a crucial part of modern life. However, today's systems continue to suffer from poor user experience: in particular, poor video quality and unwelcome disruptions are all too common. Many of these problems are rooted in the inability of today's applications to operate in low-bandwidth scenarios. For instance, Zoom recommends a minimum bandwidth of 1.2 Mbps for one-on-one meetings and 2-3 Mbps for group meetings [1]. In certain parts of the world, Internet broadband speeds remain insufficient for reliable video conferencing. For example, large swaths of the population in Africa and Asia had average Internet broadband speeds less than 10 Mbps in 2022 [2], with the five slowest countries having speeds under 1 Mbps. Mobile bandwidth is even more restricted: SpeedTest's Global Index [3] suggest that global mobile bandwidth average is 50% of broadband speeds. Even in regions of North America and Europe with high broadband speeds [2], over 30% of users surveyed about their video conferencing experience claimed that "video quality" issues were their biggest pain point [4]. This is because the user experience is not just determined by the average bandwidth, but rather by tail events of low bandwidth (a few seconds every 5–10 minutes) that cause glitches and disrupt the video call. When the network deteriorates, even briefly, existing video conferencing solutions cope to an extent by lowering quality, but below a certain bandwidth (e.g., 100s of Kbps for HD video), they must either suspend the transmission altogether or risk packet loss and frame corruption.

Recently, several neural approaches for face image synthesis have been proposed that deliver extreme compression by generating each video frame from a sparse representation (*e.g.,* keypoints) [5–9]. These techniques have the potential to enable video conferencing with one to two orders of magnitude reduction in bandwidth (as low as ∼10 Kbps [6, 8]), but their lack of robustness and high computational complexity hampers their practicality. Specifically, synthesis approaches work by "warping" a reference image into different target poses and orientations captured by such sparse keypoints. These methods produce good reconstructions when the difference between the reference and the target image is small, but they fail (possibly catastrophically) in the presence of large movements or occlusions. In such cases, they produce poor reconstructions, for both low-frequency content (*e.g.,* missing the presence of a hand in a frame altogether) and high-frequency content (*e.g.,* details of clothing and facial hair). As a result, while synthesis approaches show promising average-case behavior, their performance at the tail is riddled with inconsistencies in practice. Furthermore, real-time reconstruction is only feasible at low resolution for such models [5, 8], even on high-end GPUs, while typical video conference applications are designed for HD, Full HD, and even 4K videos [10, 11]. Naïvely reusing these models on larger input frames can quickly become prohibitively expensive as the resolution is increased.

We present Gemino, a neural compression system for low-bitrate video conferencing, designed to overcome the above robustness and compute complexity challenges. Gemino targets extreme compression scenarios, such as delivering video in ∼100 Kbps or less. At such bitrates, the bandwidth required for video becomes comparable to a typical audio call [12], greatly expanding the range of networks that can support video conferencing.

Gemino's design begins with the observation that current

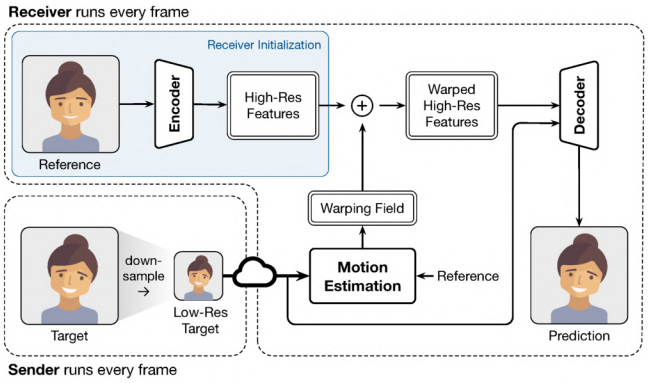


Figure 1: Gemino's design. The sender sends a downsampled version of the target frame across the network to the receiver that has a reference image of the speaker in the current video conference setting. The encoder network encodes and warps features from the reference image based on the motion estimated between the reference and target frames. The decoder network at the receiver upsamples the downsampled frame with help from the warped encoded reference features.

synthesis approaches, in an effort to squeeze the most compression, overly rely on keypoints, a modality with limited semantic information about the target frame. This causes several inevitable failures (§2). For example, if a user's hand is absent in the reference frame but appears in the target, there is no way to reconstruct the hand by warping the reference image. We would need to send a new reference frame that includes the hand, but sending a high-resolution frame (even occasionally) incurs significant cost. Instead, Gemino directly transmits low-resolution video, which includes significantly more information about the target frame, and upsamples it to the desired resolution at the receiver. Using low-resolution video is viable because modern codecs [13–16] compress them very efficiently. For example, sending 128×128 resolution video in our implementation consumes ∼15 Kbps, only slightly more than would be needed to transmit keypoints [6, 8]. We posit that the robustness benefits of providing the receiver with more information for reconstruction far outweigh the marginal bandwidth cost.

It is challenging to upsample a video significantly while reconstructing high-frequency details accurately. For example, at 1024×1024 resolution, we can see high-frequency texture details of skin, hair, and clothing, that are not visible at 128×128 resolution. To improve high-frequency reconstruction fidelity, Gemino uses a reference frame that provides such texture information in a different pose than the target frame. Like synthesis approaches, it warps features extracted from this reference frame based on the motion between the reference and target frames, but it combines it with information extracted from the low-resolution target image to generate the final reconstruction (Fig. 1). We call this new approach *high-frequency-conditional super-resolution*.

Gemino uses several further optimizations to improve reconstruction fidelity and reduce computation cost. First, we *personalize* the model by fine-tuning it on videos of a specific person so that it can learn the high-frequency detail associated with that person (e.g., hair, skin wrinkles, *etc.*). Personalizing the model makes it easier to transition texture information from the reference to the target frame. Second, we train the neural network using decompressed low-resolution frames obtained from a standard codec so that it learns to reconstruct accurate target frames despite codec-induced artifacts. We design Gemino to work with any starting resolution so that it can achieve different rate-distortion tradeoffs based on the available network bandwidth. The resolution at a particular bitrate is chosen by profiling standard codecs and picking the highest resolution that can achieve that bitrate.

Lastly, as the target resolution increases, it is essential to reduce the number of operations required per-pixel for synthesis. Otherwise, the compute overheads of running neural networks at higher resolutions become prohibitively high. To achieve this compute reduction, we design a *multi-scale* architecture wherein different parts of the model operate at different resolutions. For instance, the module that produces the warping field uses low-resolution versions of the reference and target images, while the encoder and decoder networks[1] operate at full resolution but are equipped with additional downsampling blocks to reduce the operations per pixel. This design allows us to obtain good reconstruction quality while keeping the reconstruction real time. The multi-scale architecture will be particularly salient as we transition to higher resolution video conferencing applications in the future. We also further optimize our model using neural architecture search techniques such as NetAdapt [18] to reduce the compute footprint.

We implement and evaluate Gemino within *aiortc* [19], a Python implementation of WebRTC, and show the following:

1. Gemino achieves a perceptual quality (LPIPS) [20] of 0.21 at 105 Kbps, a 5× and 2.2× reduction from VP8 and VP9's default Chromium and WebRTC settings respectively.
2. In lower bitrate regimes where current video conferencing applications *cannot* operate, Gemino outperforms bicubic upsampling and SwinIR [21] super-resolution from compressed downsampled VPX [2] frames. Gemino requires 50 Kbps to achieve an LPIPS of 0.23, a 2× and 3.9× reduction compared to bicubic and super-resolution.
3. Our model transitions smoothly across resolutions, and between generated and synthetic video, achieving different rate-distortion tradeoffs based on the target bitrate.
4. Our model benefits considerably (∼ 0.04 in LPIPS) from personalization and including the VPX codec at train-time (∼0.05 in LPIPS). Our optimizations atop the multi-scale architecture enable real-time inference on 1024×1024 frames on a Titan X GPU.

[1] "Encoder" and "decoder" throughout this paper refer to the neural network pair typically used in GANs [17]. When we specify *VPX* encoder or decoder, we refer to the video codec's encoder and decoder.

[2] VPX refers jointly to VP8 and VP9; we delineate them when necessary.

2 Related Work

Traditional Codecs. Most video applications rely on standard video compression modules (codecs) such as H.264/H.265 [22, 23], VP8/VP9 [24, 25], and AV1 [13]. These codecs separate video frames into keyframes (I-frames) that exploit spatial redundancies within a frame, and predicted frames (P-/B-frames) that exploit temporal—as well as spatial—redundancies across frames. Over the years, these standards have been improved through ideas like variable block sizes [23] and low-resolution encoding for lower bitrates [13]. These codecs are particularly efficient in their *slow* modes when they have generous time and compute budget to compress a video at high quality. However, these codecs still require a few hundred Kbps for real-time applications such as video conferencing, even at moderate resolutions like 720p. In low-bandwidth scenarios, these codecs cannot do much other than transmit at the worst quality, and suffer packet loss and frame corruption [26]. To circumvent this, some applications [27] switch to lower resolutions when the network degrades. However, as new video conferencing solutions such as Google's Starline [11] with a large bandwidth footprint are introduced, these concerns with current codecs become more acute.

Super-resolution. Linear single-image super-resolution (SR) methods [28, 29] provide robust quality enhancements in various contexts. Neural SR methods have further enhanced the upsampling quality by learning better interpolation or in-painting methods [21, 30–32]. Video SR methods [33, 34] build on image SR but further improve the reconstruction by exploiting redundant information in adjacent low-res video frames. Certain approaches like FAST [35] and Nemo [36] further optimize SR for video generation by performing SR only on "anchor frames" and generating the rest by upsampling motion vectors and residuals. For video conferencing, domain-specific SR has also shown promising outcomes utilizing facial characteristics and training losses in their models [37, 38]. However, to the best of our knowledge, none of these prior methods study upsampling conditioned on a high-resolution image from the same context. Unlike pure SR methods, Gemino provides access to high-resolution reference frames and learns models that jointly in-paint and propagate high-frequency details from the reference frame. In recent work, SRVC [39] uses content-specific super-resolution to upsample a low-resolution video stream. Our approach is similar to SRVC in that it designs a model adapted to a specific person. However, to enable real-time encoding, Gemino only customizes the model once per person rather than continuously adapting it throughout the video.

Neural Codecs. The inability of traditional codecs to operate at extremely low bitrates for high-resolution videos has led researchers to consider neural approaches that reconstruct video from very compact representations. Neural codecs have been designed for video-streaming [39–41], live-video [42], and video video conferencing [6, 8]. Swift [41] learns to compress and decompress based on the residuals in a layered-encoding stack. Both NAS [40] and LiveNAS [42] enhance video quality using one or more DNN models at either the client for video streaming, or the ingest server for live video. The models have knobs to control the compute overheads by using a smaller Deep Neural Network (DNN) [40], or by adjusting the number of epochs over which they are fine-tuned online [42]. All of these approaches have shown improvements in the bits-per-pixel consumption across a wide range of videos.

However, video conferencing differs from other video applications in a few ways. First, the video is unavailable ahead of time to optimize for the best compression-quality tradeoff. Moreover, the interactivity of the application demands that the video be both compressed and decompressed with low-latency. Second, the videos belong to a specific distribution consisting primarily of facial data. This allows for a more targeted model for generating videos of faces. A number of such models have been proposed [5–9, 43, 44] over the years. These models use keypoints or facial landmarks as a compact intermediary representation of a specific pose, to compute the movement between two poses before generating the reconstruction. The models may use 3D keypoints [6], off-the-shelf keypoint detectors [7], or multiple reference frames [9] to enhance prediction.

Challenges for neural face image synthesis. Neural synthesis approaches and specifically, keypoint-based models fall short in a number of ways that make them impractical in a video conferencing setting. These models operate similarly to the model described in Fig. 1 but do not transmit or use the downsampled target frame. They extract keypoints from the downsampled target frame, and transmit those instead. This choice causes major reconstruction failures when the reference and target frames are not close. Fig. 2 shows the reconstruction produced by the First-Order-Motion Model (FOMM) [5], a keypoint-based model, on 1024×1024 frames. We focus on the FOMM as a representative keypoint-based model, but these limitations extend to other such models. The FOMM only produces blurry outlines of the faces in rows 1 and 3 where the reference and target differ in orientation and zoom level respectively. In row 2, the FOMM misses the arm altogether because it was not present in the reference frame and warping alone cannot convey the arm's presence. Such failures occur because keypoints are limited in their representation of differences across frames, and most warping fields cannot be modeled as small deformations around each keypoint. Poor prediction quality in the event of such movements in video calls seriously disrupts the user experience.

Secondly, even in regions without much movement between the reference and the target frames, current approaches do not have good fidelity to high-frequency details. In row 2 of Fig. 2, the microphone does not move much between the reference and the target, but possesses a lot of high-frequency detail in its grille and stand. Yet, the FOMM has a poor reconstruction of that area. In row 1 of Fig. 2, it misses even those details in the hair that are similar between the reference and the target. This issue becomes more pronounced at higher resolutions where more high-frequency content is present in each frame,

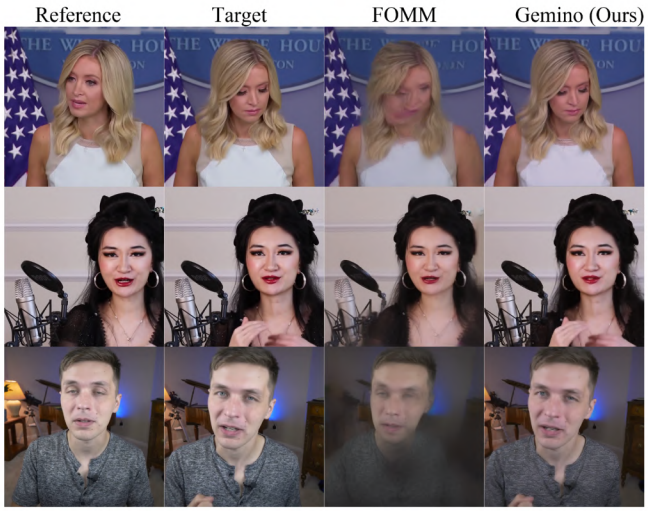


Figure 2: Failure cases in the FOMM's [5] reconstruction when the reference and target differ. FOMM misses the hand (row 2) because the reference frame does not contain it. In rows 1 and 3, FOMM only produces a blurry outline of the face (and torso). Gemino improves on these reconstructions by utilizing the downsampled target frame in its architecture and capturing the difference between the reference and the target better.

and where the human eye is more sensitive to missing details.

Lastly, many existing approaches [5, 8, 9] are evaluated on 256×256 images [45]. However, typical video conferences are at least 720p, especially in the full-speaker view. While the convolutions in these models allow them to scale to larger resolutions with the same kernels, the inference times often exceed the real-time deadline (33ms per frame for 30 frames-per-second) at high resolutions. We observe this in Tab. 1 which shows the inference time of the FOMM [5] on different GPU systems at different resolutions. This is unsurprising: running the FOMM unmodified at different resolutions performs the same number of operations per-pixel on 1024×1024 and 256×256 frames even though each pixel represents a smaller region of interest in the former case. With 16× more input pixels, the model's encoded features are also larger. This suggests that we need to redesign neural codecs to operate at higher resolutions without significant compute overheads, particularly as we move towards 4K videos.

3 System Design

3.1 Overview

Our goal is to design a robust neural compression system that reconstructs both low and high frequency content in the target frame with high fidelity even under extreme motion and variance across target frames. Ideally, such a system should also be able to operate at high resolution without significant compute overheads. Our prototype operates at 1024×1024 resolution.

Our key insight is that we need to go beyond the limited representation of movement that keypoints or facial landmarks alone provide. Fortunately, modern video codecs are very efficient in compressing low-resolution (LR) video frames. For instance, VPX compresses 128×128 resolution frames (8× downsampled in each dimension from our target resolution) using only 15 Kbps. Motivated by this observation and the fact that downsampled frames possess much more semantic information than keypoints, we design a model that performs *super-resolution conditioned on high-frequency textures* from a full-resolution reference frame. More specifically, our model resolves a downsampled frame at the receiver to its full resolution, aided by features derived from a full-resolution reference frame.

	First-Order-Motion Model			Gemino	Gemino (Opt.)
GPU	256×256	512×512	1024×1024	1024×1024	1024×1024
Titan X	17.35 ms	50.13 ms	187.30 ms	68.08ms	26.81ms
V100	12.48 ms	33.11 ms	117.27 ms	41.23ms	17.42ms
A100	8.19 ms	12.76 ms	32.96 ms	17.66ms	13.89ms

Table 1: Inference time of the First-Order-Motion Model [5] at different resolutions on different GPU systems. Prediction time increases with resolution, making it hard to reconstruct using the FOMM at 1024×1024 in real-time. The multi-scale architecture (third column) and further optimizations (fourth column) allow Gemino to achieve real-time inference at 1024×1024 on NVIDIA V100, A100, and Titan X systems.

3.2 Model Architecture

Fig. 3 describes Gemino's model architecture running at the receiver of a video conferencing session. We assume that the receiver has access to a single *high-resolution reference frame*, capturing what the speaker looks like in that particular session. It also receives a stream of *low-resolution target frames* that are compressed by a traditional video codec. ① The receiver first takes the decompressed low-resolution (LR) target that it receives from the sender over the network and runs it through a single convolutional layer to produce LR features. ② Next, the reference frame is downsampled and supplied, along with the (decompressed) LR target, to a motion estimation module that consists of a UNet [46]. ③ Since the two frames typically differ in their axes alignment, the motion estimator uses the keypoints to output a "warping field" or a transformation to move the reference frame features into the target frame's coordinates. ④ Meanwhile, the full-resolution reference frame is fed through a series of convolutional layers that encode it to extract high-resolution (HR) features. ⑤ The model applies the transformation produced by the motion estimator to the HR features to produce a set of warped (or rotated) HR features in the target frame's coordinate system.

Once the warped and encoded features from both the LR and HR pipelines are obtained, the model combines them before decoding the features to synthesize the target image. Specifically, ⑥ the model uses three inputs to produce the final reconstruction: Ⓐ the warped HR features, Ⓑ the HR features prior to warping, and Ⓒ the LR features. Each of these inputs on a feature-by-feature basis with three occlusion masks of the same dimension, produced by the motion estimator. These occlusion masks sum up to 1 and describe how to weigh

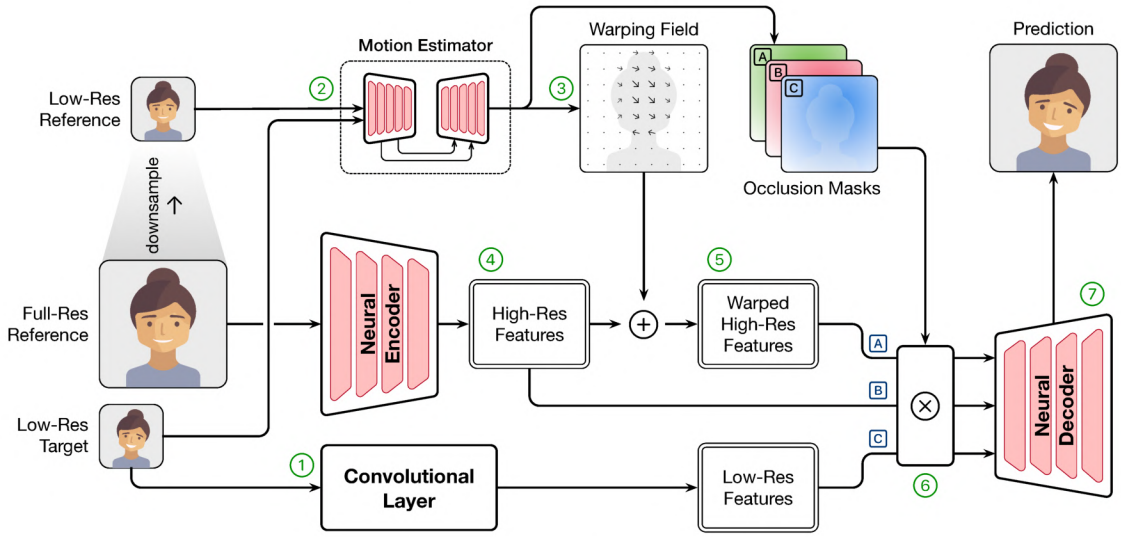


Figure 3: Gemino's high-frequency-conditional super-resolution model at the receiving client. The model first obtains features from the low-resolution target. It combines the low-resolution reference and target frames to produce a warping field based on the motion between them. The warping field is applied on encoded features from a full-res reference frame. The low-resolution and full-resolution features are jointly decoded by the neural decoder to produce the prediction at the receiver.

different regions of the input features when reconstructing each region of target frame. For instance, mask (A) maps to parts of the face or body that have moved in the reference and need to be regenerated in the target pose, mask (B) maps to the regions that don't move between the reference and target (*e.g.,* background), while mask (C) maps to new regions (*e.g.,* hands) that are only visible in the LR frame. (7) Once multiplied with their respective masks, the combined input features are fed through convolutional layers in the decoding blocks that upsample them back into the target resolution. A more detailed description of the model with figures can be found in App. A.

Why is super-resolution insufficient? A natural question is whether the high-resolution information from the HR reference frame is essential or if we can simply perform super-resolution on the LR input frame through a series of upsampling blocks. While the downsampled LR frame is great at conveying low-frequency details from the target frame, it possesses little high-frequency information. This often manifests in the predicted frames as a blur or lack of detail in clothing, hair, or facial features such as teeth or eye details. To synthesize frames that are faithful to both the low and high-frequency content in the target, it is important to combine super-resolution from the LR frame with features extracted from the HR reference frame (§5.3). The former ensures that we good low-frequency fidelity, while the latter handles the high-frequency details. The last column of Fig. 2 shows how much the reconstruction improves with our high-frequency-conditional super-resolution approach. Specifically, we are able to capture the hand movement in row 2, the head movement in row 1, and the large motion in row 3. Gemino also produces a sharper reconstruction of facial features and the high-frequency content in the grille in row 2.

Bitrate Regime	Resolution	Codec
<30 Kbps	128×128	VP8
30 Kbps – 75 Kbps	256×256	VP8
75 Kbps – 200 Kbps	512×512	VP9
>200 Kbps	1024×1024	VP9

Table 2: Mapping between desired bitrate regime and chosen resolution in Gemino.

3.3 Optimizations To Improve Fidelity

Codec-in-the-loop training. Our design decision to transmit low-resolution frames instead of keypoints is motivated by modern codecs' ability [24, 25, 47] to efficiently compress videos at lower resolutions. Downsampled frames can be compressed at a wide range of bitrates depending on the resolution and quantization level. However, the LR video resolution determines the required upsampling factor. Note that different upsampling factors normally require different variants of the deep neural model (*e.g.,* different number of upsampling stages, different feature sizes, *etc.*). This means that for each target bitrate, we first need to choose a LR video resolution, and then train a model to reconstruct high-res frames from that resolution. We first create a reverse map from bitrate ranges to resolutions by profiling VPX codecs to identify the bitrate regime that can be achieved at each resolution. We use this map to select an appropriate resolution and codec for the LR video stream at each target bitrate. Tab. 2 shows the resolution and codec Gemino uses in each bitrate range. Once a resolution is chosen, we train the model to upsample *decompressed LR frames* (from the VPX codec) of that resolution to the desired output resolution. This results in separate models for each target bitrate regime, each optimized to learn the nature of frames (and artifacts)

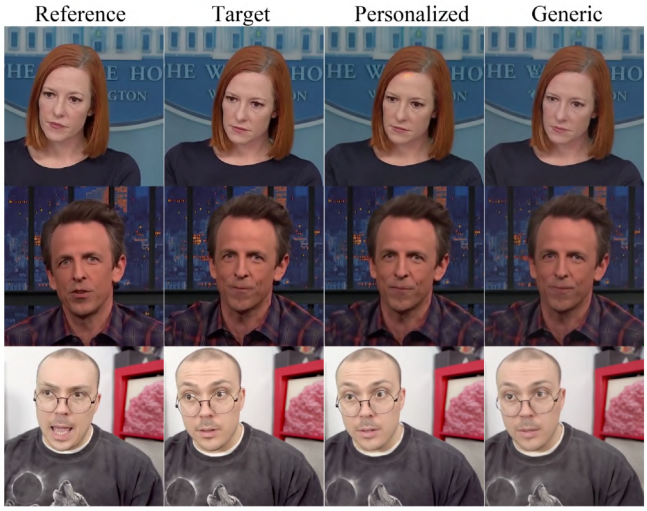


Figure 4: Performance of a model trained on a generic corpus of 512×512 videos compared to a personalized model fine-tuned on the specific person in the video. The personalized model better captures the eye gaze, dimples, and rim of the glasses.

produced by the codec at that particular resolution and bitrate. Since each such model requires 30 hours of training time on a single A100 GPU, we show that it is prudent to downsample to the highest resolution compressible at a particular bitrate, and use the model trained with a target bitrate at the lowest end of the achievable bitrate range for that resolution (§5.4).

Personalization. To improve the fidelity of our reconstructions, we explore training Gemino in two ways: *personalized* to each individual, and *generic*. To train the personalized model, we separate videos of each person into non-overlapping test and train data; the model is not exposed to the test videos, but learns a person's facial features from training videos of that specific person. The generic model is trained on a corpus of videos consisting of many different people. Fig. 4 visually compares these two approaches. The personalized model is better at reconstructing specific high-frequency details of the person, e.g., eye gaze (row 1), dimples (row 2), and the rim of the glasses (row 3), compared to the generic model. We envision the model in neural video conferencing systems to be personalized to each individual using a few hours of their video calls, and cached at the systems of receivers who frequently converse with them. An unoptimized (full-precision) checkpoint of the model's 82M parameters is about a GB in size. However, we show that the model can be compressed in §5.4.

3.4 Reducing Computational Overheads

Multi-scale architecture. As we scale up the desired output resolution, it is crucial to perform fewer operations per pixel to reduce the compute overheads. We carefully examine different parts of the model in Fig. 3, and design a multi-scale architecture where we separate those modules that require fine-grained detail from those that only need coarse-grained information. Specifically, the motion estimation module is responsible for obtaining high-level motion information from the model. Even as the input resolution is increased, this coarse information can be inferred from low-resolution frames which retain low-frequency details. Consequently, in Gemino, we operate the motion estimation module on low-resolution reference and target frames (64×64). In contrast, the neural encoder and decoders are responsible for reconstructing high-frequency content and require fine-grained details from the high-resolution reference frame.

To further reduce computational overheads, we adjust the number of downsampling and upsampling blocks in the neural encoder and decoder based on the input resolution, keeping the size of the intermediate bottleneck tensor manageable. For example, for 1024×1024 input, we use 4 down/up-sampling blocks in the encoder and decoder (Fig. 3). So as to not lose information through the bottleneck, we equip the first two blocks with skip connections [46] that directly provide encoded features to the decoder. As seen in the third column of Tab. 1, our multi-scale architecture reduces the reconstruction time of the model by 2.75× on older Titan X and NVIDIA V100 GPUs, while allowing us to easily meet the real-time deadline on an NVIDIA A100 GPU.

Model Optimizations. While the multi-scale architecture achieves real-time inference on A100 GPU system, it still fails to meet the 30 fps requirement on older systems such as Titan X. To enable this, we optimize our model further through a suite of techniques aimed at lowering the compute overheads of large neural networks. Specifically, we focus on reducing the number of operations or Multiply-Add Cumulations (MACs) involved in the expensive decoding layers which are run on every frame. While a reduction in MACs does not always translate to reduction in latency on the targeted platform [18], it acts as a reasonable proxy for our case. We first replace the regular convolutional blocks in Gemino with depthwise separable convolutions (DSC) followed by a pointwise convolution [48, 49], a commonly used architecture for mobile inference.

To decrease the model MACs even further without impacting its fidelity, we run NetAdapt [18], a neural-architecture search algorithm that iteratively prunes the number of channels layer-by-layer until the compute overheads fall below a target threshold. NetAdapt can directly optimize for latency on different platforms, but we instead use it to optimize the MACs to reduce the overheads of running separate architecture searches for each platform. NetAdapt achieves its MAC reduction in iterations, each of which targets a small decrease (*e.g.,* 3%) towards its target. In each iteration, NetAdapt prunes the layer of the model which decreases its MACs with the least loss in accuracy after "short-term finetuning" on a small subset of the data. The pruned model acts as the starting point for the next iteration, and this process continues until the model has been shrunk sufficiently. Finally, the model is "long-term fine-tuned" on the entire dataset to recover its lost accuracy.

Since Gemino uses personalized models, we adapt NetAdapt's logic to suit Gemino's training paradigm. We first

apply NetAdapt on the generic Gemino model with short-term finetuning on the generic dataset. This is because neural architecture search is expensive to run, and we observe that the pruned architecture at the end of NetAdapt is the same for generic and personalized models. We finally long-term finetune the shrunk model in a personalized manner to better recover its accuracy (§5.4). These optimizations allow us to run inference on the Titan X GPU in 27 ms with barely any loss in visual quality (last column of Tab. 1) when upsampling 128×128 frames at 15 Kbps to 1024×1024. Given that video conferencing applications tolerate latencies of up to 200 ms (5–6 frames) in their jitter buffers [50], we believe that the additional delay from generating the received frame will be negligible.

3.5 Operational Flow

Training Procedure. To train Gemino, we first obtain weights from a trained FOMM model on the entire VoxCeleb dataset [45] at 256×256 resolution. We choose the appropriate training data for the specific person we want to train a Gemino model for, and train from scratch the additional downsampling and upsampling layers in the HR pipeline as well as all layers in the LR pipeline, while fine-tuning the rest of the model for 30 epochs. We repeat this procedure for different LR frame resolutions and target bitrate regimes mention in Tab. 2, and different people. In parallel, using the same procedure, we also train a generic version of the model on a larger corpus of people. Both models are replaced with depthwise-separable convolutions and optimized using NetAdapt [18] to produce the final model.

Inference Routine. Once versions of the model have been trained and optimized for different LR resolutions and target bitrates, we simply use the appropriate model for the current target bitrate regime and the person on the video call. The sender and the receiver pre-negotiate the reference frame at the beginning of the video call. This model performs inference on a frame-by-frame basis in real-time to synthesize the video stream at the receiver. We detail our prototype implementation and WebRTC pipeline further in §4.

4 Implementation

Basic WebRTC Pipeline. Our neural video conferencing solution uses WebRTC [51], an open-source framework that enables video and audio conferencing atop the real-time transport protocol (RTP) [52]. Since we perform neural frame synthesis, we use a Python implementation of WebRTC called *aiortc* [19] that allows easy interfacing with PyTorch. Aiortc handles the initial signaling and the peer-to-peer connection setup. A typical video call has two streams (video and audio) that are multiplexed onto a single connection. The sender extracts raw frames from the display, and compresses the video and audio components separately using standard codecs like VPX [24, 25], H.264/5 [22, 23], Opus [53], *etc.*. The receiver decompresses the received data in both streams before synchronizing them and displaying each frame to the client.

New Streams. We extend the standard WebRTC stack to use

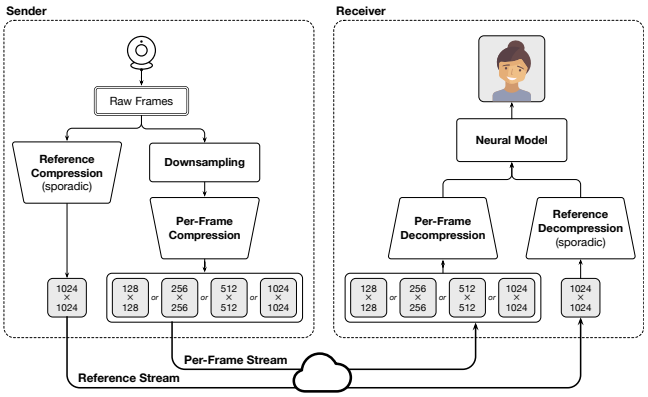


Figure 5: Neural video compression pipeline atop WebRTC [51]. We use two RTP streams: A sparse reference stream that sporadically sends high-resolution reference frames, and a per-frame (PF) stream that is used on every frame. The PF stream sends downsampled frames of the highest resolution that the current bandwidth can support, and thus has separate VP8 compression modules for each resolution. The receiver decompresses the downsampled frames, and supplies them, along with the latest reference frame, to the neural network that reconstructs the target video. If bandwidth is high enough, the PF stream is used for full-resolution VP8 frames without synthesis.

two distinct streams for video: a *per-frame stream* (PF stream) that transmits downsampled video (e.g. 64×64 frames) *on every frame*, and a reference stream that transmits occasional but high-resolution *reference* frames that improve the synthesis fidelity. We anticipate using the reference stream extremely sparsely. For instance, in our implementation, we use the first frame of the video as the *only* reference frame. However, more reference frames may help recover high-frequency fidelity as it worsens when the reference and target frames drift apart. But, most low-frequency changes between the reference and the target can be communicated simply through the downsampled target in the PF stream[3]. The receiver uses the per-frame information in the PF stream, with the reference information, to synthesize each high-resolution frame. Fig. 5 illustrates the expanded WebRTC architecture to accommodate the Gemino design.

The PF stream is implemented as a new RTP-enabled stream on the same peer connection between the sender and the receiver. We downsample each input frame to the desired resolution at the sender and compress it using the appropriate VPX codec. The frame is decompressed at the receiver. The bitrate achieved is controlled by supplying a target bitrate to VPX. Our PF stream can support full-resolution video that is typical in most video conferencing applications, while also supporting a range of lower resolutions for the model to upsample from. To enable this flexibility, we design the PF stream to have multiple VPX

[3] We observe that sending reference frames with any fixed frequency adds significant bandwidth overheads. So, we only use a single reference frame in our evaluations. We leave an investigation of mechanisms to detect the need for a new reference frame (speaker moves significantly, high-frequency content or background changes) to future work.

encoder-decoder pairs, one for each resolution that it operates at. When the sender transmits a frame, it chooses an appropriate resolution and codec based on the target bitrate, and compresses the video at that resolution and target bitrate. The resolution information is embedded in the payload of the RTP packet carrying the frame data. When the receiver receives each RTP packet, it infers the resolution and sends it to the VPX decoder for that resolution. Once decompressed, the low-resolution frame is upsampled by Gemino to the appropriate full-resolution frame. If the PF stream consists of 1024×1024 frames, Gemino falls back onto the regular codec and stops using the reference stream. The reference stream is repurposed from the existing video stream.

Model Wrapper. To enable neural frame synthesis, we define a wrapper that allows the *aiortc* pipeline to interface with the model. We reuse most of the pipeline from frame read at the sender to display at the receiver, except for introducing a downsampling module right after frame read, and a prediction function right before frame display. The wrapper is structured to perform format conversions and data movement from the AudioVisual [54] frames on the CPU that *aiortc* needs, to the PyTorch [55] tensors on the GPU required by the model. We initialize models separately for the sending and receiving clients. The wrapper also allows us to save (and periodically update) state at the sender and receiver which is useful for reducing the overheads from modules where we can reuse old computation (*e.g.,* run the encoder for high-resolution reference features only when the reference changes).

Further Optimizations. We optimize a number of other aspects of the *aiortc* pipeline. For instance, we move data between the CPU and GPU multiple times in each step of the pipeline. Batching these operations is difficult when maintaining low latency on each frame. However, to minimize PCIe overheads from repetitive data movement, we use *uint8* variables instead of *float*. We also keep reference frames and their encoded features stored as model state on the GPU. We pipeline as many operations as possible by running keypoint extraction, model reconstruction, and conversions between data formats in separate threads.

5 Evaluation

We evaluate Gemino in a simulation environment and atop a WebRTC-based implementation. We describe our setup in §5.1 and use it to compare existing baselines in §5.2. §5.3 motivates our model design, §5.4 discusses the impact of having the codec in our training process, and §5.5 shows that Gemino closely matches a time-varying target bitrate.

5.1 Setup

Dataset. Since most widely used datasets are of low-resolution videos [6, 45, 56] and lack diversity in the extent of the torso or face-zoom level, we collected our own dataset comprising of videos of five Youtubers with publicly available HD (1920×1080) videos. For each Youtuber, we curate a set of 20 distinct videos or URLs that differ in clothing, hairstyle, accessories, or background. The 20 videos of each Youtuber are separated into 15 training videos and 5 test videos. For each video, we manually record and trim the segments that consist of talking individuals; we ignore parts that pan to news segments or different clips. The segments are further split into 10s chunks to generate easily loadable videos for training, while the segments of the test video are combined to form a longer video. We also spatially crop each frame into our desired dimensions (typically 1024× 1024). based on the average location of the person across all frames of the video. Note that 720p and 1024×1024 frames have similar numbers of pixels. We strip the audio since our focus is on video synthesis. Tab. 8 in App. A.3 describes the details of the dataset. We do not own any of these videos, and we only use images of frames produced by our evaluation pipeline in this paper. We use the 512×512 dataset from NVIDIA [6] to train a generic model to illustrate the benefits of personalization. Our evaluation focuses on reconstructing a single front-facing person in a video call; Gemino can be extended to multiple speakers if there are application-level techniques to separate speakers into individual streams [57, 58].

Model Details. The main model we evaluate is our high-frequency conditional super-resolution model that consists of an upsampling module that takes in features from a low-resolution (LR) frame, and upsamples it to 1024×1024. To provide the high-frequency details, it uses two pathways consisting of warped and unwarped features from the high-resolution (HR) reference image (Fig. 3). We use the first frame of the video as the *sole reference* image for the entire test duration. The warping field is produced by a motion estimation network that uses the first-order approximation near 10 keypoints [5]. Our multi-scale architecture runs motion estimation always at 64×64 irrespective of the input video resolution. The neural encoder (for the HR features) and decoder (for both LR and HR features) consist of four down and upsample blocks. The discriminator operates at multiple scales and uses spectral normalization [59] for stability. Layers of our model that are identical in dimensions to those from the FOMM are initialized from a public FOMM checkpoint trained on the VoxCeleb dataset [45], and fine-tuned on a per-person basis. The remaining layers are randomly initialized and trained on a per-person basis over 30 epochs. We fine-tune the FOMM baseline also in the same personalized manner. We use Adam optimizer [60] to update the model weights with a learning rate of 0.0002, and first and second momentum decay rates of 0.5 and 0.999. We use equally weighted multi-scale VGG perceptual loss [61], a feature-matching loss [62], and a pixel-wise loss. We also use an adversarial loss [17] with one-tenth the weight of remaining losses. The keypoints use an equivariance loss similar to the FOMM [5]. We train our models to reconstruct from decompressed VPX frames corresponding to the low-resolution target frame so that the model learns to correct any artifacts produced by VPX.

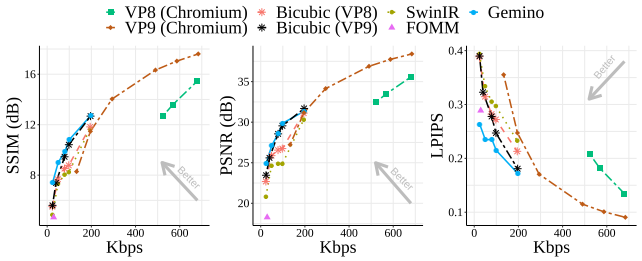
Evaluation Infrastructure. We evaluate our neural compression system in a simulation environment where frames are read from a video, downsampled (if needed) for the low-resolution

PF stream, compressed using VPX's chromium codec [63], and passed to the model (or other baselines) to synthesize the target frame. Note that the FOMM [5] uses keypoints and four "Jacobian" values around each keypoint for producing its warping, and transmits them over the network. We design a new codec for the keypoint data that achieves nearly lossless compression and a bitrate of about 30 Kbps. For VPX, we compress and decompress the full-resolution frame at different target bitrates, and measure the difference in visual quality between the output and the original frame.

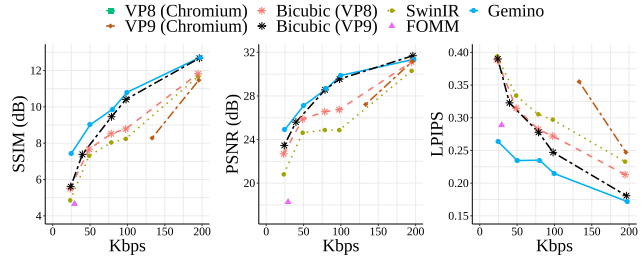
To obtain end-to-end latency measurements and to demonstrate Gemino's adaptability to different target bitrates, (§5.5), we use our *aiortc* [19] implementation. A sending process reads video from a file frame-by-frame and transmits it to a receiving process that records each received frame. The two processes, running on the same server, use the ICE signaling [64] mechanism to establish a peer-to-peer connection over a UNIX socket, which then supports video frame transmission using the Real-Transport Protocol (RTP). We timestamp each frame as it is sent and received, and save the sent and received frames in their uncompressed forms to compute latency and visual metrics. We also log RTP packet sizes to compute the bitrate.

Metrics. To quantify the aesthetics of the generated video, we use standard visual metrics such as PSNR (peak signal-to-noise ratio), SSIM (structural similarity index) in decibels [65], and LPIPS (learned perceptual image patch similarity) [20]. For PSNR and SSIM, higher is better; while for LPIPS, lower is better. We observe that differences in LPIPS are more reflective of how natural the synthesized frame feels and use that as our main comparison metric (§B.2); we also show visual strips where appropriate. We report the bitrate consumed to achieve a particular visual quality by measuring the total data transferred (size of compressed frames or RTP packet sizes) over the duration of the video, and dividing it by the duration itself. To measure the end-to-end latency, we record the time at which the frame is read from the disk at the sender as well as the time at which prediction completes at the receiver. We report the difference between these two timestamps as our per-frame latency metric. We also report the inference time per frame when running the trained model in simulation; this does not capture the overheads of data conversion or movement in an end-to-end pipeline. This inference time needs to be <33ms to maintain a 30fps video call. We run all our experiments for the entire duration of each test video in our dataset (Tab. 8), and report the average over all frames for each metric.

Baselines. We obtain the bitrate for *VP8*, the default codec in its Chromium settings [63] that comes with the *aiortc* codebase. We also implement and evaluate VP9 in the same setup. To evaluate the benefits of using a neural approach to video conferencing, particularly at lower bitrates, we compare a few different models: (1) *FOMM* [5], a keypoint-based model for face animation, (2) our approach, Gemino, (3) state-of-the-art super-resolution model based on *SwinIR* [21], and (4) *bicubic upsampling* [28] applied to the low-resolution VPX target frame. All



(a) Overall rate-distortion curve for all schemes



(b) Rate-distortion curve in low-bitrate regimes.

Figure 6: Rate-distortion curve for Gemino compared with existing baselines. VP8 and VP9 require ∼5× and ∼3× the bitrate consumed by Gemino to achieve comparable LPIPS. At lower bitrates, Gemino outperforms other approaches that upsample low-resolution video frames. Gemino's benefits become prominent as the bitrate regime is lowered.

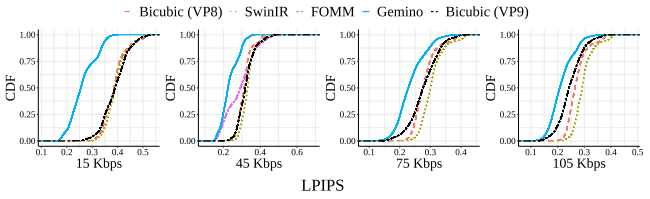


Figure 7: CDF of reconstruction quality across all video frames as that shows that, as we move from higher bitrates to lower, the improvement from Gemino relative to Bicubic, particularly over VP9, becomes more pronounced.

of the compared models generate 1024× 1024 frames except for the generic model that uses NVIDIA's 512×512 corpus [6].

5.2 Overall Bitrate vs. Quality Tradeoff

To quantify the improvements of our neural compression system, we first compare Gemino with VP8 and VP9 in their chromium configuration [63]. Fig. 6 shows the rate-distortion curve for all schemes. For VPX, we alter the target bitrate alone for full-resolution (1024×1024) frames in the PF stream. For Gemino, bicubic, and SwinIR, we vary the resolution and target bitrate of the low-resolution (LR) frame in the per-frame (PF) stream. For each point on the rate-distortion curve for Gemino, we train a personalized model to reconstruct full-resolution frames from LR frames encoded at the highest resolution supported by that target bitrate. We motivate using

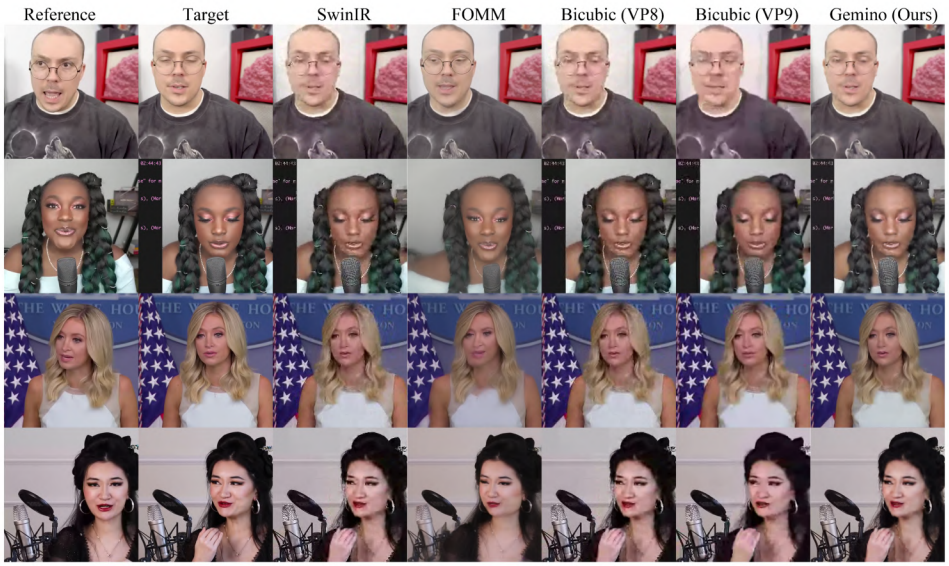


Figure 8: Visual comparison across low-bitrate baselines. All but FOMM upsample a 256×256 frame at 45 Kbps. Gemino's reconstructions contain a smoother output than the blocky artifacts observed with Bicubic and SwinIR. The FOMM completely fails when the reference and the target differ considerably in rows 1 and 2.

the codec in training and choosing the resolution in §5.4. The PF stream is compressed with VPX's Chromium settings at the target bitrate for its resolution. We configure Gemino to choose the base codec (VP8 or VP9) that has the better visual quality for a given target bitrate. We plot the resulting bitrate and visual metrics averaged across all 25 test videos' (5 speakers; 5 videos each) frames. Fig. 6(a) shows that VP8 operates in a different bitrate regime than all other schemes. VP9 improves the dynamic range to about 130 Kbps, but still can't compress below that. Both VPX codecs operate on full-resolution frames, which cannot be compressed as efficiently as LR frames by the codec in its real-time mode. We observe that VP8 and VP9 consume 523 Kbps and 230 Kbps respectively to achieve an LPIPS of 0.21. In contrast, our approach is able to achieve the same visual quality with only 105 Kbps, providing nearly 5× and 2.2× in improvement over VP8 and VP9 respectively.

However, since our goal is to enable video conferencing in bitrate regimes where current codecs fail, we focus on bitrates < 200 Kbps in Fig. 6(b). Gemino uses VP8 at 15 Kbps and 45 Kbps but VP9 at higher bitrates; VP8 and VP9 do not differ much at lower bitrates, but VP9 allows even 512×512 to be compressed to 75 Kbps. Fig. 6(b) suggests that while LPIPS always shows considerable variation across schemes, small differences (1–2 dB) in PSNR and SSIM start manifesting only at lower bitrates. Specifically, Gemino achieves over 1 dB better SSIM and PSNR, and 0.08 lesser LPIPS than Bicubic around 50 Kbps. To map these metrics to improvements in visual quality, we show some snippets in Fig. 8 and enlarged frames (with per-frame metrics) in §B.2. Fig. 8 shows that compared to Gemino, reconstructions from bicubic have more block-based artifacts in the face, and the FOMM misses parts of the frame (row 4) or distorts the face. SwinIR, a super-resolution model, performs worse than bicubic. We suspect this is because SwinIR is not specifically trained on faces, and is also oblivious to artifacts from the codec that it encounters in our video conferencing pipeline. To account for this, we evaluate a simple upsampling model personalized on faces in §5.3.

We also show CDFs of the visual quality in LPIPS across all frames in our corpus in Fig. 7 at 15 Kbps, 45 Kbps, 75 Kbps and 105 Kbps. The same CDFs for PSNR and SSIM can be found in §B.1. At each target bitrate, we use the largest resolution supported by the underlying video codec. The CDFs show that Gemino's reconstructions are robust to variations across frames and orientation changes over the course of a video. Gemino outperforms all other baselines across all frames and metrics. Specifically, its synthesized frames at 45 Kbps are better than FOMM by 0.05–0.1. It also outperforms Bicubic and SwinIR by 0.05 and 0.1 in LPIPS at the median and tail respectively. As we move from higher bitrates to lower, we observe that the improvement from Gemino relative to Bicubic, particularly over VP9, becomes more pronounced. While the gap in LPIPS at the median between Bicubic (VP9) and Gemino is less than 0.05 at 105 Kbps, it increases to nearly 0.2 at 15 Kbps. Gemino increases the end-to-end frame latency to ∼87 ms, from VPX's 47 ms. However, our carefully pipelined operations are optimized to enable a throughput of 30 fps.[4]

5.3 Model Design

Architectural Elements. To understand the impact of our model design, we compare Gemino to a *pure upsampling* approach from the LR target features with no pathways from an HR reference frame (only Ⓒ in Fig. 3) at decompressing VP8 128×128 frames at 15 Kbps to its full 1024×1024 resolution. Tab. 3 reports the average visual quality while Fig. 10 shows a CDF of performance across all frames and videos. Gemino

[4]We ignore the delay due to the expensive conversion between NumPy arrays and Python's Audio Visual Frames in calculating latency because the the current PyAV library [54] is not optimized for conversion at higher resolutions.

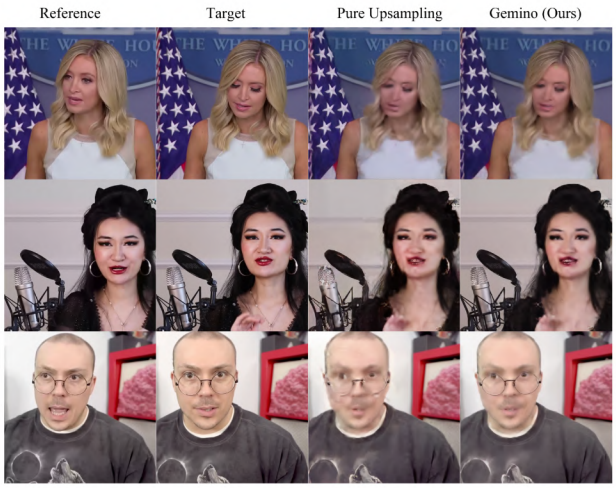


Figure 9: Visual comparison across different model architectures. Compared to Gemino, Pure upsampling misses the high-frequency details of the microphone (row 2) and lends block-based artifacts on the face (row 3).

Model Architecture	PSNR (dB)	SSIM (dB)	LPIPS
Upsampling	24.72	7.02	0.30
Gemino	**24.90**	**7.42**	**0.26**

Table 3: Average visual quality of synthesized frames from Gemino and "Pure Upsampling" when reconstructing from decompressed 128×128 frames.

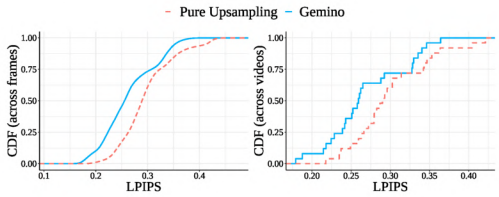


Figure 10: CDF of reconstruction quality of different model architectures across frames and videos in our test corpus. Gemino outperforms pure upsampling by nearly 0.05 in LPIPS at the median of both frames and videos. It also outperforms the approach that relies on only warped HR, and the RGB-based warping across most frames in the corpus.

outperforms pure upsampling on average by 0.4 dB on SSIM and 0.04 on LPIPS. This is also visible in rows 2 and 3 of Fig. 9 where the pure upsampling approach misses the high-frequency details of the microphone grille and lends blocky artifacts to the face respectively. This manifests as a 0.05 difference in the LPIPS of the median frame reconstruction and a nearly 0.1 difference between Gemino and Pure Upsampling at the tail in Fig. 10. Since one of our goals is to have a robust neural compression system that has good reconstruction quality across a wide variety of frames, this difference at the tail across approaches is particularly salient. On the other hand, Gemino synthesizes the mic better, and has fewer blocky artifacts on the face.

Personalization. To quantify the impact of personalization,

Model	Training Regime	PSNR (dB)	SSIM (dB)	LPIPS
Gemino	Personalized	**30.59**	**11.11**	**0.13**
	Generic	29.48	10.55	0.13
Upsampling	Personalized	**30.32**	**11.10**	**0.13**
	Generic	28.89	9.59	0.17

Table 4: Impact of personalization on different models.

Convolution Type	**Regular**			**Depthwise**		
NetAdapt Target	None	10%	1.5%	None	10%	1.5%
# of Decoder MACs	195B	14.7B	2.4B	22.7B	1.7B	0.3B
# of Decoder Parameters	30M	1.3M	151K	3.4M	186K	26K
A100 Inference	13ms	9ms	8ms	11ms	8ms	7ms
V100 Inference	14ms	11ms	8ms	14ms	8ms	6ms
Titan X Inference	46ms	17ms	12ms	53ms	26ms	13ms
Jetson TX2 Inference	800ms	274ms	165ms	436ms	183ms	87ms
LPIPS (Generic)	0.13	0.15	0.17	0.14	0.16	0.19
LPIPS (Personalized)	0.13	0.13	0.15	0.14	0.14	0.17

Table 5: Accuracy and compute overheads of different versions of our model operating on 512×512 resolution.

we compare two versions of Gemino and Pure Upsampling: a *generic model* trained on a large corpus of 512×512 videos [6] of different people and a personalized model fine-tuned on videos of only a specific person. We compare the visual metrics averaged across all videos of all people when synthesized from a single generic model against the case when each person's videos are synthesized from their specific model. All models upsample 64×64 LR frames (without codec compression) to 512×512. Fig. 4 visually shows that the personalized Gemino model reconstructs the details of the face (row 1), dimples (row 2) and rim of the glasses (row 3) better but Tab. 4 shows that personalization provides limited benefits for Gemino (PSNR and SSIM improvements of 1.1 dB and 0.56 dB). However, personalization provides more pronounced benefits of nearly 1.5 dB in SSIM and PSNR, and 0.04 in LPIPS for the Pure Upsampling architecture. This suggests that the benefits from personalization are most notable when the underlying model architecture itself is limited in its representational power: the personalized pure upsampling architecture is able to encode high-frequency information in its weights when trained in a personalized manner, while it has no pathways to obtain that information in a generic model. It also illustrates that Gemino is more robust; it operates better than pure upsampling in the harder settings of a generic model and extreme upsampling from a 128×128 frame compressed to 15 Kbps (Fig. 10).

5.4 Operational Considerations

Computational Overheads. Tab. 5 explores the accuracy *vs.* compute tradeoffs of different models at 512×512 resolution with and without personalization as measured on different GPU systems with the various optimizations described in §3.4. We focus on optimizing the decoding layers (bottleneck and up-sampling) of the generator (Fig. 14) since they are the most compute intensive layers, and are also run on every received

frame. The accuracy is reported in the form of the average LPIPS [20] across all frames of our corpus. We observe that DSC reduces the decoder to 11% of its original MACs. While this gives limited improvements on large GPU systems, it improves the inference time on Jetson TX2, an embedded AI device, by 1.84×. Running NetAdapt further reduces the inference time to 87 ms at 1.5% of the model MACs on the TX2. The NVIDIA compiler on the Titan X GPU and the Jetson TX2 is not optimized for DSC [49]; this can be improved with a TVM compiler stack [66] and optimized engines such as TensorRT [67]. However, running NetAdapt produces a real-time model for Titan X even at 10% of the original model MACs. As expected though, there is a loss of accuracy as the models become smaller. This loss is negligible in moving from the full model MACs to 10%, particularly when personalizing, but is more significant at 1.5%. The trend with personalization is expected since smaller models do not generalize well with their limited capacity, however it does not help if the optimizations are extreme. This illustrates that there is a sweet spot (such as decreasing MACs to 10%) wherein the gains from decreased compute outweigh the loss (or lack thereof) in accuracy.

Choosing PF Stream Resolution. Gemino is designed flexibly to work with LR frames of any size (64×64, 128×128, 256×256, 512×512) to resolve them to 1024×1024 frames, and to fall back to VPX at full resolution if it can be supported. VP8 and VP9 achieve different bitrate ranges at every resolution by varying how the video is quantized. For instance, on our corpus, we observe that 256×256 frames can be compressed with VP8 in the 45 Kbps–180 Kbps range, but VP9 can compress even 512×512 frames from 75 Kbps onwards. These bitrate ranges often overlap partially across resolutions. This begs the question: given a target bitrate, what resolution and codec should the model use to achieve the best quality? To answer this, we compare the synthesis quality with Gemino atop VP8 from three PF resolutions, all at 45 Kbps in Tab. 6. Upsampling 256×256 frames, even though they have been compressed more to achieve the same bitrate, gives a nearly 4 dB improvement in PSNR, more than 2 dB improvement in SSIM, and a 0.03 improvement in LPIPS, over upsampling lower resolution frames. This is because the extent of super-resolution that the model performs decreases dramatically at higher starting resolutions. This suggests that for any given bitrate budget, we should start with the highest resolution frames that the PF stream supports at that bitrate, even at the cost of more quantization. This also means that if VP9 can compress higher resolution frames than VP8 at the same target bitrate, we should pick VP9. Tab. 2 shows the resolution and codec we choose for different target bitrate ranges in our implementation.

Encoding Video During Training. A key insight in the design of Gemino is that we need to design the neural compression pipeline to leverage the latest developments in codec design. One way to do so is to allow the model to see decompressed frames at the chosen bitrate and PF resolution during the training process so that it learns the artifacts produced by the codec.

PF Stream Resolution	PSNR (dB)	SSIM (dB)	LPIPS
64×64	23.80	6.77	0.27
128×128	25.72	7.86	0.27
256×256	**27.12**	**9.01**	**0.24**

Table 6: Reconstruction quality from different resolution PF stream frames at the same bitrate of 45 Kbps. Gemino reconstructs better from higher resolution frames.

Training Regime	PF @ 15 Kbps	PF @45 Kbps	PF @75 Kbps
No Codec	0.32	0.30	0.28
VP8 @ 15 Kbps	**0.26**	**0.25**	**0.23**
VP8 @ 45 Kbps	0.28	0.27	0.25
VP8 @ 75 Kbps	0.30	0.28	0.26
VP8 @ [15, 75] Kbps	0.28	0.26	0.25

Table 7: LPIPS for different regimes wherein we include the VP8 codec in the training pipeline. The model trained with the lowest bitrate videos at a given resolution performs best regardless of what the bitrate of the video is at inference time.

This allows us to get extremely low bitrates for LR frames (which often causes color shifts or other artifacts) while maintaining good visual quality. To evaluate the benefit of this approach, we compare five training regimes for Gemino when upsampling 128×128 video to 1024×1024: (1) no codec, (2) VP8 frames at 15 Kbps, (3) VP8 frames at 45 Kbps, (4) VP8 frames at 75 Kbps, (5) VP8 frames at a bitrate uniformly sampled from 15 Kbps to 75 Kbps. We evaluate all five models at upsampling decompressed frames at 15 Kbps, 45 Kbps, and 75 Kbps.

Tab. 7 shows the LPIPS achieved by all the models in each reconstruction regime. All models trained on decompressed frames perform better than the model trained without the codec. Further, the model trained at the lowest bitrate (15 Kbps) performs the best even when provided decompressed frames at a higher bitrate at test time because it has learned the most challenging Super-Resolution task from the worst LR frames, and performs well even with easier instances or higher bitrate frames. This suggests that we only need to train one personalized model per PF resolution at the lowest bitrate supported by a resolution, and then we can reuse it across the entire bitrate range that the PF resolution can support.

5.5 Adaptation to Network Conditions

To understand the adaptability of Gemino, we explore how it responds to changes in the target bitrate over the course of a video. We remove any conflating effects from bandwidth prediction by directly supplying the target bitrate as a decreasing function of time to both Gemino and the VP8 codec. Gemino uses only VP8 through all bitrates for a fair comparison. Fig. 11 shows (in black) the target bitrate, along with the achieved bitrates of both schemes, and the associated perceptual quality [20] for a single video over the course of 220s of video-time[5]. We observe that initially (first 120s), at

[5]The timeseries are aligned to ensure that VP8 and Gemino receive the same video frames to remove confounding effects of differing latencies.

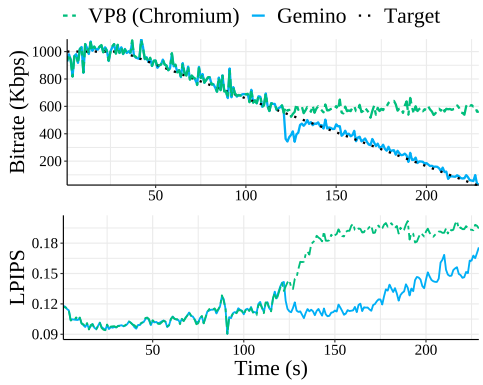


Figure 11: Gemino's ability to adapt to a time-varying target bitrate. As the target bitrate reduces, Gemino gradually lowers its PF stream resolution trading off more upsampling and less quality (increased LPIPS) for a reduction in achieved bitrate. VP8, in contrast, lowers the bitrate initially, but once at its minimum quality, it stops responding to the target bitrate.

high target bitrates, Gemino and VP8 perform very similarly because they are both transmitting just VP8 compressed frames at full (1024×1024) resolution. Once VP8 has hit its minimum achievable bitrate of ∼550 Kbps (after 120s), there is nothing more it can do, and it stops responding to the input target bitrate. However, Gemino continues to lower its PF stream resolution and/or bitrate in small steps all the way to the lowest target bitrate of 20 Kbps. Since Gemino is only using VP8 here, it switches to 512×512 at 550 Kbps, 256×256 at 180 Kbps, and 128×128 at 30 Kbps. This design choice might cause abrupt shifts in quality around the transition points between resolutions. However, Gemino prioritizes responsiveness to the target bitrate over the hysteresis that classical encoders experience which, in turn, leads to packet losses due to overshooting and glitches. As the resolution of the PF stream decreases, as expected, the perceptual quality of Gemino worsens but is still better than VP8's visual quality. This shows that Gemino can adapt well to bandwidth variations, though we leave the design of a transport and adaptation layer that provides fast and accurate feedback to Gemino for future work.

6 Limitations and Future Work

While Gemino greatly expands the operating regime for video conferencing to very low bitrates, it incurs significant overheads in the form of training costs for codec-in-the-loop training and personalization. It compresses better than VPX, but the encoding and decoding processes are quite a bit slower than VPX, and not as widely supported on devices without access to some graphical processing engine. However, we believe that device improvements year on year are trending in a favorable direction, particularly with the emergence of optimized runtimes and hardware for running machine learning workloads on both Apple and Android devices [68,69]. Further, NetAdapt [18] and layer-by-layer pruning is only one technique amongst a large suite of model optimization approaches. We believe that with more targeted optimizations for particular devices, we can do better. Such optimizations become more salient when operating on higher-resolution video (*e.g.,* 4K, UltraHD) and in higher bandwidth regimes (∼ 5 Mbps). We leave an exploration of such optimizations to future work.

Gemino, though trained on random pairs of reference and target frames, always uses the first frame of the test video as its reference frame. The reconstruction fidelity can be improved by using reference frames close to each target frame. However, sending more frequent reference frames incurs very high bitrate costs due to their high resolution. We leave to future work a more thorough investigation of reference frame selection mechanisms that weigh these tradeoffs to squeeze the maximum accuracy for a given compression level.

7 Conclusion

This paper proposes Gemino, a neural video compression scheme for video conferencing using a new high-frequency-conditional super-resolution model. Our model combines the benefits of low-frequency reconstruction from a low-resolution target, and high-frequency reconstruction from a high-resolution reference. Our novel multi-scale architecture and personalized training synthesize good quality videos at high resolution across many scenarios. The adaptability of the compression scheme to different points on a rate-distortion curve opens up new avenues to co-design the application and transport layers for better quality video calls. However, while neural compression shows promise in enabling very low bitrate video calls, it also raises important ethical considerations about the bias that training data can introduce on the usefulness of such a technique to different segments of the human population. We believe that our personalized approach alleviates some of these concerns, but does not eliminate them entirely.

Acknowledgments

We thank our shepherd, Arpit Gupta, and our anonymous NSDI reviewers for their feedback. This work was supported by GIST, seed grants from the MIT Nano NCSOFT program and Zoom Video Communications, and NSF awards 2105819, 1751009 and 1910676.

References

[1] Zoom System Requirements. https://support.zoom.us/hc/en-us/articles/201362023-Zoom-system-requirements-Windows-macOS-Linux.

[2] Worldwide broadband speed league 2022. https://www.cable.co.uk/broadband/speed/worldwide-speed-league/}speed.

[3] Median Country Speeds March 2023. https://www.speedtest.net/global-index.

[4] The state of video conferencing 2022. https://www.dialpad.com/blog/video-conferencing-report/.

[5] Aliaksandr Siarohin, Stéphane Lathuilière, Sergey Tulyakov, Elisa Ricci, and Nicu Sebe. First order motion model for image animation. In *Conference on Neural Information Processing Systems (NeurIPS)*, December 2019.

[6] Ting-Chun Wang, Arun Mallya, and Ming-Yu Liu. One-shot free-view neural talking-head synthesis for video conferencing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10039–10049, 2021.

[7] Egor Zakharov, Aleksei Ivakhnenko, Aliaksandra Shysheya, and Victor Lempitsky. Fast bi-layer neural synthesis of one-shot realistic head avatars. In *European Conference of Computer vision (ECCV)*, August 2020.

[8] Maxime Oquab, Pierre Stock, Daniel Haziza, Tao Xu, Peizhao Zhang, Onur Celebi, Yana Hasson, Patrick Labatut, Bobo Bose-Kolanu, Thibault Peyronel, et al. Low bandwidth video-chat compression using deep generative models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2388–2397, 2021.

[9] Anna Volokitin, Stefan Brugger, Ali Benlalah, Sebastian Martin, Brian Amberg, and Michael Tschannen. Neural face video compression using multiple views, 2022.

[10] Unlimited HD Video Calls. https://trueconf.com/features/modes/videocall.html, 2021.

[11] Project Starline: Feel like you're there, together. https://blog.google/technology/research/project-starline/, 2021.

[12] What is the Best Audio Codec for Online Video Streaming? https://www.dacast.com/blog/best-audio-codec/.

[13] Yue Chen, Debargha Murherjee, Jingning Han, Adrian Grange, Yaowu Xu, Zoe Liu, Sarah Parker, Cheng Chen, Hui Su, Urvang Joshi, et al. An overview of core coding tools in the AV1 video codec. In *2018 Picture Coding Symposium (PCS)*, pages 41–45. IEEE, 2018.

[14] Debargha Mukherjee, Jingning Han, Jim Bankoski, Ronald Bultje, Adrian Grange, John Koleszar, Paul Wilkins, and Yaowu Xu. A technical overview of VP9, the latest open-source video codec. *SMPTE Motion Imaging Journal*, 124(1):44–54, 2015.

[15] Jim Bankoski, Paul Wilkins, and Yaowu Xu. Technical overview of VP8, an open source video codec for the web. In *2011 IEEE International Conference on Multimedia and Expo*, pages 1–6. IEEE, 2011.

[16] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the H.264/AVC standard. *IEEE Transactions on circuits and systems for video technology*, 17(9):1103–1120, 2007.

[17] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[18] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018.

[19] aiortc. https://github.com/aiortc/aiortc.

[20] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 586–595, 2018.

[21] Jingyun Liang, Jiezhang Cao, Guolei Sun, Kai Zhang, Luc Van Gool, and Radu Timofte. Swinir: Image restoration using swin transformer. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1833–1844, 2021.

[22] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the h. 264/avc standard. *IEEE Transactions on circuits and systems for video technology*, 17(9):1103–1120, 2007.

[23] Gary J Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on circuits and systems for video technology*, 22(12):1649–1668, 2012.

[24] Jim Bankoski, Paul Wilkins, and Yaowu Xu. Technical overview of VP8, an open source video codec for the web. In *2011 IEEE International Conference on Multimedia and Expo*, pages 1–6. IEEE, 2011.

[25] Debargha Mukherjee, Jingning Han, Jim Bankoski, Ronald Bultje, Adrian Grange, John Koleszar, Paul Wilkins, and Yaowu Xu. A technical overview of VP9, the latest open-source video codec. *SMPTE Motion Imaging Journal*, 124(1):44–54, 2015.

[26] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 267–282, 2018.

[27] Change the quality of your video. https://support.google.com/youtube/answer/91449?hl=en.

[28] Robert Keys. Cubic convolution interpolation for digital image processing. *IEEE transactions on acoustics, speech, and signal processing*, 29(6):1153–1160, 1981.

[29] Pascal Getreuer. Linear Methods for Image Interpolation. *Image Processing On Line*, 1:238–259, 2011. https://doi.org/10.5201/ipol.2011.g_lmii.

[30] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Learning a deep convolutional network for image super-resolution. In *IEEE European Conference on Computer Vision (ECCV)*, 2014.

[31] Xintao Wang, Ke Yu, Shixiang Wu, Jinjin Gu, Yihao Liu, Chao Dong, Yu Qiao, and Chen Change Loy. Esrgan: Enhanced super-resolution generative adversarial networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 0–0, 2018.

[32] Bee Lim, Sanghyun Son, Heewon Kim, Seungjun Nah, and Kyoung Mu Lee. Enhanced deep residual networks for single image super-resolution. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 136–144, 2017.

[33] Jose Caballero, Christian Ledig, Andrew Aitken, Alejandro Acosta, Johannes Totz, Zehan Wang, and Wenzhe Shi. Real-time video super-resolution with spatio-temporal networks and motion compensation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4778–4787, 2017.

[34] Jingyun Liang, Jiezhang Cao, Yuchen Fan, Kai Zhang, Rakesh Ranjan, Yawei Li, Radu Timofte, and Luc Van Gool. Vrt: A video restoration transformer. *arXiv preprint arXiv:2201.12288*, 2022.

[35] Zhengdong Zhang and Vivienne Sze. FAST: A framework to accelerate super-resolution processing on compressed videos. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 19–28, 2017.

[36] Hyunho Yeo, Chan Ju Chong, Youngmok Jung, Juncheol Ye, and Dongsu Han. Nemo: Enabling neural-enhanced video streaming on commodity mobile devices. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, MobiCom '20, New York, NY, USA, 2020. Association for Computing Machinery.

[37] Yu Chen, Ying Tai, Xiaoming Liu, Chunhua Shen, and Jian Yang. Fsrnet: End-to-end learning face super-resolution with facial priors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2492–2501, 2018.

[38] Cheng Ma, Zhenyu Jiang, Yongming Rao, Jiwen Lu, and Jie Zhou. Deep face super-resolution with iterative collaboration between attentive recovery and landmark estimation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 5569–5578, 2020.

[39] Mehrdad Khani, Vibhaalakshmi Sivaraman, and Mohammad Alizadeh. Efficient video compression via content-adaptive super-resolution. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4521–4530, 2021.

[40] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. Neural adaptive content-aware internet video delivery. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 645–661, 2018.

[41] Mallesham Dasari, Kumara Kahatapitiya, Samir R. Das, Aruna Balasubramanian, and Dimitris Samaras. Swift: Adaptive video streaming with layered neural codecs. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 103–118, Renton, WA, April 2022. USENIX Association.

[42] Jaehong Kim, Youngmok Jung, Hyunho Yeo, Juncheol Ye, and Dongsu Han. Neural-enhanced live streaming: Improving live video ingest via online learning. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 107–125, 2020.

[43] Pan Hu, Rakesh Misra, and Sachin Katti. Dejavu: Enhancing videoconferencing with prior knowledge. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 63–68, 2019.

[44] Arun Mallya, Ting-Chun Wang, and Ming-Yu Liu. Implicit warping for animation with image sets, 2022.

[45] Arsha Nagrani, Joon Son Chung, and Andrew Zisserman. Voxceleb: a large-scale speaker identification dataset. *arXiv preprint arXiv:1706.08612*, 2017.

[46] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[47] AV1 bitstream & decoding process specification. http://aomedia.org/av1/specification/.

[48] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient

convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[49] Diana Wofk, Fangchang Ma, Tien-Ju Yang, Sertac Karaman, and Vivienne Sze. FastDepth: Fast Monocular Depth Estimation on Embedded Systems. *2019 International Conference on Robotics and Automation (ICRA)*, pages 6101–6108, 2019.

[50] International Telecommunication Union. ITU-T G.1010: End-user multimedia QoS categories. In *Series G: Transmission Systems and Media, Digital Systems and Networks*, 2001.

[51] WebRTC. https://webrtc.org/.

[52] Henning Schulzrinne, Stephen Casner, Ron Frederick, and Van Jacobson. Rtp: A transport protocol for real-time applications, 1996.

[53] Opus interactive audio codec. https://opus-codec.org/.

[54] Pyav documentation. https://pyav.org/docs/stable/.

[55] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[56] Joon Son Chung, Arsha Nagrani, and Andrew Zisserman. Voxceleb2: Deep speaker recognition. *arXiv preprint arXiv:1806.05622*, 2018.

[57] Google Meet Hardware. "https://workspace.google.com/products/meet-hardware/".

[58] Explore hardware options to enable Zoom. https://explore.zoom.us/en/workspaces/conference-room/.

[59] Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks. *arXiv preprint arXiv:1802.05957*, 2018.

[60] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[61] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In *European conference on computer vision*, pages 694–711. Springer, 2016.

[62] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Guilin Liu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. Video-to-video synthesis. *arXiv preprint arXiv:1808.06601*, 2018.

[63] VP8 Chromium Implementation. https://chromium.googlesource.com/external/webrtc/+/143cec1cc68b9ba44f3ef4467f1422704f2395f0/webrtc/modules/video_coding/codecs/vp8/vp8_impl.cc.

[64] WebRTC connectivity. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity.

[65] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[66] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. *arXiv preprint arXiv:1802.04799*, 2018.

[67] NVIDIA TensorRT. https://developer.nvidia.com/tensorrt.

[68] What Is Apple's Neural Engine and How Does It Work? https://www.makeuseof.com/what-is-a-neural-engine-how-does-it-work/.

[69] Android Neural Networks API. https://source.android.com/docs/core/ota/modular-system/nnapi.

[70] Kunihiko Fukushima and Sei Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.

[71] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[72] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[73] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

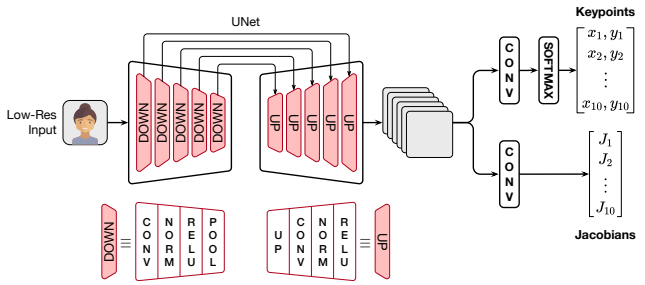


Figure 12: Keypoint Detector used as a precursor for computing the warping field between the reference and target images. Low-resolution versions of both frames are supplied to a UNet architecture [46], and then put through convolutional layers to generate keypoint locations and four "Jacobian" values in the neighborhood of each keypoint.

A Model Details

In the following subsections, we detail the structure of the motion estimator that produces the warping field for Gemino and the neural encoder-decoder pair that produce the prediction. We also describe additional details about the training procedure.

A.1 Motion Estimator

UNet Structure. The keypoint detector and the motion estimator use identical UNet structures (Fig. 12 and Fig. 13) to extract features from their respective inputs before they are post-processed. In both cases, the UNet consists of five up and down-sampling blocks each. Each downsampling block consists of a 2D convolutional layer [70], a batch normalization layer [71], a Rectified Linear Unit Non-linearity (ReLU) layer [72], and a pooling layer that downsamples by 2× in each dimension. The batch normalization helps normalize inputs and outputs across layers, while the ReLU layer helps speed up training. Each upsampling block first performs a 2× interpolation, followed by a convolutional layer, a batch normalization layer, and a ReLU layer. Thus, every downsampling layer reduces the spatial dimensions of the input but instead extracts features in a third "channel" or "depth" dimension by doubling the third dimension. On the other hand, every upsampling layer doubles in each spatial dimension, while halving the number of features in the depth dimension. In our implementation, the UNet structure always produces 64 features after its first encoder downsampling layer, and doubles from there on. The reverse happens with the decoder ending with 64 features after its last layer. Since the UNet structure operates on low-resolution input (as part of the keypoint detector and motion estimator), its kernel size is set to 3×3 to capture reasonably sized fields of interest.

Keypoint Detector. To obtain the warping field between the reference frame and the target, Gemino first uses a keypoint detector to locate key facial landmarks. It then uses a first-order approximation in the neighborhood of these keypoints similar to the FOMM [5]. To extract keypoints, we first downsample the input image to 64×64, and then feed it into the UNet structure described above in its RGB space itself. The UNet structure produces a set of output features from its decoder, which are then put through two separate pipelines to extract the keypoint locations and the "Jacobians." The keypoint locations are extracted via a single 7×7 convolutional layer, which is then put through a softmax to extract probabilities for keypoint presence at each spatial location. This is then converted to actual keypoint locations by performing a weighted average of these probabilities across the entire spatial grid. Note that this process is replicated 10 times by having 10 separate channels to extract 10 keypoints. The Jacobians are simply four floating point numbers that are used to approximate the movement (derivatives) in the neighborhood of each keypoint. This is used for the first-order approximation when computing the motion around each keypoint. To generate these Jacobians, the output from the UNet is simply put through a single 7x7 convolutional layer. Fig. 12 describes this architecture. Note that both the reference and the target images are fed to this pipeline independently to generate two separate sets of reference and target keypoints and Jacobians.

Motion Estimation Fig. 13 describes the working of the motion estimator in Gemino's design in detail. First, the motion estimator creates Gaussian *heatmaps* corresponding to the keypoint locations from both the reference and the target frames. It subtracts the two on a per-keypoint basis to generate the difference between the two frames' keypoints. It adds a separate heatmap consisting of zeros to denote the fact that the background is identical in the two frames. The motion estimator then generates *sparse motion* vectors or motion vectors in the neighborhood of each keypoint using the first-order Taylor series approximation [5] and the Jacobian values from the keypoint detector. These motion vectors (along with an identity for the background) are applied to the low-resolution reference frame to obtain a set of *deformed references*. This effectively generates 11 heatmaps (10 keypoints + 1 for background), and 11 different RGB (3 channels) deformed references. The 44 resulting channels are provided as input along with 3 RGB features from the low-resolution target image to another replica of the UNet structure described above. This UNet's decoder also outputs a set of predicted features based on all the provided 47 input features.

The predicted features are put through three separate 7×7 convolutional layer followed by Sigmoid layers and a Softmax layer to produce *three occlusion masks*. Each occlusion mask is later used in the decoding pipeline to convey how to combine information from three pathways: the warped high-resolution features, the non-warped high-resolution features, and the low-resolution features to generate the prediction. We use a Softmax layer to enforce that the sum of these three occlusion masks is 1 at every spatial location so that they do not compete in later parts of the decoding pipeline. Intuitively, this forces each pixel to be generated from one out of the three pathways. If a feature represents a part of the frame that has moved between the reference and the target frames, reconstruction relies on the HR warped pathway, while if it represents a part of the frame that has not moved, it relies on the non-warped

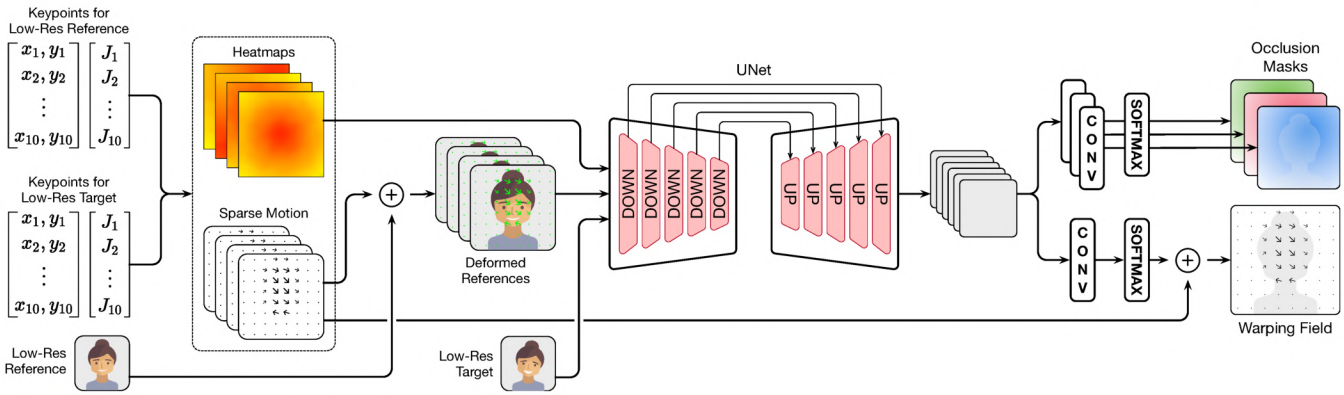


Figure 13: Gemino's motion estimation module that takes as input reference and target keypoints, along with a low-resolution reference and target frames, and produces a warping field and occlusion masks. The warping field helps the generator move encoded features from a full-resolution reference frame into the target frame's coordinate space, while the occlusion masks inform the decoder how to combine information from the low-resolution target frame with high-resolution (warped and unwarped) features from the reference frame.

HR pathway. Regions that are significantly different between the reference and the target use the LR features instead.

The predicted features are also fed into a single 7×7 convolutional layer followed by a Softmax to generate a *deformation mask* that reflects how to combine the sparse motion vectors previously obtained in the neighborhood of each location. The Softmax ensures that across every spatial location, different vectors are weighted appropriately to sum up to 1 finally. This deformation mask is applied to the sparse motion vectors to obtain the final warping field.

A.2 Image Synthesis

Fig. 14 describes the generative parts of Gemino design in more detail. First, the low-resolution target frames are put through a single 7×7 convolutional layer to produce low-resolution (LR) features to be used later during decoding. The high-resolution reference RGB frame is also fed through a single 7×7 convolutional layer to produce 32 high-resolution (HR) features with the same spatial dimensions before it is fed to the neural encoder. The neural encoder consists of four downsample blocks, each of which has the same structure as the downsample blocks in our UNet structure in the keypoint detector (Fig. 12). The four blocks ensure that we start at full-resolution (1024×1024 frames) with 32 features and end up with 256 separate 64×64 encoded HR features at the bottleneck. However, unlike the UNet, not all blocks in the neural encoder are equipped with skip connections. Specifically, only the first two have skip connections to the corresponding last two decoder blocks. The decoder also consists of upsampling blocks with the same composition as the UNet's downsampling blocks. Prior to decoding, a copy of the HR features are warped using the warping field from the motion estimator (Ⓐ). The warped HR features and the unwarped features (Ⓑ) are fed through five Residual Blocks [73] that refine them and also help prevent diminishing gradients during training. Each residual block consists of batch normalization, ReLU, and convolutional layers. The refined HR features, along with the LR features, and appropriate skip connections are combined using occlusion masks obtained from the motion estimator. Once combined, all three sets of features are fed through decoder's upsampling blocks to spatial dimensions of 1024×1024 frames with 32 features. This output is fed through a final 7×7 convolutional layer to bring it back to its RGB space as the prediction.

A.3 Training Details

To train our model, we use equally weighted (with a weight 10) multi-scale VGG perceptual loss [61], a feature-matching loss [62], and an L1 pixelwise loss. The multi-scale VGG loss and feature matching losses operate at scales of 1, 0.5, 0.25, and 0.125 of the image size. All of these scales have equal loss weights. We also use an adversarial loss [17] with a weight of one (one-tenth of the other losses). The keypoints use an equivariance loss similar to the FOMM [5]. Our keypoint detector is unchanged relative to the FOMM, and so we reuse a trained checkpoint on the VoxCeleb dataset [45], but fine-tune it on a per-person basis. For other layers, we adopt a simple strategy: if the dimensions match the equivalent layers in the FOMM (*e.g.*, encoder layers, residual layers), we initialize them based on a FOMM checkpoint, and if the dimensions do not match, they are initialized randomly. All models are personalized (either trained from scratch or fine-tuned if the layers are initialized to FOMM checkpoints). All models for 1024×1024 output resolution were trained with a batch size of 2 on NVIDIA A100 GPUs while models for 512×512 output resolution were trained with a batch size of 2 on NVIDIA V100 GPUs. The details of the dataset we curate is shown in Tab. 8 As discussed earlier, models are trained on decompressed VPX frames at different bitrates. To support this, we encode individual frames at the target bitrate during train-time *after* they are sampled from the dataset. Since we encode individual frames, this only results in keyframes at train-time, but these frames are still representative of what compressed video frames more generally look like at that bitrate.

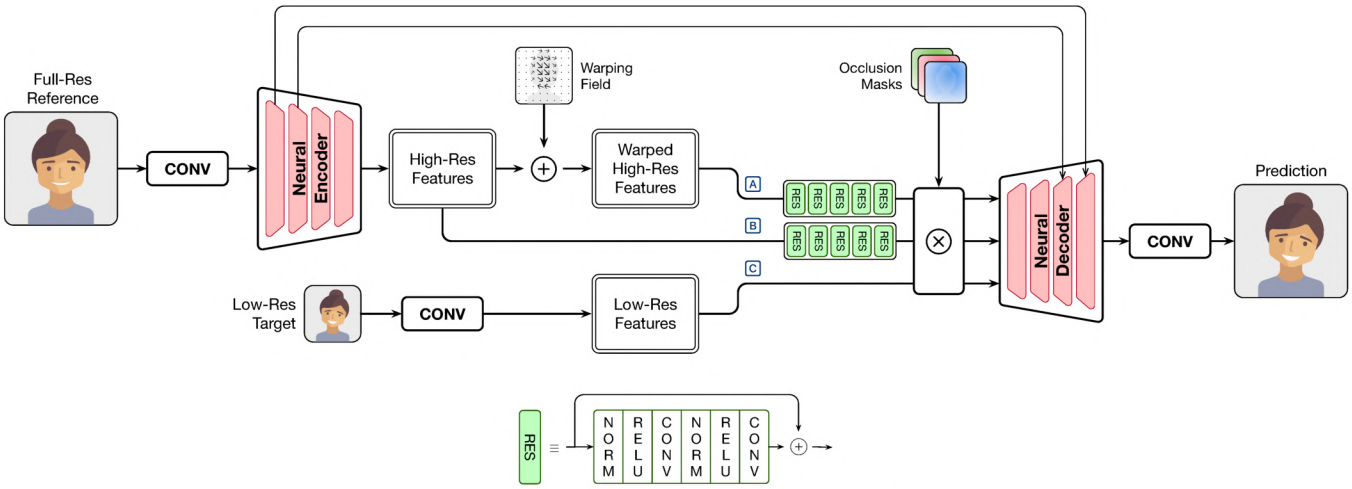


Figure 14: Gemino's encoder-decoder pair that is responsible for synthesizing the prediction. The encoder runs the high-resolution reference image through a series of downsampling layers to produce encoded features. A copy of these encoded HR features is warped, and both the warped and non-warped features are refined through a sequence of residual blocks. Meanwhile, a convolutional layer extracts low-resolution features from a low-resolution target. The three sets of features are combined based on occlusion masks from the motion estimator before they are decoded to result in the final prediction.

	Training Videos		Test Videos	
Youtuber	Total Len.	Avg. Bitrate	Avg. Len.	Avg. Bitrate
Adam Neely	31 min	1082 Kbps	146 s	1303 Kbps
Xiran Jay Zhao	30 min	2815 Kbps	180 s	1560 Kbps
The Needle Drop	33 min	2013 Kbps	206 s	1286 Kbps
fancy fueko	15 min	4064 Kbps	124 s	1607 Kbps
Kayleigh McEnany	28 min	2521 Kbps	180 s	1247 Kbps

Table 8: Details of our dataset. All videos are at 1024×1024.

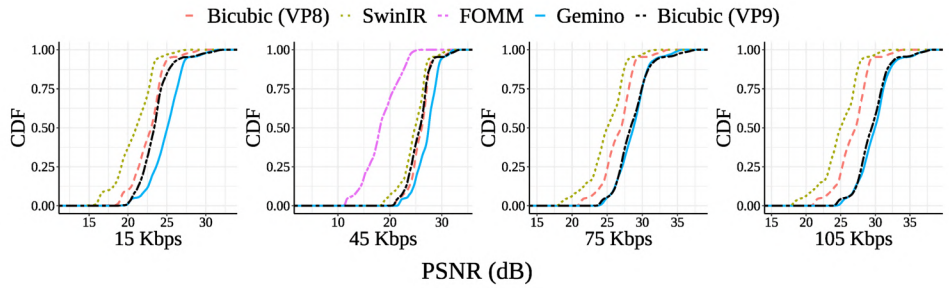
B Extended Evaluation

B.1 Low-bitrate Regime Comparisons

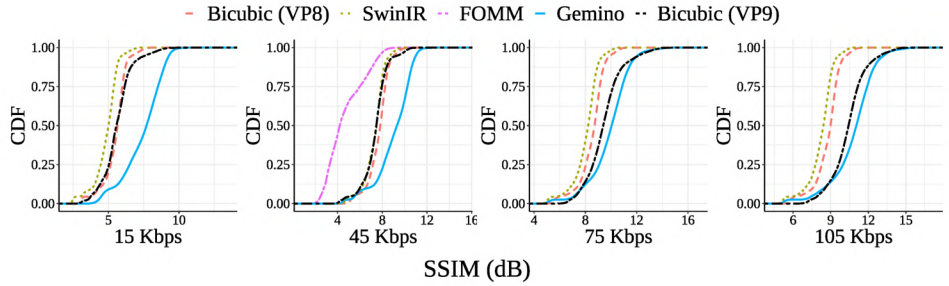
We plot a CDF of the visual quality in the context of PSNR and SSIM across all frames in our corpus in Fig. 15 at 15 Kbps, 45 Kbps, 75 Kbps and 105 Kbps. At each target bitrate, we use the largest resolution supported by the underlying video codec. Gemino uses VP8 at 15 Kbps and 45 Kbps but VP9 at higher bitrates; VP8 and VP9 do not differ much at lower bitrates, but VP9 allows 512×512 to be compressed all the way to 75 Kbps while VP8 cannot. Since our results in §5.4 suggest using the highest resolution at any target bitrate, we use VP9 at 75 Kbps and 105 Kbps. The CDF shows that Gemino's reconstructions are robust to variations across frames and orientation changes over the course of a video. At 45 Kbps when upsampling from a 256×256 frame, Gemino outperforms all other baselines across all frames and both metrics. Specifically, its synthesized frames are better than FOMM by nearly 5 dB in SSIM and 10 dB in PSNR throughout. While the differences in PSNR between Bicubic atop VP9 and Gemino are small at 105 Kbps, the difference at the median is over 2 dB at 15 Kbps. The differences in SSIM is even more pronounced at 15 Kbps with Gemino outperforming both bicubic approaches by nearly 2.5 dB at the median. As the bitrate is lowered, the differences between VP8 and VP9 also shrink, resulting in very similar visual quality for both of their bicubic upsampling approaches. As we show in §B.2, differences in PSNR are not as reflective of how natural the image looks to the human eye, and so we rely on LPIPS [20] primarily, and SSIM when relevant.

B.2 Comparing the metrics.

To understand the difference between different quantitative metrics and the visual quality each of them map to, we show a strip (with the associated metrics) that compares Gemino with the reconstruction of bicubic atop VP8 and VP9 when upsampling 256×256 frames at 45 Kbps in Fig. 16, 17, 18 and 19. For example, in Fig. 16, while the output produced by Gemino is significantly better than that of both bicubic approaches, it manifests as only a 2 dB difference in PSNR and SSIM, both of which have very wide ranges. In contrast, we see an improvement of 0.1 in LPIPS. LPIPS is constrained to be between 0 and 1, making it easier to put the 0.1 improvement in context. We see similar trends in Fig. 17 and in Fig. 18, where a difference of 0.13-0.14 in LPIPS, and ∼2 dB PSNR manifests as smoother output on the facial regions. In contrast, while the PSNRs in Fig. 19 are fairly close across the bicubic approaches and Gemino, the SSIM and LPIPS differences reflect how much more natural Gemino's facial reconstruction looks. Frames like these explain why the PSNR graphs in Fig. 6 are a lot closer than the other two metrics, and motivate the use of LPIPS as our main metric. However, even though the two bicubic approaches in Fig. 17 differ significantly in the smoothness of the face, all three metrics are close enough that the difference would not be very perceptible on a plot. As a result, we acknowledge that none of these metrics are truly perfect, and a combination of them



(a) PSNR achieved by different schemes in the low-bitrate regime



(b) SSIM achieved by different schemes in the low-bitrate regime

Figure 15: CDF of PSNR and SSIM achieved by different schemes on all frames in low-bitrate regimes. The differences between Gemino and other approaches becomes more pronounced in lower-bitrate regimes.

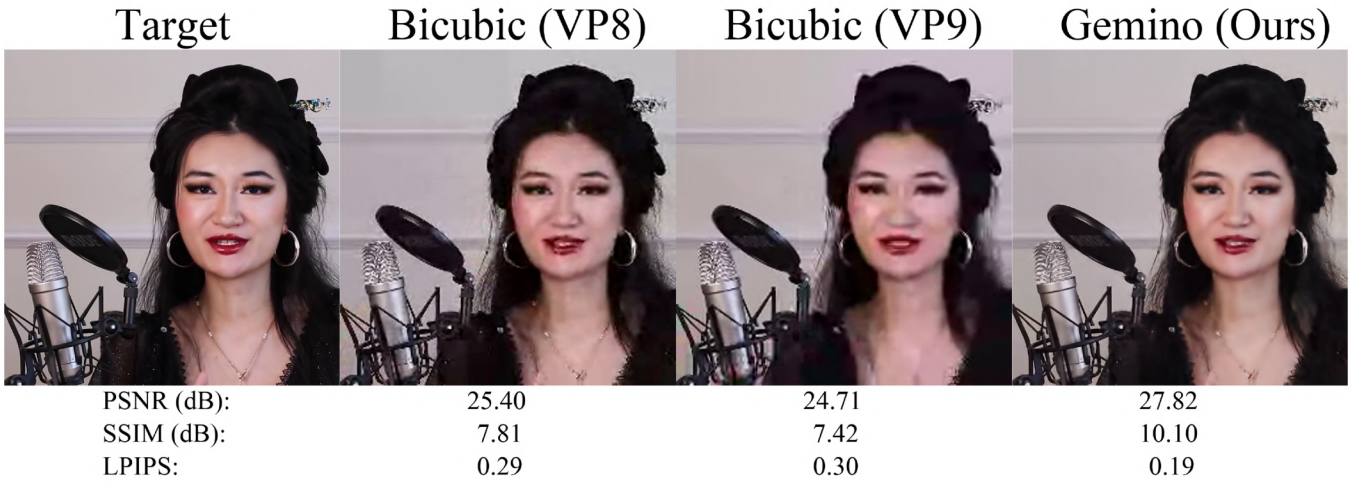
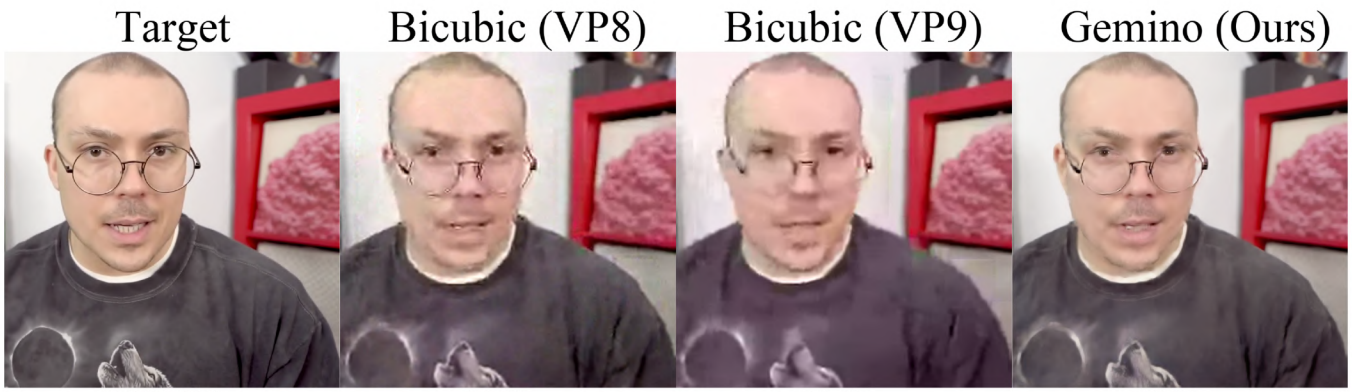


Figure 16: Full-size examples to illustrate the visual quality differences and their correlation with the metric values.

all along with visual strips is needed for a thorough evaluation.



PSNR (dB):	26.80	26.40	28.85
SSIM (dB):	7.44	7.22	9.91
LPIPS:	0.34	0.33	0.20

Figure 17: Full-size examples to illustrate the visual quality differences and their correlation with the metric values.

PSNR (dB):	26.80	26.40	28.85
SSIM (dB):	7.44	7.22	9.91
LPIPS:	0.34	0.33	0.20

Figure 18: Full-size examples to illustrate the visual quality differences and their correlation with the metric values.

PSNR (dB):	25.47	24.59	25.64
SSIM (dB):	8.31	7.46	9.29
LPIPS:	0.28	0.31	0.24

Figure 19: Full-size examples to illustrate the visual quality differences and their correlation with the metric values.

ARTEMIS: Adaptive Bitrate Ladder Optimization for Live Video Streaming

Farzad Tashtarian[†] Abdelhak Bentaleb[§] Hadi Amirpour[†] Sergey Gorinsky[*]
Junchen Jiang[$] Hermann Hellwagner[†] Christian Timmerer[†]

[†]*Christian Doppler Laboratory ATHENA, Alpen-Adria Universität Klagenfurt*
[§]*Concordia University* [*]*IMDEA Networks Institute* [$]*University of Chicago*

Abstract

Live streaming of segmented videos over the Hypertext Transfer Protocol (HTTP) is increasingly popular and serves heterogeneous clients by offering each segment in multiple representations. A bitrate ladder expresses this choice as a list of bitrate-resolution pairs. Whereas existing solutions for HTTP-based live streaming use a static bitrate ladder, the fixed ladders struggle to appropriately accommodate the dynamics in the video content and network-conditioned client capabilities. This paper proposes ARTEMIS as a practical scalable alternative that dynamically configures the bitrate ladder depending on the content complexity, network conditions, and clients' statistics. ARTEMIS seamlessly integrates with the end-to-end streaming pipeline and operates transparently to video encoders and clients. We develop a cloud-based implementation of ARTEMIS and conduct extensive real-world and trace-driven experiments. The experimental comparison vs. existing prominent bitrate ladders demonstrates that live streaming with ARTEMIS outperforms all baseline solutions, reduces encoding computation by 25%, end-to-end latency by 18%, and increases the quality of experience by 11%.

1 Introduction

Live streaming is an increasingly prominent variant of video streaming. Video applications keep gaining popularity in general, with the Internet experiencing a 24% increase in video traffic during 2022 [55]. Live streaming constitutes a major contributor to this growth and feeds on support by social media platforms and streaming services such as Facebook Live, Twitch, and YouTube Live. The skyrocketing rise of live streaming in all Internet traffic from less than 1% to nearly 18% during the 2015-2022 period substantiates the importance of this streaming mode.

HTTP Adaptive Streaming (HAS) comprises an attractive option for not only Video On Demand (VOD) but also live streaming. The two leading representatives of the HAS paradigm are Apple's HTTP Live Streaming (HLS) [52] and standardized Dynamic Adaptive Streaming over HTTP (DASH) [36]. Both formats have low-latency extensions [20, 26] that accept the same Common Media Application Format (CMAF) [35] for video packaging. While WebRTC [38] enables streaming with subsecond latency and creates a promising alternative to HAS formats for interactive applications, the HAS paradigm maintains its dominance in live streaming distribution [63] due to its easy deployment, great scalability, and good performance.

In HAS, an origin server partitions a video into segments and encodes each segment into multiple representations characterized typically by a bitrate and resolution. A *bitrate ladder*, aka an *encoding ladder* or simply a *ladder*, comprises a list of the bitrate-resolution pairs. Each client, aka player, downloads from the server a *manifest* file that describes the bitrate ladder and other metadata. The video delivery is segment by segment, with the client requesting the next segment in a representation chosen by an *adaptive bitrate (ABR) algorithm* [24]. The choice strives to accommodate dynamic network conditions and balance conflicting performance objectives. For example, a higher bitrate might increase both video quality and stall duration because the client does not receive the segment in time for its playback.

A key metric of HAS performance is Quality of Experience (QoE) which captures the overall satisfaction of the user with the streaming service [56]. In this paper, we evaluate QoE using the model introduced by Comyco [33]. This QoE model measures video quality by means of Video Multimethod Assessment Fusion (VMAF) [45] and expresses QoE as a weighted sum of VMAF, stall duration, and VMAF instability over a sequence of segments. VMAF relies on machine learning to account for human perception, spatial and temporal video characteristics, and many other factors. Appendix A describes the QoE model in more detail.

HAS scales up to millions of clients due to two features. First, because the number of representations, rather than the number of clients, controls the encoding, storage, and bandwidth overhead, bitrate ladders contain relatively few representations, *e.g.*, Twitch and Apple use ladders with six and nine representations, respectively [21, 22]. When the network bandwidth available for a client aligns with a bitrate imperfectly, the discrepancy does not disrupt QoE as long as it lies within the Just-Noticeable Difference (JND) [18]. Second, HAS employs Content Delivery Networks (CDNs) which deploy edge caches around the world to serve end users with low latency [31].

In comparison to VOD, live streaming faces new challenges. The fundamental difference lies in the real-time operation of the end-to-end pipeline from the video ingestion at the camera

to the playback at the client. Specifically, VOD encoding is offline and free from stringent latency constraints. The state of the art in ladder construction goes beyond one-size-fits-all ladders [32, 42] and leverages video content and viewing context to build more effective ladders [28, 32, 37, 40, 47, 53, 54, 64, 65]. The construction of content-aware and context-aware ladders commonly relies on exhaustive search or other time-consuming methods. In contrast, live streaming is subject to tight constraints on end-to-end latency and calls for new designs. For example, live streams encode on a subsegment level, *e.g.*, all the way down to individual frames. Also, a live encoder is unable to assess video quality via VMAF because the computation of VMAF would introduce substantial latency which might even exceed the encoding latency. On the ABR side, L2A [39] and LoL$^+$ [22] represent prominent low-latency ABR algorithms that leverage the subsegment encoding structure and incorporate additional mechanisms. For example, LoL$^+$ includes a learning-based technique that proactively controls the playback speed by considering both current latency and buffer occupancy level so as to jointly handle stalls and achieve the latency target.

The live mode also creates opportunities for more effective streaming. In VOD, the origin server stores the video in all representations of the bitrate ladder for future streaming to a priori unknown clients. In live streaming, the server does not necessarily need to store the streams for future playback. Apart from making video storage less of a concern, this aspect of live streaming opens the possibility of dynamically configuring the bitrate ladder of the session to improve the encoding efficiency and QoE of current clients.

In this paper, we design ARTEMIS[1], a system that dynamically constructs effective ladders for the live encoder during the live video session without any pre-encoding step. The system objectives are to: (1) support end-to-end streaming latency of a few seconds, (2) provide heterogeneous clients with high QoE, (3) utilize network bandwidth efficiently with low storage and processing overhead, and (4) operate transparently with existing video codecs, players, and ABR algorithms.

ARTEMIS achieves its key innovation by collecting client-state information via CDN log files and leveraging this information to dynamically configure the bitrate ladder. The added mechanisms preserve the scalability of CDN-assisted HAS. In particular, whereas the number of representations encoded in real time constitutes the most critical resource in a live streaming system, the encoding of the live video into the representations that fully accommodate the capabilities of all individual clients in the session does not constitute a practical option, and ARTEMIS instead constructs bitrate ladders with a relatively small number of representations. Another important innovation of ARTEMIS is a mega-manifest file that advertises a large number of representations. ARTEMIS utilizes the mega-manifest to collect information about the bitrate preferences of the clients with a higher fidelity.

[1] *ARTEMIS* is an abbreviation of the following full name of the design: *A*daptive bit*RaTE* ladder opti*MI*zation for live video *S*treaming.

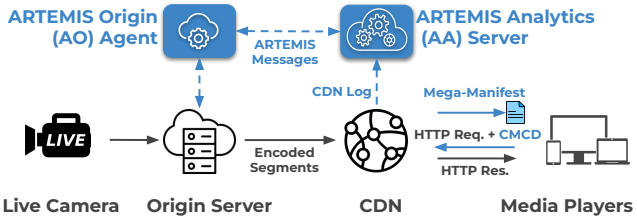


Figure 1: Conceptual architecture of live video streaming with ARTEMIS enhancements (in blue).

Figure 1 depicts the ARTEMIS architecture and its role in live streaming. ARTEMIS consists of an *ARTEMIS analytics (AA) server* and *ARTEMIS origin (AO) agent* with limited communications between these two components. Operating in real time to dynamically compose the bitrate ladder from the representations advertised in the mega-manifest, the AA server utilizes CDN logs in the Common Media Client Data (CMCD) format [7] to scalably collect client-side information about stall duration and numbers of requests for each bitrate in the mega-manifest. ARTEMIS uses the Peak Signal-to-Noise Ratio (PSNR) [34] as a metric of video quality because the latency of computing a PSNR value is negligible, within a millisecond. Specifically, based on the PSNR values computed by the live encoder for past segments, the AO agent trains a function predicting the video quality of future segments and communicates this quality indicator function to the AA server. Because the AO agent runs on the machine of the live encoder, ARTEMIS trains the quality indicator function at the AO agent, rather than at the AA server, in order to reduce traffic from the live encoder to the AA server. Utilizing the quality indicator function received from the AO agent, the AA server solves a Mixed Integer Linear Program (MILP) to update the bitrate ladder. Upon receiving the updated ladder from the AA server, the AO agent instructs the origin server to encode the subsequent segments of the video according to this new ladder. In the most likely economic realization, the streaming service provider administers both components of ARTEMIS.

We implement the ARTEMIS system in the Amazon cloud infrastructure and report an extensive real-world evaluation of ARTEMIS-enhanced live streaming. Specifically, our Python implementations of the AA server and the AO agent run in the Amazon Elastic Compute Cloud (EC2) [2]. To complete the end-to-end streaming pipeline, we utilize the Bitmovin Live Encoder [5] for the origin-server encoding of videos, Amazon CloudFront as the CDN, and multiple instances of the DASH JavaScript Player (dash.js) [27] as video clients on the CAdViSE platform [57]. The Bitmovin Live Encoder and CAdViSE also run in EC2. The live streaming experiments confirm that ARTEMIS achieves its design objectives and significantly outperforms existing

techniques for ladder construction in terms of provided QoE and network utilization while imposing low computation and communication overhead.

Our paper makes the following main contributions:

1. We design ARTEMIS, a practical system that enhances live video streaming by dynamically constructing the bitrate ladder accounting for the content complexity and network conditions. The construction leverages the clients' fine-grained bitrate preferences and stall information collected scalably through the CDN.

2. The paper reports a cloud-based implementation of ARTEMIS and an extensive real-world evaluation of its utility for end-to-end live streaming.

3. The experimental comparison vs. existing prominent bitrate ladders demonstrates that ARTEMIS delivers multi-objective improvements over the static ladders to reduce encoding computation by 25%, end-to-end latency by 18%, and increase QoE by 11%.

2 Related Work

The traditional HAS approach relies on the same bitrate ladder throughout the streaming session. The fixed ladder might be agnostic of the video content, *e.g.*, Apple's ladder [51], or depend on the content type, *e.g.*, per-title encoding by Netflix [28, 47]. One can classify recent advanced solutions into content-aware and context-aware categories.

Content-aware ladder construction accounts for the content complexity. Whereas the initial proposal of per-title encoding [28] seeks to improve the resolution for each bitrate in the ladder, [19] aims at content-aware improvements in both resolution and frame rate. [46] considers the content complexity to partition the video offline for streaming with segments of variable duration and then augments the bitrate ladder with additional representations to improve QoE via fine-grained adaptation. [40] extracts content features and applies machine learning to configure each quantization parameter (QP) based on the rate-distortion curves of different resolutions. [25] uses metrics of video quality to select a resolution for each bitrate in a perception-aware manner. [49, 50] extract low-complexity video features to set the resolutions and QPs in bitrate ladders for live streaming. While all the above solutions successfully leverage content awareness to improve bitrate ladders, [19, 28, 46] rely on brute force to construct the ladders, which is computationally untenable in live streaming. On the other hand, [25, 40, 49, 50] do not consider network conditions. In contrast, ARTEMIS accounts for both content complexity and dynamic network-conditioned client capabilities to efficiently construct dynamic ladders for live streaming.

Context-aware ladder construction addresses the importance of the context such as the network bandwidth available for the segment download by the clients. [54] models the available network bandwidth as a continuous random variable and uses its probability density function to build ladders. [43] constructs bitrate ladders by analyzing the content complexity and historical data on the available network bandwidth in order to reduce the likelihood that the user quits watching the streamed video. [60] accounts for the user population, network dynamics, video content, and other factors in its ladder construction with the objective to improve user satisfaction. [32] applies deep reinforcement learning to build ladders with features that describe the content, available network bandwidth, and storage overhead. [41] uses a Markov model to predict the bitrate of the next requested segment so as to support transcoding and ladder construction in VOD settings. [58] modifies ABR algorithms with a plug-in to obtain feedback from the clients, characterizes video quality via VMAF estimates, and evaluates the utility of the constructed bitrate ladders via trace-driven emulations. Although sharing some aspirations and techniques with ARTEMIS, the above context-aware proposals tend to make problematic design choices that jeopardize their chances for wide deployment in practice, *e.g.*, deep learning or other computationally expensive techniques unsuitable for live streaming. The key innovation of ARTEMIS as a practical scalable system for live streaming lies in collecting client-side information via CDN logs and leveraging this information to construct dynamic context-aware bitrate ladders. ARTEMIS operates transparently to ABR algorithms and uses a mega-manifest file to accurately detect the bitrate preferences of the clients.

3 Problem Description and Motivation

This paper deals with the problem of constructing improved bitrate ladders for live streaming. As shown in Figure 1, the end-to-end streaming pipeline passes through a CDN and has two ends on the origin and client sides. On the origin side, the encoder creates a bitrate ladder from the live video captured by a camera. On the client side, the ABR algorithm continuously requests segments of the video at a dynamically chosen representation, with the client subsequently buffering and playing back the received segments. In this section, we discuss the utility and challenges of using information from both origin and client sides to improve the ladder construction.

Client-side motivation. While ARTEMIS constructs the bitrate ladder based on the bitrates requested by the clients, an alternative foundation might be the network bandwidth available for the segment download. ABR algorithms typically predict the future network bandwidth through various techniques, *e.g.*, based on historical measurements of throughput and occupancy level of the playback buffer [24]. However, ABR algorithms differ greatly in their logic of selecting the bitrate for the next requested segment: whereas the greedy approach requests the highest bitrate that does not exceed the predicted available bandwidth, many ABR

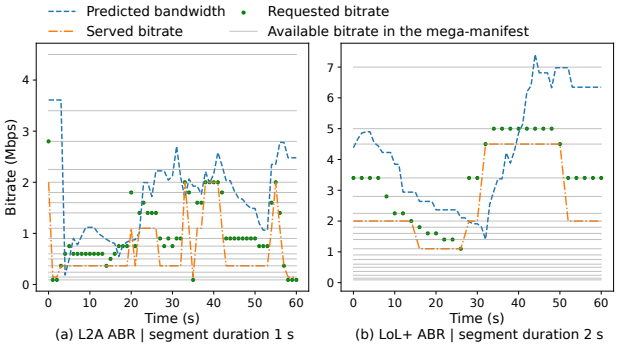


Figure 2: The bitrate ladder advertised in the mega-manifest, predicted bandwidth and requested bitrates by two ABR algorithms, and actually served bitrates at the origin side.

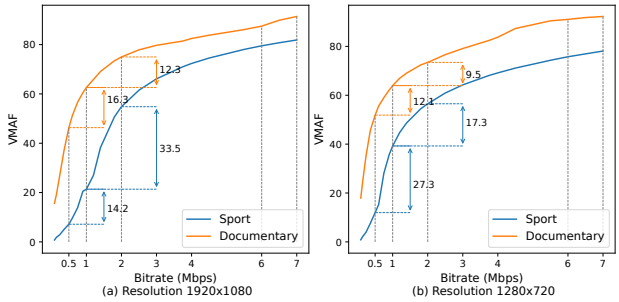


Figure 3: Dependence of video quality on the bitrate for two different content types.

algorithms choose to keep the requested bitrate stable over consecutive segments because QoE might benefit from the bitrate stability even when the bitrate is significantly below or above the available network bandwidth.

To support live streaming with low latency and utilize efficiently the limited computation, bandwidth, and storage resources, ARTEMIS constrains the number of encodings to yield fewer representations than advertised in the mega-manifest, and a client might receive a segment in a representation with a lower bitrate than requested. Although serving the segment at a lower bitrate might degrade QoE for the client, prior work adopting this technique for similar overhead reasons shows that the technique works well in conjunction with client-side ABR algorithms and supports an effective trade-off between the complexity and QoE [44, 59].

We explore the loose coupling between the client-side and origin-side logics in experiments where the mega-manifest advertises 19 representations[2]. Out of the 19 representations, the origin server actually maintains only the five representations with the bitrates of 0.145, 0.365, 1.1, 2, and 4.5 Mbps. When the client requests a representation with a different bitrate, the server delivers the maintained representation with the highest bitrate that does not exceed the requested one. The experiments evaluate the L2A ABR algorithm [39] with the segment duration of 1 s and the LTE network trace [61] as well as the LoL$^+$ ABR algorithm [22] with the segment duration of 2 s and the Cascade network trace [23]. In both experiments, the content type is animation. Figures 2a and 2b plot the results for L2A and LoL$^+$, respectively, and depict the 19 representations in the mega-manifest as horizontal gray lines, bandwidth predictions by the ABR algorithm as blue dashed lines, its requested bitrates as green dots, and actually served bitrates as orange dash-dotted lines.

[2](0.09 Mbps, 270p), (0.145 Mbps, 270p), (0.24 Mbps, 270p), (0.365 Mbps, 360p), (0.5 Mbps, 360p), (0.6 Mbps, 360p), (0.75 Mbps, 360p), (0.9 Mbps, 360p), (1.1 Mbps, 480p), (1.4 Mbps, 480p), (1.6 Mbps, 720p), (1.8 Mbps, 720p), (2 Mbps, 720p), (2.25 Mbps, 720p), (2.8 Mbps, 720p), (3.4 Mbps, 720p), (4.5 Mbps, 1080p), (5 Mbps, 1080p), and (7 Mbps, 1080p).

Figure 2 provides two relevant observations. First, the predicted available bandwidth is a poor proxy of the bitrate requested by the ABR algorithm. It is common for the discrepancy between these two values to span multiple representations. Therefore, *compared to the available network bandwidth, the bitrates requested by the clients constitute a better foundation for constructing a dynamic bitrate ladder.* Second, the L2A and LoL$^+$ algorithms adapt the bitrate effectively even though the origin server encodes segments into a smaller number of representations than advertised in the mega-manifest. Thus, *the mega-manifest empowers the origin server to detect the bitrate preference of the client with a higher accuracy and yet preserves the effectiveness of diverse ABR algorithms*. The above two observations substantiate a promise of our approach where the origin server leverages the mega-manifest technique and relies on the bitrates requested by the clients, rather than the available network bandwidth, to construct the dynamic bitrate ladder.

Origin-side motivation. Even if the origin server collects real-time fine-grained feedback about the bitrate preferences of the clients, the dynamic construction of an effective bitrate ladder requires additional information from the origin side. In particular, video quality is relevant because it strongly correlates with QoE. Figure 3 plots VMAF as a function of the bitrate for two different resolutions in the experiments that consider two content types of sport and documentary. The graphs depict the average VMAF over 100 segments, where each segment has the duration of 2 s, and show that VMAF grows sublinearly with the bitrate and greatly depends on the content type. Hence, in addition to the bitrate preferences of the clients, the construction of effective bitrate ladders should also consider the sublinear impact of the chosen bitrate on video quality as well as the specific content type. Thus, *origin-side awareness of video quality and content complexity is critical for constructing an effective bitrate ladder*.

Exploitation of content awareness in live streaming has additional challenges compared to VOD. Content-aware VOD solutions typically rely on offline search in the bitrate and resolution spaces to find a bitrate ladder where the

chosen list of bitrate-resolution pairs provides a good coverage of the VMAF space for the given content. However, the exhaustive search requires extensive computations and imposes extra latency unacceptable for real-time operation. Similarly, the computation of VMAF is untenably slow for live streaming. Therefore, *construction of a dynamic bitrate ladder for live streaming requires efficient techniques to quickly measure content complexity and select a content-aware ladder configuration.*

4 System Design

4.1 Design Principles

Based on the motivation in Section 3, we now present our ARTEMIS system and start by establishing its design principles. ARTEMIS aspires to build a dynamic bitrate ladder for a live streaming session by leveraging real-time information from both origin and client sides of the streaming pipeline. However, the provided improvements in the bitrate ladder should not undermine the traditional ability of HAS to scale up well with respect to the number of clients in the session, computation and storage overhead, and bandwidth consumption. This constraint forms a basis for our first design principle:

Design Principle 1 *Enhancements of the ladder construction should preserve scalability of live streaming and, in particular, require only affordable amounts of extra feedback from both client and origin sides.*

The existing ecosystem of live streaming is diverse in regard to deployed clients and their ABR algorithms. At the same time, ABR streaming also has standard features, e.g., the client learns about available representations by receiving from the origin server a manifest file and reacts by requesting one of the advertised representations. For ease of deployment, the proposed design should exploit common elements of end-to-end streaming and avoid modifications in the components that exist in heterogeneous instances. The above considerations lead us to the following design principle:

Design Principle 2 *ARTEMIS should seamlessly integrate with today's streaming ecosystem by leveraging its standard features and, in particular, operate transparently to heterogeneous clients and their ABR algorithms.*

Constructing an effective bitrate ladder with inexpensive overhead constitutes an important but not the only goal in live streaming. A major objective of end-to-end streaming is to support high QoE. Furthermore, because live streaming imposes tight constraints on end-to-end latency, the ladder construction should be sufficiently fast so that the streaming session fulfills the end-to-end latency constraints. Thus, we establish the final design principle for ARTEMIS:

Design Principle 3 *The ladder construction should simultaneously achieve multiple interdependent objectives that include the ladder effectiveness, reasonable overhead, high QoE, and satisfaction of end-to-end latency constraints.*

4.2 ARTEMIS Overview

We derive the ARTEMIS design from the principles established in Section 4.1. Figure 1 shows the conceptual architecture of our proposal with the AA server and AO agent as its two primary components. We envision that the provider of the streaming service administers both components and instantiates them in a cloud, *e.g.*, in EC2.

To fulfill Design Principle 1 on the client side, the AA server in ARTEMIS utilizes a CDN service to scalably collect client-state information. Akamai [1] and other major CDNs offer such services that transmit log files to third parties, *e.g.*, Conviva [6] and Datazoom [8] analytics platforms. When a streaming session uses multiple CDNs, the analytics platform obtains the client-state information separately from each individual CDN. Without loss of generality, the rest of our paper assumes that each streaming session utilizes a single CDN. In ARTEMIS, each player appends its stall information and unique player identifier (*pid*) in the CMCD format to the Uniform Resource Locator (URL) of the HTTP request message sent to the CDN edge server [3, 7]. The URL also includes the ID of the segment representation requested by the player. The CDN compiles both kinds of information provided by the players to the CDN edge servers and periodically provides the AA server with log files that contain information about stall duration and requested representations for all players in the session during the covered period. By processing the CDN log files with negligible computation overhead, the AA server extracts aggregate statistics in the form of average stall duration and request counts for each bitrate in the mega-manifest. The AA server leverages these two kinds of statistics in its computation of a new bitrate ladder.

To satisfy Design Principle 1 on the origin side, the AO agent acts as an intermediary between the origin server and the AA server in order to reduce traffic from the live encoder to the AA server. The AO agent runs on the machine of the live encoder, trains a quality indicator function on the segments produced by the encoder, and sends the quality indicator function to the AA server. The reliance of ARTEMIS on PSNR as the quality metric enables low end-to-end latency of a few seconds while accomplishing the QoE, overhead, and other objectives in compliance with Design Principle 3. Because the computation of VMAF is unacceptably slow for live streaming, Figure 14 in Appendix B shows a strong correlation between PSNR and VMAF across different segment durations and content types.

The AA server utilizes the collected origin-side and client-side information to solve an optimization program that

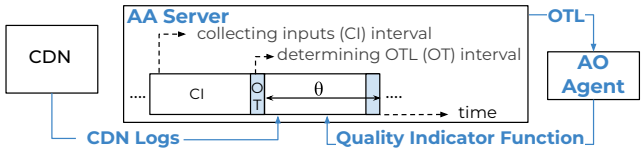


Figure 4: Time-slotted operation by ARTEMIS.

produces an *optimized temporary ladder (OTL)*. Because the number of representations encoded in real time constitutes the critical resource in a live streaming system, the AA server dynamically composes the OTL from only some of the representations listed in the mega-manifest and instructs the origin server through the AO agent to use the OTL as the current ladder for segment encoding. The origin server does not encode any mega-manifest representations that do not appear in the OTL. When a client requests a representation absent from the OTL, ARTEMIS instructs the origin server to stream the segment in the closest lower representation in the OTL. Hence, ARTEMIS operates transparently to the clients and origin server in accordance with Design Principle 2.

ARTEMIS relies on time slots in its internal and external communications. As shown in Figure 4, each time slot is θ seconds long and consists of two distinct intervals: the collecting inputs (CI) interval and determining OTL (OT) interval. During the CI interval, the AA server collects inputs from the CDN and AO agent. In the OT interval, the AA server uses the collected data to update the OTL. Below, we detail the methodology and explain our algorithm for determining the OTL based on the data collected during the CI interval.

4.3 OTL Selection

During each OT interval, the AA server decides whether to update the OTL. In the following, we formulate a Mixed Integer Linear Program (MILP) that computes a new OTL. Table 2 in Appendix C summarizes our notation. Without loss of generality, we assume only one fixed resolution for each bitrate in the mega-manifest. The optimal resolution for a bitrate is determinable in an online manner [50]. Set $B = \{1,2,\ldots,m\}$ indexes the m mega-manifest representations in increasing order of their bitrates, and b_i refers to the bitrate of representation i. List $R = \{r_1, r_2, \ldots, r_m\}$ consists of m nonnegative integers, where r_i denotes the number of requests for bitate b_i.

The OTL contains at most ℓ of the m representations. For each representation i, we define binary variable x_i that indicates whether bitrate b_i appears in the OTL ($x_i = 1$) or not ($x_i = 0$). When the OTL does not include bitrate b_i, and the AA server observes requests for this bitrate (*i.e.*, $r_i > 0$), ARTEMIS selects a lower bitrate in the OTL to serve the requests. To model such situations, our MILP incorporates a list of $i-1$ binary variables $Y_i = \{y_{1,i}, y_{2,i}, \ldots, y_{i-1,i}\}$ and the following two constraints:

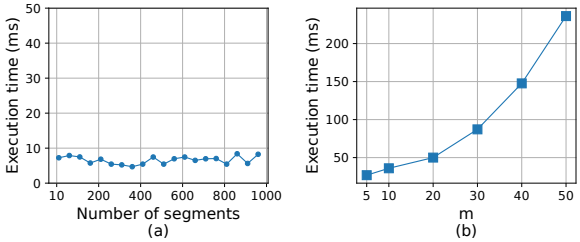


Figure 5: Execution time to: (a) train quality indicator function F for different numbers of segments and (b) solve the MILP as a function of m, the number of representations in the mega-manifest.

$$\sum_{j \in B \,\&\, j<i} y_{j,i} + x_i = 1 \quad \forall i \in B, \tag{1}$$

$$\sum_{i \in B \,\&\, j<i} y_{j,i} \leq x_j \times m \quad \forall j \in B \tag{2}$$

where $y_{j,i} = 1$ indicates that ARTEMIS instructs the origin server to transmit the segment at bitrate b_j which is lower (*i.e.*, $j < i$) than requested bitrate b_i. Constraint 1 guarantees that ARTEMIS accommodates the requests for bitrate b_i by either including this bitrate into the OTL ($x_i = 1$) or serving them at a lower bitrate in the OTL ($\sum_{j \in B \,\&\, j<i} y_{j,i} = 1$). Constraint 2 enforces $x_j = 1$ when the OTL includes bitrate b_j to accommodate requests for higher bitrates than b_j.

The following constraint ensures that the OTL comprises at most ℓ representations:

$$\sum_{i \in B} x_i \leq \ell. \tag{3}$$

If all bitrates in the OTL would change every OT interval, excessive bitrate switching might degrade the QoE. To prevent frequent bitrate changes, the MILP imposes an upper limit on the number of changes in the OTL over two consecutive OT intervals:

$$\sum_{i \in B} |x_i - \bar{x}_i| \leq \beta, \tag{4}$$

where $\beta > 0$ and $\bar{x}_i \in \{0,1\}$ indicates whether bitrate b_i appears in the OTL of the previous OT interval.

The live encoder calculates the PSNR value of a segment on the fly during the encoding process, *e.g.*, by appending `-psnr` to the FFmpeg command line [10]. Based on the PSNR values collected from the encoder for previous segments, the AO agent computes a function estimating the quality of upcoming segments. Specifically, the AO agent uses linear regression to train quality indicator function F that maps a bitrate to PSNR. Figure 5a plots the execution time to train function F. The AA server leverages quality indicator function F to estimate the PSNR values for all bitrates in the mega-manifest for subsequent segments. When the OTL relies on bitrate b_j to accommodate the requests for higher bitrate b_i, the AA server utilizes function F to measure the respective change in PSNR. The following inequality introduces the nonpositive real variable q to express the quality improvement that the requested bitrates would provide compared to the bitrates chosen in the OTL:

$$q \times \sum_{i \in B} r_i \leq \sum_{i \in B} \sum_{j \in B \,\&\, j<i} r_i \times y_{j,i} \times \big(F(b_j) - F(b_i)\big). \quad (5)$$

The final constraint of our MILP brings in the nonnegative real variable s to represent the traffic reduction that happens because clients receive segments in lower representations than the requested ones:

$$s \times \sum_{i \in B} r_i \leq \sum_{i \in B} \sum_{j \in B \,\&\, j<i} r_i \times y_{j,i} \times (b_i - b_j). \quad (6)$$

For the sake of normalization, we introduce Q and S as bounds on the possible values of quality improvement q and traffic reduction s, respectively, and define the following Multi-Objective Optimization (MOO) function:

$$\text{MOO} = \alpha \times \frac{q}{Q} + (1-\alpha) \times \frac{s}{S}. \quad (7)$$

Our MILP strives to maximize the MOO function where weight α controls the relative priorities of the quality improvement vs. traffic reduction. The two individual objectives are in conflict: increasing q decreases s and vice versa. Note that $\alpha = 1$ emphasizes increasing the video quality, which conforms to the OTL selecting its bitrates as close as possible to the requested bitrates. On the other hand, $\alpha = 0$ places the only focus on minimizing the traffic, which corresponds to serving all requests with the lowest bitrate. We detail our algorithm for determining the value of α in Section 4.4. To summarize, ARTEMIS relies on the following MILP:

Maximize: MOO in Equation 7 (8)

subject to: Constraints 1 through 6 (9)

variables: $x_i,\ y_{j,i} \in \{0,1\},\ q \leq 0,$ and $s \geq 0.$ (10)

The computational complexity of the formulated MILP depends on the number of representations in the mega-manifest, and not on the number of clients in the session. The total number of variables equals $m + \frac{m(m-1)}{2} + 2$ because each bitrate b_i contributes $i-1$ binary variables $y_{j,i}$ and one binary variable x_i, with two real variables q and s completing the total. The number of individual constraints behind Constraints 1 through 6 amounts to $2m+4$. Despite being an NP-complete problem [30], the MILP is applicable for optimizing the OTL in live streaming due to the relatively low numbers of variables and constraints. We solve the MILP by using the PuLP library [14]. Figure 5b reports the respective execution time as a function of m, the number of representations in the mega-manifest.

4.4 Algorithm Details

Input processing. Algorithm 1 constitutes the core algorithm of ARTEMIS. Lines 3-6 of the pseudocode describe the operation during the CI interval when the AA server retrieves the latest version of quality indicator function F and client-state log files from the AO agent and CDN, respectively. After extracting the request counts and stall information from the CDN logs, the AA server stores the data as list $R = \{r_1, r_2, \ldots, r_m\}$ and set T that consists of tuples (pid, t_s, t_e), where pid denotes the unique player identifier included in the URLs of HTTP requests according to the CMCD specification, and t_s and t_e represent the start and end times of a stall event. A player that experiences a stall event sends the respective (pid, t_s, t_e) tuple to the CDN and requests to buffer all encoded segments that reach the CDN edge server. The time-slot duration divided by the segment duration imposes the upper limit on the number of requests that the player contributes to the R list during the time slot. For each of the players that request segments and do not report any stalls, set T includes tuple $(pid, 0, 0)$.

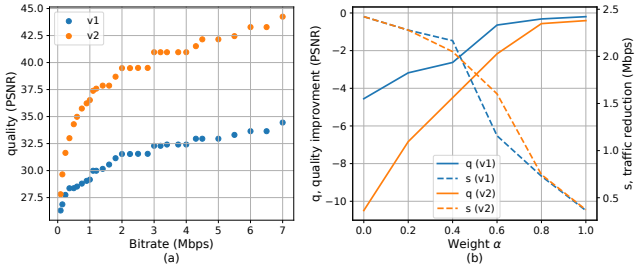


Figure 6: (a) Rate-distortion curves $v1$ and $v2$ with 30 bitrates between 0.09 Mbps and 7 Mbps and (b) impact of the α value on quality improvement and traffic reduction.

Algorithm for determining α. Before answering the question how to determine the best α value, we use Figure 6 to illustrate the impact of α on quality improvement q and traffic reduction s, *i.e.*, the two individual objectives of our MILP. Figure 6a depicts rate-distortion curves $v1$ and $v2$ associated with two different content types and $m = 30$ bitrates. For each bitrate b_i, we randomly pick a value between 50 and 100 for request count r_i and configure $\bar{x}_i$ to 0. The example also assumes $\ell = \beta = 8$. Figure 6b reveals the opposite impact of changes in α on q and s. When α equals 0, the MILP produces OTLs that offer the worst video quality, with q being about -4.2 and -10.2 for curves $v1$ and $v2$, respectively. However, the traffic reduction in this α setting is the largest, with s reaching around 2.4 Mbps for both curves. Increasing the α value improves quality and increases traffic. When α becomes 1, the MILP-produced OTLs support video quality within 0.2% and 3% of the PSNR values with the requested bitrates, and traffic reduction s drops to 0.35 Mbps for both curves $v1$ and $v2$.

Because Figure 6b shows that α significantly affects both video quality and traffic volume, ARTEMIS adjusts the α value based on the origin-side and client-side information. For instance, larger traffic with a greater α value might cause not only improved video quality but also increased stall duration, which the AA server detects by observing the increased average stall duration in the client-side feedback provided via the CDN. In adjusting the α value based on the stall information, ARTEMIS avoids dramatic changes over consecutive OT intervals so that the resulting updates in the

Algorithm 1 ARTEMIS Algorithm

```
1: for each time slot do
2:     R ←[], T ←[], F ← ∅, O* ← ∅
3:     while in CI do                ▷ CI interval starts
4:         T,R ← ProcessCDNlogs()
5:         F ← QualityFunction()
6:     end while                     ▷ OT interval starts
7:     α, f1 ←StallAnalysis(T) ▷ Algorithm 2
8:     O,q ← Optimization(α, O*,R,F)
9:     if f1 then
10:        SendOTLtoAOagent(O)
11:        O* ← O
12:    else
13:        f2 ←QualityAnalysis(q,F) ▷ Alg. 3
14:        if f2 then
15:            SendOTLtoAOagent(O)
16:            O* ← O
17:        end if
18:    end if
19: end for
```

Algorithm 2 StallAnalysis Function

```
1: Inputs: LastStall, StallAlpha
2: function StallAnalysis(T)
3:     l* ← mean(T)
4:     α ←SelectAlpha(StallAlpha,l*)
5:     l ←LastStall
6:     if l == 0 then
7:         t ← min(1,l*)
8:     else
9:         t ← min(1, (l*−l)/l)
10:    end if
11:    LastStall← l*
12:    p ←GenerateRandom(uniform[0,1))
13:    if p ≤ t then
14:        f ← True
15:    else
16:        f ← False
17:    end if
18:    return α, f
19: end function
```

Algorithm 3 QualityAnalysis Function

```
1: Inputs: O*
2: function QualityAnalysis(q,F)
3:     d ←[]
4:     for i ∈ B do
5:         d.append(DiffQuality(b_i,r_i,a_i,F,i))
6:     end for
7:     q* ← mean(d)
8:     if q == 0 then
9:         t ← min(1,q*)
10:    else
11:        t ← min(1, (q*−q)/q)
12:    end if
13:    p ←GenerateRandom(uniform[0,1))
14:    if p ≤ t then
15:        return True
16:    else
17:        return False
18:    end if
19: end function
```

OTL do not degrade QoE for the clients, *e.g.*, due to frequent bitrate switching.

Algorithm 2 presents pseudocode of the *StallAnalysis*() function that determines the α value. The function seeks an appropriate balance between high video quality and low stall duration. In addition to set T, the function takes *LastStall* and *StallAlpha* as two other local inputs. *LastStall* refers to the average stall duration calculated during the previous time slot. *StallAlpha* is a dictionary that specifies the α value for each range of stall duration. For example, *StallAlpha*=`{(0,2):1.0,(2,'inf'):0.8}` means $\alpha = 1.0$ for the average stall duration between 0 and 2 s and $\alpha = 0.8$ for the average stall duration exceeding 2 s. Supplied by the streaming service provider, the *StallAlpha* dictionary is updatable during the streaming session.

Line 3 of Algorithm 2 computes l^* as the average stall duration in set T. Line 4 configures α according to the *StallAlpha* dictionary. With variable l initialized to *LastStall*, Lines 6-10 adjust threshold t according to the difference between l^* and l. Line 12 draws random number p uniformly between 0 and 1. If p is at most threshold t, *i.e.*, the stalling is significantly higher than in the previous time slot, Line 14 sets flag f to *True*, indicating the need for a new OTL to decrease the stalling. Otherwise, Line 16 sets flag f to *False*. Line 18 returns the α and f values. The core algorithm of ARTEMIS obtains these values as α and f_1, respectively, by calling the *StallAnalysis*() function in Line 7 of Algorithm 1.

OTL computation. Based on the determined α value, previous OTL O^*, list R of request counts, and quality indicator function F, Line 8 of Algorithm 1 computes new OTL O as described in Section 4.3. Solving the MILP also returns quality improvement q that satisfies Constraint 5.

Communication of the OTL to the AO agent. If flag f_1 is set to *True*, Line 10 of Algorithm 1 sends the new OTL from the AA server to the AO agent. Otherwise, with flag f_1 set to *False* and indicating insignificant stalling, Line 13 calls a *QualityAnalysis*() function to explore whether quality improvement justifies adopting the new OTL.

Algorithm 3 shows the pseudocode of the *QualityAnalysis*() function, with previous OTL O^* as an extra input. Lines 4-6 of Algorithm 3 measure how much the requested bitrates would improve video quality compared to the bitrates in the previous OTL. While r_i refers to the number of requests for bitrate b_i, input a_i in Line 5 indicates how many requests the previous OTL accommodates with bitrate b_i. Lines 7-12 introduce threshold t that captures the difference between q^* and q, which refer to the quality improvement for the previous and new OTL, respectively. Line 14 compares the threshold with random number p drawn uniformly between 0 and 1. The *QualityAnalysis*() function returns a *True* flag if p is at most t. Otherwise, Algorithm 3 returns a *False* flag.

When Line 13 of Algorithm 1 receives a *True* flag from the *QualityAnalysis*() function, Line 15 of the core algorithm sends the new OTL to the AO agent. With this flag f_2 set to *False*, ARTEMIS continues operation with the previous OTL. Hence, in deciding whether to update the OTL, ARTEMIS considers not only the *StallAlpha* dictionary but also video quality and stalling information across two consecutive time slots.

5 Performance Evaluation

This section describes our practical implementation of ARTEMIS as well as the evaluation methodology and results of our real-world experiments. The evaluation intends to answer the following questions: (*i*) What is the impact of ARTEMIS on QoE? How do the constructed dynamic bitrate ladders fare against existing fixed-length and content-based ladders? (*ii*) What is the influence of ARTEMIS on resource

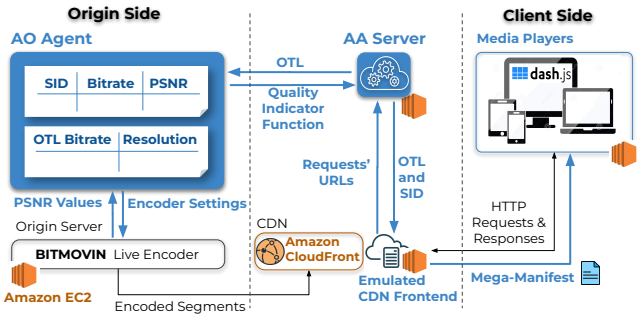


Figure 7: Cloud-based implementation of ARTEMIS.

consumption? (*iii*) How do different real-world network traces affect the ARTEMIS performance? (*iv*) How sensitive is the ARTEMIS performance to changes in the number of players, segment duration, content type, end-to-end latency, and α value?

5.1 ARTEMIS Implementation

Our real-world experimentation combines existing infrastructure with our own cloud-based implementations of the ARTEMIS functionalities. The existing infrastructure includes Amazon CloudFront as the CDN, Bitmovin Live Encoder [5] as the origin server, and CAdViSE [57] for hosting media players. Both Bitmovin Live Encoder and CAdViSE operate as EC2 implementations. We implement the ARTEMIS functionalities in EC2 as well. Figure 7 depicts our implementations and their interactions with the existing infrastructure.

Media players. CAdViSE is a cloud-based platform for automated testing of media players. We use CAdViSE to run multiple players on different kinds of EC2 machines. Each EC2 instance fetches from Docker Hub [9] a Docker container holding the DASH JavaScript Player [27]. We modify the player to send the *pid* and stall information in the CMCD format as a query argument in the URL of every HTTP request for a segment.

Emulated CDN frontend. To support the mega-manifest feature of ARTEMIS, we emulate a CDN frontend by implementing it in Python and running the frontend on an EC2 machine. The emulated CDN frontend faces the players and receives their HTTP requests. Also, the AA server provides the CDN frontend with an OTL and Segment ID (SID) to indicate the validity of the given OTL for segments with numerical IDs that are equal or greater than this SID. Before forwarding each HTTP request for a segment to the CDN, the CDN frontend checks whether the requested representation appears in the OTL associated with this segment. If the representation ID is not in the OTL, the CDN frontend changes the representation ID in the HTTP request to the ID of the representation with the closest lower bitrate in the OTL. The emulated CDN frontend also provides the AA server with CDN logs in the form of requests' URLs.

AA server. We implement the AA server in Python and, specifically, employ the Pulp library [14] to solve the MILP. Apart from determining the OTL and communicating it to the CDN frontend and AO agent, the AA server also coordinates the time-slotted operation of ARTEMIS. The two main threads of the AA server utilize Transmission Control Protocol (TCP) sockets to receive the requests' URLs and quality indicator function F from the CDN frontend and AO agent, respectively.

AO agent. Our Python implementation of the AO agent runs in EC2 on the same machine with the Bitmovin Live Encoder. The encoder uses DASH packaging, computes PSNR values during the encoding process, and records a (SID, bitrate, PSNR) tuple for each encoded segment. The computational burden of extracting the PSNR values during the encoding process is negligible [62]. The main computational task of the AO agent is to train quality indicator function F by means of linear regression. Figure 5a shows that the execution time to train function F is low, below 20 ms, and largely independent from the number of segments. In addition, based on the OTL received from the AA server, the AO agent updates the encoder settings of the Bitmovin Live Encoder. The AO agent also stores the bitrates of the previous OTL along with their resolutions. Our experiments employ predetermined resolutions for all bitrates.

5.2 Evaluation Methodology

Bitrate ladders. We consider two groups of baselines for evaluating the dynamic bitrate ladders constructed by ARTEMIS. The first group consists of common fixed-length ladders used by conventional HAS solutions for live streaming that do not account for the content type or network conditions. Table 1 presents the Theo [16], Bitmovin [4], Mux [11], Pensieve [48], and Twitch [17] ladders that comprise this first group. The second group consists of three bitrate ladders designed for particular content types and/or specific average available bandwidth. Table 3 in Appendix D describes these three ladders. Designed offline for VOD scenarios, the ILP ladder [60] targets animation videos with the average available bandwidth of the clients being around 3 Mbps. The other two baselines in the second group are the Netflix ladders for animation [12] and movie [13] as the content type and do not account for the available network bandwidth.

Network traces. Our experiments use the LTE [61], AmazonFCC [15], Cascade-5, and Cascade-20 traces. Figure 15 in Appendix D depicts the four traces for 500 s, *i.e.*, the total duration of each streaming session. Cascade-5 and Cascade-20 are synthetic network traces generated using five distinct bandwidth values of 0.5, 1, 2, 4, and 7 Mbps, where labels 5 and 20 indicate the the available bandwidth remains constant for 5 and 20 s, respectively. To allocate a network trace to each player, we convert a longer network trace into circular arrays. Then, we generate a random number uniformly distributed

Table 1: Representations advertised by the mega-manifest of ARTEMIS vs. the five static bitrate ladders.

BL \ Res.@Mbps	240p@0.145	240p@0.240	360p@0.365	360p@0.5	360p@0.6	360p@.75	360p@0.9	540p@1.0	480p@1.1	480p@1.2	480p@1.4	720p@1.6	720p@1.8	720p@2.0	720p@2.25	720p@2.5	1080p@2.8	720p@3.0	720p@3.2	720p@3.4	720p@3.75	1080p@4.0	1080p@4.3	1080p@4.5	1080p@5.0	1080p@5.5	1080p@6.0	1080p@6.5	1080p@7.0
Theo			✓					✓								✓						✓							
Bitmovin	✓		✓						✓					✓										✓					
Mux						✓		✓								✓								✓					
Pensieve			✓			✓				✓			✓				✓						✓						
Twitch				✓					✓			✓							✓		✓								✓
ARTEMIS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

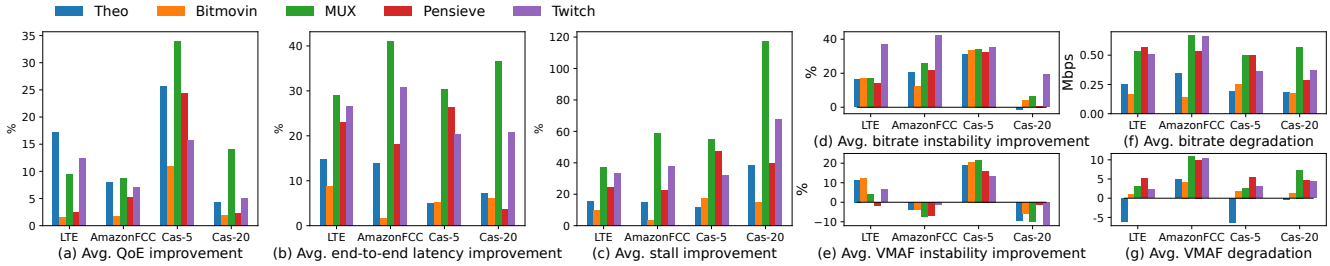


Figure 8: QoE, latency, stall, bitrate, and VMAF performance of ARTEMIS' dynamic ladders vs. the five static ladders.

between 0 and 500. This random number determines the time at which the allocated network trace starts for the specific player. Thereby, we have a distinct network trace for each player.

Content types and encoding parameters. Animation, sport, movie, and documentary are four distinct content types in our experiments. The Bitmovin Live Encoder encodes segments at a constant bitrate with segment duration of 1, 2, and 4 s and extracts the PSNR values of the segments during the encoding process.

Default ARTEMIS parameters. By default, we set maximum OTL length ℓ to 5 representations, dictionary *StallAlpha* to $\{1:[0,1], 0.9:[1,2], 0.8:[2,3], 0.7:[3,4], 0.6:[4,5], 0.5:[5,100]\}$, time-slot duration θ to 10 s, segment duration to 2 s, and target end-to-end latency to 4 s. The default number of players, ABR algorithm, and content type are 50 players, L2A, and animation, respectively.

Experimental scenarios. We experiment in two scenarios. Using the default parameter settings, Scenarios I and II evaluate ARTEMIS' dynamic ladders against the bitrate ladders from the first and second baseline groups, respectively. We also analyze performance sensitivity to changes in the experimental settings, including the ABR algorithm, content type, ARTEMIS parameters α, ℓ, and θ, and the number of players. While individual experiments in our preliminary investigation substantiate the potential of ARTEMIS in sessions with as many as 15,000 players, the evaluation in this paper accomplishes its multifaceted agenda in the settings where each session serves 10, 20, 50, or 100 players. For each experiment, we average the results over five runs.

Evaluation metrics. We evaluate QoE via the QoE model of Appendix A and also zoom in on instability of the

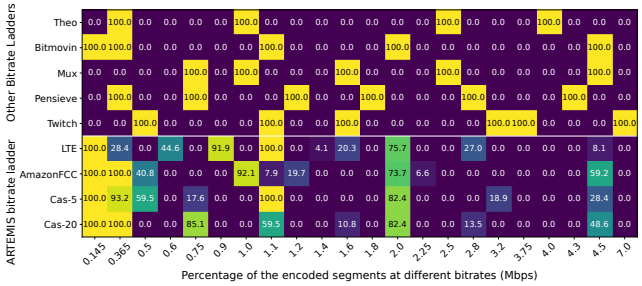


Figure 9: Bitrates used by the five static ladders and ARTEMIS' dynamic ladders on different network traces.

bitrate and VMAF. For each of these two underlying metrics, we calculate its instability as the average of the absolute differences between their consecutive values throughout a sequence of segments. Besides, we calculate ladder efficiency and resource cost. To compute ladder efficiency, we divide the average served bitrate of the clients by the average encoded bitrate of the ladder. Instead of directly measuring computation and bandwidth costs, we report encoding time and volume of traffic from the live encoder to the CDN and from the CDN to the player. We choose these metrics because the computation and bandwidth costs are proportional to the processing time and traffic volume.

5.3 Results and Analysis

Scenario I. Our evaluation of ARTEMIS starts by comparing its dynamic bitrate ladders with the five fixed-length ladders. Figures 8a-8e and, in more detail, Table 4 in Appendix E show the average relative improvement in QoE, end-to-end latency,

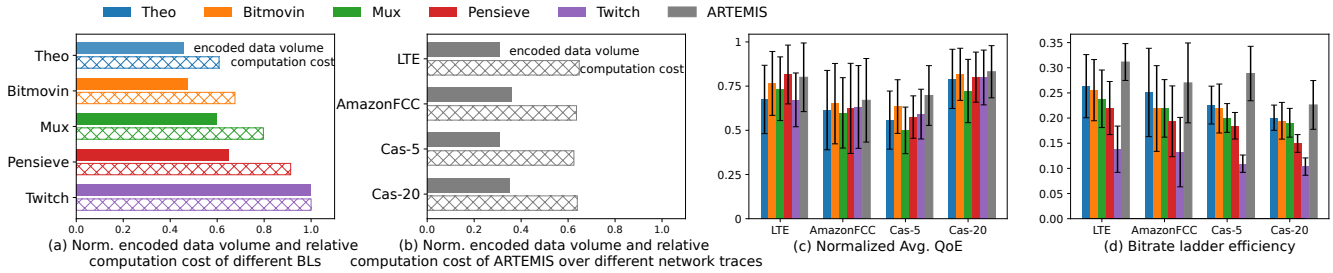


Figure 10: Computation cost, traffic volume, and ladder efficiency with ARTEMIS vs. the five static ladders.

stall duration, bitrate and VMAF instability for ARTEMIS vs. the baselines. The results demonstrate that ARTEMIS consistently improves QoE compared to the baselines, with the average QoE improvement ranging from 4% to 16.5% with respect to the Bitmovin and Mux ladders, respectively. Figure 8a shows that ARTEMIS yields the largest average QoE improvement in comparison to the Mux ladder because ARTEMIS mitigates stalls for the players by selecting lower bitrates than 0.75 Mbps, which is the lowest bitrate in the Mux ladder. Although the Bitmovin ladder includes the lowest bitrate of 0.145 Mbps, ARTEMIS outperforms this ladder as well, *e.g.*, by reducing the stall duration by 17.3% and improving QoE by 10.9% on the Cascade-5 network trace. Similarly, Figure 8b reveals that ARTEMIS outperforms the baselines by reducing the end-to-end latency. The latency reduction occurs due to shorter encoding time arising from the more effective ladder selection in ARTEMIS.

Figures 8d and 8e demonstrate that ARTEMIS decreases the bitrate instability compared to the baseline ladders, particularly on the LTE, AmazonFCC, and Cascade-5 network traces. While the VMAF instability gets worse with ARTEMIS on the AmazonFCC and Cascade-20 traces, ARTEMIS still improves the overall QoE on these traces. Furthermore, the observed increase in the VMAF instability lies within the JND, meaning that humans do not perceive the quality decrease.

Figures 8f and 8g offer further insights into the impact of ARTEMIS on the decrease in the average bitrate and VMAF compared to the baseline ladders. Although ARTEMIS serves the players with lower bitrates, the decrease in VMAF is negligible for some combinations of ladders and network traces, and ARTEMIS improves the VMAF value in other cases. ARTEMIS attains this performance by using PSNR as a proxy of VMAF in predicting the video quality of the subsequent segments. Figure 8g shows that ARTEMIS provides higher VMAF compared to the Theo ladder on the LTE and Cascade-20 network traces. The Theo ladder yields the lower VMAF values due to employing only four representations with the largest bitrate of 4 Mbps only, whereas ARTEMIS includes higher bitrates into its dynamic ladders and thereby improves the video quality.

Figure 9 reports on the bitrates used by ARTEMIS' dynamic ladders vs. the static baselines. While the static ladders always employ the same bitrates regardless of the network trace, ARTEMIS adapts its ladders to network conditions. Only the lowest mega-manifest bitrate of 0.145 Mbps always appears in ARTEMIS' OTL. The second lowest bitrate of 0.365 Mbps is always present in the OTL for the AmazonFCC and Cascade-20 network traces only. The bitrate of 1.1 Mbps is consistently in demand on the LTE and Cascade-5 traces. For all four traces, ARTEMIS includes the bitrate of 2 Mbps in the OTL most of the time but not always. Finally, ARTEMIS never puts into the OTL the largest mega-manifest bitrate of 7 Mbps on all four traces and occasionally employs the second largest bitrate of 4.5 Mbps in the OTL. Overall, Figure 9 corroborates that ARTEMIS effectively adjusts its ladders for the current network conditions.

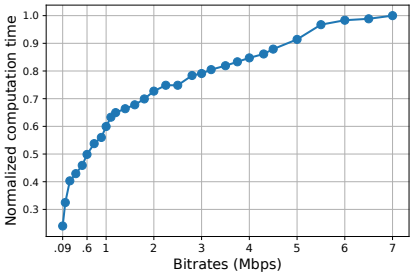


Figure 11: Normalized average computation time for encoding each bitrate in ARTEMIS.

Figure 10 evaluates ARTEMIS vs. the baselines in regard to computation cost, traffic volume, and ladder efficiency. While Figures 10a and 10b focus on the computation cost, we also investigate the relationship between the computation cost and processing time by measuring the time to encode each bitrate to the ultrafast preset on an EC2 machine and by calculating the average processing time to encode each segment according to the selected OTL. Figure 11 shows the normalized average computation time for encoding each bitrate. The findings corroborate earlier research conclusions that the computation cost is proportional to the processing time [29]. Figures 10a and 10b demonstrate that, by accounting for the network conditions, ARTEMIS' dynamic ladders provide lower computation cost than the baseline ladders, except for the Theo ladder that comprises

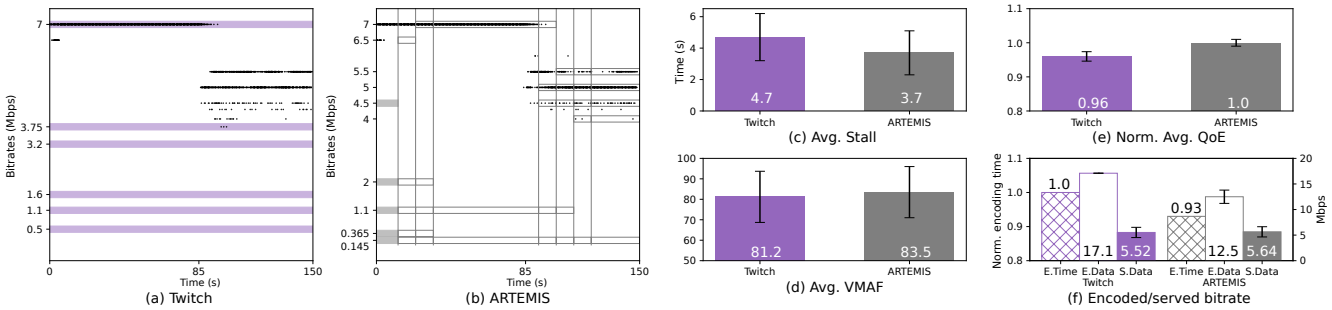


Figure 12: Requested and served bitrates, stall duration, VMAF, QoE, and encoding time with ARTEMIS' dynamic ladder vs. the static Twitch ladder when both approaches advertise the mega-manifest bitrates to the players.

only four representations. Despite limiting the maximum ladder length to five representations, ARTEMIS requires less computation than the Bitmovin ladder due to selecting a lower bitrate to cope with fluctuations in the available bandwidth. Additionally, ARTEMIS selects a smaller ladder when the bitrate preferences of the clients indicate a need in less than the maximum ℓ representations.

Figure 10c demonstrates that ARTEMIS' dynamic ladders increase the normalized average QoE compared to almost all baselines across the four network traces. This finding highlights the effectiveness of ARTEMIS even under challenging network conditions. While the Pensieve ladder on the LTE network trace delivers nearly the same QoE as ARTEMIS' dynamic ladders, the computation cost incurred by the Pensieve ladder is 44% higher than with ARTEMIS. Figure 10d shows that ARTEMIS substantially outperforms the static baselines with respect to ladder efficiency across all four network traces within a time slot of at most θ.

The network traces in our study exhibit high bandwidth fluctuations resulting in frequent stalls for the players. Whereas Figure 9 unveils that ARTEMIS reacts to the highly variable bandwidth by limiting the highest bitrate in the OTL to 4.5 Mbps, we conduct an additional experiment on a network trace with minimal bandwidth fluctuations. Specifically, the available bandwidth for each player is 10 Mbps initially, decreases to 6.5 Mbps after 85 s, and remains at this level for the subsequent 65 s. The experiment evaluates ARTEMIS' dynamic ladder against the static Twitch ladder, which is the only baseline that includes the largest considered bitrate of 7 Mbps. Additionally, we expose the mega-manifest bitrate preferences of the players not only to ARTEMIS but also to the Twitch alternative. In this setup, we anticipate ARTEMIS to serve the players with the highest available bitrate until 85 s into the experiment.

For this additional experiment, Figures 12a and 12b depict the bitrates requested by the players as black dots and the bitrates available for serving the requests as horizontal purple lines. The Twitch alternative supports the bitrate of 7 Mbps and affirmatively responds to all requests for this highest bitrate in the mega-manifest until 85 s into the experiment. After 85 s however, the players request bitrates that are below 7 Mbps and above 3.75 Mbps, which is the second highest bitrate in the Twitch ladder. Hence, the Twitch alternative serves the requests with the bitrate of 3.75 Mbps, which degrades the video quality by underutilizing the available bandwidth. On the other hand, Figure 12b shows that ARTEMIS efficiently selects the OTL to serve the requests with higher bitrates. The gray vertical lines indicate the times when ARTEMIS changes its OTL. During the first 10 s that comprise the first slot of the time-slotted operation, ARTEMIS uses the OTL predefined by the streaming service provider and depicted as gray rectangles in Figure 12b. Upon receiving requests for the bitrates of 6.5 and 7 Mbps during the first time slot, ARTEMIS updates the OTL and, due to low stalling and frequent requests for the bitrate of 7 Mbps, converges by the third time slot to serving the players with the bitrate of 7 Mbps until time 85 s. During the five consequent time slots, ARTEMIS instructs the live encoder to use the OTL with only three bitrates of 0.145, 1.1, and 7 Mbps, which efficiently reduces the computation and bandwidth costs. However, when the clients request lower bitrates at time 85 s, ARTEMIS reacts quickly, and Figure 12b shows that ARTEMIS determines the four bitrates that closely match all bitrates requested by the players.

Figures 12c-12f compare ARTEMIS with the Twitch alternative in regard to the average stall duration, VMAF, and normalized QoE as well as encoding time (labeled as E. Time), encoded bitrate (E. Data) and served bitrate (S. Data). The results show that ARTEMIS' dynamic ladders perform better than the static Twitch ladder in all metrics while reducing the traffic volume by 0.3%.

We further extend our exploration of Scenario I by evaluating QoE in Figure 16 of Appendix E, LoL$^+$ as an alternative low-latency ABR algorithm in Figure 17 of Appendix F, and influence of the content type in Figures 18 and 19 of Appendix G. We also study how the ARTEMIS performance depends on weight α in Figure 20 of Appendix H, maximum OTL length ℓ in Figures 21 and 22 of Appendix I, and time-slot duration θ in Figures 23 and 24 of Appendix J.

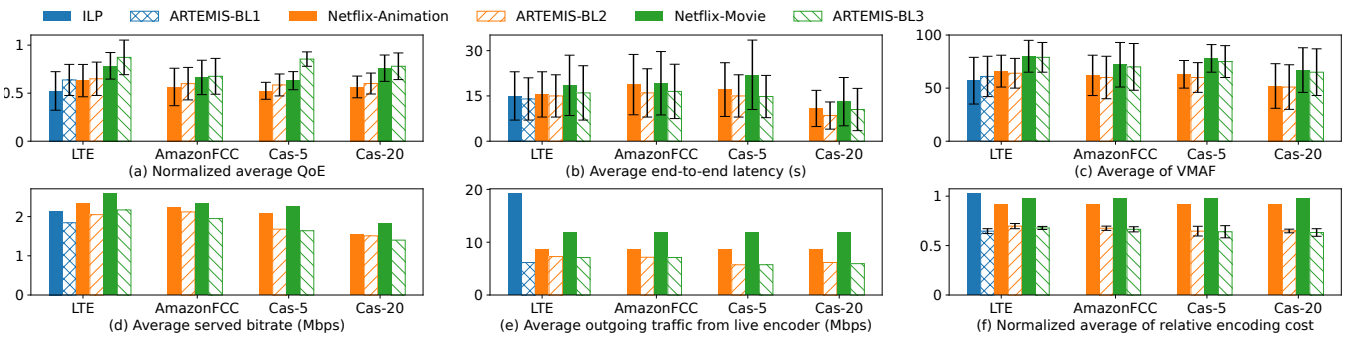


Figure 13: QoE, end-to-end latency, VMAF, served bitrates, traffic volume, and encoding cost for ARTEMIS' customized BL1, BL2, and BL3 ladders vs. the ILP, Netflix-Animation, and Netflix-Movie ladders.

Scenario II. We also assess ARTEMIS' dynamic ladders in comparison to the three fixed-length ladders described in Table 3 of Appendix D. These baselines aim at particular content types and/or available network bandwidth. Because the average bandwidth in the LTE network trace is nearly 3 Mbps, *i.e.*, as targeted by the ILP ladder, we evaluate ARTEMIS against the ILP ladder specifically on the LTE trace. However, we consider all available network traces for the two Netflix ladders. To cover the representations in the considered baselines, we customize ARTEMIS' mega-manifest by introducing additional representations into it. For example, the comparison with the ILP ladder customizes ARTEMIS' mega-manifest by including all representations of the ILP ladder and the additional seven representations described in Table 3.

Figure 13, along with Table 5 in Appendix K, shows that ARTEMIS consistently outperforms the baselines across all network traces in terms of QoE and end-to-end latency by selecting the maximum five representations. ARTEMIS also exhibits better performance in terms of stall duration, bitrate instability, and VMAF instability. While Figure 13c reveals a negligible decrease in VMAF, Figure 13d demonstrates that ARTEMIS effectively serves the players with lower bitrates, thereby reducing the traffic from the CDN to the players. Figure 13e illustrates that ARTEMIS decreases the volume of encoded data, reducing the consumption of computational resources in the live encoder and also lowering the traffic from the encoder to the CDN.

6 Conclusion

This paper presents ARTEMIS, a practical scalable system for efficient dynamic construction of effective bitrate ladders during a live session of video streaming. ARTEMIS seamlessly enhances the end-to-end HAS pipeline and innovatively leverages its standard features, such as the advertisement of representations via the mega-manifest to collect fine-grained client-side information transparently to heterogeneous ABR algorithms. The context-aware system is also efficient in its capture of the content complexity via PSNR. We provide a cloud-based implementation of ARTEMIS and extensively evaluate it in real-world and trace-driven experiments.

Whereas ARTEMIS pursues multifaceted goals that include low end-to-end latency, high QoE, and low encoding computation, the evaluation against prominent static bitrate ladders shows that ARTEMIS successfully delivers multi-objective improvements to reduce end-to-end latency by 18%, increase QoE by 11%, and decrease encoding computation by 25%. The deeper analysis of ARTEMIS' success reveals that its accounting for dynamic network conditions is crucial. When the available network bandwidth fluctuates significantly to increase the likelihood of stall events, ARTEMIS prioritizes lower bitrates in the ladder construction while maintaining QoE close to the maximum achievable under the poor network conditions. By scaling down the bitrate ladder, ARTEMIS also reduces encoding computation, decreases storage and bandwidth costs (especially at the network edge), and thereby enables the streaming service to support more clients. Under stable network conditions, ARTEMIS also lowers the resource consumption by composing a ladder with a smaller number of representations and improves QoE through a better alignment of the representations with the bitrate preferences of the clients.

Acknowledgments

We thank the anonymous NSDI 2024 reviewers and our shepherd Dave Oran for their invaluable feedback and guidance. The research is supported in part by the Austrian Federal Ministry for Digital and Economic Affairs, National Foundation for Research, Technology and Development, and Christian Doppler Research Association with project Christian Doppler Laboratory ATHENA (https://athena.itec.aau.at/) and the Spanish Ministry of Science and Innovation with grants TED2021-131264B-I00 (SocialProbing) and PID2021-128223OA-I00 (GreenEdge).

References

[1] Akamai, Content Delivery Network (CDN). [Online] Available: https://www.akamai.com/solutions/content-delivery-network.

[2] Amazon Elastic Compute Cloud (Amazon EC2). [Online] Available: https://aws.amazon.com/ec2/.

[3] Bitmovin and Akamai to Debut Joint CMCD Solution at NAB Show 2023. [Online] Available: https://bitmovin.com/press-room/akamai-bitmovin-create-joint-cmcd-solution/.

[4] Bitmovin Dashboard, Live Encoder. [Online] Available: https://bitmovin.com/dashboard/live.

[5] Bitmovin Live Encoder. [Online] Available: https://bitmovin.com/live-encoding-live-streaming/.

[6] Conviva. [Online] Available: https://www.conviva.com/technology/.

[7] CTA-5004: Web Application Video Ecosystem – Common Media Client Data. [Online] Available: https://cdn.cta.tech/cta/media/media/resources/standards/pdfs/cta-5004-final.pdf.

[8] Datazoom, Streaming Video Monitoring. [Online] Available: https://www.datazoom.io/streaming-video-monitoring/.

[9] Docker Hub. [Online] Available: https://hub.docker.com/.

[10] FFmpeg. [Online] Available: https://https://ffmpeg.org/.

[11] Mux Video. [Online] Available: https://docs.mux.com/guides/video/configure-broadcast-software#good---720p-30fps.

[12] Netflix Animation Bitrate Ladder. [Online] Available: https://ottverse.com/creating-the-perfect-encoding-ladder/,.

[13] Netflix Movie Bitrate Ladder. [Online] Available: https://ottverse.com/creating-the-perfect-encoding-ladder/,.

[14] Optimization with PuLP. [Online] Available: https://coin-or.github.io/pulp/.

[15] Raw Data — Measuring Broadband America. [Online] Available: https://www.fcc.gov/reportsresearch/reports.

[16] Theo Player. [Online] Available: https://www.theoplayer.com/blog/encoding-cost-efficient-streaming.

[17] Twitch. [Online] Available: https://stream.twitch.tv/encoding/.

[18] Hadi Amirpour, Raimund Schatz, and Christian Timmerer. Between Two and Six? Towards Correct Estimation of JND Step Sizes for VMAF-Based Bitrate Laddering. In *QoMEX*, pages 1–4, September 2022.

[19] Hadi Amirpour, Christian Timmerer, and Mohammad Ghanbari. PSTR: Per-Title Encoding Using Spatio-Temporal Resolutions. In *IEEE ICME*, pages 1–6, July 2021.

[20] Apple. Enabling Low-Latency HTTP Live Streaming (HLS). [Online] Available: https://developer.apple.com/documentation/http_live_streaming/enabling_low-latency_http_live_streaming_hls, 2021. Accessed on Jan. 26, 2023.

[21] Apple Inc. HTTP Live Streaming (HLS) Authoring Specification for Apple Devices. [Online] Available: http://bit.ly/hls_spec_2017, 2015. Online; accessed on Jan. 05, 2022.

[22] Abdelhak Bentaleb, Mehmet N. Akcay, May Lim, Ali C. Begen, and Roger Zimmermann. Catching the Moment with LoL+ in Twitch-like Low-Latency Live Streaming Platforms. *IEEE Transactions on Multimedia*, 24:2300–2314, 2021.

[23] Abdelhak Bentaleb, May Lim, Mehmet N. Akcay, Ali C. Begen, and Roger Zimmermann. Common Media Client Data (CMCD): Initial Findings. In *NOSSDAV*, pages 25–33, July 2021.

[24] Abdelhak Bentaleb, Bayan Taani, Ali C. Begen, Christian Timmerer, and Roger Zimmermann. A Survey on Bitrate Adaptation Schemes for Streaming Media Over HTTP. *IEEE Communications Surveys & Tutorials*, 21(1):562–585, 2019.

[25] Madhukar Bhat, Jean-Marc Thiesse, and Patrick Le Callet. Combining Video Quality Metrics To Select Perceptually Accurate Resolution In A Wide Quality Range: A Case Study. In *IEEE ICIP*, pages 2164–2168, September 2021.

[26] DASH-IF. Low-Latency Modes for DASH. [Online] Available: https://dashif.org/docs/CR-Low-Latency-Live-r8.pdf, 2020. Accessed on Jan. 26, 2023.

[27] DASH-IF. DASH Reference Player (dash.js). [Online] Available: https://reference.dashif.org/dash.js/, 2021. Online; accessed on Jan. 22, 2022.

[28] Jan De Cock, Zhi Li, Megha Manohara, and Anne Aaron. Complexity-Based Consistent-Quality Encoding in the Cloud. In *IEEE ICIP*, pages 1484–1488, September 2016.

[29] Alireza Erfanian, Hadi Amirpour, Farzad Tashtarian, Christian Timmerer, and Hermann Hellwagner. LwTE: Light-Weight Transcoding at the Edge. *IEEE Access*, 9:112276–112289, 2021.

[30] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. 1990.

[31] Syed Hasan, Sergey Gorinsky, Constantine Dovrolis, and Ramesh K. Sitaraman. Trade-Offs in Optimizing the Cache Deployments of CDNs. In *IEEE INFOCOM*, pages 460–468. IEEE, 2014.

[32] Tianchi Huang, Rui-Xiao Zhang, and Lifeng Sun. Deep Reinforced Bitrate Ladders for Adaptive Video Streaming. In *NOSSDAV*, pages 66–73, 2021.

[33] Tianchi Huang, Chao Zhou, Rui-Xiao Zhang, Chenglei Wu, Xin Yao, and Lifeng Sun. Comyco: Quality-Aware Adaptive Video Streaming via Imitation Learning. In *ACM MM*, pages 429–437, 2019.

[34] Quan Huynh-Thu and Mohammad Ghanbari. Scope of Validity of PSNR in Image/Video Quality Assessment. *Electronics Letters*, 44:800–801(1), June 2008.

[35] ISO/IEC. ISO/IEC 23000-19:2020 Information Technology – Multimedia Application Format (MPEG-A) – Part 19: Common Media Application Format (CMAF) for Segmented Media. [Online] Available: https://www.iso.org/standard/79106.html, 2020. Accessed on Jan. 26, 2023.

[36] ISO/IEC. 2019. Information Technology — Dynamic Adaptive Streaming over HTTP (DASH) — Part 1: Media Presentation Description and Segment Formats. International standard 23009-1:2019, International Organization for Standardization, December 2019.

[37] Jan Ozer. Saving on H.264 Encoding and Streaming: Deploy Capped CRF. [Online] Available: https://tinyurl.com/CappedCRF, 2018. Online; accessed on Feb. 12, 2022.

[38] Alan B. Johnston and Daniel C. Burnett. *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web*. Digital Codex LLC, 2012.

[39] Theo Karagkioules, Rufael Mekuria, Dirk Griffioen, and Arjen Wagenaar. Online Learning for Low-Latency Adaptive Streaming. In *ACM MMSys*, pages 315–320, 2020.

[40] Angeliki V. Katsenou, Fan Zhang, Kyle Swanson, Mariana Afonso, Joel Sole, and David R. Bull. VMAF-Based Bitrate Ladder Estimation for Adaptive Streaming. In *PCS*, pages 1–5, June 2021.

[41] Dilip Kumar Krishnappa, Michael Zink, and Ramesh K. Sitaraman. Optimizing the Video Transcoding Workflow in Content Delivery Networks. In *ACM MMSys*, pages 37–48, 2015.

[42] Pierre Lebreton and Kazuhisa Yamagishi. Predicting User Quitting Ratio in Adaptive Bitrate Video Streaming. *IEEE Transactions on Multimedia*, 23:4526–4540, 2021.

[43] Pierre Lebreton and Kazuhisa Yamagishi. Quitting Ratio-Based Bitrate Ladder Selection Mechanism for Adaptive Bitrate Video Streaming. *IEEE Transactions on Multimedia*, pages 1–14, 2023.

[44] Dayoung Lee, Jungwoo Lee, and Minseok Song. Video Quality Adaptation for Limiting Transcoding Energy Consumption in Video Servers. *IEEE Access*, 7:126253–126264, 2019.

[45] Zhi Li, Christos Bampis, Julie Novak, Anne Aaron, Kyle Swanson, Anush Moorthy, and Jan De Cock. VMAF: The Journey Continues. *Netflix Technology Blog*, 25(1), 2018.

[46] Melissa Licciardello, Lukas Humbel, Fabian Rohr, Maximilian Grüner, and Ankit Singla. [Solution] Prepare Your Video for Streaming with Segue. *Journal of Systems Research*, 2(1), 2022.

[47] Megha Manohara, Anush Moorthy, Jan De Cock, Ioannis Katsavounidis, and Anne Aaron. Optimized Shot-Based Encodes: Now Streaming. *The Netflix Tech Blog*, 2018.

[48] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *ACM SIGCOMM*, pages 197–210, 2017.

[49] Vignesh V. Menon, Hadi Amirpour, Mohammad Ghanbari, and Christian Timmerer. ETPS: Efficient Two-Pass Encoding Scheme for Adaptive Live Streaming. In *IEEE ICIP*, pages 1516–1520, October 2022.

[50] Vignesh V. Menon, Hadi Amirpour, Mohammad Ghanbari, and Christian Timmerer. OPTE: Online Per-Title Encoding for Live Video Streaming. In *IEEE ICASSP*, pages 1865–1869, May 2022.

[51] Jan Ozer. Encoding Ladders: What You Need to Know. https://www.wowza.com/blog/encoding-ladders-what-you-need-to-know, 2022. [Online; accessed 27-April-2023].

[52] Roger Pantos and William May. HTTP Live Streaming. RFC 8216, August 2017.

[53] Yuriy A. Reznik, Xiangbo Li, Karl O. Lillevold, Abhijith Jagannath, and Justin Greer. Optimal Multi-Codec Adaptive Bitrate Streaming. In *IEEE ICMEW*, pages 348–353, 2019.

[54] Yuriy A. Reznik, Karl O. Lillevold, Abhijith Jagannath, Justin Greer, and Jon Corley. Optimal Design of Encoding Profiles for ABR Streaming. In *PV Workshop*, pages 43–47, June 2018.

[55] Sandvine Inc. The Global Internet Phenomena Report. [Online] Available: https://www.sandvine.com/phenomena, 2022. Online; accessed on March. 15, 2022.

[56] Babak Taraghi, Minh Nguyen, Hadi Amirpour, and Christian Timmerer. Intense: In-Depth Studies on Stall Events and Quality Switches and Their Impact on the Quality of Experience in HTTP Adaptive Streaming. *IEEE Access*, 9:118087–118098, 2021.

[57] Babak Taraghi, Anatoliy Zabrovskiy, Christian Timmerer, and Hermann Hellwagner. CAdViSE: Cloud-Based Adaptive Video Streaming evaluation Framework for the Automated Testing of Media Players. In *ACM MMSys*, pages 349–352, 2020.

[58] Farzad Tashtarian, Abdelhak Bentaleb, Hadi Amirpour, Babak Taraghi, Christian Timmerer, Hermann Hellwagner, and Roger Zimmermann. LALISA: Adaptive Bitrate Ladder Optimization in HTTP-Based Adaptive Live Streaming. In *IEEE/IFIP NOMS*, 2023.

[59] Farzad Tashtarian, Abdelhak Bentaleb, Alireza Erfanian, Hermann Hellwagner, Christian Timmerer, and Roger Zimmermann. HxL3: Optimized Delivery Architecture for HTTP Low-Latency Live Streaming. *IEEE Transactions on Multimedia*, 25:2585–2600, 2022.

[60] Laura Toni, Ramon Aparicio-Pardo, Karine Pires, Gwendal Simon, Alberto Blanc, and Pascal Frossard. Optimal Selection of Adaptive Streaming Representations. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 11(2s):1–26, February 2015.

[61] Jeroen Van der Hooft, Stefano Petrangeli, Tim Wauters, Rafael Huysegems, Patrice Rondao Alface, Tom Bostoen, and Filip De Turck. HTTP/2-Based Adaptive Streaming of HEVC Video Over 4G/LTE Networks. *IEEE Communications Letters*, 20(11):2177–2180, 2016.

[62] Abhinau K. Venkataramanan, Cosmin Stejerean, and Alan C. Bovik. FUNQUE: Fusion of Unified Quality Evaluators. In *IEEE ICIP*, pages 2147–2151, October 2022.

[63] Wowza Media Systems. Video Streaming Latency Report. September 2019, Report, https://www.wowza.com/wp-content/uploads/Streaming-Video-Latency-Report-Interactive-2019.pdf.

[64] Xu Zhang, Yiyang Ou, Siddhartha Sen, and Junchen Jiang. SENSEI: Aligning Video Streaming Quality with Dynamic User Sensitivity. In *NSDI*, pages 303–320, 2021.

[65] Hongcheng Zhong, Jun Xu, Chen Zhu, Donghui Feng, and Li Song. Complexity-Oriented Per-Shot Video Coding Optimization. In *IEEE ICME*, pages 1–6, 2022.

A QoE Model

Our work and, specifically, the evaluation metrics in Section 5.2 adopt the following QoE model to express QoE as a weighted sum of four terms [33]:

$$QoE = \omega_1 \sum_{n=1}^{N} q(R_n) + \omega_2 \sum_{n=1}^{N} T_n + \omega_3 \sum_{n=1}^{N-1} [q(R_{n+1}) - q(R_n)]_+ + \omega_4 \sum_{n=1}^{N-1} [q(R_{n+1}) - q(R_n)]_- \quad (11)$$

where bitrate R_n and stall time T_n characterize each segment n, function $q(R_n)$ computes VMAF of the segment, and the last two terms express VMAF instability over N consecutive segments, with different weights for VMAF increases and decreases. The QoE model uses weights $\omega_1 = 0.8469$, $\omega_2 = -28.7959$, $\omega_3 = 0.2979$, and $\omega_4 = -1.0610$.

B Correlation between PSNR and VMAF

To measure video quality quickly enough for live streaming, the ARTEMIS design in Section 4.2 relies on PSNR rather than VMAF. For different segment durations and content types, Figure 14 demonstrates that PSNR strongly correlates with VMAF.

C Notation in the OTL Selection

Table 2 sums up the notation in Section 4.3 that formulates and solves the MILP to determine a new bitrate ladder.

D Experimental Settings

To elaborate on the setup presented in Section 5.2, Table 3 describes the three static content-aware ILP, Netflix-Animation, and Netflix-Movie bitrate ladders in contrast with three customized ARTEMIS ladders. Figure 15 illustrates bandwidth fluctuations in the LTE, AmazonFCC, Cascade-5, and Cascade-20 network traces used in our experiments.

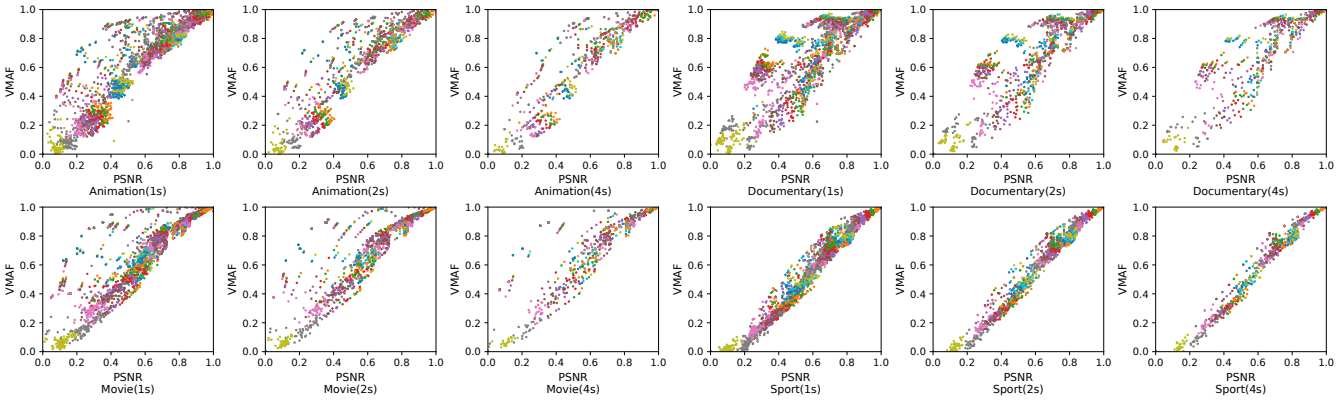


Figure 14: Correlation between PSNR and VMAF.

Table 2: Notation summary.

Symbol	*Description*
	Input parameters
B	Set with indexes of the m representations in the mega-manifest
b_i	Bitrate of representation i
r_i	Number of requests for bitrate b_i
$F(b_i)$	Quality indicator function that predicts the PSNR value of the segment with bitrate b_i
ℓ	Maximum number of representations in the OTL
β	Maximum changes between two successive OTLs
α	Weight in the objective function
$\bar{x}_i$	Indicator whether bitrate b_i appears in the previous OTL
	Variables
x_i	Indicator whether bitrate b_i appears in the the new OTL
$y_{j,i}$	Indicator whether the OTL uses bitrate b_j to accommodate the requests for higher bitrate b_i
q	Quality improvement that the requested bitrates would provide compared to the OTL bitrates
s	Traffic reduction due to using the OTL bitrates instead of the requested bitrates

E Additional Results for Scenario I

To complement the experimental results presented for Scenario I in Figures 8a-8e of Section 5.3, Table 4 depicts the mean and standard deviation for the stall duration, bitrate, and VMAF. The results on the stall duration show that ARTEMIS outperforms the baselines by efficiently updating the OTL. Due to leveraging the client-side information, ARTEMIS succeeds in detecting the stalls and adjusts the OTL by selecting lower bitrates to mitigate the stalls while maintaining an acceptable average VMAF value. Table 4 reveals that ARTEMIS serves the players with lower average bitrates for almost all network traces and numbers of players. However, because ARTEMIS considers the estimated PSNR values of the subsequent segments, the lower served bitrates decrease VMAF insignificantly. The comparison with the Theo ladder on the LTE network trace shows that ARTEMIS even improves the average VMAF in some situations despite decreasing the average served bitrates. Figure 16 evaluates the normalized average QoE with ARTEMIS' dynamic ladders vs. the five static baselines.

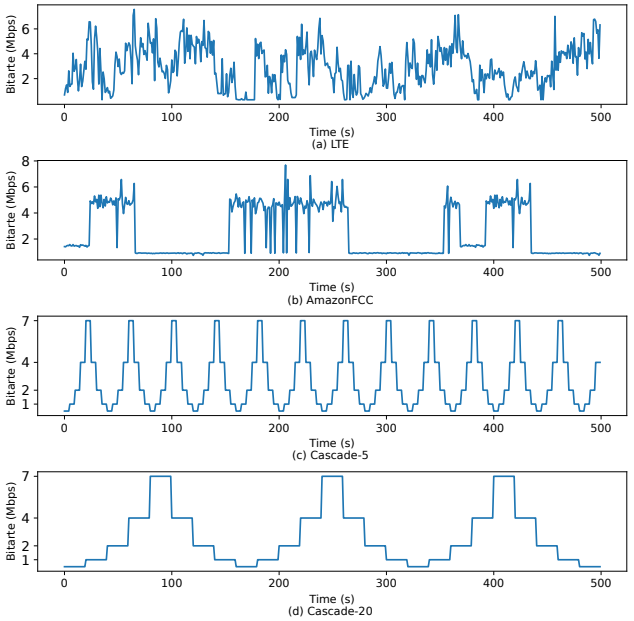


Figure 15: Bandwidth fluctuations in the network traces.

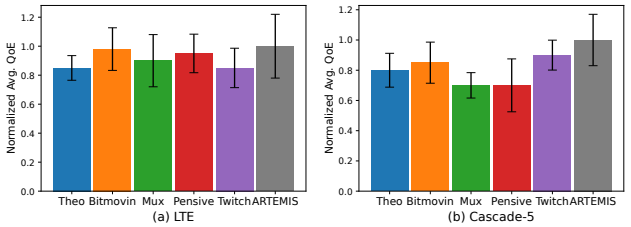


Figure 16: Normalized average QoE with ARTEMIS' dynamic ladders vs. the five static baselines.

Table 3: Representations in the three static content-aware ILP, Netflix-Animation, and Netflix-Movie ladders vs. three customized ARTEMIS ladders.

BL \ Res.@Mbps	224p@0.052	360p@0.082	360p@0.283	720p@0.451	720p@1.0	720p@1.4	720p@1.625	720p@1.8	720p@2.25	720p@3.002	720p@3.4	720p@3.75	720p@4.3	72pp@5.32	1080p@8.34
ILP [60]	✓	✓	✓	✓			✓			✓				✓	✓
ARTEMIS-BL1 [60]	✓	✓	✓	✓	+	+	✓	+	+	✓	+	+	+	✓	✓

	360p@0.2	440p@0.333	640p@0.533	630p@0.725	630p@1.0	810p@1.275	810p@1.4	810p@1.6	1080p@1.8	1080p@2.1	1080p@2.25	1080p@2.8	1080p@3.2	1080p@3.5	
Netflix-Animation [12]	✓	✓	✓	✓		✓				✓				✓	
ARTEMIS-BL2	✓	✓	✓	✓	+	✓	+	+	+	✓	+	+	+	✓	

	315p@0.2	503p@0.4	743p@0.638	743p@0.75	743p@1.088	900p@1.775	900p@2.0	900p@2.25	900p@2.5	1013p@2.95	1013p@3.4	1013p@3.75	1013p@4.3	1080p@4.875	
Netflix-Movie [13]	✓	✓	✓		✓	✓				✓				✓	
ARTEMIS-BL3	✓	✓	✓	+	✓	✓	+	+	+	✓	+	+	+	✓	

Table 4: Impact by the Theo (T), ARTEMIS (A), Bitmovin (B), Mux (M), Pensieve (P), and Twitch (W) bitrate ladders (BLs) on the stall duration, bitrate, and VMAF of served segments on four network traces (NTs).

NT	BL	Stall duration (s)				Bitrate (Mbps)				VMAF			
		#10	#20	#50	#100	#10	#20	#50	#100	#10	#20	#50	#100
LTE	T	37.1±13.9	41.3±17.4	37.5±14.8	36.7±13.1	2.2±0.5	2.3±0.5	2.1±0.5	2.2±0.6	57.0±21.0	58.5±20.5	57.5±20.9	57.9±20.8
	A	36.4±11.6	29.9±10.6	30.6±9.9	33.5±10.4	2.0±0.4	1.7±0.3	1.7±0.2	2.1±0.4	60.0±18.5	60.4±18.5	60.6±18.4	62.5±18.9
	B	36.1±15.9	38.6±14.1	37.1±11.3	34.3±13.4	2.2±0.6	2.1±0.5	2.1±0.5	2.1±0.5	62.6±19.4	61.7±18.7	62.2±18.9	62.3±19.2
	M	44.0±15.4	42.2±14.1	47.3±11.3	43.7±13.4	2.4±0.6	2.3±0.5	2.5±0.6	2.5±0.6	62.5±19.1	62.2±18.3	64.2±17.9	63.4±18.4
	P	45.0±16.6	42.4±14.8	40.5±14.2	39.4±13.0	2.6±0.7	2.5±0.6	2.5±0.6	2.5±0.6	64.8±17.6	64.0±17.7	65.0±17.8	64.8±17.4
	W	41.9±14.6	43.8±17.7	42.1±12.2	43.7±15.7	2.3±0.6	2.4±0.8	2.4±0.8	2.5±0.8	62.1±18.4	63.1±17.5	62.5±17.6	63.2±17.8
AmazonFCC	T	40.7±14.8	44.0±16.1	40.2±15.2	40.8±17.5	2.0±0.7	2.1±0.7	2.0±0.7	2.0±0.8	53.9±22.6	55.1±22.2	55.8±23.2	55.4±22.3
	A	39.0±16.4	35.9±14.4	35.1±15.2	35.0±14.3	1.5±0.6	1.5±0.5	1.7±0.5	1.7±0.6	51.3±22.1	50.9±22.8	53.2±22.0	53.5±22.2
	B	44.1±20.0	39.6±21.8	35.7±19.1	36.1±20.2	1.9±0.8	1.8±0.7	1.8±0.7	1.8±0.7	56.2±22.1	55.1±23.1	54.0±23.5	55.2±22.7
	M	62.0±13.9	63.1±17.9	53.3±18.5	55.2±19.7	2.5±0.7	2.5±0.6	2.3±0.6	2.3±0.7	61.4±20.8	62.4±19.3	54.3±21.4	55.4±20.9
	P	51.3±19.0	46.7±18.2	53.5±17.2	42.2±18.6	2.2±0.7	2.2±0.8	2.2±0.8	2.2±0.7	58.3±21.9	57.7±21.2	57.2±22.1	58.0±21.7
	W	52.1±20.9	50.2±21.4	48.5±19.2	48.9±20.7	2.4±1.2	2.5±1.2	2.3±1.2	2.3±1.1	58.5±20.9	59.1±20.8	57.3±21.0	58.1±21.3
Cascade-5	T	37.1±10.9	36.6±10.8	32.6±8.8	32.0±11	1.8±0.3	1.7±0.4	1.8±0.3	1.7±0.3	52.7±22.5	51.0±22.5	52.1±23.4	51.8±22.4
	A	29.6±8.9	30.7±10.3	32.6±11.2	27.7±10.3	1.5±0.2	1.6±0.2	1.6±0.3	1.5±0.3	56.5±20.2	55.3±20.6	56.7±20.8	54.5±21.7
	B	36.0±10.1	36.3±9.5	35.7±10.5	33.6±12.5	1.8±0.3	1.7±0.3	1.8±0.4	1.7±0.3	57.4±21.9	56.0±22.3	56.4±22.0	56.3±21.6
	M	45.1±10.6	48.1±10.3	46.8±10.6	44.7±10.3	2.1±0.3	2.0±0.3	2.0±0.3	2.0±0.3	48.4±21.3	57.7±21.2	57.4±20.9	57.2±21.1
	P	43.5±11.2	46.0±12.0	44.2±9.6	42.5±12.6	2.1±0.4	2.1±0.4	2.1±0.3	2.0±0.4	58.6±21.2	58.8±19.9	59.1±20.4	57.9±20.6
	W	37.3±10.7	43.3±12.0	38.5±11.3	38.4±12.9	1.9±0.3	2.0±0.4	1.8±0.3	1.9±0.4	57.2±20.5	57.4±20.5	56.4±21.0	57.1±20.4
Cascade-20	T	26.4±15.7	21.8±12.4	19.3±13.7	20.5±14.5	1.8±0.2	1.6±0.3	1.6±0.2	1.6±0.3	54.1±22.4	51.3±23.6	51.2±23.8	49.0±23.0
	A	11.4±8.2	13.3±11.9	16.7±10.5	15.1±11.2	1.7±0.4	1.5±0.2	1.4±0.3	1.4±0.3	55.2±22.7	52.5±25.3	51.1±23.3	50.3±21.8
	B	15.4±10.9	18.8±13.3	17.3±14.2	17.4±13.9	1.6±0.1	1.6±0.3	1.6±0.3	1.6±0.3	53.1±23.9	52.5±25.2	52.2±24.9	51.9±25.0
	M	30.0±15.1	34.5±13.7	32.3±14.9	32.0±15.0	2.0±0.2	2.0±0.2	2.0±0.3	2.0±0.3	55.2±21.9	55.3±22.3	56.9±23.3	54.0±23.0
	P	20.4±13.4	22.5±14.6	20.8±14.5	21.0±14.1	1.8±0.2	1.8±0.3	1.7±0.2	1.7±0.3	55.0±22.6	54.7±23.7	54.7±23.5	52.0±24.0
	W	23.9±15.0	26.5±14.5	24.8±15.2	25.2±15.5	1.8±0.2	1.8±0.3	1.8±0.3	1.8±0.2	54.7±22.8	54.2±23.4	54.6±23.8	51.9±23.0

F Sensitivity to the ABR Algorithm

While the default ABR algorithm in our experiments is L2A, we also examine how the choice of the ABR algorithm affects the ARTEMIS performance. Specifically, we consider LoL$^+$ as an alternative to L2A. Figure 17 evaluates ARTEMIS' dynamic ladders against the five static baselines on the four network traces. The results demonstrate that ARTEMIS consistently outperforms the baselines in terms of the stall duration, bitrate switching, VMAF switching, bitrate instability, and VMAF instability regardless of the chosen ABR algorithm. Although Figure 17b shows that ARTEMIS serves the players with lower bitrates, Figures 17g and 17h reveal that the decreases in VMAF and PSNR are negligible.

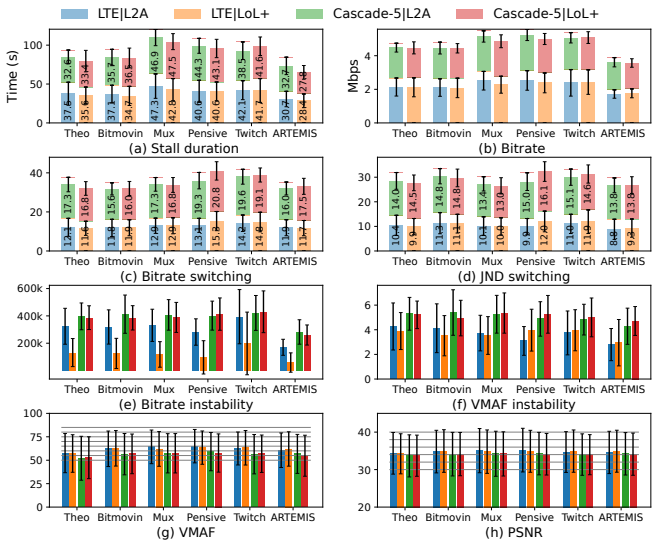


Figure 17: Impact of choosing L2A vs. LoL+ as the ABR algorithm on the performance with the five static ladders and ARTEMIS' dynamic ladders.

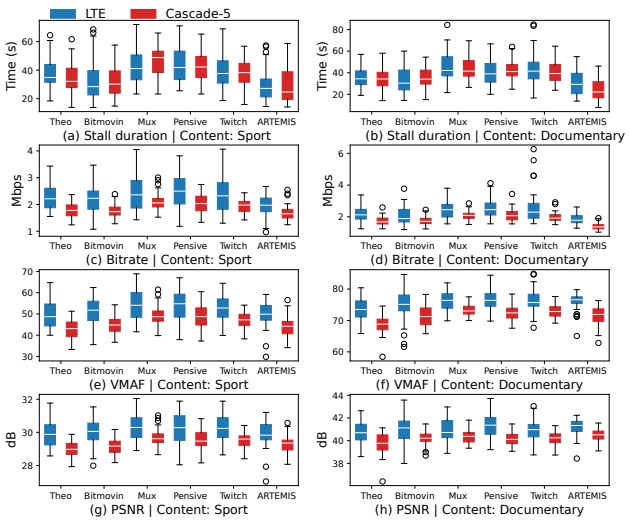


Figure 18: Stall duration, served bitrates, VMAF, and PSNR under the five static ladders and ARTEMIS' dynamic ladders for sport vs. documentary as the content type.

G Influence of the Content Type

We assess ARTEMIS' dynamic ladders against the five static baselines on the LTE and Cascade-5 network traces. Figure 18 reports on the stall duration, bitrates, VMAF, and PSNR. ARTEMIS reduces the stall reduction compared to the static fixed-length baselines for two different content types. We attribute the reduction to the substantial bandwidth fluctuations in both considered network traces, which leads to frequent stalls. Figures 18c and 18d illustrate that ARTEMIS mitigates the stalls by adding representations with lower bitrates to the OTL. Despite the reduction in the served bitrates, ARTEMIS maintains an acceptable VMAF value for the two content types. ARTEMIS surpasses the baseline performance because its ladder construction accounts for video quality measured via PSNR which, unlike VMAF, is quickly computable. The comparison of Figures 18c and 18d with Figures 18g and 18h suggests a strong correlation between PSNR and VMAF. Figure 14 in Appendix B corroborates this correlation.

Figure 19 extends the evaluation by reporting on the bitrate switching, bitrate instability, JND switching, and VMAF instability. We set JND to 6 [18]. While the bitrate switching is similar with ARTEMIS' dynamic ladders and baselines, ARTEMIS reduces the bitrate instability compared to the baseline ladders. We attribute this reduction to composing the mega-manifest from the large number of representations. Overall, the evaluation shows that ARTEMIS outperforms the baselines with different content types.

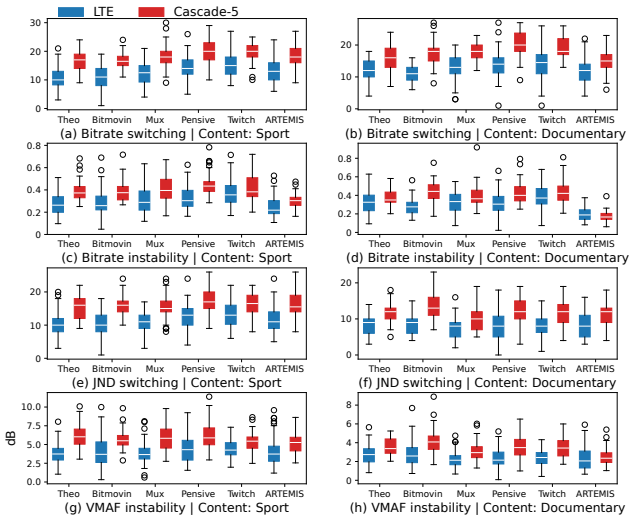


Figure 19: Bitrate switching, bitrate instability, JND switching, and VMAF instability under the five static ladders and ARTEMIS' dynamic ladders for sport vs. documentary as the content type.

H Impact of Weight α

We evaluate the impact of weight α on the LTE trace and a constant-bandwidth trace where the network bandwidth available for each player remains fixed at 7 Mbps throughout the streaming session. We assess how ARTEMIS performs when provided with the following three *StallAlpha* dictionaries:

- D-I = {1:[0,1], 0.9:[1,2], 0.8:[2,3], 0.7:[3,4], 0.6:[4,5], 0.5:[5,100]},
- D-II = {1:[0,2], 0.9:[2,4], 0.8:[4,6], 0.7:[6,8], 0.6:[8,10], 0.5:[10,100]}, and

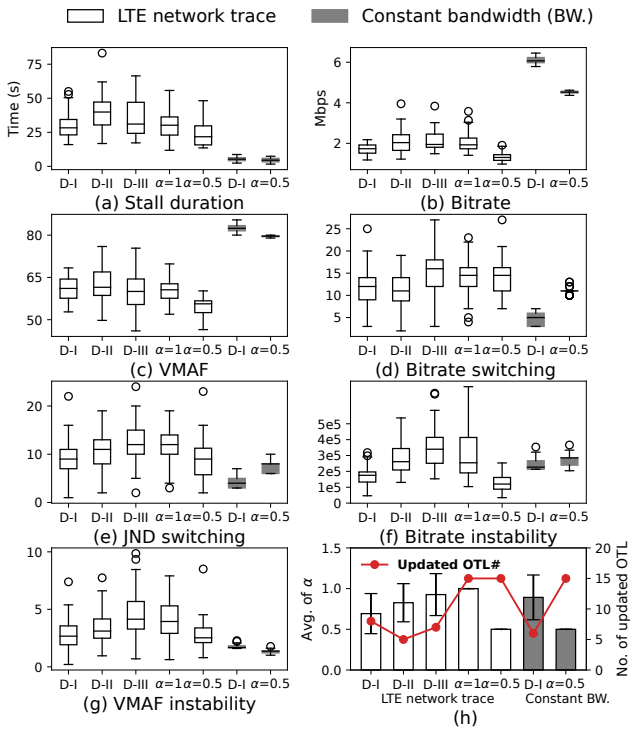


Figure 20: Impact of using the D-I, D-II, and D-III dictionaries to dynamically determine the α value vs. employing the static α values of 0.5 and 1.

- D-III = {1:[0,5], 0.9:[5,10], 0.8:[10,15], 0.7:[15,100]}.

The D-I, D-II, and D-III *StallAlpha* dictionaries exhibit different sensitivity levels to stall events. Dictionary D-I prioritizes stall duration over video quality and decreases α in response to short stalls. For example, ARTEMIS with dictionary D-I sets α to 1 when the average stall duration in the current time slot, which lasts 10 s, is less than 1 s. On the other hand, dictionary D-III sets α to 1 when the average stall duration is below 5 s. Figure 20 shows that reduction of this sensitivity causes more stalls and increases the bitrate instability, JND switching, bitrate instability, and VMAF instability.

We also consider an ARTEMIS variant that uses a static α value instead of relying on the *StallAnalysis*() function to determine α dynamically. Figure 20 indicates that the ARTEMIS instances with dictionary D-I and constant α of 0.5 behave similarly on the LTE trace. The similar behavior occurs because the bandwidth in the LTE trace fluctuates significantly and forces ARTEMIS with dictionary D-I to update the OTL eight times in order to handle stalls. Figure 20h demonstrates that the average of dynamic α values for ARTEMIS with dictionary D-I is 0.69, which is close to 0.5 in the static α setting. However, a static α value makes ARTEMIS perform suboptimally on the constant-bandwidth network trace by considerably reducing the video quality. Based on the experiments, we select D-I as the default

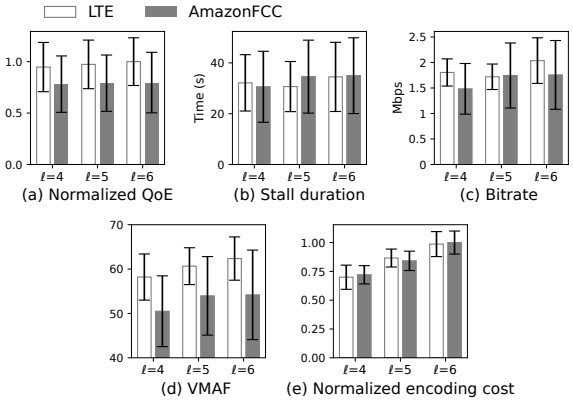


Figure 21: Impact of maximum OTL length ℓ on the ARTEMIS performance.

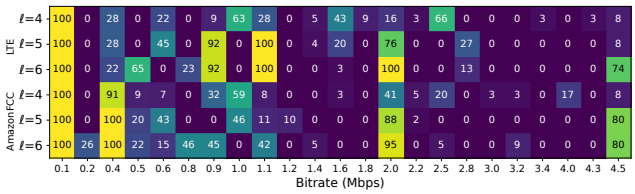


Figure 22: Bitrates selected by ARTEMIS with different values of maximum OTL length ℓ.

StallAlpha dictionary of ARTEMIS because dictionary D-I supports the best response to different network conditions while maintaining high video quality.

I Impact of Maximum OTL Length ℓ

We run ARTEMIS on the LTE and AmazonFCC network traces with different values of maximum OTL length ℓ. Figure 21 presents an exciting result that an increase in ℓ improves the served bitrates, VMAF, and encoding cost. However, these improvements do not guarantee higher QoE, *i.e.*, longer OTLs increase the resource cost without achieving a remarkable improvement in QoE. With relatively short OTLs, the difference between two adjacent bitrates in the OTL is typically large. Thus, the ABR algorithm keeps requesting the same bitrate when the available network bandwidth fluctuates mildly. On the other hand, when the bitrate switching does occur, the amplitude of the bitrate change is high. With a longer OTL, the bitrate switching becomes more frequent, and the bitrate-change amplitude diminishes. Figure 22 depicts the bitrates selected by ARTEMIS for its OTL with different values of the maximum OTL length.

J Impact of Time-Slot Duration θ

We investigate the impact of the time-slot duration on the ARTEMIS performance. It is essential for the time-slot duration to be neither too long nor too short. When the

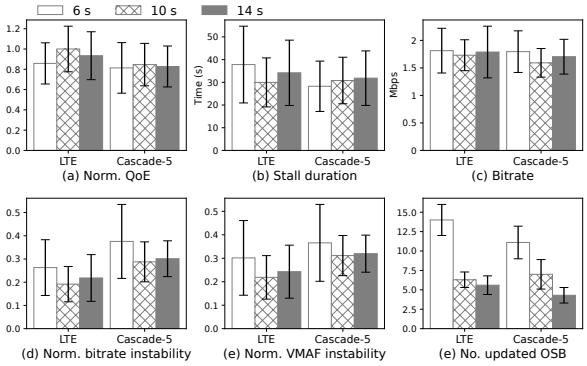


Figure 23: Impact of the time-slot duration on the ARTEMIS performance with the segment duration of 2 s.

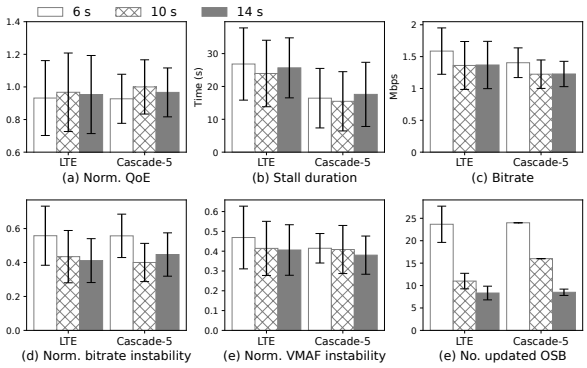


Figure 24: Impact of the time-slot duration on the ARTEMIS performance with the segment duration of 1 s.

time-slot duration is too long, ARTEMIS might be unable to promptly react to condition changes because ARTEMIS checks for a need to update the OTL only at the end of each time slot. The slow response degrades QoE. On the other hand, if the time-slot duration is too short, ARTEMIS incurs larger overhead, *e.g.*, due to solving the MILP more frequently. Other potential concerns include unnecessary updates of the OTL, higher bitrate instability for the clients, drifts in the live encoder, and emergence of video compression artifacts due to frequent changes in the OTL.

Figures 23 and 24 evaluate the impact of the time-slot duration on the ARTEMIS performance when the segment duration is set to 2 s or 1 s, respectively. The experiments rely on the LTE and Cascade-5 network traces and consider 6, 10, and 14 s as three values of the time-slot duration. For both values of the segment duration, our results show that the time-slot duration of 10 s provides the best performance. An algorithm for dynamic configuration of the time-slot duration is an exciting direction for future research on ARTEMIS.

Table 5: Stall duration, bitrate instability, and VMAF instability under ARTEMIS' dynamic ladders compared to the ILP and Netflix ladders in Scenario II.

NT	Bitrate ladder	Stall (s)	Bitrate instability (×1e6)	VMAF instability
LTE	ILP BL [60]	32.3±14.6	0.5±0.2	6.3±2.8
	ARTEMIS-BL1 [60]	31.1±12.7	0.2±0.08	3.2±1.4
	Netflix-Animation	35.5±11.8	0.27±0.1	3.9±1.8
	ARTEMIS-BL2	33.2±11.1	0.19±0.07	3.6±2.1
	Netflix-Movie	40.6±13.5	0.37±0.13	4.0±2.1
	ARTEMIS-BL3	36.1±16.6	0.22±0.077	3.0±1.6
AmazonFCC	Netflix-Animation	36.5±12.7	0.21±0.032	3.29±1.4
	ARTEMIS-BL2	34.5±14.4	0.18±0.069	3.22±1.6
	Netflix-Movie	40.8±13.8	0.35±0.12	4.8±2.2
	ARTEMIS-BL3	35.1±10.11	0.20±0.069	4.5±2.3
Cascade-5	Netflix-Animation	37.2±9.0	0.296±0.065	4.5±1.41
	ARTEMIS-BL2	29.5±9.1	0.185±0.041	3.4±1.07
	Netflix-Movie	45.5±0.2	0.42±0.087	4.5±1.5
	ARTEMIS-BL3	31.5±12.3	0.22±0.081	3.5±1.4
Cascade-20	Netflix-Animation	20.7±11.2	0.1±0.026	2.4±0.5
	ARTEMIS-BL2	17.1±13.2	0.1±0.038	2.1±0.7
	Netflix-Movie	24.5±11.6	0.16±0.035	3.2±1.09
	ARTEMIS-BL3	17.9±13.8	0.13±0.032	2.9±0.92

K Extra Results for Scenario II

Table 5 characterizes the performance of the ILP, Netflix-Animation, and Netflix-Movie bitrate ladders compared to ARTEMIS' three customized ladders in Scenario II.

Credence: Augmenting Datacenter Switch Buffer Sharing with ML Predictions

Vamsi Addanki
TU Berlin

Maciej Pacut
TU Berlin

Stefan Schmid
TU Berlin

Abstract

Packet buffers in datacenter switches are shared across all the switch ports in order to improve the overall throughput. The trend of shrinking buffer sizes in datacenter switches makes buffer sharing extremely challenging and a critical performance issue. Literature suggests that push-out buffer sharing algorithms have significantly better performance guarantees compared to drop-tail algorithms. Unfortunately, switches are unable to benefit from these algorithms due to lack of support for push-out operations in hardware. Our key observation is that drop-tail buffers can emulate push-out buffers if the future packet arrivals are known ahead of time. This suggests that augmenting drop-tail algorithms with predictions about the future arrivals has the potential to significantly improve performance.

This paper is the first research attempt in this direction. We propose CREDENCE, a drop-tail buffer sharing algorithm augmented with machine-learned predictions. CREDENCE can unlock the performance only attainable by push-out algorithms so far. Its performance hinges on the accuracy of predictions. Specifically, CREDENCE achieves near-optimal performance of the best known push-out algorithm LQD (Longest Queue Drop) with perfect predictions, but *gracefully* degrades to the performance of the simplest drop-tail algorithm Complete Sharing when the prediction error gets arbitrarily worse. Our evaluations show that CREDENCE improves throughput by 1.5x compared to traditional approaches. In terms of flow completion times, we show that CREDENCE improves upon the state-of-the-art approaches by up to 95% using off-the-shelf machine learning techniques that are also practical in today's hardware. We believe this work opens several interesting future work opportunities both in systems and theory that we discuss at the end of this paper.

1 Introduction

Datacenter switches come equipped with an on-chip packet buffer that is shared across all the device ports in order to improve the overall throughput and to reduce packet drops. Unfortunately, buffers have become increasingly expensive and chip-manufacturers are unable to scale up buffer sizes proportional to capacity increase [13]. As a result, the buffer available per port per unit capacity of datacenter switches has been gradually reducing over time. Worse yet, datacenter traffic is bursty even at microsecond timescales [22, 57]. This makes it challenging for a buffer sharing algorithm to maximize throughput. Recent measurement studies in large scale datacenters point-out the need for improved buffer sharing algorithms in order to reduce packet drops during congestion events [22]. To this end, buffer sharing under shallow buffers is an emerging critical problem in datacenters [1, 12].

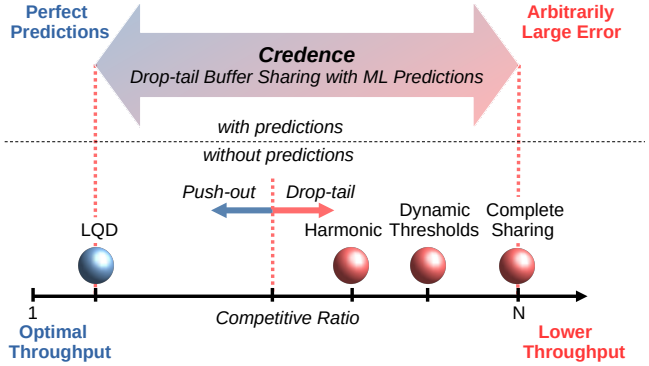


Figure 1: Augmenting drop-tail buffer sharing with ML predictions has the potential to significantly improve throughput compared to the best possible drop-tail algorithm (without predictions), and unlock the performance that was only attainable by push-out so far.

The buffer sharing problem has been widely studied in the literature from an online perspective [15] with the objective to maximize throughput [9, 24, 26, 33, 34]. Traditional online algorithms for buffer sharing can be classified into two types: **drop-tail** e.g., Dynamic Thresholds (DT) [20], Harmonic [33], ABM [1] and **push-out** e.g., Longest Queue Drop (LQD). The performance gap of these algorithms compared to an offline optimal (clairvoyant) algorithm can be expressed in terms of the competitive ratio [15]. For instance, we say that an online algorithm is 2-competitive if it performs at most 2x worse compared to an offline optimal algorithm. Figure 1 illustrates

Algorithm	Competitive Ratio
Complete Sharing [26]	$N+1$
Dynamic Thresholds [20, 26]	$O(N)$
Harmonic [33]	$\ln(N)+2$
LQD (push-out) [9, 26]	1.707
LateQD (clairvoyant) [14]	1
CREDENCE	$\min(1.707\,\eta, N)$

Table 1: CREDENCE's performance smoothly depends on the prediction error (η). CREDENCE outperforms traditional drop-tail buffer sharing algorithms and performs as good as push-out when the predictions are perfect ($\eta = 1$) but is also never worse than Complete Sharing even when the predictions are bad ($\eta \to \infty$). N denotes the number of ports.

the performance spectrum of drop-tail and push out buffer sharing algorithms. In terms of throughput-competitiveness, it is well-known that push-out algorithms perform significantly better than drop-tail algorithms. In fact, no deterministic drop-tail algorithm can perform better than a certain throughput (a lower bound for competitive ratio), beyond which only push-out algorithms exist (Figure 1). Table 1 presents the competitive ratios of known algorithms. Interestingly, LQD pushes out packets when the buffer is full, and it is $\approx$ 2-competitive whereas Complete Sharing drops packets when the buffer is full, but it is $N+1$-competitive.

Intuitively, the poor throughput-competitiveness of drop-tail buffers owes it to the fundamental challenge that utilizing the buffer for some queues comes at the cost of deprivation of buffer for others [1]. To this end, drop-tail algorithms proactively drop packets i.e., packets are dropped even when the buffer has remaining space [11, 20, 26, 28, 33]. On one hand, maintaining remaining buffer space is necessary to serve future packet arrivals. On the other hand, maintaining remaining buffer space could lead to under-utilization, throughput loss and excessive packet drops. In contrast, the superior throughput-competitiveness of push-out algorithms owes it to their fundamental advantage to push out packets instead of dropping them[1]. Hence, push-out algorithms can utilize the entire buffer as needed and only push out packets when multiple ports contend for buffer space. Although push-out algorithms offer far superior performance guarantees compared to drop-tail, hardly any datacenter switch supports push-out operations for the on-chip shared buffer. This begs the question: Are drop-tail buffer sharing algorithms ready for the trend of shrinking buffer sizes?

Our key observation is that every push-out algorithm can be converted to a drop-tail algorithm. However, such a conversion requires certain (limited) visibility into the future packet arrivals. Specifically, pushing out a packet is equivalent to dropping the packet when it arrives. Recent advancements in dataplane programmability and traffic predictions play a pivotal role in providing such visibility into the future packet arrivals [4, 17, 31, 37]: paving a way for better drop-tail buffer sharing algorithms.

In this paper, we take the first step in this direction. Figure 1 illustrates our perspective. We propose CREDENCE, a drop-tail buffer sharing algorithm augmented with machine-learned predictions. CREDENCE's performance is tied to the accuracy of these predictions. As the prediction error decreases, CREDENCE unlocks the performance of push-out algorithms and reaches the performance of the best-known algorithm. Even when the prediction error grows arbitrarily large, CREDENCE offers at least the performance of the simplest drop-tail algorithm Complete Sharing. Table 1 gives the competitive ratio of CREDENCE as a function of the prediction error η. Importantly, CREDENCE's performance *smoothly* varies with the prediction error, generalizing the performance space between the known push-out and drop-tail algorithms. Hence, CREDENCE achieves the three goals of prediction-augmented algorithms, in the literature referred to as consistency, robustness and smoothness [42, 46].

In addition to the theoretical guarantees for CREDENCE's performance, our goal is also its practicality. Specifically, without predictions, CREDENCE's core logic only uses additions, subtractions, and does not add additional complexity compared to existing approaches. For predictions, we currently use random forests, which have been recently shown to be feasible on programmable switches at line rate [4, 17]. A full implementation of CREDENCE in hardware unfortunately requires switch vendor involvement since buffer sharing is merely a blackbox even in programmable switches. With this paper, we wish to gain attention from switch vendors on the fundamental blocks required for such algorithms to be deployed in the dataplane. We currently implement CREDENCE in NS3 to evaluate its performance using realistic datacenter workloads. We present a detailed discussion on the practicality of CREDENCE later in this paper.

Our evaluations show that CREDENCE performs 1.5x better in terms of throughput and up to 95% better in terms of flow completion times, compared to alternative approaches.

We believe CREDENCE is a stepping stone towards further improving buffer sharing algorithms. Especially, achieving better performance than CREDENCE under large prediction error remains an interesting open question. Our approach of augmenting buffer sharing with predictions is not limited to drop-tail algorithms, but push-out algorithms can also benefit from predictions. We discuss exciting future research directions both in systems and theory at the end of this paper.

In summary, our key contributions in this paper are:

- CREDENCE, the first buffer sharing algorithm augmented with predictions, achieving near-optimal performance with perfect predictions while also guaranteeing performance under arbitrarily large prediction error, and gradually degrading the performance as the prediction error increases.

[1] Push-out operation, similar to a drop operation, does not incur any transmission delays, unlike extract-out [51].

- Extensive evaluations using realistic datacenter workloads, showing that CREDENCE outperforms existing approaches in terms of flow completion times.
- All our artifacts have been made publicly available at https://github.com/inet-tub/ns3-datacenter.

2 Motivation

In this section, we provide a brief background and motivate our approach by highlighting the drawbacks of traditional approaches. We show the potential for reaching close-to-optimal performance when buffer sharing algorithms are augmented with machine-learned predictions. To this end, we first describe our model and throughput competitiveness (§2.1). We then discuss the drawbacks of existing approaches (§2.2). We show that a renewed hope for improved buffer sharing is enabled by the recent rise in algorithms with predictions (§2.3).

2.1 Buffer Sharing from Online Perspective

A network switch receives packets one after the other at each of its ports. The switch does not know the packet arrivals ahead of time. This makes buffer sharing inherently an *online* problem i.e., algorithms must take instantaneous decisions upon packet arrivals without the knowledge of the future. In order to systematically understand the performance of such algorithms, we take an online approach following the classical model in the literature [9, 24, 26, 33, 34]. In this section, we describe our model intuitively, and we refer to Appendix A for formal definitions. Figure 2 illustrates the model.

Buffer model: We consider an output-queued switch with N ports and a buffer size of B. Buffer is shared across all the ports. A buffer sharing algorithm takes buffering decisions that we describe next. We assume that time is discrete. At most N packets can arrive in a single timeslot (since there are N ports), and each port removes at most one packet in a timeslot.

Online algorithm: When a packet arrives, a buffer sharing algorithm determines whether it should be accepted into the available buffer space. Drop-tail algorithms can only accept or discard incoming packets, while push-out algorithms can also remove packets from the buffer.

Objective: The network throughput is of utmost importance for datacenter operators since throughput often relates to the cost in typical business models (e.g., $ per bandwidth usage). We hence consider throughput as an objective function, following the literature. Specifically, for any packet arrival sequence, our objective is to maximize the total number of transmitted packets. The throughput maximization objective is closely related to packet drops minimization objective. In this sense, our objective captures two important performance metrics i.e., throughput and packet drops.

Competitive Ratio: We use competitive ratio as a measure to compare the performance of an online algorithm to the optimal offline algorithm. Specifically, let ALG and OPT be an online and optimal offline algorithm correspondingly. Let $ALG(\sigma)$ be the throughput of ALG for the packet arrival sequence σ. We say an algorithm ALG is c-competitive if the following relation holds for any packet arrival sequence.

$$OPT(\sigma) \leq c \cdot ALG(\sigma)$$

Competitive ratio is a particularly interesting metric for buffer sharing since it offers performance guarantees without any assumptions on specific traffic patterns. For example, the buffer may face excessive packet drops or may temporarily experience throughput loss due to bursty traffic. One could argue that the buffer sharing algorithm is the culprit and should have allocated more buffer to the bursty traffic. While this may have solved the problem for a particular bursty arrival, the same solution could result in unexpected drops and throughput loss if there were excessive bursty arrivals i.e., large bursts could monopolize the buffer. Instead, from an online perspective, better competitive ratio indicates that the buffer sharing algorithm performs close to optimal under any traffic conditions.

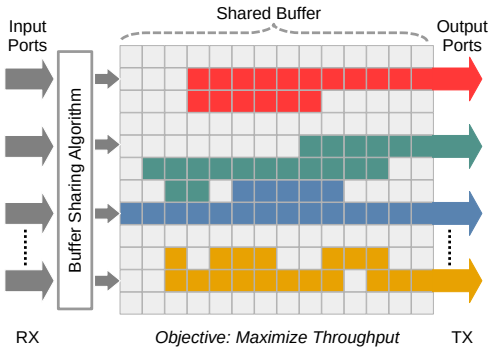


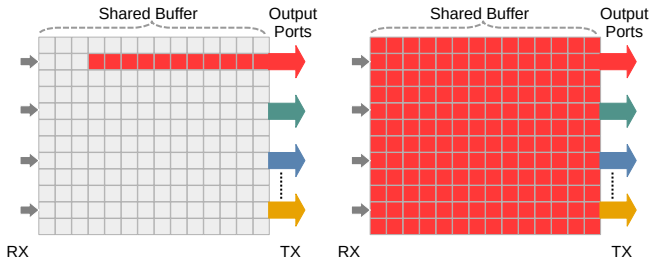
Figure 2: The switch has a buffer size of B shared across N output ports. Each color indicates the packets residing in the shared buffer corresponding to each port. A buffer sharing algorithm takes decisions (accept or drop) for each input packet.

■ ***Takeaway.*** *A buffer sharing algorithm with lower competitive ratio improves the throughput of the switch and reduces packet drops under worst-case packet arrival patterns.*

2.2 Drawbacks of Traditional Approaches

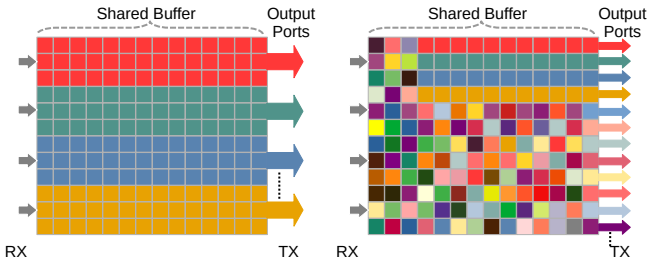
We observe two main drawbacks of traditional buffer sharing algorithms, both affecting the competitive ratio. First, algorithms proactively and unnecessarily drop packets in view of accommodating future packet arrivals. Second, algorithms reactively drop packets when the buffer is full and incur throughput loss, which could have been avoided. We argue that these drawbacks are rather fundamental to drop-tail algorithms and cannot be addressed by traditional online approaches.

Proactive unnecessary packet drops → throughput loss: A drop-tail buffer sharing algorithm typically drops packets even



(a) ALG (b) OPT

Figure 3: Upon a large burst arrival, a typical drop-tail algorithm (ALG) *proactively* drops the incoming packets in anticipation of future bursts and significantly under-utilizes the buffer. In this case, an optimal offline algorithm accepts the entire burst without any packet drops.



(a) ALG (b) OPT

Figure 4: In pursuit of high burst absorption, a drop-tail algorithm ALG may absorb bursts but this results in excessive *reactive* drops for the future packet arrivals. In this case, an optimal offline algorithm OPT drops few packets such that the overall throughput is maximized.

if there is remaining buffer space available [1, 11, 20]. We refer to such drops as proactive drops. Being proactive is indeed necessary in order to accommodate transient bursts. However, proactive packet drops and the corresponding remaining buffer space ends up being wasteful if the future packet arrivals do not need additional buffer space (if the anticipated burst does not arrive). Figure 3a and Figure 3b illustrate an example. Consider a traffic pattern where there is little to no congestion on all the ports but once in a while, a large burst appears. Specifically, the buffer is empty initially and a large burst of size B appears. A deterministic drop-tail algorithm has two choices: *(i)* accept a portion of the burst and proactively drop the rest of the burst or *(ii)* accept the entire burst. Typical algorithms in the literature choose the former in view of accommodating future packet arrivals. An optimal offline algorithm that knows the arrivals ahead of time would accept the entire burst of size B in this case. This makes an online algorithm at least c-competitive for this particular arrival pattern, where $\frac{1}{c}$ is the fraction of the burst accepted: since the optimal solution accepts and transmits B packets over time, whereas an online algorithm only accepts and transmits only $\frac{B}{c}$ packets over time. We observe that recent works focus on minimizing proactive unnecessary packet drops by prioritizing bursty traffic to the extent that they allow burst on a single port to monopolize the buffer [1, 11, 28]. However, note that competitive ratio is not defined for a particular arrival sequence, but over all scenarios. To this end, accepting a larger burst size may be helpful in the above example but if there were indeed future packet arrivals on other ports that need buffer, the algorithm incurs excessive reactive drops (described next) and throughput loss.

Reactive avoidable packet drops → throughput loss: Any drop-tail algorithm is forced to drop the incoming packets once the shared buffer is full. We call such drops reactive drops. Reactive drops result in throughput loss if the algorithm fills up significant portion of the buffer on a small set of ports but reactively drops incoming packets to other ports. Figure 4a and Figure 4b illustrate an example. Consider that the buffer is initially empty and four simultaneous bursts each of size B arrive to four ports. If an algorithm proactively drops a significant portion of the bursts, it would suffer under arrival sequences such as in the previous example (Figure 3a). Alternatively, the algorithm may choose to accept a larger portion of the bursts and ends up filling up the entire buffer in aggregate. At this point, several short bursts arrive to multiple other ports. An optimal offline algorithm accepts only a fraction of the large bursts such that it is able to accommodate upcoming short bursts. In doing so, the optimal algorithm benefits in throughput since the switch transmits packets from more number of ports. However, since the online algorithm fills up the entire buffer due to the initial large bursts, it is forced to reactively drop the upcoming short bursts, losing throughput. In fact, a similar arrival pattern for Dynamic Thresholds yields at least $\Omega\left(\sqrt{\frac{N}{\log(N)}}\right)$-competitiveness [26]. The known upper bound for Dynamic Thresholds is $O(N)$ [26]. Further, it has been shown in the literature that no deterministic drop-tail algorithm can be better than $\Omega\left(\frac{\log(N)}{\log(\log(N))}\right)$-competitive [33].

Interestingly, push-out algorithms are not prone to the problems discussed above, since they can take revocable decisions i.e., to accept a packet and drop it later. Hence, push-out algorithms do not have to maintain free space in the buffer in order to accommodate transient bursts. Instead, such algorithms can defer the dropping decision until the moment the drop turns out to be necessary.

■ ***Takeaway.*** *Traditional drop-tail algorithms are fundamentally limited in throughput-competitiveness as they are unable to effectively navigate proactive and reactive drops due to the online nature of the problem i.e., future packet arrivals are unknown to the algorithm.*

2.3 Predictions: A Hope for Competitiveness

Given that the fundamental barrier in improving drop-tail buffer sharing algorithms is the lack of visibility into the future arrivals, we turn towards predictions. The recent rise of algorithms with predictions offers a renewed hope for competitive buffer sharing. Algorithms with predictions

successfully enabled close to optimal performance for various classic problems [46]. The core idea is to guide the underlying online algorithm with certain knowledge about the future obtained via predictions. The machine-learned oracle that produces predictions is considered a blackbox with a certain error. The main challenge is to offer performance guarantees at the extremes i.e., close to optimal performance under perfect predictions and a minimum performance guarantee when the prediction error gets arbitrarily large. Further, it is desirable that the competitiveness of the algorithm *smoothly* degrades as the prediction error grows.

2.3.1 Prediction Model

In the context of the buffer sharing problem, there are several prediction models that can be considered e.g., drops or packet arrivals. In this paper, we assume that a blackbox machine-learned oracle predicts packet drops. Our choice is due to the fact that packet drops are the basic decisions made by an algorithm. Concretely, we consider an oracle that predicts whether an incoming packet would eventually be dropped (or pushed out) by the Longest Queue Drop (LQD) algorithm serving the same packet arrival sequence. We classify the predictions into four types: *(i)* true positive i.e., a correct prediction that a packet is eventually dropped by LQD, *(ii)* false negative i.e., an incorrect prediction that a packet is eventually transmitted by LQD, *(iii)* false positive i.e., an incorrect prediction that a packet is eventually dropped by LQD and *(iv)* true negative i.e., a correct prediction that a packet is eventually transmitted by LQD. Figure 5 summarizes this classification. Following the literature [42, 46], our goals for prediction-augmented buffer sharing are consistency, robustness and smoothness.

α-Consistent buffer sharing algorithm has a competitive ratio α when the predictions are all true i.e., perfect predictions.

β-Robust buffer sharing algorithm has a competitive ratio β when the predictions are all false i.e., large prediction error.

Smoothness is a desirable property such that the competitive ratio degrades smoothly as the prediction error grows i.e., a small change in error does not drastically influence the competitive ratio.

Our goal is to design a prediction-augmented buffer sharing algorithm that is close to 1-consistent (with perfect predictions) i.e., near-optimal, at most N-robust (with arbitrarily large error) i.e., not worse than Complete Sharing algorithm, and has the desirable property of smoothness.

2.3.2 Common Pitfalls

It is intuitive that predictions can potentially improve the performance of a drop-tail algorithm. For instance, in the examples from Figure 3 and Figure 4, our prediction-augmented online algorithm could take nearly the same decisions as a push-out algorithm. However, the main challenge is to ensure robustness and smoothness. If an algorithm blindly trusts the predictions, we observe that false positive and false negative predictions have a significantly different impact on the performance.

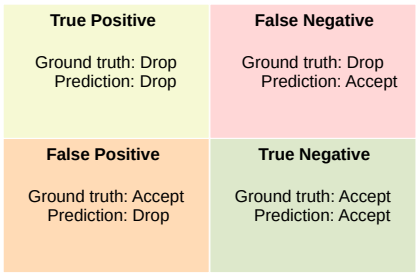


Figure 5: Confusion matrix for our prediction model.

Excessive false positives can lead to starvation: The worst case for a naive algorithm that blindly trusts predictions is when all the predictions are false positives. In this case, the algorithm ends up dropping every incoming packet. Blindly trusting false predictions could lead to a competitive ratio worse than the simplest drop-tail algorithm Complete Sharing i.e., the competitive ratio becomes unbounded (∞-robust) if the predictions are mostly false positives.

A single false negative can hurt throughput forever: A naive algorithm that blindly relies on false negative predictions is susceptible to adverse effects that propagate over time. Consider a packet arrival sequence that hits only one queue initially and consider that the predictions are all true negatives until the queue length reaches $B-1$, where B is the total buffer size. At this point, one more packet arrives and our prediction is a false negative. As a result, our naive algorithm has a queue of size B and the optimal algorithm has a queue of size $B-1$. Note that all non-empty queues drain one packet after each timeslot. From here on, in every timeslot, one packet (first) arrives to the large queue and one packet (second) arrives to any other queue. Also consider that all the predictions are true from now on. The optimal algorithm accepts both first and second packet in every timeslot. However, in every timeslot our naive approach can only accept the first packet to the large queue and cannot accept the second packet since the buffer is full. Notice that relying on just one false negative resulted in cumulative drops in this case even though all other predictions were true. In fact, a tiny error such as just N number of false negatives even with all other predictions being true could result in a competitive ratio for a naive approach as worse as Complete Sharing.

■ ***Takeaway.*** *Augmenting drop-tail algorithms with predictions has the potential to unlock the optimal performance. Ensuring performance guarantees with inaccurate predictions remains a challenge.*

3 Prediction-Augmented Buffer Sharing

Reflecting on our observations in §2, our goal is to design a drop-tail buffer sharing algorithm that performs close to optimal with perfect predictions but also provides a minimum performance guarantee when the prediction error is

arbitrarily large. In essence, our aim is to enable performance improvement in terms of throughput and packet drops in datacenter switches. To this end, we first present an overview of our algorithm (§3.1). We then present the workings of CREDENCE (§3.2) and discuss its properties (§3.3). Finally, we discuss the practicality of CREDENCE (§3.4).

3.1 Overview

In a nutshell, CREDENCE relies on predictions and *follows* a push-out algorithm, reaching close to optimal performance under perfect predictions. CREDENCE cleverly takes certain decisions independent of the predictions in order to guarantee a minimum performance. Further, CREDENCE's competitiveness gradually degrades as prediction error grows (a property known as smoothness [42]), hence the algorithm still performs near-optimally when predictions are slightly inaccurate.

CREDENCE *follows* Longest Queue Drop algorithm: Our design of CREDENCE consists of two key ingredients. First, CREDENCE uses thresholds as a drop condition irrespective of the predictions. CREDENCE treats thresholds as queue lengths of LQD and updates the thresholds based on the LQD algorithm (simply arithmetic) upon every packet arrival. Second, CREDENCE relies on predictions as long as the queue lengths satisfy the corresponding thresholds. The combination of thresholds and predictions allows CREDENCE to closely *follow* the Longest Queue Drop algorithm (LQD) without requiring push-out operations[2].

CREDENCE guarantees performance under extremities: When all the predictions are perfectly accurate, CREDENCE achieves a competitive ratio of 1.707 (consistency) due to the straight-forward argument that the drops by CREDENCE and LQD are equivalent for true predictions. In order to guarantee a minimum performance even with arbitrarily large prediction error (robustness), CREDENCE bypasses the threshold and predictions as long as the longest queue is within $\frac{B}{N}$ size. Here, B is the buffer size and N is the number of ports. This allows CREDENCE to be most N-competitive even under large prediction error, similar to the Complete Sharing algorithm. We prove our claim formally in Appendix C.

CREDENCE smoothly degrades with prediction error: We design our error function in terms of the performance of LQD and the predicted drops. We analyze the types of drops incurred by CREDENCE due to false positive and false negative predictions. This allows us to show that CREDENCE satisfies the smoothness property i.e., the competitive ratio smoothly degrades from 1.707 to N as the prediction error grows.

3.2 CREDENCE

We now present CREDENCE and explain how it operates. Algorithm 1 presents the pseudocode of CREDENCE. Our pseudocode is simplified to discrete time for ease of presentation and for simplicity of analysis. It can be trivially extended to continuous time, and our implementation incorporates it[3].

Arrival: Upon a packet arrival, CREDENCE has three important steps that are highlighted in Algorithm 1. First, CREDENCE updates the threshold for the current queue (highlighted in blue). Second, CREDENCE takes a decision based on the thresholds and predictions whether or not to accept the incoming packet (highlighted in yellow). Finally, the packet is either accepted or dropped. We next describe each of these steps in detail. Third, depending on the state of the buffer, CREDENCE bypasses the thresholds and predictions with a safeguard condition in order to accept or drop the incoming packet (highlighted in green).

Thresholds: CREDENCE updates its thresholds based on the longest queue drop algorithm. Specifically, upon a packet arrival at time t to a queue i, CREDENCE increments the threshold $T_i(t)$ for queue i by the packet size. If upon arrival the sum of thresholds $\Gamma(t)$ is equal to the buffer size B, then CREDENCE first decrements the longest queue threshold by packet size and then increments the threshold for queue i by the packet size. Note that upon a packet arrival to a queue, the corresponding threshold is updated before accepting or dropping the packet.

Drop criterion: Similar to existing threshold-based algorithms, CREDENCE also uses thresholds as a drop criterion. CREDENCE compares the queue length $q_i(t)$ of a queue i against its threshold $T_i(t)$ and drops an incoming packet if the queue length is larger than or equal to the corresponding threshold. If and only if an incoming packet satisfies the thresholds, then CREDENCE takes input from a machine-learned oracle that predicts whether to accept or drop according to our prediction model discussed in §2.3.1. Finally, based on the thresholds and predictions, CREDENCE either accepts or drop the incoming packet.

Safeguard: In order to bound CREDENCE's competitiveness under arbitrarily large prediction error, we bypass the above drop criterion under certain cases. Specifically, when the longest queue length is less than $\frac{B}{N}$, CREDENCE always accepts a packet irrespective of the thresholds and predictions. This ensures that CREDENCE is at least N-competitive even with large prediction error. Our safeguard is based on the observation that even the push-out longest queue drop algorithm cannot push out a packet from a queue less than $\frac{B}{N}$ size since the longest queue must be at least $\frac{B}{N}$ size when the buffer is full. In essence, CREDENCE circumvents the impact of large prediction error by accepting packets until a certain amount of buffer is filled up.

Predictions: CREDENCE can be used with any ML oracle that predicts whether to accept or drop a packet, according to our prediction model (see §2.3.1). We do not rely on the internal details of the oracle. However, certain choices of ML oracles are better suited to operate within the limited resources available in a switch hardware. We discuss further on our choice of oracle later in §3.4.

[2] Recall that LQD is close to optimal with a competitive ratio of 1.707.

[3] Our source code is available is at https://github.com/inet-tub/ns3-datacenter.

Algorithm 1: CREDENCE

Input : Packet arrivals σ,
Drop predictions $\phi'(\sigma)$

1 **procedure** ARRIVAL $(\sigma(t))$**:**
2 **for** *each packet* $p \in \sigma(t)$ **do**
3 Let i be the destination queue for the packet p
4 UPDATETHRESHOLD$(i, arrival)$
 ▷ *Guarantees N-competitiveness*
 Let j be the longest queue
5 **if** $q_j(t) < \frac{B}{N}$ **then**
 $q_i(t) \leftarrow q_i(t)+1$ ▷ Accept
 Continue to next packet
 ▷ *Enables* 1.707 η*-competitiveness*
 if $q_i(t) < T_i(t)$ **then**
 if $Q(t) < B$ **then**
 $drop$ = GETPREDICTION()
 if $drop$ **then**
6 ▷ Drop
 else
 $q_i(t) \leftarrow q_i(t)+1$ ▷ Accept
 else
 ▷ Drop
7 **procedure** DEPARTURE (i)**:**
8 **if** $q_i(t) > 0$ **then**
9 $q_i(t) \leftarrow q_i(t)-1$ ▷ Drain one packet
10 UPDATETHRESHOLD$(i, departure)$
11 **procedure** UPDATETHRESHOLD $(i,\ event)$**:**
12 **if** *event* = *arrival* **then**
 ▷ *Thresholds are treated as LQD queue lengths*
 if $\Gamma(t) = B$ **then** ▷ Sum of thresholds
 Let $T_j(t)$ be the highest threshold
 $T_j(t) \leftarrow T_j(t)-1$ ▷ Decrease
13 $T_i(t) \leftarrow T_i(t)+1$ ▷ Increase
 else
 $T_i(t) \leftarrow T_i(t)+1$ ▷ Increase
 $\Gamma(t) \leftarrow \Gamma(t)+1$
14 **if** *event* = *departure* **then**
15 **if** $T_i(t) > 0$ **then**
16 $T_i(t) \leftarrow T_i(t)-1$ ▷ Decrease
17 $\Gamma(t) \leftarrow \Gamma(t)-1$

3.3 Properties of CREDENCE

CREDENCE offers attractive theoretical guarantees in terms of competitive ratio. In this section, for simplicity, we refer an offline optimal algorithm as *OPT*.

Although we have so far discussed the prediction error more intuitively, it requires a quantitative measure in order to analyze the performance of an algorithm relying on predictions. There are two important considerations in defining a suitable error function. First, following the literature, an error function must be independent of the state and actions of our algorithm, so that we can train a predictor without considering all possible states of the algorithm [29]. Second, it is desirable that the performance of our algorithm can be related to the error function in an uncomplicated manner. Taking these into consideration, we define our error function in Definition 1. Our definition captures the prediction error in terms of the performance of LQD (push-out) and the performance of an algorithm *FollowLQD*. Here, *FollowLQD* (Algorithm 2 in Appendix B) is a deterministic drop-tail algorithm (without predictions) with thresholds similar to CREDENCE.

Definition 1 (Error function)**.** *Let LQD(σ) and FollowLQD(σ) denote the total number of packets transmitted by the online push-out algorithm LQD and the online drop-tail algorithm FollowLQD over the arrival sequence σ. Let ϕ denote the sequence indicating drop by LQD for each packet in the arrival sequence σ. Let ϕ' denote the sequence of drops predicted by the machine-learned oracle. Let ϕ'_{TP}, ϕ'_{FP}, ϕ'_{TN}, and ϕ'_{FN} denote the sequence of true positive, false positive, true negative and false negative predictions for the arrival sequence σ. We define the error function $\eta(\phi,\phi')$ as follows:*

$$\eta(\phi,\phi') = \frac{LQD(\sigma)}{FollowLQD\left(\sigma - \phi'_{TP} - \phi'_{FP}\right)} \tag{1}$$

Using Definition 1, we analyze the throughput of CREDENCE over an entire packet arrival sequence σ based on the predictions ϕ'. In fact, our error function is upper bounded by an intuitive closed form expression, in terms of the number of true and false predictions, as follows, that can be easily computed:[4]

$$\eta(\phi,\phi') \leq \frac{\phi'_{TN} + \phi'_{FP}}{\phi'_{TN} - \min\left((N-1)\cdot\phi'_{FN}, \phi'_{TN}\right)}$$

The upper bound of our error function indicates intuitively that *(i)* the error decreases as the total number of true negative predictions dominate the total false predictions, *(ii)* the error increases with each false positive prediction and *(iii)* the error increases with each false negative with a larger weight. Lemma 1 states the relation between the throughput of CREDENCE, throughput of *LQD* and the prediction error.

Lemma 1. *The total number of packets transmitted by* CREDENCE *for an arrival sequence σ, a drop sequence ϕ by LQD and the predicted drop sequence ϕ' is given by*

$$\text{CREDENCE}(\sigma) \geq \frac{LQD(\sigma)}{\underbrace{\eta(\phi,\phi')}_{error}} \tag{2}$$

Equation 2 shows that the throughput of CREDENCE reaches closer to (moves away from) *LQD* as the prediction error becomes smaller (larger). We present a sketch of our proof here. Our full proof appears in Appendix C. We begin by analyzing the drops incurred by CREDENCE based on the drop criterion described in §3.2. We argue that for every true positive and false positive predictions, there is at most one

[4] We prove our upper bound in Theorem 2 (Appendix C).

drop by CREDENCE. All other drops incurred by CREDENCE are due to the thresholds. Using these observations, we show that CREDENCE transmits at least the number of packets transmitted by *FollowLQD* over the arrival sequence $\sigma - \phi'_{TP} - \phi'_{FP}$. This leads us to Equation 2.

Recall that CREDENCE bypasses the drop criterion and accepts packets through a safeguard condition under certain cases (see §3.2). Based on this, we obtain another bound for the throughput of CREDENCE in Lemma 2, which is independent of the prediction error.

Lemma 2. CREDENCE *transmits at least* $\frac{1}{N}$ *times the number of packets transmitted by an offline optimal algorithm OPT i.e.,* $\text{CREDENCE}(\sigma) \geq \frac{1}{N} \cdot OPT(\sigma)$.

Lemma 2 shows that irrespective of the prediction error (even under large error), CREDENCE can always transmit at least $\frac{1}{N}$ fraction of the packets transmitted by an optimal solution. Our proof of Lemma 2 is based on the fact that upon a drop by CREDENCE, there is at least one queue with length $\frac{B}{N}$ (the safeguard condition). As a result, for every B packets transmitted by OPT, there are at least $\frac{B}{N}$ number of packets transmitted by CREDENCE over the arrival sequence σ. This leads us to the bound expressed in Lemma 2.

Finally, using the above two results, we prove the competitive ratio of CREDENCE as a function of the prediction error. CREDENCE's competitive ratio satisfies the three desirable properties: 2-consistent, N-robust and exhibits smoothness.

Theorem 1. *The competitive ratio of* CREDENCE *grows linearly from* 1.707 *to N based on the prediction error* $\eta(\phi, \phi')$*, where N is the number of ports,* ϕ *is the drop sequence of LQD and* ϕ' *is the predicted sequence of drops i.e., the competitive ratio is at most* $\min(1.707\,\eta(\phi, \phi'), N)$.

Our proof follows from Lemmas 1 and 2 (see Appendix C). Theorem 1 essentially shows how CREDENCE's competitive ratio in terms of throughput improves from N to 1.707 as the prediction error (Definition 1) decreases. Interestingly, CREDENCE's competitive ratio is independent of the buffer size B i.e., CREDENCE is compatible for shallow buffers as well as deep buffers. We note that our analysis compares an algorithm against an optimal offline algorithm over a fixed packet arrival sequence. This allows us to analyze the competitive ratio via an error function defined over the corresponding arrival sequence. However, real-world traffic is responsive in nature due to congestion control and packet retransmissions. Although we have used η as our error function to express the competitiveness of CREDENCE, in our evaluation (§4), we compare CREDENCE with state-of-the-art approaches under realistic datacenter workloads and we also present the quality of our predictions using more natural error functions that are widely used for machine learning models.

3.4 Practicality of CREDENCE

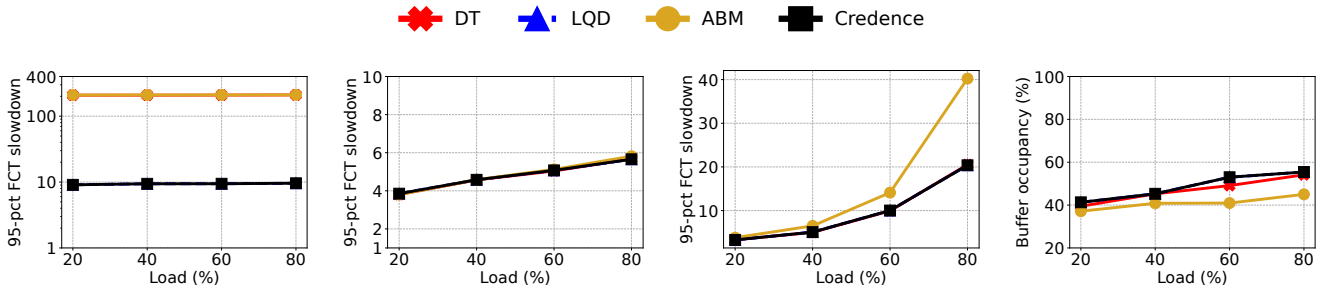
CREDENCE's algorithm itself is simple and close to complexity of the longest queue drop (push-out). However, the machine-learned oracle producing the predictions adds additional complexity in order to deploy CREDENCE on switches. Overall, there are three main parts of CREDENCE that contribute to additional complexity in terms of memory and computation: *(i)* finding the longest queue (and its threshold), *(ii)* remembering thresholds and *(iii)* obtaining predictions.

Finding the longest queue (and its threshold): For every packet arrival, CREDENCE requires finding the longest queue for the safeguard condition described in §3.2. Additionally, CREDENCE requires finding the largest threshold during the threshold updates upon every packet arrival. The maximum value search operation has a run-time complexity of $O(N)$, where N is the number of ports. Note that typical datacenter switches have a relatively small number of ports e.g., 64 ports in Broadcom Tomahawk4 [16]. Prior work in the context of LQD proposes an approximation to further reduce the complexity of finding the longest queue [52] to $O(1)$. The average case complexity can further be reduced by only maintaining the list of queue lengths (and their thresholds) that are larger than $\frac{B}{N}$. This is sufficient since the safeguard condition checks whether the longest queue is less than $\frac{B}{N}$, which is the same as checking that no queue is longer than $\frac{B}{N}$. Similarly, the largest threshold search during the threshold updates is only triggered when the buffer is full. In this case, the longest queue must be at least $\frac{B}{N}$. Given that switches are becoming more and more computationally capable, we believe that a basic function such as finding the maximum value in a small list is feasible to implement within the available resources.

Thresholds memory: In contrast to existing threshold-based algorithms, CREDENCE's thresholds depend on their previous value i.e., thresholds must be remembered. As a result, CREDENCE adds a small memory overhead of $O(N)$ for the thresholds. The threshold calculations are in fact much simpler than existing schemes and do not add any further computational complexity since CREDENCE only requires adding and subtracting the threshold values by the packet size.

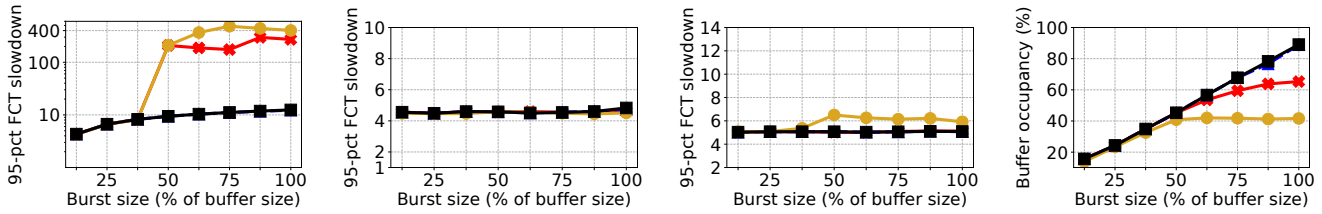
Predictions: Our prediction model (drop or accept) essentially boils down to binary classification problem. To this end, numerous ML techniques exist ranging from linear classifiers to more advanced neural networks. In view of practicality, we consider random forests as they are implementable in programmable hardware [4, 17, 31]. In order to reduce the prediction latency, we also limit the number of trees and the maximum depth of our trained random forest model. We find that, even a model trained with a maximum depth of four, and as low as four to eight trees achieves reasonable prediction error (precision $\approx$ 0.65). Further, to reduce the complexity of the model, we also limit the number of features to four: queue length, total shared buffer occupancy and their corresponding moving averages (exponentially weighted) over one round-trip time (baseRTT).

The fundamental blocks required for CREDENCE are all individually practical in today's hardware. Unfortunately, modifying the buffer sharing algorithm and integrating it



(a) Incast flows (b) Short flows (c) Long flows (d) Shared buffer occupancy

Figure 6: Performance of CREDENCE across various loads of websearch workload and incast workload at a burst size 50% of the buffer size, with DCTCP as the transport protocol. As the load increases, ABM penalizes long flows. DT and ABM are unable to absorb bursts of size 50% of the buffer size. CREDENCE achieves superior burst absorption and does not penalize long flows.



(a) Incast flows (b) Short flows (c) Long flows (d) Shared buffer occupancy

Figure 7: Performance of CREDENCE across various burst sizes of incast workload and websearch workload at 40% load, with DCTCP as the transport protocol. At small burst sizes, DT and ABM achieve similar performance compared to CREDENCE but as the burst size increases, CREDENCE outperforms DT and ABM in terms of FCTs for incast flows (burst absorption).

with predictions requires switch vendor support. Even in programmable switches, the traffic manager is merely a blackbox that implements Dynamic Thresholds with a single parameter exposed to the user [8, 10]. Given the superior performance of CREDENCE (§4), we wish to gain attention from switch vendors to discuss further on the implementation of CREDENCE.

4 Evaluation

We evaluate the performance of CREDENCE and compare it against state-of-the-art buffer sharing algorithms in the context of datacenter networks. Our evaluation aims at answering three main questions:

(Q1) *Does* CREDENCE *improve the burst absorption?*
Our evaluation shows that CREDENCE significantly improves the burst absorption capabilities of switches. We find that CREDENCE improves the 95-percentile flow completion times for incast flows by up to 95.4% compared to Dynamic Thresholds (DT) and by up to 96.9% compared to ABM.

(Q2) *Can* CREDENCE *improve the flow completion times for short flows as well as long flows?*
We find that CREDENCE performs similar to existing approaches in terms of 95-percentile flow completion times for short flows and improves upon ABM by up to 22% correspondingly for long flows.

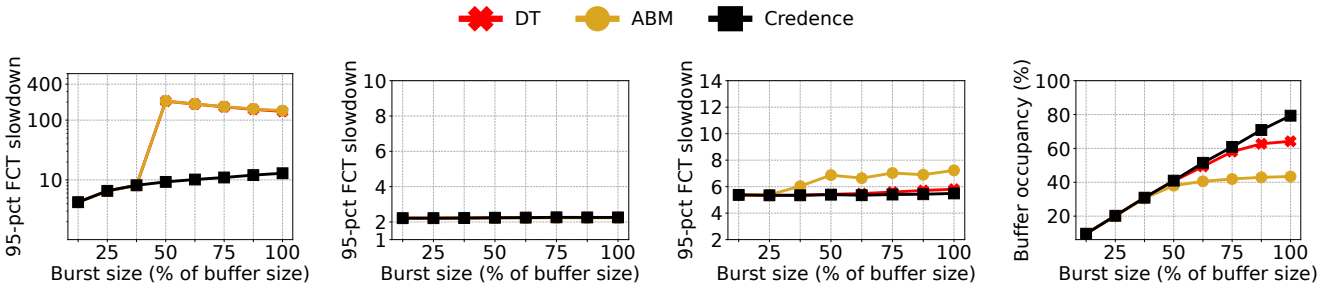
Q3 *How does prediction error impact the performance of* CREDENCE *in terms of flow completion times?*
We increase the error of our prediction by artificially flipping the predictions with a probability. As the probability increases (error increases), we find that CREDENCE sustains performance up to 0.01 probability and smoothly degrades in performance beyond 0.01.

4.1 Setup

Our evaluation is based on packet-level simulations in NS3 [44]. We embed a Python interpreter within NS3 using pybind11 [47] in order to obtain predictions from a random forest model trained with scikit-learn [49].

Topology: We consider a leaf-spine topology with 256 servers organized into 4 spines and 16 leaves. Each link has a propagation delay of 3μs leading to a round-trip-time of 25.2μs. The capacity is set to 10Gbps for all the links leading to 4 : 1 oversubscription similar to prior works [1, 3, 48]. All the switches in our topology have 5.12KB buffer-per-port-per-Gbps similar to Broadcom Tomahawk [16].

Workloads: We generate traffic using websearch [6] flow size distribution that is based on measurements from real-world datacenter workloads. We vary the load on the network in the range 20-80%. We additionally generate traffic using a synthetic incast workload similar to prior work [1]. Our incast workload mimics the query-response behavior of a



(a) Incast flows (b) Short flows (c) Long flows (d) Shared buffer occupancy

Figure 8: Performance of CREDENCE across various burst sizes of incast workload and websearch workload at 40% load, with PowerTCP as the transport protocol. Even with advanced congestion control, DT and ABM only benefit in terms of FCTs for long flows, but CREDENCE outperforms in terms of FCTs for incast flows (burst absorption) as well as FCTs for long flows.

distributed file storage system where each query results in a bursty response from multiple servers. We set the query request rate to 2 per second from each server, and we vary the burst size in the range 10-100%[5] of the switch buffer size. We use DCTCP [6] and PowerTCP [3] as transport protocols.

Comparisons & metrics: We compare CREDENCE with Dynamic Thresholds [20] (the default algorithm in datacenter switches), ABM [1] and LQD (push-out). Hereafter, we refer to Dynamic Thresholds as DT. We report four performance metrics: 95-percentile flow completion times for short flows ($\leq$ 100KB), incast flows (incast workload), long flows ($\geq$ 1MB), and the 99-percentile shared buffer occupancy. We present the CDFs of flow completion times in Appendix D.

Predictions: We collect packet-level traces from each switch in our topology when using LQD (push-out) as the buffer sharing algorithm. Each trace consists of five values corresponding to each packet arrival: *(i)* queue length, *(ii)* average queue length, *(iii)* shared buffer occupancy, *(iv)* average shared buffer occupancy and *(v)* accept (or drop). We train a random forest classifier using queue length and shared buffer occupancy as features and the model predicts packet drops. We set the maximum depth for each tree to 4 in view of practicality. At a train-test split 0.6 of our LQD trace, based on our parameter sweep across the number of trees used for our classifier (Figure 15 in Appendix D), we set the number of trees to 4. We observe that the quality of our predictions does not improve significantly beyond four trees in our datasets. This gives us an accuracy of 0.99, error score $\frac{1}{\eta}$ 0.99[6] (inverse of our error function based on Definition 1), precision of 0.65, recall of 0.35 and F1 score of 0.45. We defer the definitions of these prediction scores to Appendix C as they are standard in the literature. We train our model with an LQD trace corresponding to websearch workload at 80% load, and a burst size of 75% buffer size for the incast workload, using DCTCP as the transport protocol. We use the same trained model in all our evaluations. We ensure that our test scenarios are different from the training dataset by using different random seeds in addition to different traffic conditions (different loads and different burst sizes) in each experiment in our evaluation.

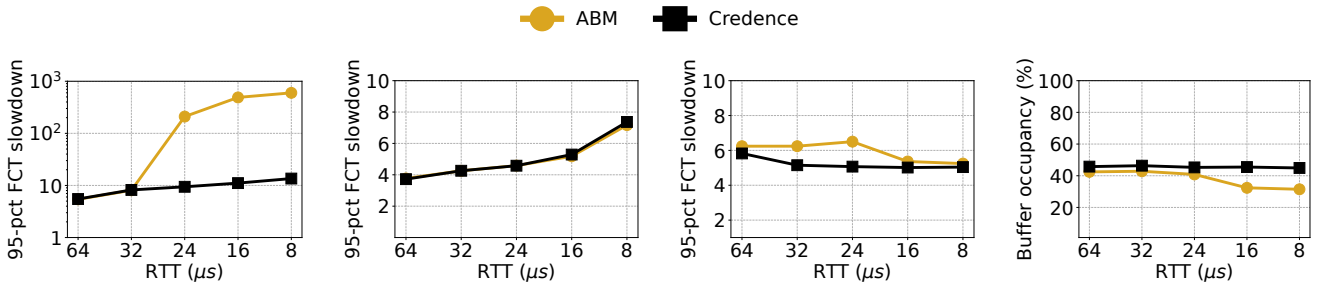
Configuration: CREDENCE is parameter-less, and it takes input from an oracle (described above) that predicts packet drops. We set $\alpha = 0.5$ for DT and ABM similar to prior work [1]. ABM uses $\alpha = 64$ for all the packets which arrive during the first round-trip-time [1]. We configure DCTCP according to [6] and PowerTCP according to [3].

4.2 Results

CREDENCE significantly improves burst absorption: In Figure 6a, using DCTCP as the transport protocol, we generate websearch traffic across various loads in the range 20-80% and generate incast traffic with a burst size of 50% buffer size. We observe that CREDENCE performs close to the optimal performance of LQD. CREDENCE improves the 95-percentile flow complete times for incast flows by 95.50% compared to DT, and by 95.53% compared to ABM. In Figure 7a, we set the load of websearch traffic at 40% and vary the burst size for incast workload in the range 10-100% buffer size. CREDENCE performs similar to DT and ABM for small burst sizes. As the burst size increases, CREDENCE improves the 95-percentile flow completion times for incast workload by 95% on average compared to DT, and by 96.92% on average compared to ABM. In Figure 8a, even when using PowerTCP as the transport protocol, we see that CREDENCE improves the 95-percentile flow completion times for incast flow by 93.27% on average compared to DT and by 93.36% on average compared to ABM. In terms of burst absorption, both DT and ABM are drop-tail algorithms, hence they face the drawbacks discussed in §2.2. CREDENCE relies on predictions and unlocks the performance of LQD (push-out) as shown by our results in Figure 6a and Figure 7a.

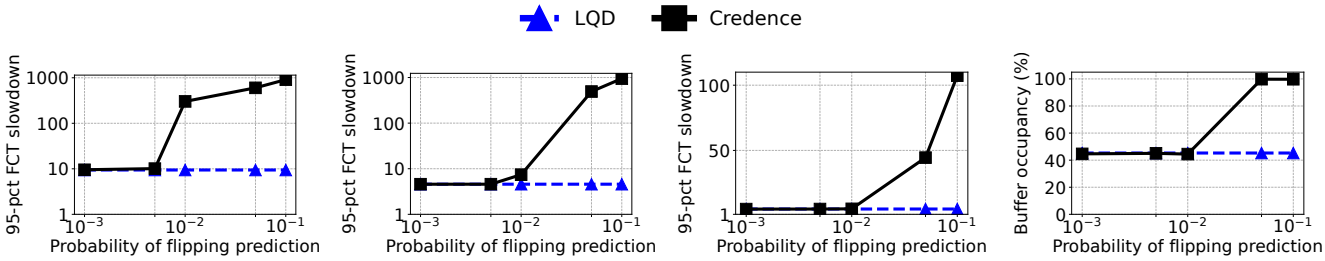
[5] We note that if the burst size exceeds the buffer size ($> 100\%$), then no buffer sharing algorithm can prevent excessive packet drops. As such, controlling and mitigating the extent of incast scenarios can be better addressed by congestion control and scheduling techniques.

[6] The high values of accuracy and our error score $\frac{1}{\eta}$ are attributed to the dataset being skewed i.e., congestion is not persistent.



(a) Incast flows (b) Short flows (c) Long flows (d) Shared buffer occupancy

Figure 9: ABM is sensitive to RTT and performs significantly worse compared to CREDENCE at low RTTs. At high RTTs, ABM performs similar to CREDENCE.



(a) Incast flows (b) Short flows (c) Long flows (d) Shared buffer occupancy

Figure 10: Even though the predictions from our random forest classifier are intentionally flipped (to increase error), CREDENCE performs close to LQD up to 0.005 flipping probability but smoothly diverges from LQD at 0.01 flipping probability.

CREDENCE improves long flows FCTs: CREDENCE not only improves the burst absorption but also improves the flow completion times for long flows. In Figure 6c, at 50% burst size for incast workload and across various loads of websearch workload, we observe that CREDENCE performs similar to DT in terms of 95-percentile flow completion times for long flows and improves upon ABM on average by 28.49%. At 80% load, CREDENCE improves upon ABM by 49.34%. Across various burst sizes, and at 40% load of websearch workload, we observe from Figure 7c that CREDENCE improves upon ABM by up to 22.02% and by 12.02% on average. With PowerTCP as the transport protocol (Figure 8c), CREDENCE improves the 95-percentile flow completion times for long flows by 3.31% on average compared to DT and by 17.35% compared to ABM. At a burst size of 100% buffer size, CREDENCE improves the flow completion times by 5.49% compared to DT and by 24.09% compared to ABM. As described in §2.2, drop-tail algorithms such as DT and ABM cannot effectively navigate proactive and reactive drops, resulting in throughput loss i.e., high flow completion times for long flows. In contrast, predictions guide CREDENCE to effectively navigate proactive and reactive drops. This allows CREDENCE to achieve better flow completion times even for long flows.

CREDENCE does not waste buffer resources: In anticipation of future burst arrivals, both DT and ABM buffer resources. We show the 99.99-percentile buffer occupancies[7] in Figure 6d for various loads of websearch workload and at a burst size of 50% buffer size for incast workload. We observe that, DT (ABM) utilizes 3.77% (18.68%) lower buffer space on average compared to CREDENCE, at the cost of increased flow completion times even for long flows. Even as the burst size increases (Figure 7d), DT and ABM are unable to efficiently utilize the buffer space. In contrast, CREDENCE efficiently utilizes the available buffer space as the burst size increases, improving burst absorption without sacrificing flow completion times for long flows.

ABM is sensitive to RTT: Although ABM is expected to outperform DT, our evaluation results especially in terms of flow completion times for incast flows contradict the results presented in [1]. We ran several simulations varying all the parameters in our setup in order to better understand the performance of ABM. We found that ABM is in fact sensitive to round-trip-time (RTT). We vary the base RTT of our topology in Figure 9 and compare CREDENCE with ABM. At high RTTs, we observe that ABM performs close to CREDENCE, but degrades in performance as RTT decreases. Specifically, at 8μs RTT, ABM performs 97.73% worse compared to CREDENCE in terms of flow completion times for incast flows. Although ABM achieves on-par flow completion

[7] DT, ABM and CREDENCE have similar tail occupancies (100-percentile) that occurs at rare congestion events in our simulations.

times for short flows, we observe that ABM degrades in flow completion times for long times as well as under-utilizes the buffer as RTT decreases. The poor performance of ABM at low RTTs is due to the fact that ABM prioritizes the first RTT packets and considers the rest of the traffic as steady-state traffic. However, it is not uncommon that datacenter switches experience bursts for several RTTs [22]. Further, congestion control algorithms require multiples RTTs to converge to steady-state. In contrast, CREDENCE is parameter-less and does not make such assumptions. CREDENCE performs significantly better than existing approaches even with an off-the-shelf machine-learned predictor with a simple model.

CREDENCE gradually degrades with prediction error: Our random forest classifier that we used in our evaluations so far, has a precision close to 0.65. In order to evaluate the performance of CREDENCE with even worse prediction error, we artificially introduce error by flipping every prediction obtained from our random forest classifier with a certain probability. We consider LQD (push-out) as a baseline since CREDENCE is expected to perform close to LQD and degrade as the prediction error grows large. Figure 10 presents our evaluation results, under websearch workload at 40% and burst size 50% of the buffer size for incast workload. At 0.001 flipping probability, CREDENCE performs close to LQD. However, at 0.01 flipping probability CREDENCE starts to diverge[8] from LQD and gets significantly worse at 0.1 flipping probability. Figure 10 gives practical insights into smoothness of CREDENCE in addition to our analysis.

5 Related Work

The buffer sharing problem has been widely studied for many decades. Research works in the literature range from push-out as well as drop-tail algorithms tailored for ATM networks [19,20,35,52,54] to more recent drop-tail algorithms tailored for datacenter networks [1, 2, 8, 11, 28, 50]. While we focus on the buffer sharing problem in this paper, several related but orthogonal approaches also tackle buffer problems in datacenter networks e.g., end-to-end congestion control [3, 6, 18, 25, 36, 38], AQM [21, 43, 45], scheduling [7, 27], packet deflection [56] and load-balancing [5, 23, 32]. These approaches aim at reducing congestion events and the overall buffer requirements, but they cannot fundamentally address buffer contention across multiple switch ports sharing the same buffer. Research on algorithms with predictions for various problems has recently been an active field of research [30, 39--42, 46] but ours is the first approach tackling the buffer sharing problem with predictions. Ongoing research efforts show the feasibility of deploying machine-learned predictions in the network data plane [4, 17, 31, 55].

[8] The extent of divergence in FCT slowdown relates to minRTO (set to 10 ms) due to packet drops that result in timeouts, and is not explicitly bounded by $O(N)$ similar to throughput. Yet, incast and short flows can be protected by incorporating packet priorities (discussed in §6).

6 Future Research Directions

CREDENCE is the first approach showing the performance benefits and guarantees by augmenting buffer sharing algorithms with predictions. This work barely scratches the surface and leaves intriguing open questions: *(i)* practically training the prediction oracle in the real-world, *(ii)* accounting for packet priorities in taking buffering decisions, *(iii)* integrating predictions with buffer sharing in hardware. We believe that this paper opens several interesting avenues for future work both in systems and theory. In this section, we discuss some of the future work directions to push approaches such as CREDENCE to be deployed in the real-world (§6.1), as well as to improve the performance guarantees offered by such approaches (§6.2).

6.1 Systems for In-Network Predictions

In this paper, we show how predictions can improve the performance of drop-tail buffer sharing. Many interesting systems research questions remain in order to integrate buffer sharing and predictions in the network data plane.

Training the model: Training a prediction oracle based on the model described in §2.3.1 involves collecting the buffering decisions (ground truth) of Longest Queue Drop algorithm (LQD), i.e., push-out, along with a set of features e.g., queue lengths, overall buffer occupancy. We envision two approaches to collect the training data in practice. First, similar to our approach in §4, the training data can be obtained from packet-level simulations that implement LQD on the switches. While this approach is simple, it has the limitation that the training data does not necessarily reflect real-world traffic arrivals and the corresponding buffering decisions of LQD. However, simulation-based training data could still be suitable for traffic patterns of a datacenter that can be explicitly modeled e.g., collective communications in GPU clusters [53, 58]. Second, the buffering decisions of the LQD algorithm can be exported by each switch in a real-world datacenter. This can be achieved by implementing LQD *virtually* without taking any buffering decisions on packets, along side any underlying buffer sharing algorithm i.e., maintaining per-queue *counters* that are incremented and decremented upon packet arrival, departure and drop (virtually based on LQD) events, similar to CREDENCE's thresholds. Any feature values exported must also correspond to these counter values. Exporting LQD's events even in a coarse-grained manner would suffice if the trace is sufficiently large. We believe that future work on implementing a virtual LQD (CREDENCE's thresholds) in hardware would not only strengthen CREDENCE's practicality but also improves the accuracy of the trained model, thereby also improving the performance. For simplicity, our random forest model in § 4 uses only four features, yet exploring the tradeoff between prediction error and model complexity in terms of space and time would be valuable. Further, the trained model must be simple enough that fits within the resources available in the data plane. Developing such trained models is an important step forward.

Deploying the model: Recent works propose practical implementations for in-network machine-learning models e.g., in the context of traffic classification [17]. P4 implementation of a model that predicts drops would enhance not only the practicality but also stimulate further research to design algorithms with performance guarantees better than CREDENCE.

Alternative predictions: As mentioned in §2.3, there are several different prediction models that can be considered for the buffer sharing problem. For instance, instead of predicting the drops, an oracle could predict packet arrivals just for a tiny window of the near future. Alternatively, techniques such as online reinforcement learning can be adopted, where rewards are based on how closely the queue lengths track their thresholds[9]. Systems research on studying the practicality and deployability of different prediction models is a valuable future direction that would better guide the design of algorithms with predictions for the buffer sharing problem.

Understanding push-out complexity: While push-out algorithms raised much interest initially, over the last years, research on this approach has been less active. We believe this is partly due to the lack of support from switch vendors. It is an open question how the complexity of obtaining drop predictions and the complexity of push-out fare against each other. Although we focused on augmenting drop-tail algorithms with predictions, we believe that our approach of using predictions has much potential also in other types of buffer algorithms. While switch vendors may be better informed about the complexity of push-out buffers, an understanding of this complexity in the scientific community is much needed in order to navigate the complexity vs performance spectrum.

6.2 Theory for Performance Guarantees

We believe the performance guarantees offered by CREDENCE can be improved in the future. Further, considering packet priorities and traffic classes in the competitive analysis is an open question.

Improving consistency and robustness: An open question is whether an algorithm could be designed to improve the competitive ratio under perfect predictions (consistency) better than 1.707, while also improving the ratio under large error (robustness) better than N. Further research in this direction would enable a better understanding whether a consistency-robustness tradeoff exists for the buffer sharing problem.

Competitive analysis with packet priorities: Literature in theory considers that all packets are of the same priority in the context of competitive analysis. One of the current limitations of CREDENCE is its obliviousness to packet priorities. It is well-known that preferential treatment of packets has various performance benefits, especially in terms of flow completion times when short flow packets are prioritized. The performance degradation of CREDENCE for short flows and incast flows (Figure 10a, 10b) can potentially be shielded from prediction errors by employing packet priorities. We believe that defining throughput (objective function) as the weighted sum of the number of transmitted packets of different priorities would enable the design of online algorithms that prioritize higher-priority packets, e.g., bursts or short flows, in order to be competitive. For instance, throughput can be defined as $\sum \alpha_p \cdot n_p$, where α_p is the relative importance of a priority p and n_p is the number of transmitted packets of priority p by a buffer sharing algorithm. To this end, developing analysis techniques for such a setup is an interesting future research direction.

[9] CREDENCE's thresholds are equivalent to LQD's (push-out) queue lengths for the same packet arrivals (see §3.2).

7 Conclusion

We presented CREDENCE, the first buffer sharing algorithm augmented with predictions that not only reaches close to optimal performance given low prediction error but also guarantees performance with arbitrarily large prediction error, while maintaining smoothness. We analytically proved our claims and our evaluations show the superior performance of CREDENCE even with an off-the-shelf machine-learned predictor, compared to the state-of-the-art buffer sharing algorithms. The building blocks required for CREDENCE are all individually practical in today's hardware. In future, we plan to pursue switch vendors to further discuss the integration of predictions with buffer sharing algorithm in hardware.

Acknowledgements

We would like to thank our shepherd, Marco Chiesa, as well as the anonymous reviewers for their useful feedback. This work is part of a project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, consolidator project Self-Adjusting Networks (AdjustNet), grant agreement No. 864228, Horizon 2020, 2020-2025.

References

[1] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. Abm: Active buffer management in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 36–52, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3544216.3544252.

[2] Vamsi Addanki, Wei Bai, Stefan Schmid, and Maria Apostolaki. Reverie: Low pass filter-based switch buffer sharing for datacenters with rdma and tcp traffic. In *21th USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, Santa Clara, CA, 2024. USENIX Association. URL: https://www.usenix.org/conference/nsdi24/presentation/addanki-reverie.

[3] Vamsi Addanki, Oliver Michel, and Stefan Schmid. PowerTCP: Pushing the performance limits of datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 51--70, Renton, WA, April 2022. USENIX Association. URL: https://www.usenix.org/conference/nsdi22/presentation/addanki.

[4] Aristide Tanyi-Jong Akem, Michele Gucciardo, and Marco Fiore. Flowrest: Practical flow-level inference in programmable switches with random forests. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pages 1--10, 2023. doi:10.1109/INFOCOM53939.2023.10229100.

[5] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 503–514, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2619239.2626316.

[6] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1851182.1851192.

[7] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. Pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 435–446, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2486001.2486031.

[8] Hamidreza Almasi, Rohan Vardekar, and Balajee Vamanan. Protean: Adaptive management of shared-memory in datacenter switches. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pages 1--10, 2023. doi:10.1109/INFOCOM53939.2023.10229046.

[9] Antonios Antoniadis, Matthias Englert, Nicolaos Matsakis, and Pavel Veselý. Breaking the Barrier Of 2 for the Competitiveness of Longest Queue Drop. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1--17:20, Dagstuhl, Germany, 2021. Schloss Dagstuhl -- Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2021/14086, doi:10.4230/LIPIcs.ICALP.2021.17.

[10] Maria Apostolaki, Vamsi Addanki, Manya Ghobadi, and Laurent Vanbever. Fb: A flexible buffer management scheme for data center switches. *arXiv preprint arXiv:2105.10553*, 2021.

[11] Maria Apostolaki, Laurent Vanbever, and Manya Ghobadi. Fab: Toward flow-aware buffer sharing on programmable switches. In *Proceedings of the 2019 Workshop on Buffer Sizing*, BS '19, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3375235.3375237.

[12] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering azure storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation*

(NSDI 23), pages 49--67, Boston, MA, April 2023. USENIX Association. URL: https://www.usenix.org/conference/nsdi23/presentation/bai.

[13] Wei Bai, Shuihai Hu, Kai Chen, Kun Tan, and Yongqiang Xiong. One more config is enough: Saving (dc)tcp for high-speed extremely shallow-buffered datacenters. *IEEE/ACM Transactions on Networking*, 29(2):489--502, 2021. doi:10.1109/TNET.2020.3032999.

[14] Ivan A. Bochkov, Alex Davydow, Nikita Gaevoy, and Sergey I. Nikolenko. New competitiveness bounds for the shared memory switch. *CoRR*, abs/1907.04399, 2019. URL: http://arxiv.org/abs/1907.04399.

[15] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. 1998.

[16] Broadcom. StrataXGS® Switch Solutions. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs.

[17] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. pforest: In-network inference with random forests. *arXiv preprint arXiv:1909.05680*, 2019. URL: http://arxiv.org/abs/1909.05680.

[18] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. Dcpim: Near-optimal proactive datacenter transport. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 53–65, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3544216.3544235.

[19] J.W. Causey and H.S. Kim. Comparison of buffer allocation schemes in atm switches: complete sharing, partial sharing, and dedicated allocation. In *Proceedings of ICC/SUPERCOMM'94 - 1994 International Conference on Communications*, pages 1164--1168 vol.2, 1994. doi:10.1109/ICC.1994.368919.

[20] A.K. Choudhury and E.L. Hahne. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Transactions on Networking*, 6(2):130--140, 1998. doi:10.1109/90.664262.

[21] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397--413, Aug 1993. doi:10.1109/90.251892.

[22] Ehab Ghabashneh, Yimeng Zhao, Cristian Lumezanu, Neil Spring, Srikanth Sundaresan, and Sanjay Rao. A microscopic view of bursts, buffer contention, and loss in data centers. In *Proceedings of the 22nd ACM Internet Measurement Conference*, IMC '22, page 567–580, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3517745.3561430.

[23] Soudeh Ghorbani, Zibin Yang, P. Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Drill: Micro load balancing for low-latency data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 225–238, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3098822.3098839.

[24] Michael H. Goldwasser. A survey of buffer management policies for packet switches. *SIGACT News*, 41(1):100–128, mar 2010. doi:10.1145/1753171.1753195.

[25] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. Backpressure flow control. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 779--805, Renton, WA, April 2022. USENIX Association. URL: https://www.usenix.org/conference/nsdi22/presentation/goyal.

[26] Ellen L. Hahne, Alexander Kesselman, and Yishay Mansour. Competitve buffer management for shared-memory switches. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '01, page 53–58, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/378580.378589.

[27] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing flows quickly with preemptive scheduling. *SIGCOMM Comput. Commun. Rev.*, 42(4):127–138, aug 2012. doi:10.1145/2377677.2377710.

[28] Sijiang Huang, Mowei Wang, and Yong Cui. Traffic-aware buffer management in shared memory switches. *IEEE/ACM Transactions on Networking*, 30(6):2559--2573, 2022. doi:10.1109/TNET.2022.3173930.

[29] Sungjin Im, Ravi Kumar, Mahshid Montazer Qaem, and Manish Purohit. Non-clairvoyant scheduling with predictions. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, page 285–294, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3409964.3461790.

[30] Sungjin Im, Benjamin Moseley, Chenyang Xu, and Ruilong Zhang. Online dynamic acknowledgement with learned predictions. *arXiv preprint arXiv:2305.18227*, 2023. URL: https://arxiv.org/abs/2305.18227.

[31] Syed Usman Jafri, Sanjay Rao, Vishal Shrivastav, and Mohit Tawarmalani. Leo: Online traffic classification at Multi-Terabit line rate. In *21th USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, Santa Clara, CA, 2024. USENIX Association.

[32] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, SOSR '16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2890955.2890968.

[33] Alexander Kesselman and Yishay Mansour. Harmonic buffer management policy for shared memory switches. *Theoretical Computer Science*, 324(2):161--182, 2004. Online Algorithms: In Memoriam, Steve Seiden. URL: https://www.sciencedirect.com/science/article/pii/S0304397504003779, doi:https://doi.org/10.1016/j.tcs.2004.05.014.

[34] Koji Kobayashi, Shuichi Miyazaki, and Yasuo Okabe. A tight bound on online buffer management for two-port shared-memory switches. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, page 358–364, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1248377.1248437.

[35] S. Krishnan, A.K. Choudhury, and F.M. Chiussi. Dynamic partitioning: a mechanism for shared memory management. In *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*, volume 1, pages 144--152 vol.1, 1999. doi:10.1109/INFCOM.1999.749262.

[36] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3387514.3406591.

[37] Jason Lei and Vishal Shrivastav. Seer: Future-Aware caching system for network processors. In *21th USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, Santa Clara, CA, 2024. USENIX Association.

[38] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3341302.3342085.

[39] Michael Mitzenmacher. A model for learned bloom filters and optimizing by sandwiching. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper_files/paper/2018/file/0f49c89d1e7298bb9930789c8ed59d48-Paper.pdf.

[40] Michael Mitzenmacher. Scheduling with predictions and the price of misprediction. *arXiv preprint arXiv:1902.00732*, 2019. URL: http://arxiv.org/abs/1902.00732.

[41] Michael Mitzenmacher. Queues with small advice. In *Proceedings of the 2021 SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*, pages 1--12. SIAM, 2021. doi:10.1137/1.9781611976830.1.

[42] Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions. *Commun. ACM*, 65(7):33–35, jun 2022. doi:10.1145/3528087.

[43] Kathleen Nichols and Van Jacobson. Controlling queue delay: A modern aqm is just one piece of the solution to bufferbloat. *Queue*, 10(5):20–34, may 2012. doi:10.1145/2208917.2209336.

[44] ns-3. Network Simulator. https://www.nsnam.org/.

[45] R. Pan, P. Natarajan, C. Piglione, M. S. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg. Pie: A lightweight control scheme to address the bufferbloat problem. In *2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR)*, pages 148--155, July 2013. doi:10.1109/HPSR.2013.6602305.

[46] Manish Purohit, Zoya Svitkina, and Ravi Kumar. Improving online algorithms via ml predictions. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper_files/paper/2018/file/73a427badebe0e32caa2e1fc7530b7f3-Paper.pdf.

[47] Pybind11. Seamless operability between C++11 and Python. https://pybind11.readthedocs.io/en/stable/.

[48] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa Ammar, Ellen Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani,

and Amin Vahdat. Annulus: A dual congestion control loop for datacenter and wan traffic aggregates. SIGCOMM '20, page 735–749, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3387514.3405899.

[49] scikit-learn. Machine Learning in Python. https://scikit-learn.org/stable/.

[50] Danfeng Shan, Wanchun Jiang, and Fengyuan Ren. Analyzing and enhancing dynamic threshold policy of data center switches. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2454--2470, 2017. doi:10.1109/TPDS.2017.2671429.

[51] Vishal Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 367–379, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3341302.3342090.

[52] B. Suter, T.V. Lakshman, D. Stiliadis, and A.K. Choudhury. Buffer management schemes for supporting tcp in gigabit routers with per-flow queueing. *IEEE Journal on Selected Areas in Communications*, 17(6):1159--1169, 1999. doi:10.1109/49.772451.

[53] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 739--767, Boston, MA, April 2023. USENIX Association. URL: https://www.usenix.org/conference/nsdi23/presentation/wang-weiyang.

[54] Guo-Liang Wu and J.W. Mark. A buffer allocation scheme for atm networks: complete sharing based on virtual partition. *IEEE/ACM Transactions on Networking*, 3(6):660--670, 1995. doi:10.1109/90.477712.

[55] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning? toward in-network classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 25–33, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3365609.3365864.

[56] Kyriakos Zarifis, Rui Miao, Matt Calder, Ethan Katz-Bassett, Minlan Yu, and Jitendra Padhye. Dibs: Just-in-time congestion mitigation for data centers. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2592798.2592806.

[57] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, page 78–85, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3131365.3131375.

[58] Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Prithwish Basu, Joud Khoury, and Arvind Krishnamurthy. Optimal direct-connect topologies for collective communications. *CoRR*, abs/2202.03356, 2022. URL: https://arxiv.org/abs/2202.03356, arXiv:2202.03356.

A Model and Definitions

We consider a network switch equipped with an on-chip buffer size of B units shared by N ports. We mainly follow the widely used model in the literature [9, 26, 33]. Time is discrete, and we refer to each step as timeslot. Packets (each of size unit 1) arrive in an online manner as time progresses. Each timeslot is divided into two phases, arrival phase and departure phase. During each arrival phase, at most N number of packets (in aggregate) arrive destined to N ports. During each departure phase, every queue drains out one packet unless the queue is empty. A buffer sharing algorithm manages the shared buffer allocation across the N ports. We next define preemptive (push-out) and non-preemptive (drop-tail) buffer sharing.

Definition 2 (Preemptive buffer sharing)**.** *During every arrival phase, the buffer sharing algorithm is allowed to preempt i.e., drop any number of existing packets in the buffer.*

Definition 3 (Non-preemptive buffer sharing)**.** *During every arrival phase, the buffer sharing algorithm is only allowed to accept or drop the incoming packet. Every accepted packet must eventually be drained out from the corresponding queue.*

We denote by $\sigma(t) = (\sigma_i(t), \sigma_i(t), ..., \sigma_N(t))$, an N-tuple, where $\sigma_i(t)$ denotes the number of packets arriving at time t to queue i. We study the performance of a buffer sharing algorithm in terms of throughput i.e., our objective is to maximize the total number of packets transmitted over the entire arrival sequence. We compare the performance of our online algorithms against an offline optimal algorithm, which has access to the entire arrival sequence at $t = 0$ and has infinite computational capacity.

Definition 4 (Competitive ratio)**.** *Let ALG be an online algorithm and OPT be an offline optimal algorithm for the buffer sharing problem. Let $ALG(\sigma)$ and $OPT(\sigma)$ be the total number of packets transmitted by ALG and OPT for the arrival sequence σ. We say ALG is c-competitive if it satisfies the following condition for any arrival sequence σ.*

$$OPT(\sigma) \leq c \cdot ALG(\sigma) \tag{3}$$

Algorithm 2: FollowLQD

```
Input  : σ(t)
procedure ARRIVAL(σ(t)):
    for each packet p ∈ σ(t) do
        Let i be the destination queue for the packet p
        UPDATETHRESHOLD(i, arrival)
        if q_i(t) < T_i(t) then
            if Q(t) < B then
                q_i(t) ← q_i(t) + 1                    ▷ accept
        else
            ▷ Drop
procedure DEPARTURE(i):
    if q_i(t) > 0 then
        q_i(t) ← q_i(t) − 1                            ▷ Drain one packet
    UPDATETHRESHOLD(i, departure)
function UPDATETHRESHOLD(i, event):
    if event = arrival then
        if Γ(t) = B then                               ▷ Sum of thresholds
            Let T_j(t) be the largest threshold
            T_j(t) ← T_j(t) − 1                        ▷ Decrease
            T_i(t) ← T_i(t) + 1                        ▷ Increase
        else
            T_i(t) ← T_i(t) + 1                        ▷ Increase
            Γ(t) ← Γ(t) + 1
    if event = departure then
        if T_i(t) > 0 then
            T_i(t) ← T_i(t) − 1                        ▷ Decrease
            Γ(t) ← Γ(t) − 1
```

B FollowLQD: A Deterministic Algorithm

In this section we propose a new online deterministic algorithm FollowLQD in the non-preemptive case which is a non-predictive building block of CREDENCE. Intuitively, FollowLQD simply *follows* the Longest Queue Drop (LQD) queues in the preemptive model. In particular, FollowLQD maintains a threshold $T_i(t)$ for each queue at time t. The thresholds are updated for every packet arrival and departure according to LQD in the preemptive model. We present the pseudocode for FollowLQD in Algorithm 2. While FollowLQD tries to *follow* LQD queue lengths by accepting packets as long as the queue lengths are smaller than the thresholds, it may happen that FollowLQD queues are larger than their thresholds. This is since FollowLQD cannot preempt (remove) existing packets in the buffer whereas LQD can preempt and correspondingly the thresholds may drop below the queue lengths. FollowLQD simply drops an incoming packet if it finds that the corresponding queue exceeds its threshold.

Although LQD is known to be 1.707-competitive, we show that FollowLQD is still at least $\frac{N+1}{2}$-competitive. We present our lower bound based on a simple arrival sequence.

Observation 1. *FollowLQD is at least $\frac{N+1}{2}$-competitive.*

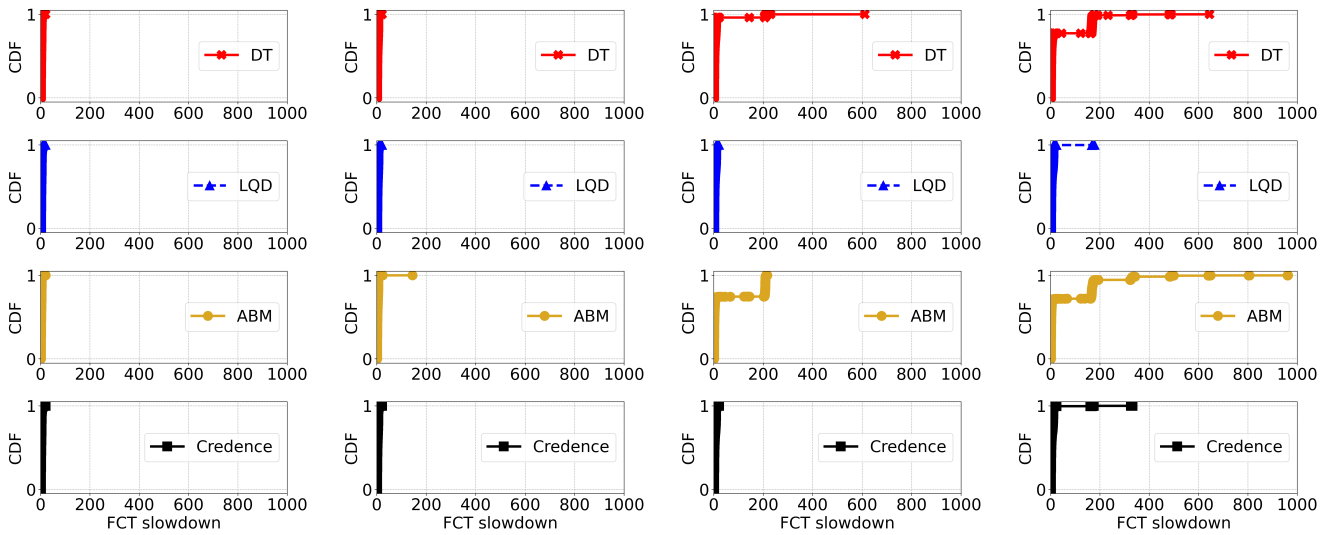
Proof. We construct an arrival sequence such that for every two packets transmitted by FollowLQD, the offline optimal algorithm OPT transmits $N+1$ packets. Consider that all the queues are empty at time $t=0$. We then burst packets to a single queue say i until its queue length reaches B. Note that this is possible since the threshold for queue i that follows the corresponding LQD queue also grows up to B. At the end of the departure phase, FollowLQD transmits one packet and the queue length becomes $B-1$. At this point, we send N packets, one packet to each of the N queues. The thresholds are updated based on LQD, which has the following actions: *(i)* preempt $N-1$ packets from queue i and *(ii)* accept all N packets to N queues. Correspondingly, the threshold for queue i of FollowLQD drops to $B-N+1$ but it still has $B-1$ packets in queue i. As a result, it can only accept one packet out of the N incoming packets. At the end of the departure phase during this timeslot, FollowLQD has $B-1$ packets in queue i and has transmitted 1 packet in total. In the next timeslot, we send N packets to the queue i so that LQD's queue i now gets back to size B again. As the threshold is larger than the queue length $(B-1)$, FollowLQD accepts 1 packet. At the end of the departure phase, FollowLQD transmits 1 packet from the queue i. Overall, FollowLQD transmitted 2 packets but OPT transmitted $N+1$ packets. We then repeat the sequence such that for every $N+1$ packets transmitted by OPT, FollowLQD transmits 2 packets. The competitive ratio is then at least $\frac{N+1}{2}$. □

C Buffer Sharing Algorithms with Predictions

In this section, we introduce our model for buffer sharing where there exists an oracle that predicts packet drop (or accept) for each packet in the arrival sequence σ, according to the prediction model introduced in §2.3.1. We denote the drop sequence of LQD for the arrival sequence σ by $\phi(\sigma)$, and the predicted drop sequence by $\phi'(\sigma)$. We classify prediction for each packet in to four types: true positive, false positive, true negative and false negative (see Figure 5. We denote the sequence of true positive predictions by $\phi'_{TP}(\sigma)$, false positive predictions by $\phi'_{FP}(\sigma)$, true negative predictions by $\phi'_{TN}(\sigma)$ and false negative predictions by $\phi'_{TP}(\sigma)$. We drop σ in our notations when the context is clear.

Hereafter, we mainly compare our online non-preemptive algorithm with predictions against online LQD (preemptive). We define the error made by the oracle by the following error function.

Definition 1 (Error function)**.** *Let LQD(σ) and FollowLQD(σ) denote the total number of packets transmitted by the online push-out algorithm LQD and the online drop-tail algorithm FollowLQD over the arrival sequence σ. Let ϕ denote the sequence indicating drop by LQD for each packet in the arrival sequence σ. Let ϕ' denote the sequence of drops predicted by the machine-learned oracle. Let ϕ'_{TP}, ϕ'_{FP}, ϕ'_{TN}, and ϕ'_{FN} denote the sequence of true positive, false positive, true negative and false negative predictions for the arrival*



(a) Burst size = 12.5% (b) Burst size = 25% (c) Burst size = 50% (d) Burst size = 75%

Figure 11: CDF of flow completion times (slowdown) for CREDENCE, DT, ABM and LQD across various burst sizes of incast workload and websearch workload at 40% load, with DCTCP as the transport protocol. Burst size is expressed as a percentage of the buffer size.

sequence σ. *We define the error function* $\eta(\phi,\phi')$ *as follows:*

$$\eta(\phi,\phi') = \frac{LQD(\sigma)}{FollowLQD\left(\sigma - \phi'_{TP} - \phi'_{FP}\right)} \tag{1}$$

We now analyze CREDENCE that relies on drop predictions ϕ' and takes decisions in pursuit of *following* LQD more accurately. Algorithm 1 presents the pseudocode for our algorithm.

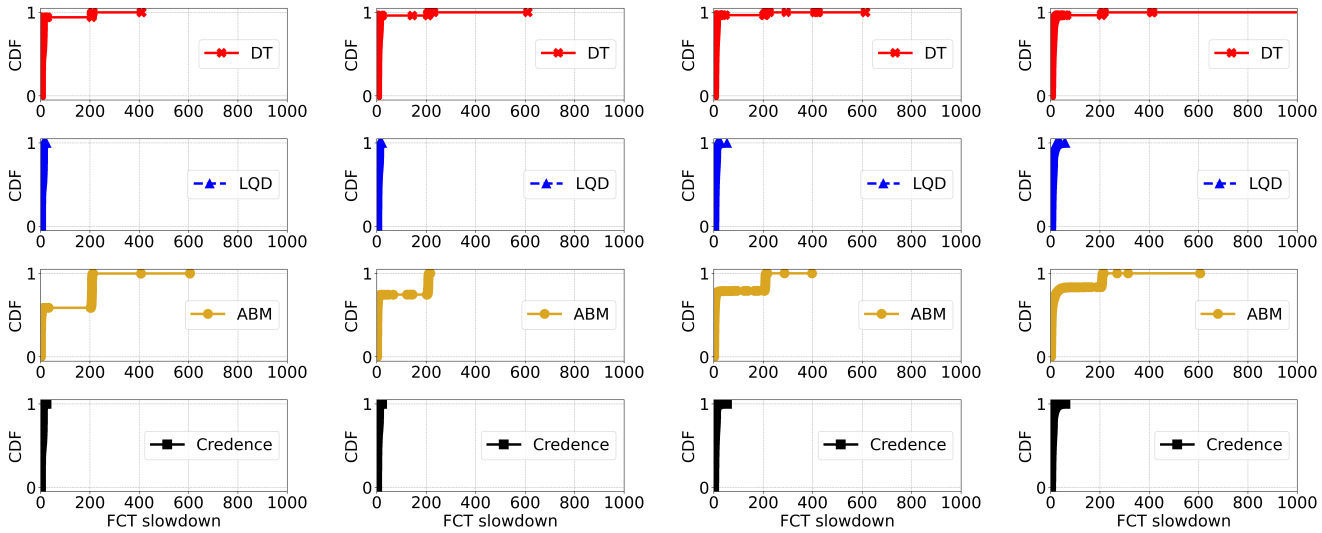
In essence, perfect predictions allows us to perfectly follow LQD queues, essentially transmitting as many packets as LQD. However, we are also concerned about the performance of the algorithm when the oracle makes mispredictions. In the following, we study the competitive ratio of CREDENCE as the error grows. We obtain the competitive ratio as a function of the error η: we show that CREDENCE is 1-competitive against LQD with perfect predictions but at most N-competitive when the error is arbitrarily large.

Theorem 1. *The competitive ratio of* CREDENCE *grows linearly from* 1.707 *to* N *based on the prediction error* $\eta(\phi,\phi')$, *where* N *is the number of ports,* ϕ *is the drop sequence of LQD and* ϕ' *is the predicted sequence of drops i.e., the competitive ratio is at most* $\min(1.707\,\eta(\phi,\phi'),N)$.

Lemma 1. *The total number of packets transmitted by* CREDENCE *for an arrival sequence* σ, *a drop sequence* ϕ *by LQD and the predicted drop sequence* ϕ' *is given by*

$$\text{CREDENCE}(\sigma) \geq \frac{LQD(\sigma)}{\underbrace{\eta(\phi,\phi')}_{error}} \tag{2}$$

Proof. For simplicity, we refer to CREDENCE as ALG in the following. We prove our claim by analyzing the drops of ALG and relating the transmitted packets by $ALG(\sigma)$ to $LQD(\sigma')$. Every drop of *ALG* arises from three types of situations. First, ALG can drop a packet due to the thresholds. Note that the thresholds used by *ALG* correspond to the queue lengths of preemptive LQD over the same arrival sequence σ. As a result, both ALG and FollowLQD algorithm have the same thresholds at any time instance. Second, ALG drops a packet if the prediction is either true positive or false positive if and only if the queue length satisfies the corresponding thresholds. This type of drops are at most the total number of true positive and false positive predictions. Third, ALG drops a packet when the buffer is full which is the same condition for FollowLQD. In essence, ALG drops at most all the positive predictions and drops at most the number of packets dropped by FollowLQD serving the arrival sequence $\sigma - \phi'_{TP} - \phi'_{FP}$ i.e., the arrival sequence in which all the packets predicted as positive are removed from σ. In order to prove our main claim, it remains to argue that the extra packets accepted by ALG due to the safeguard condition do not result in additional drops compared to FollowLQD with the arrival sequence $\sigma - \phi'_{TP} - \phi'_{FP}$. For every packet that fails to satisfy the threshold but gets accepted due to the safeguard condition by ALG, could cause at most one extra drop due to the thresholds before the buffer full again compared to FollowLQD. This is since, if FollowLQD accepts a packet, then its queue length is certainly less that the corresponding threshold (that is same for ALG). However, the queue length of ALG may have some extra packets that are accepted due to the safeguard condition. As a result, each such



(a) Load = 20% (b) Load = 40% (c) Load = 60% (d) Load = 80%

Figure 12: CDF of flow completion times (slowdown) for CREDENCE, DT, ABM and LQD across various loads of websearch workload and incast workload at a burst size 50% of the buffer size, with DCTCP as the transport protocol.

extra packet (dropped by FollowLQD) contributes to at most one drop compared compared to FollowLQD and the transmitted packets remains equivalent. Further, by the time the buffer is full in ALG, all the extra packets accepted due to the safeguard condition would have been drained out of the buffer. This is due to the fact that any such extra packet is at a queue length of at most $\frac{B}{N}$ (the safeguard condition) that drains out before the buffer fills up i.e., it takes at least $\frac{B}{N}$ timeslots to fill the buffer (only N packets can arrive in each timeslot). As a result, ALG transmits at least the total number of packets transmitted by FollowLQD over the arrival sequence $\sigma - \phi'_{TP} - \phi'_{FP}$ i.e.,

$$ALG(\sigma) \geq FollowLQD(\sigma - \phi'_{TP} - \phi'_{FP})$$

Using Definition 1, we express FollowLQD in terms of LQD and the error function $\eta(\phi,\phi')$, and obtain Equation 2. □

Lemma 2. CREDENCE *transmits at least* $\frac{1}{N}$ *times the number of packets transmitted by an offline optimal algorithm OPT i.e.,* $\text{CREDENCE}(\sigma) \geq \frac{1}{N} \cdot OPT(\sigma)$.

Proof. Irrespective of the predictions, CREDENCE always accepts an incoming packet if the longest queue is less than or equal to $\frac{B}{N}$. When CREDENCE drops a packet, there is at least one queue that has at least $\frac{B}{N}$ number of packets. Hence, every packet in OPT can be matched to at least $\frac{B}{N}$ number of packets. Consequently, the competitive ratio is at most N. □

We are now ready to prove our main claim (Theorem 1) using the above results.

Proof of Theorem 1. From Definition 4, in order to prove the competitive ratio of our CREDENCE, we are mainly concerned with the upper bound of $\frac{OPT(\sigma)}{\text{CREDENCE}(\sigma)}$ for any arrival sequence σ. Since $\frac{OPT(\sigma)}{LQD(\sigma)} \leq 1.707$ is known from literature [9, 26], we use this result to compare CREDENCE and LQD in order to argue about the competitive ratio i.e, $\frac{OPT(\sigma)}{\text{CREDENCE}(\sigma)} \leq 1.707 \cdot \frac{LQD(\sigma)}{\text{CREDENCE}(\sigma)}$ for any request sequence σ. From Lemma 1, we have the following:

$$\frac{LQD(\sigma)}{\text{CREDENCE}(\sigma)} \leq \eta(\phi,\phi')$$

From Lemma 2, irrespective of the predicted sequence, we have that $\frac{OPT(\sigma)}{\text{CREDENCE}(\sigma)} \leq N$. Finally, since $\frac{OPT(\sigma)}{\text{CREDENCE}(\sigma)} \leq 1.707 \cdot \frac{LQD(\sigma)}{\text{CREDENCE}(\sigma)}$, the competitive ratio of CREDENCE is given by:

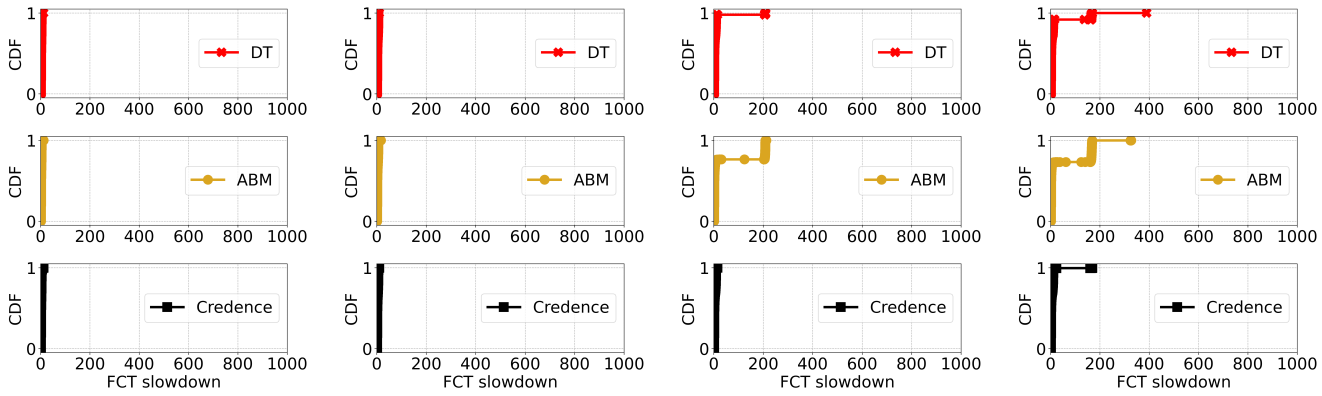
$$\frac{OPT(\sigma)}{\text{CREDENCE}(\sigma)} \leq \min\left(1.707\,\eta(\phi,\phi'), N\right)$$

The proof follows by Definition 4. □

Theorem 2. *Let* ϕ' *denote the sequence of drops predicted by the machine-learned oracle. Let* ϕ'_{TP}, ϕ'_{FP}, ϕ'_{TN}, *and* ϕ'_{FN} *denote the sequence of true positive, false positive, true negative and false negative predictions for the arrival sequence* σ. *The error function* $\eta(\phi,\phi')$ *(Definition 1) is upper upper bounded as follows:*

$$\eta(\phi,\phi') \leq \frac{\phi'_{TN} + \phi'_{FP}}{\phi'_{TN} - \min\left((N-1)\cdot\phi'_{FN}, \phi'_{TN}\right)}$$

Proof. Our proof is based on two arguments: *(i)* $LQD(\sigma) = \phi'_{TN} + \phi'_{FP}$ and *(ii)* $FollowLQD(\sigma - \phi'_{TP} - \phi'_{FP}) \geq \phi'_{TN} - \min\left((N-1)\cdot\phi'_{FN}, \phi'_{TN}\right)$.



(a) Burst size = 12.5% (b) Burst size = 25% (c) Burst size = 50% (d) Burst size = 75%

Figure 13: CDF of flow completion times (slowdown) for CREDENCE, DT, ABM and LQD across various burst sizes of incast workload and websearch workload at 40% load, with PowerTCP as the transport protocol. Burst size is expressed as a percentage of the buffer size.

First, $LQD(\sigma)$ is the total number of transmitted packets by LQD. Recall that the ground-truth (transmitted by LQD) for a prediction is an accept if and only if the prediction is either true negative or a false positive. Hence, the total number of packets transmitted by LQD i.e., $LQD(\sigma)$ is the sum of true negative predictions and the false positive predictions.

Second, $FollowLQD(\sigma - \phi'_{TP} - \phi'_{FP})$ transmits at least $\phi'_{TN} + \phi'_{FN} - Y$, where Y is the total number of drops caused by false negative predictions. Note that $\sigma = \phi'_{TN} + \phi'_{FN} + \phi'_{FP} + \phi'_{TP}$. The proof follows by showing that each false negative results in at most N extra drops due to the buffer limit. Further, these extra drops *must* be true negative predictions since we have already removed positive predictions from our arrival sequences (i.e., we assume at most all the positive predictions have be dropped). Additionally, since the extra drops are true negative predictions, it implies that LQD transmits those packets but although our prediction is true, we incur additional drop due to the buffer limit. For each false negative, there can be at most one drop in a single timeslot for up to $N-1$ distinct timeslots such drops that LQD accepts and transmits those packets but FollowLQD drops them. Beyond $N-1$ drops, there can only be at most 1 other drop upon which the existence of an additional packet (false negative) in FollowLQD's buffer would be nullified. This is since, during the initial $N-1$ drops, FollowLQD could not accept the incoming packet but after the transmission phase, the queues having false negative predictions decrement their size by 1. This leaves at least $N-1$ packets free space in FollowLQD after $N-1$ drops and LQD also has the same remaining space after those extra $N-1$ accepted by LQD are also transmitted. At this time, both LQD and FollowLQD have the same remaining space and they also transmit the same number of packets in each timeslot. One additional drop by FollowLQD corresponding to a false negative is still possible due the thresholds i.e., if there exists a packet arrival to the queue having false negative, the incoming packet is dropped since the existence of false negatives implies that the queue length is large than the threshold. As a result, there are at most N drops by FollowLQD for each false negative prediction.

The proof follows by the above two arguments. □

For completeness, although well-known in the literature, we define accuracy, precision, recall and f1 score below (used in Figure 15 in §4).

$$Accuracy = \frac{\phi'_{TP}+\phi'_{TN}}{\phi'_{TP}+\phi'_{TN}+\phi'_{FP}+\phi'_{FN}}$$

$$Precision = \frac{\phi'_{TP}}{\phi'_{TP}+\phi'_{FP}}$$

$$Recall = \frac{\phi'_{TP}}{\phi'_{TP}+\phi'_{FN}}$$

$$F1\ score = \frac{2\cdot\phi'_{TP}}{2\cdot\phi'_{TP}+\phi'_{FP}+\phi'_{FN}}$$

D Additional Results

In this section, we present additional results from our evaluations. Figures 11, 12, 13 present the CDF of flow completion times for each experiment in our evaluations (§4), showing the complete performance profile of each algorithm.

Figure 14 presents our numerical results based on a custom simulator in discrete time. Note that Figure 14 shows the throughput *ratio* of an algorithm vs LQD. We perform this experiment using custom simulator in order to fully control the prediction error (artificially).

We generate large bursts of the size of the total buffer, where each such burst arrives according to a poisson process (which is fixed in subsequent runs). We then collect a trace of per-packet drop (or accept) trace using LQD as the buffer sharing algorithm. This trace serves as the ground-truth as well

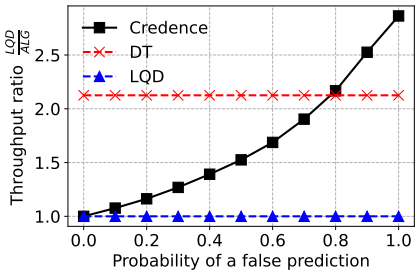


Figure 14: As the probability of false predictions increases, CREDENCE's throughput compared to LQD (push-out) i.e., the ratio $\frac{LQD}{ALG}$ increases from 1 to 2.9 (lower values are better). CREDENCE performs significantly better than DT even when the probability of false predictions is as high as 0.7.

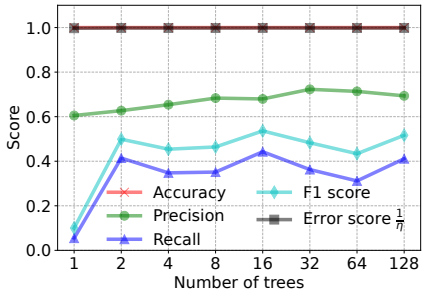


Figure 15: The quality of our predictions does not improve significantly beyond 4 trees in our random forest classifier.

as the case for perfect predictions for CREDENCE. We then run CREDENCE over the same packet arrival sequence from above, and use the drop trace of LQD as predictions. With full access to this trace i.e., perfect predictions case, CREDENCE performs exactly as LQD as expected. However, in order to study the performance of CREDENCE with increasing error, in a controlled manner, we flip each packet drop (or accept) from our LQD's drop trace i.e., each flip becomes a false prediction. We control the error via the flipping probability i.e., the false prediction rate. We observe from Figure 14 that CREDENCE degrade in throughput as the probability of false predictions increases i.e., as the prediction error increases. However, even at as high as 0.7 probability of false predictions, CREDENCE still out-performs DT.

In Figure 15, we present our results obtained from a parameter sweep across the number of trees used for random forest model vs prediction scores.

Seer: Enabling Future-Aware Online Caching in Networked Systems

Jason Lei
Purdue University

Vishal Shrivastav
Purdue University

Abstract

State-intensive network and distributed applications rely heavily on online caching heuristics for high performance. However, there remains a fundamental performance gap between online caching heuristics and the optimal offline caching algorithm due to the lack of visibility into future state access requests in an online setting. Driven by the observation that state access requests in network and distributed applications are often carried in incoming network packets, we present Seer, an online caching solution for networked systems, that exploits the delays experienced by a packet inside a network – most prominently, transmission and queuing delays – to notify in advance of future packet arrivals to the target network nodes (switches/routers/middleboxes/end-hosts) implementing caching. Using this as a building block, Seer presents the design of an online cache manager that leverages visibility into (partial) set of future state access requests to make smarter prefetching and cache eviction decisions. Our evaluations show that Seer achieves up to 65% lower cache miss ratio and up to 78% lower flow completion time compared to LRU for key network applications over realistic workloads.

1 Introduction

Online caching is a key component of every class of computer systems, ranging from microprocessors [19] to file and storage systems [35] to networked and distributed systems [8, 15, 37]. It is well-known that the optimal (offline) caching algorithm for minimizing cache misses, namely Belady [7], requires visibility into all future access requests, thus making it impractical in an online setting. Hence, there have been several works [6, 13, 30, 56, 59, 60] over the past several decades designing online caching heuristics that closely emulate Belady. However, there remains a fundamental gap between the performance of online caching heuristics and the optimal offline algorithm due to the lack of effective mechanisms to provide visibility into future access requests in an online setting.

In this paper, we present Seer, that aims to bridge the performance gap between online and optimal offline caching in a networked system, by providing a perfectly accurate visibility into (partial) set of future access requests to the target network nodes (switches/routers/middleboxes/end-hosts) implementing caching. We assume that the target nodes have a small cache with a larger backing store, and run network/distributed applications that operate over large amounts of state that may not fit entirely in the cache. Examples of such applications include virtual switching [15, 37], stateful load balancing [31], NATs and firewalls [25], receive-side host network stack processing [10], CDN caching [8], distributed key-value [34], network monitoring [33], content-based networking [9], and network intrusion detection [21]. Further, we note that the state access requests in such applications are often carried in incoming network packets, e.g., NAT accesses the address translation table based on the address carried in incoming packets, virtual switch and stateful load balancer access the flow table based on the flow id carried in received packets, and CDN server accesses the content based on content id carried in incoming client packets. *Thus there lies an opportunity to provide visibility into future access requests to target nodes implementing caching for such applications, if only one could notify them of the state access metadata carried in future incoming packets well in advance before those packets arrive.*

To achieve this, Seer leverages the fact that packets experience various forms of delays in the network – most prominently, transmission and queuing delays – and while the packets are waiting at a network node to be transmitted, one could put that delay to good use by notifying in advance the target nodes implementing caching about the future incoming packets. More specifically, in Seer, the directly connected neighbors of each target node continuously forward the relevant state access metadata carried in the packets (e.g., the flow id or the object id) for all the packets in their egress queues to the respective target nodes. Thus, while the packets are still waiting in the neighbor queues for transmission, the forwarded metadata about the future state access requests allow the target nodes to make closer to optimal caching decisions in terms of what to prefetch to the cache and what to evict from the cache.

However, implementing the above idea in practice requires solving several key challenges.

Challenge # 1. *Notifying of future state access requests in a timely manner with low bandwidth overhead.*

A neighbor node must forward the relevant state access metadata in every queued packet destined to the target node as soon as possible, to provide the target node maximum visibility into future access requests. However, doing this naively would require generating a control packet corresponding to every packet destined to the target node, thus resulting in high bandwidth overhead.

In §3.1, we describe Seer's solution to reduce the bandwidth overhead of control packets. The key idea is to leverage the inter-packet gap (IPG) to exchange control information at zero bandwidth overhead.

Challenge # 2. *Caching with partial future visibility.*

In an online system, it is impractical to provide full visibility into all future access requests. Hence fundamentally, Seer could only provide a partial visibility into future requests, and in practice, at any given time, Seer only provides visibility into the future requests currently queued at the directly connected neighbors of a target node. This presents the challenge of designing a cache manager for a new caching design point, that sits somewhere in between the two previously explored design points of optimal offline caching (that assumes full visibility into future access requests) and online caching heuristics (that assume no visibility into future access requests).

In §3.2, we describe the design of Seer's cache manager. The cache manager implements cache prefetching and cache eviction algorithms in Seer. For prefetching, the cache manager uses the knowledge of future access requests to prefetch corresponding state to the cache in the order of future request arrival. This reduces cache misses for future arrivals. For cache eviction, Seer *dynamically* combines a default online caching heuristic[1] with Belady's optimal offline cache eviction policy, based on the current degree of visibility into future requests. In the best case, Seer emulates Belady, while in the worst case, Seer simply reduces to the default online caching heuristic.

Challenge # 3. *Limited time budget for caching decisions.*

Seer's cache manager has a limited time budget to make the prefetching and cache eviction decisions, as determined by the access time of the backing store. In particular, Seer must be able to make caching decisions in lesser time than the backing store access time in order to ensure that fetching data from the backing store remains the bottleneck for cache replacement throughput. To make matters worse, the time budget for caching decisions can be as small as 10s to 100s of nanoseconds for a backing store such as DRAM, which is a common backing store for several state-intensive network applications [15, 31, 38].

In §4, we describe the hardware implementation of Seer's cache manager, that exploits massive hardware parallelism to implement both the prefetching and cache eviction algorithms in a total of $log(P)+2k+1$ clock cycles, where k is the cache set size (in a set associative cache) and P is the number of ingress ports in the target network node implementing caching. In practice, this translates to ∼100 ns latency on FPGA/NIC target and ∼25 ns latency on ASIC switch/router target.

We implement and prototype Seer's design on an FPGA (§5). Our evaluations on a small hardware testbed show that Seer achieves up to 80% fewer cache misses compared to LRU while remaining within 20% of Belady. Based on larger-scale network simulations (§6), we show that Seer achieves up to 65% lower cache miss ratio and up to 78% lower flow completion time compared to LRU for key network applications over realistic workloads.

[1] Any existing online caching heuristic can be used for this purpose.

2 Seer: Overview and Insights

Seer is an online caching system for state-intensive network and distributed applications. Seer is designed for networked systems, where a subset of network nodes implement online caching (called "target nodes"). A network node in Seer could be any networking device, including switches, routers, middleboxes, or end-host NICs and processors.

Network node model. Seer assumes an abstract model of a network node, where a node has N ingress and egress ports. To support differential service, each egress port has k priority classes implemented using k *strict priority* FIFO queues, i.e., packets from a lower priority FIFO are scheduled if and only if all the higher priority FIFOs are empty. Strict priority FIFOs are widely implemented in modern networking devices due to their scalability, e.g., datacenter switches typically have 8 strict priority FIFO classes [18]. Further, recent works [2] have shown that strict priority FIFOs are also expressive and can implement wide range of packet scheduling algorithms. Finally, each target node in Seer is assumed to have a small local cache (e.g., SRAM) and a larger backing store (e.g., DRAM). The backing store could either be local or remote.

Seer's protocol runs between a target node and its directly connected neighbor node(s), with the goal to provide the target node a visibility into future state access requests. The objective in Seer is to **minimize cache misses** using the future visibility. Seer's design is based on following key insights.

Insight # 1. *For network/distributed applications, state access requests are often carried in incoming network packets.*

State accesses in several key network and distributed applications are triggered by incoming network packets. The state in these applications are typically indexed by some metadata carried in the incoming packets, e.g., address translation table in NAT is indexed by the address field of incoming packets, flow table in a virtual switch or a stateful load balancer is indexed by the flow id (such as hash of 5-tuple) of incoming packets, and the content store in a CDN is indexed by the content id carried in client packets. Thus, by notifying the target nodes in advance of the state access metadata carried in future incoming packets, one could provide the target nodes a visibility into future state access requests.

Insight # 2. *Network delays can be leveraged to provide visibility into future state access requests.*

In a networked system, a packet typically hops through multiple network nodes before arriving at the destination. At each hop, the packet can experience various delays in the form of transmission, propagation, processing, and queuing delays. The key insight in Seer is that these delays can be leveraged to notify the target nodes of future state access requests. Suppose a packet p arrives at a node Y at time 0, and is destined to target node X after a further network delay of T time units. Thus, if Y notifies X of the state access metadata in p by time $t < T$, X will get visibility into a future state access request.

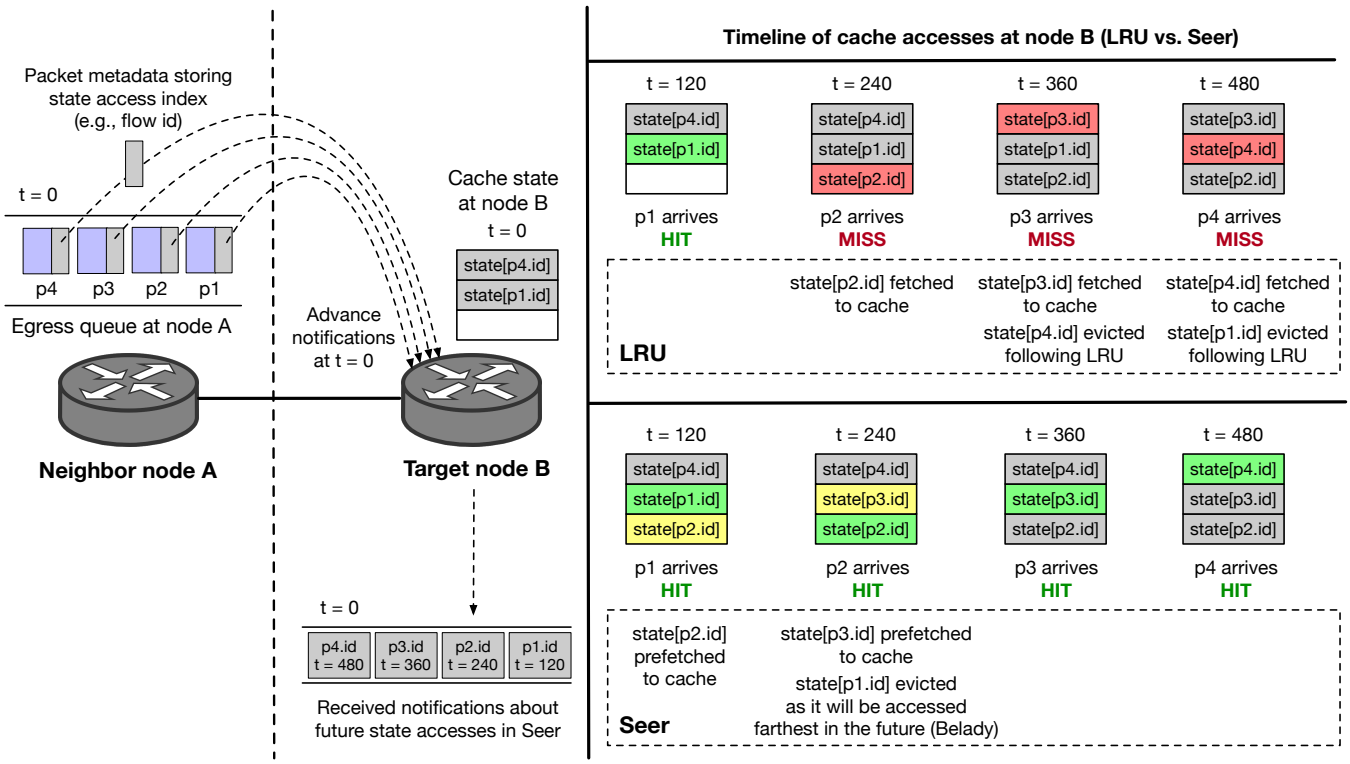


Figure 1: *A toy example illustrating the potential of Seer in terms of minimizing cache misses over Least Recently Used (LRU) [56] policy. Example assumes two directly connected nodes A and B. Node A has four packets $p1, p2, p3, p4$ in its egress queue. Transmission delay of each packet is 120 ns and propagation delay is 0. The time to fetch a state from the backing store is 100 ns (e.g., DRAM). In Seer, node B has already received notifications from node A at time $t = 0$ about the future state accesses along with the expected time of arrival of corresponding packets. Seer is able to use this information to minimize cache misses by (i) prefetching states into the cache well before the packets that will access those states arrive (done at t=120 and t =240), and (ii) evicting cache entries that will be accessed farthest in the future (done at t=240) following Belady's optimal offline algorithm.*

Insight # 3. *Even small network delays can provide enough visibility into future requests to result in significant gains.*

We illustrate this insight using an example. Consider an extreme case of zero queuing delay, where an MTU-sized packet p arrives at $t = 0$ in an empty egress queue of node X. Over a 100 Gbps link, it would take 120 ns to transmit this packet. Thus the packet will reach the next hop node Y at $t = 120 + \varepsilon$ ns, where ε is the propagation delay. Hence, if Seer could notify Y of the arrival of p at $t = 0$, Y could potentially prefetch the state p would access into the cache before p arrives at Y, assuming a backing store such as DRAM with access time of 50–100 ns. Thus one can avoid a cache miss even in this extreme case scenario of zero queuing delay. In practice, due to many-to-one (incast) traffic pattern, bursty traffic, routing inefficiencies, and bandwidth oversubscription, the queuing delay experienced by a packet is much higher.

Insight # 4. *An accurate estimation of the time of arrival of future packets is necessary for optimal caching.*

Notifying the target nodes of future state access requests is not sufficient for optimal caching. Target nodes also need an accurate estimation of when in the future those states will be accessed, i.e., an accurate estimate of the arrival times of packets corresponding to each future state access request. This knowledge would allow the target nodes to create a global arrival order of future packets (potentially coming from different sources over different paths), which can then be leveraged to prefetch states in the order of their (future) access time, and implement optimal cache eviction policy that evicts state that will be accessed farthest in the future [7].

Insight # 5. *A directly connected neighbor can provide a perfectly accurate estimation of future packet arrival times.*

If two network nodes X and Y are separated by multiple hops, it becomes extremely challenging to provide an accurate estimate of when a packet p currently queued at node X would eventually arrive at Y, due to non-deterministic queuing along the path from X to Y. In fact, in the worst case, p could be dropped along the path from X to Y and may never arrive at Y. However, if X and Y are directly connected, X could accurately calculate when a packet queued at X would arrive at Y. Consider an egress queue Q at X directly connected

to Y with B bytes of queued data. If a packet p of size P bytes arrives at queue Q at time t, then p will arrive at Y at time $T = t + (B+P)/L + \varepsilon$ time units (assuming a FIFO queue), where L is the link speed and ε is the link propagation delay. Based on this insight, Seer's protocol only runs between directly connected nodes in a network.

Putting it all together, Figure 1 shows the potential of Seer in terms of minimizing cache misses over the most popular online caching heuristic, LRU [56], that assumes no visibility into future state access requests.

3 Design

In this section, we describe the two key design components in Seer—(i) a low overhead notification protocol (§3.1) running between directly connected nodes in a network, that notifies target nodes of future state access requests and corresponding packet arrival times in a timely manner, and (ii) a cache manager (§3.2) that leverages the future visibility into state access requests and packet arrivals to make smarter caching decisions in terms of prefetching and cache eviction. Finally, in §3.3, we discuss the limits of Seer's design.

3.1 Future Packet Notification Protocol

As mentioned in §2, an N-port network node in Seer has k FIFO queues (priority classes) per egress port, thus totaling $N*k$ egress queues per node. Each neighbor node of a target node in Seer maintains a **Future Packet Metadata (FPM)** queue per egress queue, thus totaling $N*k$ FPM queues per node, as shown in Figure 2. Each FPM queue is a FIFO queue. Every time a packet is added to an egress queue, Seer enqueues a corresponding FPM of the form <*request id, pkt size, expected delay*> to the tail of the corresponding FPM queue. A FPM encodes the state access request corresponding to a given packet. Seer assumes that the state at the target node is indexed by request id. For most network applications, the request id is the hash of a subset of packet header fields, e.g., the flow table (state) in a virtual switch or a stateful load balancer is indexed by the hash of 5-tuple (request id), whereas the address translation table (state) in a NAT is indexed by the hash of <IP address, port> (request id). Seer assumes that a target node and its neighbors share the same hash function.

The expected delay field in a FPM is written once a FPM is dequeued and ready to be transmitted on the link. This field represents the expected delay since the FPM transmission until the corresponding packet arrives at the target node. The expected delay for a packet is calculated as follows:

Assume at egress port P, the currently dequeued FPM corresponds to a packet p in priority class i. Assume priority classes at port P are indexed from 1 to k, such that for any two priority classes i and j, if $i < j$, then i has higher priority. Next, let t_j = total transmission delay[2] of all packets queued in priority class j ($j \neq i$). Let t_i = total transmission delay of packet p plus all packets queued ahead of p in priority class i. And let

[2] transmission delay = packet size / link bandwidth.

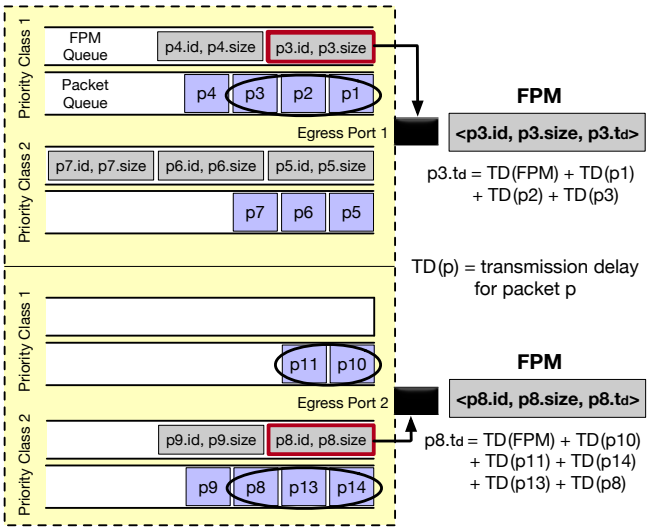


Figure 2: *Illustrates the functioning of FPM queues in Seer. It assumes the network node has two egress ports and each port has two priority classes (Priority Class 1 has higher priority). It also assumes that the FPMs corresponding to packets p1 and p2 have already been transmitted via egress port 1, as well as FPMs corresponding to packets p10, p11, p13 and p14 have already been transmitted via egress port 2. The figure shows the next FPMs being transmitted via the two egress ports, and the calculation of expected delay for the packets corresponding to those FPMs.*

t_{FPM} = transmission delay of packet carrying FPM. Then the expected delay for packet p is $t_d = \sum_{j=1}^{i-1} t_j + t_i + t_{FPM}$. This calculation is illustrated in Figure 2 for packets p_3 and p_8. Once a FPM reaches the directly connected target node, the target node can easily estimate the expected time of arrival for packet p by simply adding t_d to its current time.

Finally, each egress port continuously dequeues the FPM from the head of one of its k FPM queues, and sends it out to the target node directly connected to that egress port. The scheduling of FPM queues follow the same (strict) priority order as the packet queues, i.e., a FPM queue at an egress port is selected for dequeue only when all the FPM queues with higher priority at that port are empty (as illustrated for egress port 2 in Figure 2). This ensures that FPMs are transmitted in the same order as respective packets.

Balancing timely notification and bandwidth overhead. Seer needs to send a FPM corresponding to each packet in the egress queue. Unfortunately, if done naively, this could result in high bandwidth overhead. The naive approach would generate one control packet (of size equal to the minimum allowed packet size) for each FPM. This could result in high bandwidth overhead for applications with small sized packets—in the worst case where all packets are minimum sized, the control packets would end up consuming half the total bandwidth. One could potentially reduce the number of control packets by batching multiple FPMs into a single control packet. How-

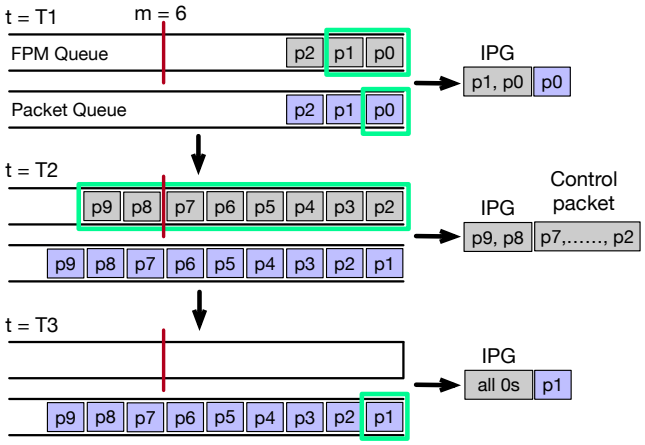


Figure 3: *Illustrates how Seer efficiently navigates the trade-off between timely delivery of FPMs and bandwidth overhead incurred. It assumes Seer only generates a control packet when FPM queue size reaches 6 or beyond, i.e., $m = 6$. Also assumes that a control packet can batch up to 6 FPMs and IPG can batch up to 2 FPMs. At $t = T1$, since FPM queue size is less than 6, Seer uses IPG to send FPM. Now suppose at $t = T2$, egress port receives a burst of 7 packets $p3 - p9$ due to incast. Now the FPM queue size exceeds 6, and hence Seer generates a control packet that can batch 6 FPMs, $p2 - p7$, while the remaining two FPMs, $p8 - p9$, can be carried in IPG. Thus, within just two packet transmission times, the neighbor knows about all the incoming packets. And this comes at the bandwidth overhead of just 1 control packet. Instead, if we had only used IPG, it would have resulted in zero bandwidth overhead, but it would have taken five packet transmission times to notify the neighbor of all incoming packets. Finally, if we had used the classic approach of control packets only with, say, a batch size of 6, to send FPMs, then it would have waited till $t = T2$ before transmitting FPMs $p0 - p5$ in 1 control packet, and again would have kept waiting starting at $t = T2$ for two more packets to arrive so it could batch them with FPMs $p6 - p9$ to form a batch size of 6. Thus, this approach would not only have required 1 extra control packet compared to Seer, but would have also added high non-deterministic latency in the delivery of the FPMs.*

ever, batching could result in delayed notifications, as Seer would need to wait for as many packets as the batch size to arrive before sending out the control packet. Ideally, Seer would like to send FPMs as soon as possible, so that at any given time, the target node has maximum possible visibility into the future state access requests.

IPG for exchanging FPMs. To overcome the aforementioned challenge, Seer uses the insight of using inter-packet gap (IPG) to exchange FPMs. A given communication protocol typically enforces a minimum IPG between consecutive packet transmissions, e.g., IEEE 802.3ae standard for Ethernet enforces a minimum IPG of 96 bits (called "idle" bits). The physical layer at the sender adds the idle bits (set to 0 by default) at the end of each transmitted packet, and the physical layer at the directly connected receiver removes the idle bits before sending the packet to higher layers. Thus, idle bits are only accessible at the physical layer.

Seer re-purposes the idle bits in IPG by overwriting the default 0s with FPMs on the transmit side, and extracting the FPMs and overwriting them with default 0s on the receive side (implementation details can be found in §5). Thus Seer effectively creates a side channel for exchanging control messages (FPMs) between directly connected nodes, with *zero* bandwidth overhead for normal data communication. This is in the same spirit as prior works that have leveraged idle bits in IPG to build systems such as a covert channel [28], a bandwidth estimator [55], and a time synchronization protocol [27, 43].

Opportunistic batching using control packets. While using IPG to exchange FPMs results in zero bandwidth overhead, it also limits the maximum rate at which Seer can exchange FPMs. Assuming one could encode X FPMs in each IPG, the rate of exchange reduces to X FPMs per packet transmission time. This is not ideal for scenarios where a burst of packets arrive at a queue in a short duration of time (e.g., incast). Assuming a burst of k packets arrive at an empty queue at time 0, it would take Seer k/X packet transmission times to transmit the FPMs for all k packets. Ideally, one would want to transmit the FPMs for all k packets at time 0 itself, so as to provide maximum future visibility to the target node. Seer achieves this by using the insight that in scenarios where the FPM queue size grows large, one can batch all the FPMs in a single control packet without incurring the batching delay. Of course this would result in bandwidth overhead, but the amount of overhead can be controlled by tuning how often Seer generates these control packets. In particular, Seer uses two tuning parameters, m and t, where Seer only generates a control packet when the total FPM queue size at an egress port exceeds m entries and at least t time units have elapsed since the last control packet was generated on that egress port. Otherwise, Seer exchanges FPMs using IPG. This way, Seer is able to balance the trade-off between timely notification and bandwidth overhead, as illustrated in Figure 3. We tune m and t for realistic experiments in §6.

3.2 Cache Manager

The cache manager in Seer is implemented on the target nodes implementing caching. The cache manager leverages the received FPMs to minimize cache misses using a novel prefetching and cache eviction algorithm as described below.

Assumptions. Seer assumes that the state stored in the backing store is indexed by request id carried in FPMs. Further, Seer assumes that the cache is a k-way set-associative cache [50] storing the <key (request id), value (state)> pairs for a subset of state from the backing store. If we set k=1, the cache becomes a direct-mapped cache, and if we set k=N, where N is the maximum number of cache lines, the cache becomes a fully-associative cache.

Data structures. The key data structure in Seer's cache manager is a set of queues, one per ingress port, storing the received FPMs from the neighbors directly connected to those ingress ports. The queue corresponding to ingress port i is shown as F_i in Algorithm 1. On receiving a FPM of the form <*m.id, m.size, m.t_d*> on ingress port i, Seer first calculates the expected time of arrival for the corresponding packet, $t_{arrival} = t_{curr} + m.t_d$, where t_{curr} is the current time at the receiver. Seer then adds the updated FPM <*m.id, m.size,* $m.t_{arrival}$> to the queue F_i. Finally, Seer also maintains a global queue, shown as X in Algorithm 1, that stores any FPM whose corresponding state has been (pre)fetched to the cache but the state has not yet been accessed by the packet corresponding to the FPM.

Prefetching algorithm. The visibility into future state access requests provides Seer the opportunity to prefetch the corresponding states to the cache *before* the packets that will access those states arrive at the target node, thus minimizing cache misses. To achieve this, Seer prefetches the states in the order of increasing expected time of arrival of packets that will access those states.

Algorithm 1 describes Seer's prefetching algorithm in its entirety. It is an endless iterative algorithm that iterates over each queue F_i in parallel (lines 7–8 in Algorithm 1). In each iteration, the algorithm finds the FPM e with the minimum $t_{arrival}$ across all the queues F_i (lines 9–18 in Algorithm 1) and whose corresponding state is not in the cache (lines 11–15 in Algorithm 1). It then fetches the corresponding state (indexed by *e.id*) from the backing store to the cache, provided the cache is not full (lines 25–27 in Algorithm 1). In case the cache is full, Seer first calls its cache eviction algorithm (lines 19-20 in Algorithm 1) to evict an entry from the cache. If the cache eviction algorithm succeeds, Seer replaces the evicted entry with the fetched state from the backing store (lines 21–24 in Algorithm 1).

One key thing to note in Algorithm 1 is that it does not destroy a FPM after the corresponding state has been fetched to the cache. Instead, it stores all such FPMs in a separate queue, X (lines 12, 22 in Algorithm 1). This is done to make sure that we don't loose the $t_{arrival}$ information for the states already fetched to the cache, as that information will be later used by Seer's cache eviction algorithm to make optimal eviction decisions.

A FPM is finally removed from all queues and destroyed once either the corresponding state has been accessed in the cache by the corresponding packet or the packet corresponding to the FPM is dropped at the target node, perhaps due to excessive queuing at the ingress (lines 29–32 in Algorithm 1).

Cache eviction algorithm. The cache eviction algorithm is triggered when Seer tries to add a state to the cache, but the cache is full. The eviction algorithm applies to a "cache set" in a set-associative cache (shown as S in Algorithm 2) to which the index of the state being added is mapped to. In a k-way set-associative cache, a cache set size is k cache lines.

Algorithm 1 Seer's Prefetching Algorithm

1: P: number of ingress ports
2: m: a FPM with attributes *m.id, m.size,* $m.t_{arrival}$
3: S_m: a cache set in a k-way set associative cache mapped to state index *m.id* (where *state[m.id]* may be cached)
4: F_i: queue storing received FPMs on ingress port i
5: X: queue storing FPMs whose corresponding states have been fetched to the cache but not yet accessed
6: $X = \{\}$
7: **while** True **do**
8: **for** i = 1 to P **do** in parallel:
9: $Y_i \leftarrow$ NULL
10: $m \leftarrow$ entry in F_i with minimum $t_{arrival}$
11: **if** $m.id \in S_m$ **then**
12: Remove m from F_i and add it to X
13: **else**
14: $Y_i \leftarrow m$
15: **end if**
16: **end for**
17: $idx \leftarrow$ index in Y with minimum $t_{arrival}$
18: $e \leftarrow Y_{idx}$
19: **if** S_e is full **then**
20: Evict an entry from S_e using Algorithm 2
21: **if** Algorithm 2 succeeds **then**
22: Remove e from F_{idx} and add it to X
23: Fetch state at index *e.id* from backing store
24: **end if**
25: **else**
26: Fetch state at index *e.id* from backing store
27: **end if**
28: **end while**
29: **Asynchronously do:**
30: **if** state indexed by *m.id* is accessed in the cache **or** packet corresponding to m is dropped at the target node **then**
31: Remove m from X and F
32: **end if**

Algorithm 2 Seer's Cache Eviction Algorithm

1: S: cache set under consideration for eviction
2: *e.id*: index of the state being considered for fetch
3: $A \leftarrow \{x.id \in S: \exists\ m$ s.t. $x.id = m.id$ and $m \in \bigcup_{i=1}^{P} Q_i \cup X\}$
4: $B \leftarrow \{x.id \in S: x.id \notin A\}$
5: **if** B not empty **then**
6: Evict an entry from B using any caching heuristic
7: **return** Success
8: **else**
9: $m.id \leftarrow$ entry in A with maximum $t_{arrival}$
10: **if** $e.t_{arrival} < m.t_{arrival}$ **then**
11: Evict *m.id* from A
12: **return** Success
13: **end if**
14: **end if**
15: **return** Failure

Intuitively, Seer's cache eviction algorithm tries to emulate the optimal (offline) algorithm for minimizing cache misses, namely Belady's algorithm [7]. Belady evicts the cache entry that will be accessed farthest in the future. However, unlike Belady, which assumes full visibility into the future requests, Seer only has a *partial* view of the future, i.e., at any given time, Seer only knows about a partial set of future state access requests (namely, the requests corresponding to packets currently queued at a neighbor node). Thus, it may be possible that there are entries in the cache for which Seer has no knowledge of when those entries will be accessed in the future (i.e., there is no received FPM corresponding to the packet that will access that cache entry). Thus, Seer's cache eviction algorithm first separates all the entries in the cache set into two sets — set A (line 3 in Algorithm 2) which includes all the entries in the cache set for which $t_{arrival}$ is known, and set B (line 4 in Algorithm 2) which includes all the remaining entries in the cache set. Thus, set A stores all the states that will be accessed in the near future, while the states in set B are expected to be accessed farther in the future, if at all. Hence, in the spirit of Belady, Seer prioritizes set B over set A for eviction (line 5 in Algorithm 2). To evict an entry from set B, Seer relies on some default caching heuristic, such as LRU [56] (line 6 in Algorithm 2). Note that any caching heuristic can be used for this purpose. On the other hand, to evict an entry from set A, Seer uses Belady's algorithm (lines 8–14 in Algorithm 2). Overall, in the best case, when set B is empty, Seer emulates Belady, while in the worst case, when set A is empty, Seer reduces to the default caching heuristic.

An interesting consequence of Seer's design is that it may result in scenarios where the cache eviction algorithm fails to evict an entry from the cache set, thus aborting the current state fetch from the backing store. This is because in Seer, states are prefetched to the cache and that too in the order of their $t_{arrival}$. Thus, if currently every entry in the cache set has $t_{arrival}$ value less than the $t_{arrival}$ value for the entry currently being considered for prefetching from the backing store (and replace an existing entry in the cache), Seer does not evict any existing entry in the cache and the fetch is aborted, otherwise eviction succeeds (lines 9–13 in Algorithm 2). This is in the spirit of Belady, where the state that will be accessed farthest in the future (in this case, the state that is currently being considered for prefetching) is evicted from the cache.

3.3 Limits of Seer

In this section, we discuss some of the limits of Seer's design.

The gains of Seer's cache eviction algorithm over existing online caching heuristics is dependent on the degree of future visibility, which in turn, is dependent on the degree of queuing at the neighbor nodes — the more the queuing, the higher the gains. In the worst case, when queuing is extremely small, Seer's cache eviction algorithm would reduce to the default caching heuristic. On the other hand, the effectiveness of Seer's prefetching algorithm is dependent on both the degree of queuing at the neighbors and the access time of the backing store. In particular, for prefetching to be effective, the amount of queuing delay for a packet must be greater than the access time of the backing store, in order to ensure that the state has been fetched to the cache before that packet arrives. Hence prefetching will be most effective for high-speed network and distributed applications that use a faster backing store such as DRAM (with SRAM as cache). However, note that even for applications that use a slower backing store compared to the typical network queuing delays, Seer's cache eviction algorithm will continue to provide gains, e.g., by ensuring one does not evict an entry that will be accessed in the near future. Finally, Seer's design is also somewhat susceptible to FPM drops. If a FPM is dropped (e.g., queue tail drop at either the neighbor or the target node) but the corresponding packet is not dropped, Seer might end up with a non-continuous view of future state access requests. This may result in priority inversion, e.g., replacing a cache entry with another entry that will be accessed farther in the future. Fortunately, it is fairly easy to avoid FPM drops in Seer, by provisioning sufficient memory for FPM queues at both the neighbors and the target nodes. Given the size of a FPM is very small, the amount of memory needed to avoid drops in practice is nominal (§6).

4 Implementation

In this section, we describe the implementation Seer's cache manager. We start by first discussing the performance goals for the cache manager, followed by an implementation that achieves those goals.

Performance goals. Seer's cache manager operates iteratively, and in each iteration it can have three potential performance bottlenecks – (i) time taken to decide what state to prefetch ($t_{prefetch}$), (ii) time taken to evict a cache entry (t_{evict}), and (iii) time taken to fetch the state from backing store into the cache ($t_{bkStore}$). Prefetching decision and eviction needs to happen sequentially in each iteration, but they both can be parallelized with the state fetch from the backing store from the previous iteration. Thus the goal of Seer's implementation is to ensure that $t_{prefetch} + t_{evict} \leq t_{bkStore}$, so that the backing store access time remains the bottleneck for cache replacement throughput. Further, we also want to ensure that the received FPMs are added to the respective queues in the cache manager in ideally $O(1)$ time, so that the prefetching and eviction algorithms get access to the FPMs as soon as they arrive at the target node.

Primitives. Next, we present the key primitives needed to implement Seer's prefetching and cache eviction algorithms.

1. **insert**(m, Q)**:** Adds an element m to queue Q (lines 12, 22 in Algorithm 1).
2. **delete**(m, Q)**:** Removes an element m from queue Q (lines 12, 22, 31 in Algorithm 1).
3. **max-min**(Q)**:** Returns the max (line 9 in Algorithm 2) or the min (line 10 in Algorithm 1) value in a queue Q.

4. **min($Y_1, Y_2, ..., Y_P$):** Returns the min value (line 18 in Algorithm 2) in an un-ordered set of entries Y_i, i=1 to P.
5. **intersect(S_1, S_2):** Returns the set of elements present in both sets S_1 and S_2 (line 3 in Algorithm 2).
6. **difference(S_1, S_2):** Returns the set of elements present in set S_1 but not in set S_2 (line 4 in Algorithm 2).

We bundle primitives 1–4 into *order primitives* and primitives 5, 6 into *set primitives*. Next, we describe the implementation of both sets of primitives.

Implementing order primitives. To find the minimum element in an un-ordered set of P elements, one fundamentally requires at least $O(log(P))$ time. Thus, Seer implements primitive 4, **min**($Y_1, Y_2, ..., Y_P$), in $log(P)$ clock cycles.

Next, we focus on primitives 1–3 that operate over a queue. The natural data structure for implementing max/min (primitive 3) would be a priority queue. A priority queue can return the max/min value in $O(1)$ time. However, insertions and deletions may take $O(log(N))$ time, where N is the queue size. To make matters worse, in Seer, adding an element to a queue may require updating the value of the attribute over which max/min is calculated for n ($n \leq N$) other elements in the queue, thus resulting in an added $O(n(log(N))$ time to re-insert each updated element to the queue. This is because of the presence of multiple priority classes. A packet p_1 in a higher priority class will be transmitted before a packet p_2 queued in a lower priority class at the same egress port, even if p_2 arrived before p_1. As a result, the $t_{arrival}$ field in the received FPM corresponding to p_2 will need to be updated once the FPM for p_1 arrives at a later time. This update can be done by adding the transmission delay for packet p_1 to p_2's current expected time of arrival, i.e., $p_2.t_{arrival}$ += $p_1.size$ / *link bandwidth*. This is illustrated in Figure 4.

Seer solves the above challenge by replacing the priority queues with *fully ordered lists*. A fully ordered list maintains the invariant that the list is always sorted (by $t_{arrival}$ in Seer) even under insertions, deletions, and updates. This automatically reduces the time complexity for max/min operations to 1 clock cycle. Seer implements a fully ordered list of size N using N flip-flops, which allows parallel access to each of the N elements in the list. To add an element m to the list, Seer first determines the right index in the list to add m that would still keep the list sorted. This can be done in 1 clock cycle by comparing the $t_{arrival}$ value for m against the $t_{arrival}$ values of each element in the list in parallel. Once the right location has been determined, Seer adds m to that location and shifts the rest of the elements in the list in parallel, again requiring only 1 clock cycle. Further, parallel access also allows Seer to update the $t_{arrival}$ values of multiple elements in the list in just 1 clock cycle. Note that in Seer, the $t_{arrival}$ values of the existing elements are all updated by the same amount (namely, the transmission delay of m), and hence their relative positions in a fully ordered list will not change. So, Seer does not need to re-insert the updated elements. This entire design is an adaptation of the classic parallel compare-and-shift architecture [32] that has also been used recently in other contexts, such as packet scheduling [40, 45] and filtering [41]. Overall, with an ordered list, Seer can implement *max-min(Q)* in 1 clock cycle, *delete(m, Q)* in 2 clock cycles, and *insert(m, Q)* in 3 clock cycles (including the updates triggered by an insert).

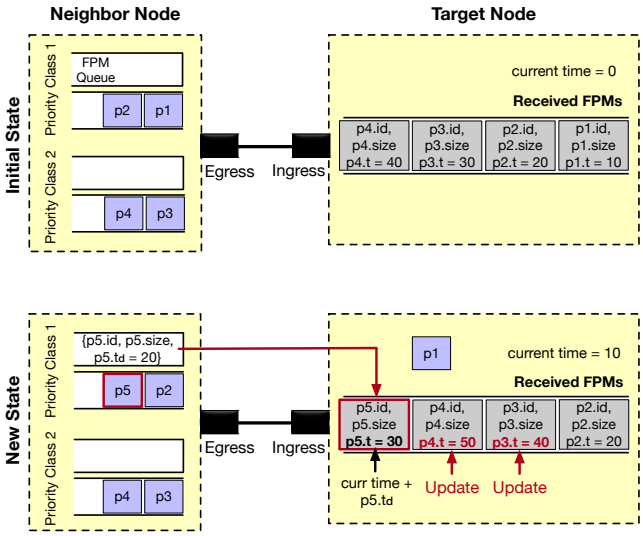


Figure 4: *Illustrates that with multiple priority classes, adding a FPM to a queue at the target node may require updating existing elements in the queue. Assumes the transmission delay for each packet is 10 time units and propagation delay 0. Initially, the FPMs for packets $p1 - p4$ were all received at the target node at time 0. Then at some later time, packet $p5$ arrived at the neighbor node, and its corresponding FPM was received at the target node at time 10. Since p_5 arrived in a higher priority class, it will be transmitted right after p_2, preempting p_3 and p_4. Thus, the expected delay for $p5$ is only 20 time units (sum of transmission delays of p_5 and p_2). But it also pushes the transmission of p_3, p_4 back by 10 time units (equal to p_5's transmission delay). Hence, the current expected time of arrival for p_3, p_4 need to be updated at the target node, by adding to them the transmission delay for p_5.*

Implementing set primitives. Implementing the queues in Seer's cache manager using flip-flops also helps with fast implementations of the two set primitives, namely *intersect(S_1, S_2)* and *difference(S_1, S_2)*. In Seer, S_1 is a cache set of size k in a k-way set-associative cache. Implementation of the cache itself is outside the purview of Seer, but we assume that it takes $O(k)$ time to locate an element in a cache set of size k. On the other hand, set S_2 in Seer refers to queues F_i and X in Algorithm 1, which we implement as fully ordered lists using flip-flops, as discussed above. Further, in Seer, the intersect and difference operations execute synchronously over the same input sets (lines 3–4 in Algorithm 2). Hence, Seer implements the two primitives jointly (in parallel) as described below.

Seer iteratively compares the id of each state s in the cache set (set S_1) with all the elements in $\bigcup F_i$ and X (set S_2) in parallel. Since flip-flops allow parallel access, this can be done in k clock cycles, where k is the cache set size. If the id of s matches the id of an element m in F_i or X, Seer adds it to set A (output of *intersect*(S_1, S_2)), else it adds it to set B (output of *difference*(S_1, S_2)). Thus, it takes a total of k clock cycles to implement both the intersect and difference primitives. Additionally, Seer also calculates the max $t_{arrival}$ within set A (line 9 in Algorithm 2) in parallel while building set A. Essentially, while adding an element m to set A, Seer updates the current max $t_{arrival}$ value, t_{max}, to $max(t_{max}, m.t_{arrival})$, and accordingly updates the state id with the current max $t_{arrival}$, id_{max}, to $m.id$ if $m.t_{arrival} > t_{max}$. Thus, after k iterations, id_{max} stores the state id in set A with max $t_{arrival}$. Hence, Seer could execute the entire cache eviction algorithm in k clock cycles.

Overall performance. Overall, Seer takes k clock cycles for the cache membership check (line 11 in Algorithm 1), $1 + log(P)$ clock cycles to find the FPM with minimum $t_{arrival}$ (lines 10 + 17 in Algorithm 1), and k clock cycles in the best case or k clock cycles plus the latency of the default caching heuristic (for LRU the best known implementation has $O(1)$ time) in the worst case, to evict an element from the cache. Thus, $t_{prefetch} + t_{evict} = log(P) + 2k + 1$ clock cycles. The value of k, the cache set size, is typically 4–8 for modern caches [52]. The value of P, number of ports, varies from 2–4 for NICs and FPGAs to few 100s for switches and routers. Assuming clock rates of around 100–200 MHz as typically observed for NICs and FPGAs, and clock rates of around 1 GHz as typically observed for ASIC switches and routers [42], the total $t_{prefetch} + t_{evict}$ time would be 100–200 ns for NICs and FPGAs and under 25 ns for ASIC switches and routers.

5 Prototype

We prototype Seer in System Verilog (∼1200 LOCs) on an Altera Stratix V [49] FPGA comprising 234 K Adaptive Logic Modules, 52 Mbits SRAM, and four 10 Gbps network ports. The architecture of the prototype is shown in Figure 5.

To implement Seer's future packet notification protocol using IPG, as described in §3.1, we modify Ethernet's physical layer (PHY) as shown in Figure 5. Once the Physical Coding Sublayer (PCS) in PHY receives a packet from higher layers to transmit, the Encoder module in PCS reformats the packet into a sequence of one /S/ block (Start of an Ethernet frame), multiple /D/ data blocks, and one /T/ block (End of an Ethernet frame). PCS inserts at least twelve 8-bit idle characters (/I/) between two Ethernet frames (IPG). The /I/ characters are set to 0 by default. The /T/ block can have 0–7 /I/ characters, and PCS inserts one or more special /E/ block with eight /I/ characters to make up the minimum requirement of twelve /I/ characters. Note that if there are no Ethernet frames to transmit, PCS continuously keeps transmitting /E/ blocks. The /T/ and /E/ blocks (and hence the /I/ characters) are accessible as part of the output from the

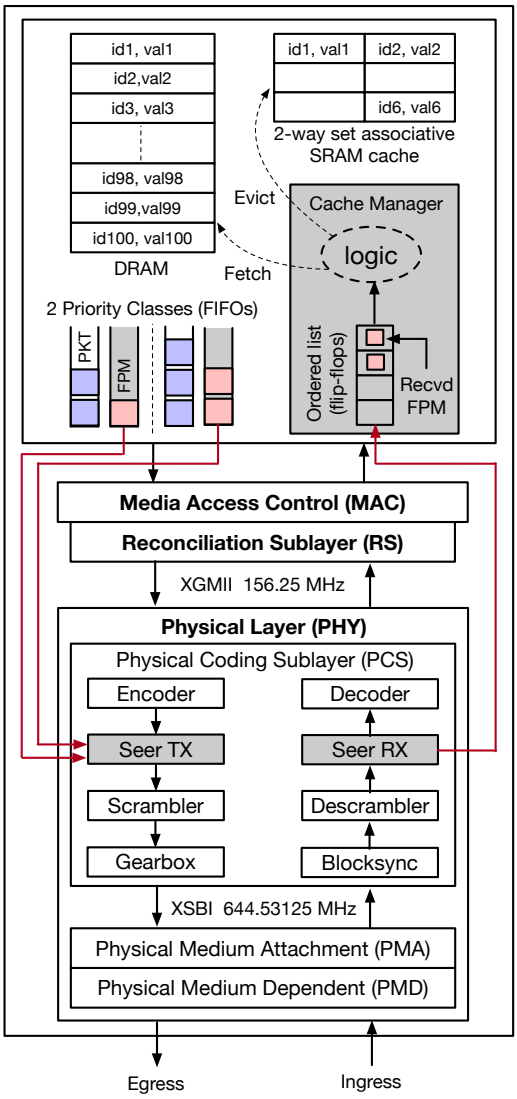


Figure 5: *Seer's FPGA prototype. Seer's modules are shown in gray boxes.*

Encoder on the TX path and input to the Decoder on RX path. Hence, we implement Seer's logic after the Encoder/Decoder modules. On the TX path, Seer creates a separate data path (shown using red lines in Figure 5) that bypasses the normal data path for Ethernet frames, and connects Seer's PHY module directly to the FPM FIFO queues. If any of the FPM queues are non-empty, Seer immediately dequeues an FPM from the queue and overwrites the outgoing /I/ characters with the FPM. On the RX path, when Seer receives a /T/ or an /E/ block, it extracts the FPM from the /I/ characters in those blocks, and overwrites those bits with all 0s before sending to Decoder, following the Ethernet standard. Seer adds the extracted FPM to the fully ordered list of FPMs through another separate data path (shown using a red line in Figure 5) bypassing the normal Ethernet frame's data path. Finally, we also implement Seer's cache manager as described in §4.

Queue size	FPGA		ASIC	
	Clock	Logic	Clock	Area
$N = 128$	170 MHz	8%	4.2 GHz	0.012 mm^2
$N = 256$	150 MHz	15%	4.1 GHz	0.022 mm^2
$N = 512$	120 MHz	30%	3.7 GHz	0.042 mm^2
$N = 1024$	100 MHz	60%	3.5 GHz	0.085 mm^2
$N = 2048$	–	>100%	3.4 GHz	0.167 mm^2
$N = 4096$	–	>100%	3.2 GHz	0.324 mm^2

Table 1: *Clock speed and resource usage for Seer's prototype with varying size of the fully ordered list data structure.*

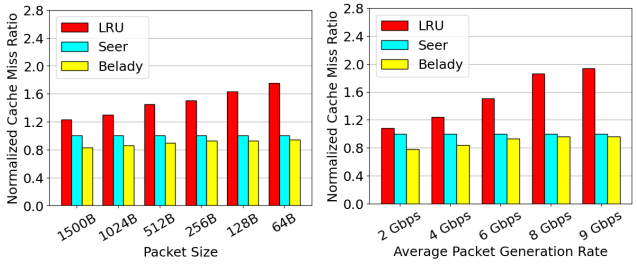
5.1 Resource Usage

The most resource consuming component of Seer's design (and also the bottleneck for clock speed) is the fully ordered list used to store the received FPMs. This is the price we way for parallelism via flip-flops. Table 1 shows Seer's overall clock speed and resource consumption for varying sizes of the fully ordered list. We also synthesize Seer's RTL design on Synopsys Design Compiler tool [53] using an open-source 15 nm process technology [29], and report the results in Table 1. On the FPGA, we are unable to synthesize a design beyond queue size of 1024, as we run out of FPGA logic resources. On the ASIC however, Seer is able to support much larger queue sizes with clock rates in excess of 3 GHz. To put this in perspective, modern switching chips typically run at around 1 GHz clock rate [42, 45]. Chip area increases linearly with queue size. Note that the numbers reported are for a single queue. If the processor has multiple ports, the area usage will be multiplied by the number of ports, as Seer maintains one queue per ingress port. Thus, for $N = 4096$, and 100 ports, the total area consumed will be 32 mm^2. This is between 5%–10% overhead for switching chips whose chip areas vary from 300–700 mm^2 [12].

5.2 Prototype Experiments

We directly connect two FPGA prototypes from Figure 5 using an optical cable of length 2 m (propagation delay of around 10 ns). One FPGA emulates the neighbor node while the other emulates the target node. On the neighbor FPGA, we implement two priority classes at the egress. We also implement a packet generator on the FPGA to feed the packets into the priority classes. Packet generator randomly decides which priority class to put a packet into. Packets arrive according to a Poisson process. We assign a random flow id to each packet, chosen uniformly at randomly from 0 to 100 K, thus emulating 100 K flows. On the target FPGA, we populate the DRAM with 100 K flow state entries. Each flow state is 512 bits. We implement a 2-way set associative cache of size 2 MB in SRAM (can cache around 30 K flow states).

Parameters. The default packet size to 256 B. The default average rate of packet generation is 6 Gbps. The FPM queue size at both the egress and ingress is 256 entries. The size of each FPM is 44 bits – 17 bits for flow id, 11 bits for packet size, and 16 bits for t_d. Thus we can send two FPMs in the minimum sized Ethernet IPG. The control packet size is 64 B, and we send a control packet only when the FPM queue size at egress exceeds 8 entries ($m = 8$) and at least 5 us have elapsed since the last control packet was sent (t=5 us). This limits the bandwidth overhead of control packets to $\sim$ 1%. The default caching heuristic used in Seer is LRU [56] (Algorithm 2).



(a) Incast workload. (b) Permutation workload.

Figure 6: *Cache miss ratio for Seer vs. LRU and Belady for different packet sizes and packet generation rates. For each data point, all the cache miss ratios are normalized w.r.t. the corresponding cache miss ratio for Seer.*

Evaluation metric. We use cache miss ratio, i.e., number of cache misses divided by total cache access requests.

Baselines. We use LRU [56] and Belady [7] as the baselines.

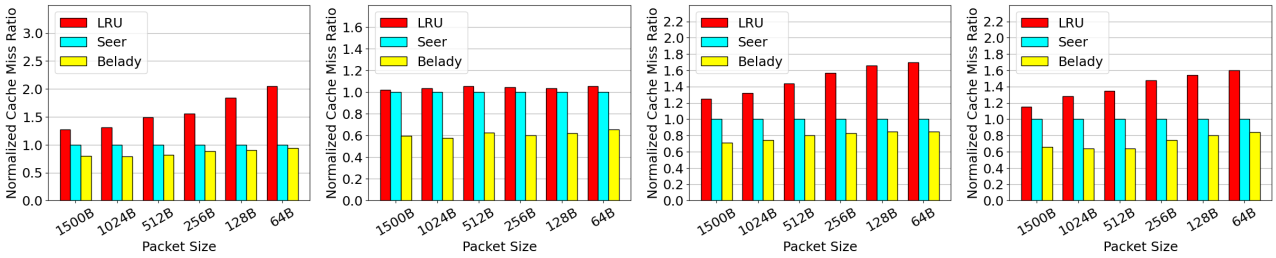
Experiment results. Figure 6a shows the cache miss ratio against packet size. Seer outperforms LRU for all packet sizes, but the gains decrease for larger packet sizes (80% gain for 64 B vs. 20% for 1500 B). This is because with smaller packet sizes, the number of packets in a queue of given size (in bytes) would be higher. This works to Seer's advantage, as Seer receives higher number of FPMs within a given time window, thus allowing it to make more informed prefetching and eviction decisions. Finally, we also note that Seer performs very close to Belady (within 20%) for all packet sizes.

Next, Figure 6b shows the cache miss ratio against different packet generation rates. As the packet generation rates increase, Seer outperforms LRU by a bigger margin (by 8% at 2 Gbps vs. 94% at 9 Gbps). Similarly, Seer performs closer to Belady at higher packet generation rates (within 5% at 9 Gbps vs. 20% at 2 Gbps). This is because higher packet generation rate leads to more queuing at the egress, thus providing Seer with more visibility into the future state access requests.

6 Simulations

In this section, we do large scale network simulations to evaluate the performance of Seer over state-intensive network applications. Our simulator is written in C built on top of [44].

Setup. We simulate a two-tier Fattree [1] topology with 16 spine switches, 9 ToR switches, and 16 hosts per ToR switch, for a total of 144 hosts. All links in the network are 100 Gbps. Per-hop propagation delay is 100 ns. Each host in the network is running DCTCP [3] congestion control and each switch supports ECN. Switches do ECMP [51] load balancing.



(a) Incast workload. (b) Permutation workload. (c) Websearch workload. (d) Datamining workload.

Figure 7: *Cache miss ratio for Seer vs. LRU and Belady for different packet sizes. For each packet size, all the cache miss ratios are normalized w.r.t. the corresponding cache miss ratio for Seer.*

Applications. Each spine switch in the network runs a stateful layer 4 load balancing application, similar to SilkRoad [31]. Each ToR switch in the network runs an intrusion detection application [57] that stores several per-flow states in the switch, e.g., packet counts, packet inter-arrival times, etc. By default, we assume the cache in each switch can store up to 20% of the total flow states. We assume DRAM as the backing store.

Workloads. We evaluate Seer against a variety of workloads – (i) a permutation workload, where each host sends and receives exactly one flow; (ii) an incast workload, where we select a rack as the incast destination and all other hosts in the network send to the hosts in that incast rack; (iii) websearch workload [3]; and (iv) datamining workload [17]. Websearch and datamining are representative datacenter workloads, with heavy-tailed flow size distribution. Flows arrive according to a Poisson process for a target network load of 0.6.

Baselines. We evaluate Seer against a variety of state-of-the-art online caching algorithms – LRU [56], LFU [13], ARC [30], S3-FIFO [59], and SIEVE [60]. We also evaluate Seer against the optimal offline algorithm, Belady [7].

Evaluation metrics. We use two evaluation metrics – (i) cache miss ratio aggregated across all the switches running the above applications, and (ii) flow completion time (FCT).

Parameters. The FPM queue size at each egress and ingress port is 1024 entries. The size of each FPM is 44 bits – 17 bits for flow id, 11 bits for packet size, and 16 bits for t_d. Thus we can send two FPMs in the minimum sized Ethernet IPG. The control packet size is 64 B, and we send a control packet only when the FPM queue size at egress exceeds 8 entries ($m = 8$) and at least 500 ns have elapsed since the last control packet was sent (t=500 ns). This limits the bandwidth overhead of control packets to $\sim$ 1%. The default caching heuristic used in Seer is LRU [56] (Algorithm 2).

Experiment results. Figure 7 shows that for incast workload, Seer significantly outperforms LRU for all packet sizes. This is due to the fact that incast workload results in significant queuing in the network, which allows Seer to get better visibility into future state access requests. In contrast, for the permutation workload, Seer performs very close to LRU, since this workload observes least queuing in the network. This is also the reason why for this workload, the gap between the performance of Seer and Belady is largest. Next, even for realistic datacenter workloads, namely websearch and datamining, Seer significantly outperforms LRU (by up to 65%) while remaining between 15–35% of Belady for all packet sizes. Finally, across all the four workloads, as the packet sizes increase, the gains of Seer over LRU decrease and the performance gap between LRU and Belady increase. The reason for this trend is explained in §5.2.

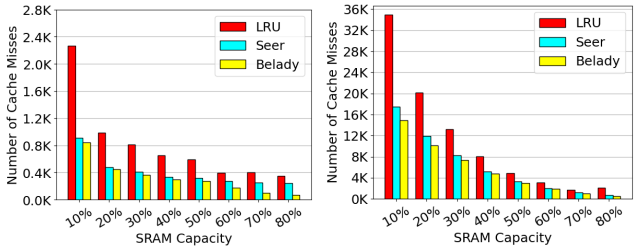
Next, Figure 8 shows the performance of Seer with varying cache capacity. As expected, as the cache capacity increases, the number of cache misses decrease for both Seer and the baselines. However, Seer performs consistently better than LRU for all cache sizes. While the gains are more for smaller cache sizes, but even for larger cache sizes, the gains are significant. This is because even with a large cache size, LRU is unable to avoid cold cache misses, where a state is fetched to the cache for the first time. However, Seer can avoid such cold misses due to its prefetching algorithm leveraging visibility into future state access requests.

Next, Figure 9 shows the performance of Seer against state-of-the-art online caching heuristics. LFU performs the worst, which is an indication that frequency is perhaps not the right metric for caching in these experiments. ARC is based upon LRU while S3-FIFO and SIEVE are recent FIFO-based caching heuristics designed to be more scalable than LRU. But ultimately, all these heuristics are limited by the lack of visibility into future state access requests, which both Seer and Belady exploit to gain much better performance.

Finally, in Figure 10, we show the performance of Seer in terms of flow completion time. The trend here is similar to the trend observed with cache miss ratio, since higher cache misses at the switches result in higher latency (and lower throughput) for in-network packet processing, ultimately resulting in higher flow completion time.

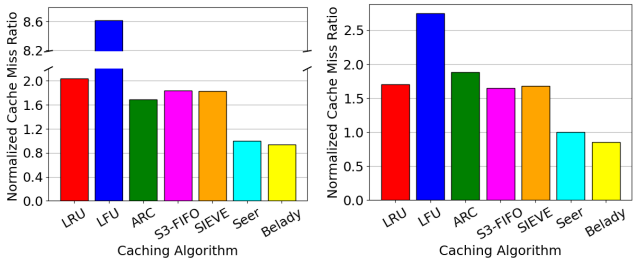
7 Related Work

Belady's algorithm [7] exemplifies an optimal caching algorithm. While a true implementation of Belady's algorithm would be ideal, it is impractical due to its fully offline nature. Seer attempts a practical, best-effort emulation of Belady.



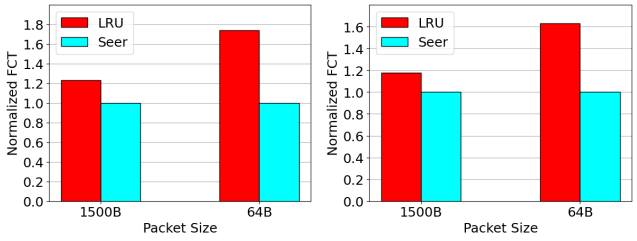
(a) Incast workload. (b) Websearch workload.

Figure 8: *Number of cache misses for different cache capacities. A cache capacity of X% means the cache can hold X% of total state. The packet size used is 64B.*



(a) Incast workload. (b) Websearch workload.

Figure 9: *Cache miss ratio for Seer vs. state-of-the-art online caching heuristics and Belady. The cache miss ratios are normalized w.r.t. the cache miss ratio for Seer. The packet size used is 64B.*



(a) Websearch workload. (b) Datamining workload.

Figure 10: *Flow completion time (FCT) for Seer vs. LRU. FCTs are normalized w.r.t. to the FCT for Seer.*

Recency and frequency metrics form the backbone of traditional caching heuristics, such as LRU [56], LFU [13], LRU-K [36], 2Q [23], ARC [30], SLRU [24], GDSF [11], EELRU [46], LRFU [26], CAR [5], CLOCK-Pro [22], TinyLFU [14], S3-FIFO [59], and SIEVE [60]. The LRU algorithm maintains an ordered queue based on access recency, evicting the oldest entry whenever necessary. LRU-K, 2Q, ARC, SLRU, and EELRU use multiple LRU queues in tandem to improve performance or cover weaknesses of LRU, such as thrashing. Frequency also serves an important role in caching, most notably with LFU. For example, LRFU combines recency and frequency into a single metric for consideration. Tiny-LFU augments any arbitrary caching algorithm with an LFU-like admission policy. Real world factors are also often taken into consideration with caching. For example, GDSF optimizes around the time cost disparity between different memory accesses. Practical-minded cache designers may instead use CLOCK-based algorithms such as CAR and CLOCK-Pro, or FIFO-based algorithms, such as S3-FIFO and SIEVE, due to their speed and ease of implementation in real-world settings. These algorithms all attempt to approximate Belady's algorithm to varying degrees of success. However, they are ultimately limited by their online design, as being oblivious to future state access requests leaves much performance on the table.

Machine learning has also been used to aid caching, especially in web caching or content delivery settings. LHD [6], Raven [20], LeCaR [54], CACHEUS [39], LRB [47], GL-Cache [58] and HALP [48] are examples, each of which integrate ML into their designs in different ways. Some algorithms like LHD and Raven are probability-based, while others such as LeCaR and CACHEUS learn weights between well-studied algorithms. Others, such as LRB and HALP use network statistics to learn an approximation of Belady's algorithm. Still others take wholly unique angles, such as GL-Cache, which classifies objects together for grouped cache management. ML-based algorithms have their time and place. However, their reliance on the historical or statistical patterns to make future predictions is their achilles heel. In contrast, Seer provides a perfectly accurate mechanism for visibility into future state access requests in a networked setting.

Some solutions do not neatly fit into the above categories, such as Belatedly and its practical approximation MAD [4], which minimize cache delay instead of cache misses as traditional algorithms do. And finally, Reframer [16] intentionally delays and reorders packets belonging to different flows to reduce end-host cache misses, however is challenging to implement at line rate, and introduces delay to reordered packets.

8 Conclusion

We presented Seer which is an online caching system for state-intensive network and distributed applications. Seer minimizes cache misses by providing visibility into future state access requests. Seer leverages the delays experienced by packets in a network to notify network nodes implementing caching of future state access requests carried in incoming network packets that are currently queued at a neighbor node. Using this as a building block, we presented the design of an online cache manager that leverages visibility into (partial) set of future state access requests to make smarter prefetching and cache eviction decisions. Seer's design has been prototyped and implemented on an FPGA. Our evaluations showed that Seer achieves up to 65% lower cache miss ratio and up to 78% lower flow completion time compared to LRU for key network applications over realistic workloads.

Acknowledgments

We thank the anonymous NSDI reviewers and our shepherd, Anuj Kalia, for their valuable feedback. This work was supported in part by an NSF CAREER Award 2239829 and a Ross Ph.D. fellowship.

References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. *A Scalable, Commodity Data Center Network Architecture*. SIGCOMM, 2008.

[2] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. *SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues*. NSDI, 2020.

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. *Data Center TCP (DCTCP)*. SIGCOMM, 2010.

[4] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S. Berger. *Caching with Delayed Hits*. SIGCOMM, 2020.

[5] Sorav Bansal and Dharmendra S. Modha. *CAR: Clock with Adaptive Replacement*. FAST, 2004.

[6] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. *LHD: Improving hit rate by maximizing hit density*. NSDI, 2018.

[7] Laszlo A. Belady. *A study of replacement algorithms for a virtual-storage computer*. IBM Systems Journal, 1966.

[8] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. *AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network*. NSDI, 2017.

[9] Daniel Bittman, Robert Soulé, Ethan L. Miller, Vishal Shrivastav, Pankaj Mehra, Matthew Boisvert, Avi Silberschatz, and Peter Alvaro. *Don't Let RPCs Constrain Your API*. HotNets, 2021.

[10] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. *Understanding host network stack overheads*. SIGCOMM, 2021.

[11] Pei Cao and Sandy Irani. *Cost-Aware WWW Proxy Caching Algorithms*. USITS, 1997.

[12] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. *dRMT: Disaggregated Programmable Switching*. SIGCOMM, 2017.

[13] John Dilley and Martin Arlitt. *Improving proxy cache performance: Analysis of three replacement policies*. IEEE Internet Computing, 1999.

[14] Gil Einziger, Roy Friedman, and Ben Manes. *TinyLFU: A Highly Efficient Cache Admission Policy*. ACM Transactions on Storage, 2017.

[15] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. *Azure Accelerated Networking: SmartNICs in the Public Cloud*. NSDI, 2018.

[16] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić. *Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets*. NSDI, 2022.

[17] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. *VL2: A Scalable and Flexible Data Center Network*. SIGCOMM, 2009.

[18] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. *Queues Don't Matter When You Can JUMP Them!* NSDI, 2015.

[19] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach Sixth Edition*. Morgan Kaufmann, 2019.

[20] Xinyue Hu, Eman Ramadan, Wei Ye, Feng Tian, and Zhi-Li Zhang. *Raven: Belady-Guided, Predictive (Deep) Learning for in-Memory and Content Caching*. CoNEXT, 2022.

[21] Syed Usman Jafri, Sanjay Rao, Vishal Shrivastav, and Mohit Tawarmalani. *Leo: Online ML-based Traffic Classification at Multi-Terabit Line Rate.* NSDI, 2024.

[22] Song Jiang, Feng Chen, and Xiaodong Zhang. *CLOCK-Pro: an effective improvement of the CLOCK replacement*. ATC, 2005.

[23] Theodore Johnson and Dennis Shasha. *2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm*. VLDB, 1994.

[24] R. Karedla, J.S. Love, and B.G. Wherry. *Caching strategies to improve disk system performance*. Computer, 1994.

[25] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. *TEA: Enabling State-Intensive Network Functions on Programmable Switches*. SIGCOMM, 2020.

[26] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, S.H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang. *LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies*. IEEE Transactions on Computers, 2001.

[27] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. *Globally Synchronized Time via Datacenter Networks*. SIGCOMM, 2016.

[28] Ki Suh Lee, Han Wang, and Hakim Weatherspoon. *PHY Covert Channels: Can you see the Idles?* NSDI, 2014.

[29] Mayler Martins, Jody Maick Matos, Renato P. Ribas, André Reis, Guilherme Schlinker, Lucio Rech, and Jens Michelsen. *Open Cell Library in 15nm FreePDK Technology*. ISPD, 2015.

[30] Nimrod Megiddo and Dharmendra S Modha. *Arc: A self-tuning, low overhead replacement cache*. FAST, 2003.

[31] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. *SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs*. SIGCOMM, 2017.

[32] Sung-Whan Moon, Jennifer Rexford, and Kang G. Shin. *Scalable hardware priority queue architectures for high-speed packet switches*. Transactions on Computers, 2000.

[33] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. *Language-Directed Hardware Design for Network Performance Monitoring*. SIGCOMM, 2018.

[34] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. NSDI, 2013.

[35] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H Noh. *Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems.* FAST, 2012.

[36] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. *The LRU-K Page Replacement Algorithm For Database Disk Buffering*. SIGMOD, 1993.

[37] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. *The design and implementation of open vSwitch*. NSDI, 2015.

[38] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. *FlowBlaze: Stateful Packet Processing in Hardware*. NSDI, 2019.

[39] Liana V. Rodrigues, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. *Learning Cache Replacement with CACHEUS*. FAST, 2021.

[40] Vishal Shrivastav. *Fast, Scalable, and Programmable Packet Scheduler in Hardware*. SIGCOMM, 2019.

[41] Vishal Shrivastav. *Programmable Multi-Dimensional Table Filters for Line Rate Network Functions*. SIGCOMM, 2022.

[42] Vishal Shrivastav. *Stateful Multi-Pipelined Programmable Switches*. SIGCOMM, 2022.

[43] Vishal Shrivastav, Ki Suh Lee, Han Wang, and Hakim Weatherspoon. *Globally Synchronized Time via Datacenter Networks*. IEEE/ACM Transactions on Networking, 2019.

[44] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. *Shoal: A Network Architecture for Disaggregated Racks*. NSDI, 2019.

[45] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. *Programmable Packet Scheduling at Line Rate*. SIGCOMM, 2016.

[46] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. *EELRU: simple and effective adaptive page replacement*. SIGMETRICS, 1999.

[47] Zhenyu Song, Daniel S. Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, et al. *Learning relaxed belady for content distribution network caching*. NSDI, 2020.

[48] Zhenyu Song, Kevin Chen, Nikhil Sarda, Deniz Altinbuken, Eugene Brevdo, Jimmy Coleman, Xiao Ji, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi. *Halp: Heuristic aided learned preference eviction policy for youtube content delivery network*. NSDI, 2023.

[49] http://de5-net.terasic.com.tw. *DE5-Net FPGA Development Kit*. Terasic, 2021.

[50] https://en.wikipedia.org/wiki/Cache_placement_policies. *Cache Placement Policies*. Wikipedia, 2023.

[51] https://en.wikipedia.org/wiki/Equal-cost_multi-path_routing. *Equal-cost multi-path routing*. Wikipedia, 2023.

[52] https://www.intel.com/content/dam/develop/external/us/en/documents/architecture-instruction-set-extensions-programming-reference.pdf. *Intel Architecture*. Intel, 2023.

[53] https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html. *DC Ultra RTL Synthesis*. Synopsys, 2021.

[54] Giuseppe Vietri, Liana V. Rodrigues, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. *Driving cache replacement with ML-based LeCaR*. hotStorage, 2018.

[55] Han Wang, Ki Suh Lee, Erluo Li, Chiun Lin Lim, Ao Tang, and Hakim Weatherspoon. *Timing is Everything: Accurate, Minimum Overhead, Available Bandwidth Estimation in High-Speed Wired Networks*. IMC, 2014.

[56] Maurice V Wilkes. *Slave memories and dynamic storage allocation*. IEEE Transactions Electronic Computers, 1965.

[57] Bruno Missi Xavier, Rafael Silva Guimarães, Giovanni Comarela, and Magnos Martinello. *Programmable Switches for in-Networking Classification*. INFOCOM, 2021.

[58] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. *GL-Cache: Group-level learning for efficient and high-performance caching*. FAST, 2023.

[59] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and K. V. Rashmi. *FIFO Queues are ALL You Need for Cache Eviction*. SOSP, 2023.

[60] Yazhuo Zhang, Juncheng Yang, Yao Yue, and Ymir Vigfusson. *SIEVE is Simpler than LRU: an Efficient Turn-Key Eviction Algorithm for Web Caches*. NSDI, 2024.

Reverie: Low Pass Filter-Based Switch Buffer Sharing for Datacenters with RDMA and TCP Traffic

Vamsi Addanki
TU Berlin

Wei Bai
Microsoft Research

Stefan Schmid
TU Berlin

Maria Apostolaki
Princeton University

Abstract

The switch buffers in datacenters today are dynamically shared by traffic classes with different loss tolerance and reaction to congestion signals. In particular, while legacy applications use loss-tolerant transport, e.g., DCTCP, newer applications require lossless datacenter transport, e.g., RDMA over Converged Ethernet. Unfortunately, as we analytically show in this paper, the buffer-sharing practices of today's datacenters pose a fundamental limitation to effectively *isolate* RDMA and TCP while also maximizing *burst absorption*. We identify two root causes: *(i)* the buffer-sharing for RDMA and TCP relies on two independent and often conflicting views of the buffer, namely ingress and egress; and *(ii)* the buffer-sharing scheme micromanages the buffer and overreacts to the changes in its occupancy during transient congestion.

In this paper, we present REVERIE, a buffer-sharing scheme, which, unlike prior works, is suitable for both lossless and loss-tolerant traffic, providing isolation and better burst absorption than state-of-the-art buffer-sharing schemes. At the core of REVERIE lies a unified (consolidated ingress and egress) admission control that jointly optimizes the buffers for both RDMA and TCP. REVERIE allocates buffer based on a low-pass filter that naturally absorbs bursty queue lengths during transient congestion within the buffer limits. Our evaluation shows that REVERIE can improve the performance of RDMA as well as TCP in terms of flow completion times by up to 33%.

1 Introduction

Network devices contain a buffer that can temporarily store excessive packets during congestion events. As the link speeds increase, maintaining a constant buffer-bandwidth ratio would require buffer memory to evolve faster than Moore's law and is hence impractical [33]. As a result, we observe buffer-per-Gbps to constantly shrink [12, 26] making performance problems rooted in buffer sharing more evident. Indeed, our expert survey, which included experts from six companies, revealed that buffer sharing is causing performance problems in most large-scale datacenters.

At a high level, the goal of a buffer-sharing scheme is to provide isolation between traffic classes, while maximizing the benefit of the buffer e.g., by absorbing bursts and achieving high throughput. Existing buffer management schemes (even recent ones) [1, 8, 15, 25] were designed considering exclusively loss-tolerant traffic (e.g., TCP variants). However, modern datacenters host traffic classes with different loss tolerance. Concretely, along with traditional loss-tolerant transport protocols, many clouds, e.g., Azure [11], Alibaba [22] and OCI [38], deploy RDMA over Converged Ethernet which requires lossless transport. In order to guarantee zero packet loss for RDMA, production datacenters enable Priority Flow Control (PFC) at the switches [11].

The co-existence of TCP and RDMA traffic in the switch buffer makes sharing the buffer particularly challenging. While, in principle, TCP and RDMA traffic have the same performance objectives (e.g., high throughput, low latency), their reaction to network events such as congestion is vastly different in terms of speed and granularity. A PFC pause proactively throttles RDMA traffic at *per-hop* granularity *before* the buffer fills up in order to prevent packet loss due to congestion. On the contrary, a packet drop throttles TCP at *per-flow* granularity once the buffer is filled up due to congestion. Moreover, the effect of PFC pause (in RDMA) on the buffer is not immediately evident as all incoming packets after the PFC has been triggered must be admitted in the buffer further increasing its occupancy. On the contrary, the effect of a packet drop (in TCP) on the buffer is immediately evident in the buffer as current packets do not further increase the buffer occupancy which can decrease proportionately to the aggregate port bandwidth. Since RDMA and TCP share the same switch buffer, congestion caused by TCP can result in excessive PFC pauses for RDMA and similarly the buffer occupied by RDMA (especially when it is paused) can result in excessive packet drops for TCP; leading to throughput degradation and poor burst absorption.

A naive approach for isolating RDMA and TCP in the shared buffer is to statically partition it e.g., dedicate 50% of the buffer to each class. However, such an approach will result in suboptimal burst absorption; and (in the worst case) poor throughput

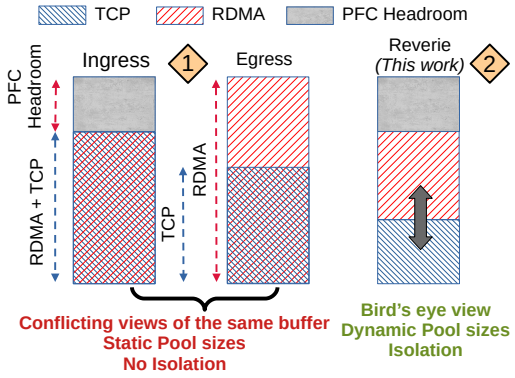


Figure 1: 1 Current buffer-sharing practices maintain two independent and at times conflicting views of the buffer i.e., ingress and egress, which are subdivided into various pools; posing a challenge to achieve isolation across RDMA and TCP traffic. 2 REVERIE maintains a bird's eye view of the buffer; effectively unifying ingress and egress admission control *without statically partitioning the buffer* to achieve isolation.

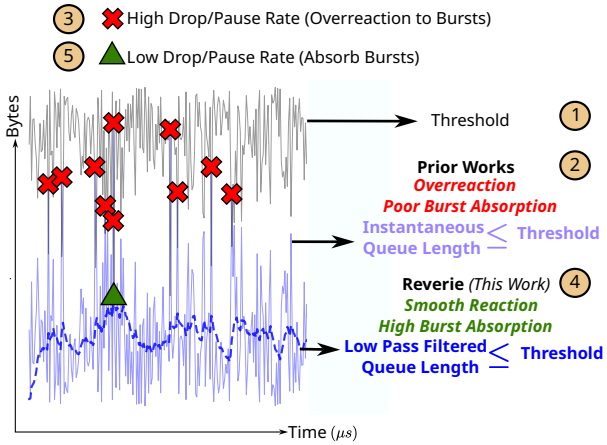


Figure 2: Prior works calculate 1 thresholds and compare against 2 instantaneous queue lengths, which leads to 3 overreaction to bursts and a high loss rate. REVERIE takes a different approach and compares the thresholds against 4 low pass filtered queue lengths. This allows REVERIE to 5 smoothly react to congestion while absorbing transient bursts.

when one of the two classes is not using its dedicated portion of the buffer. On the one hand, production-grade buffer-sharing schemes do not significantly depart from static partitioning among TCP and RDMA. The root cause of this pitfall is the unnecessary complex buffer model together with the use of pre-configured buffer pools i.e., pieces of buffer dedicated to certain queues only. On the other hand, research-grade buffer-sharing schemes such as ABM [1] can –at best– achieve steady-state isolation across traffic priorities only for loss-tolerant traffic, but would fail to isolate lossless and lossy traffic, even if we extend them to work in such settings, as we show in §2.3. Our goal in this paper is to formally navigate the trade-off between isolation and burst absorption in a setting where lossless and lossy traffic co-exist. Two key insights allow us to do so.

Our first key insight –after thoroughly studying the current buffer-sharing practices– is that although lossless and lossy traffic are, in practice, independently managed, the available buffer for both depends on each other's occupancy. Concretely, today's switches maintain two views of the buffer (i.e., ingress and egress); each of these views is virtually further split into pools i.e., buffer pieces configured to serve a subset of the queues. Figure 1 summarizes these views. The complexity of the buffer design stems from its evolution over the years from serving lossy traffic to serving both lossy and lossless. Unfortunately, as we show in this paper, this complex buffer design of today's datacenter switches leads to unexpected buffer issues e.g., lossy traffic gets more buffer allocation than lossless traffic when they compete, against the high-level objective of the configuration. Our analysis shows that buffer pools and the *independent* views of the buffer at the ingress and egress are the root causes of such issues. To tackle this problem, we propose a simple buffer-sharing scheme in which both RDMA and TCP are managed *jointly* with a bird's eye view of the buffer. Such an allocation scheme facilitates novel admission control schemes that can efficiently isolate RDMA and TCP without statically partitioning the buffer.

Our second key insight is that absorbing RDMA bursts is extremely challenging because the decisions of a buffer-sharing scheme are on a per-packet basis but PFC pause (required for RDMA) is a per-hop signal affecting not just the incoming packet but all the future arrivals from the previous hop. Moreover, sudden and concurrent fluctuations in multiple queue lengths can rapidly change the buffer occupancy. Worse yet, the incoming rate at today's link speeds can very rapidly fill in a buffer. Existing schemes [1, 8, 25] that were designed specifically for loss-tolerant traffic apply large thresholds to those packets that are classified as short flows or incast flows. Unfortunately, as we later show in this paper, these techniques cannot fundamentally achieve better burst absorption for lossless traffic since PFC works at *per-hop* or per-queue granularity and not per-packet i.e., burst absorption for RDMA requires prioritizing a queue experiencing burst (not just specific packets). To address this problem, we show that instead of increasing thresholds at per-packet granularity under bursty scenarios, it is sufficient to dampen the queue statistic (e.g., by a low-pass filter) against which thresholds are compared, when taking buffer decisions. Figure 2 illustrates our main idea. This essentially prioritizes queues experiencing bursts and improves the burst absorption capabilities of the buffer.

We present REVERIE, a buffer-sharing scheme suitable for modern datacenters hosting traffic with different loss tolerance. REVERIE jointly optimizes the buffer allocation for lossless and lossy traffic with a bird's eye view over the buffer; essentially unifying ingress and egress admission control as shown in Figure 1. Further, REVERIE significantly improves the burst absorption capabilities of the buffer by

comparing low-pass-filtered queue lengths against thresholds as illustrated in Figure 2.

Our extensive evaluation of REVERIE based on large-scale simulations in NS3 shows that REVERIE effectively isolates RDMA and TCP, reduces the overall number of PFC pauses by 60% on average and improves the flow completion times for bursty workloads by up to 33% compared to the state-of-the-art approaches.

In summary, our key contributions in this work are:

- The first analysis of a production-grade buffer model (exemplified by the open source SONiC [44]) that includes both ingress and egress admission control. This analysis generates multiple insights including the conflicting buffer views of ingress and egress that prevent effective isolation between lossless and lossy traffic.
- REVERIE, the first buffer-sharing scheme that can isolate lossless and lossy traffic while improving burst absorption for both.
- As a contribution to the research community and to facilitate future work, all our artifacts have been made publicly available at https://github.com/inet-tub/ns3-datacenter.

2 Motivation

We first present unintuitive outcomes (issues) that arise under typical configurations (§2.1). To explain the root cause of those issues (§2.3), we first describe in detail a representative buffer-sharing architecture of an open-source and widely-used switch OS, i.e., SONiC [44] (§2.2). Our buffer model has been endorsed by major Ethernet switch ASIC vendors, including Broadcom, Cisco and NVIDIA.

2.1 Buffer Issues in Datacenters

In this section, we walk through three issues that operators of large-scale RDMA deployments [11] can face while debugging buffer problems. We have verified that these issues are *(i)* possible by showing them analytically (§2.3); and *(ii)* realistic by direct communications with operators of large-scale RDMA deployments. Consider an operator who wants lossless traffic to get as much buffer as it needs, i.e., lossless is prioritized over lossy traffic under buffer contention. This is a typical use case in datacenters with large RDMA deployments. Although the operator closely follows the "best practices" (i.e., a set of heuristics) to configure the buffer, which we explain in §2.2.2, they observe the following issues.

Issue 1. *Lossy traffic gets more buffer allocation than lossless traffic when they both compete for buffer space.*

Issue 2. *Lossless traffic yields to the increase in buffer occupancy of lossy traffic, while the opposite is not true, i.e., the allocation for lossy traffic is not affected by the buffer occupancy of lossless traffic.*

Issue 3. *The buffer is more efficient in absorbing bursts of lossy traffic than bursts of lossless traffic.*

2.2 Buffer Sharing Practices

To understand the root cause of the issues (§2.3), we need to understand the buffer model used in today's datacenters and the "best practices" for configuring it. To the best of our knowledge, we are the first to present a detailed and up-to-date description of a buffer model of a datacenter switch that serves both lossless (e.g., RDMA) and lossy (e.g., TCP) traffic. We use SONiC [44], an open-source network operating system that is the closest we can get to the modus operandi for buffer management. SONiC runs on switch ASICs from multiple vendors, e.g., Broadcom, NVIDIA, Cisco and Intel, and has been widely deployed in Microsoft [11], Alibaba [48], LinkedIn [51] and Tencent. Importantly, our buffer model aligns with that of NVIDIA Onyx [47]. Hence, we believe our buffer model is representative of a broad range of scenarios and settings. We next describe the terminology and the configurable parts according to the buffer model of SONiC. We tabulate the important notations we use in Table 1.

Hereafter, we denote lossless by $\bullet$ and lossy by $\circ$. The switch uses a memory management unit (MMU) to manage the packet buffer.

Ingress and Egress Counters (Queues): The MMU maintains two types of *counters*[1], ingress denoted by $\leftarrow$ and egress denoted by $\rightarrow$ that serve admission control purposes. We henceforth refer to these counters as *queues*. Let Q be the set of all queues maintained by the MMU. Once a packet arrives at the switch, the packet is mapped to an ingress queue $(s,p) \in \overleftarrow{Q}$ based on the source port s and the packet's priority p[2]; and an egress queue $(d,p) \in \overrightarrow{Q}$ based on the destination port d. Ingress (egress) admission control acts over ingress (egress) queues. Each packet is admitted to the buffer if and only if the corresponding ingress and egress queues pass the ingress and egress admission controls. An arriving and admitted packet increases both the corresponding ingress and egress queues, while a departure packet decreases the queues. A packet is only buffered once regardless of the number of counters it is accounted by. Note that once a packet is admitted, it cannot be *pushed out* by new packet arrivals. A queue carrying lossless (lossy) traffic is known as a lossless (lossy) queue. Overall, the MMU maintains four sets of queues i.e., ingress lossless $\overleftarrow{Q}^{\bullet}$, ingress lossy $\overleftarrow{Q}^{\circ}$, egress lossless $\overrightarrow{Q}^{\bullet}$ and egress lossy $\overrightarrow{Q}^{\circ}$.

Buffer Size and Pools: The packet buffer has a total size of b. Current datacenter practices define pools that can intuitively be viewed as the buffer available for certain types of queues. In other words, the pool is a group of queues. The user can configure the buffer allocation policy, including the allocation algorithm, per-queue limit, and total size, for this group of queues. SONiC defines the following four pools:

- Ingress pool of size $\overleftarrow{b}$ shared by *both* ingress lossless and lossy queues, with an occupancy of $\overleftarrow{q}(t)$ at time t.

[1] Not to be confused with ingress and egress pipelines. Throughout this paper, ingress and egress are merely counters (referred to as queues).

[2] Most of the switch ASICs support 8 priorities. Operators typically map a packet to a priority based on its DSCP value.

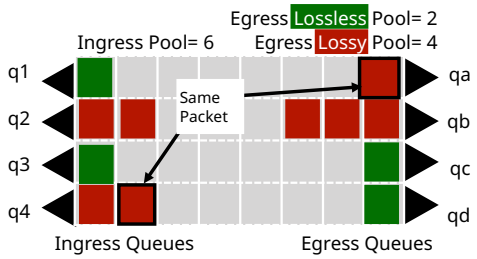


Figure 3: Buffer bookkeeping: packets are stored once but accounted twice; once in the ingress and once at the egress for admission control purposes.

- PFC headroom pool of size b_h used *only* by ingress lossless queues upon PFC pause (described next).
- Egress lossless pool of size $\overset{\bullet\rightarrow}{b}$ used by egress lossless queues, with an occupancy of $\overset{\bullet\rightarrow}{q}(t)$ at time t.
- Egress lossy pool of size $\overset{\circ\rightarrow}{b}$ used by egress lossy queues, with an occupancy of $\overset{\circ\rightarrow}{q}(t)$ at time t.

Note that pool sizes and occupancy are also *counters*. A packet can be counted in multiple pools while being buffered once (pools may overlap), thus the sum of all pools may exceed the actual buffer occupancy.

Figure 3 illustrates an example of the buffer bookkeeping. Packets are physically stored only once, but are accounted twice. For example, packet 5 is accounted in the ingress queue (counter) $q4$ and in the egress queue (counter) qa. In essence, all packets are accounted in the ingress pool, but packets of lossless queues are also accounted in the egress lossless pool, while packets of lossy queues are also accounted in the egress lossy pool.

Admission Control: Each queue i.e., counter $(i,p) \in Q$ at an input or output port denoted by i, corresponding to priority p is associated with a *threshold* $\Gamma_i^p(t)$ at time t. The admission control scheme compares the instantaneous length $q_p^i(t)$ of a queue against its threshold to make buffering decisions. Thresholds can be intuitively viewed as the maximum size of a queue. Once the queue hits the threshold, the switch will drop the incoming packet or send PFC pause frames to throttle the queue build-up. We emphasize that the switch cannot push out existing packets in the buffer to make room for the incoming packet.

2.2.1 Journey of a Packet in the Switch MMU

We walk through the various counters that are incremented and decremented during a packet's journey in the MMU. Recall that a packet can travel through the switch if and only if it satisfies both ingress and egress admission control.

Ingress Admission Control: The admission control in the ingress is different for lossy and lossless queues since as TCP (lossy) tolerates packet loss whereas RDMA (lossless) requires PFC and does not tolerate packet loss. The admission for ingress lossy queues is straightforward. If the ingress lossy queue hits the threshold (meaning if the length of the queue equals or exceeds the corresponding threshold devised by the buffer-sharing logic), the packet is dropped. Otherwise, both the ingress lossy queue and ingress pool counters are incremented upon admission and decremented as the packet departs to its destination.

In contrast, the admission control for ingress lossless queues is more complex and designed to achieve zero packet loss. If the ingress queue hits the threshold, the switch moves the queue to "paused" state and keeps sending PFC pause frames to the peer device. Then the arriving packet is admitted, but it increments the PFC headroom pool occupancy rather than the ingress pool occupancy. In other words, once an ingress lossless queue uses up its limit in the ingress pool, it starts to consume (or be accounted in) the PFC headroom pool. As the buffer drains, an ingress lossless queue under "paused" state first decrements its headroom pool occupancy, and then its ingress pool occupancy. When the headroom buffer occupancy is zero and the ingress pool occupancy is below the threshold, the switch moves the "paused" ingress lossless queue back to "resumed" state and sends PFC resume frames.

Egress Admission Control: Egress counters are straightforward. Egress queue length and pool occupancy based on the class of packet (lossy or lossless) are incremented upon admission and decremented as the buffer drains. The switch drops packets if egress queues hit thresholds.

2.2.2 Buffer Management

The MMU of the switch uses a buffer management algorithm that assigns thresholds to all ingress and egress queues. Dynamic Thresholds [15] (DT) is the state-of-the-art buffer management algorithm widely adopted by ASIC vendors [37, 45].

In a nutshell, DT calculates *dynamic* thresholds for each queue $(i,p) \in Q$ as the product of a configurable parameter α_p^i and the *remaining* buffer space in the corresponding pool. We refer the reader to Table 1 for the list of notations we use. In the following, we summarize DT's buffer management for lossless and lossy traffic, at ingress and egress queues.

$$\Gamma_p^i(t) = \alpha_p^i \times \begin{cases} \overset{\leftarrow}{b} - \overset{\leftarrow}{q}(t) & \textit{Ingress Lossless: } (i,p) \in \overset{\leftarrow\bullet}{Q} \\ \overset{\leftarrow}{b} - \overset{\leftarrow}{q}(t) & \textit{Ingress Lossy: } (i,p) \in \overset{\leftarrow\circ}{Q} \\ \overset{\bullet\rightarrow}{b} - \overset{\bullet\rightarrow}{q}(t) & \textit{Egress Lossless: } (i,p) \in \overset{\bullet\rightarrow}{Q} \\ \overset{\circ\rightarrow}{b} - \overset{\circ\rightarrow}{q}(t) & \textit{Egress Lossy: } (i,p) \in \overset{\circ\rightarrow}{Q} \end{cases} \quad (1)$$

In addition to the above threshold checks, the switch also uses the physical packet buffer limit as the last defense.

Having a better understanding of how the buffer works, we can go back to the operator's goal to prioritize lossless over lossy and explain how they would in practice configure the buffer. To avoid lossless packet drops, they would want to control lossless traffic *only* at the ingress with PFC thresholds. To make the problem easier to debug and more intuitive, they would want to control lossy traffic *only* at the egress with drop

Notation	Description
$\leftarrow$	Ingress
$\rightarrow$	Egress
$\bullet$	Lossless
$\circ$	Lossy
$*$	Shared
b	Total buffer size
b_h	Headroom pool size
Q	Set of all queues
$\overleftarrow{Q}^{\bullet}$	Set of ingress lossless queues
(i,p)	Queue at input or output port i, priority p
$q_i^p(t)$	Length of queue (i,p) at time t
$\Gamma_i^p(t)$	Threshold of queue (i,p) at time t
$\overrightarrow{q}^{\circ}(t)$	Occupancy of egress lossy pool
α_i^p	Parameter for queue (i,p)
$\overleftarrow{\alpha}^{\bullet}$	α for ingress lossless queues (for simplicity)
$\overrightarrow{\alpha}^{\circ}$	α for egress lossy queues (for simplicity)
$\overleftarrow{n}^{\bullet}$	# ingress lossless queues using buffer
$\overrightarrow{n}^{\circ}$	# egress lossy queues using buffer

Table 1: Important notations used in this paper.

thresholds. To this end, they would use the following buffer configuration heuristics. For simplicity, in this section, we set α_p^i to $\overleftarrow{\alpha}^{\bullet}$, $\overleftarrow{\alpha}^{\circ}$, $\overrightarrow{\alpha}^{\bullet}$ and $\overrightarrow{\alpha}^{\circ}$ for all ingress lossless, ingress lossy, egress lossless and egress lossy queues respectively.

Heuristic 1. *To avoid packet drops for lossless traffic at ingress, the sum of the ingress pool size $\overleftarrow{b}$ and headroom pool size b_h should be equal to (or smaller than) the total size of the switch buffer b i.e., $\overleftarrow{b}+b_h \leq b$.*

Heuristic 2. *To bypass egress admission control for lossless traffic and to allow fully utilizing the buffer space i.e., to avoid lossless packet drops at egress, we should set egress lossless pool size $\overrightarrow{b}^{\bullet}$ to total switch buffer size b and use an infinitely large egress lossless threshold i.e., $\overrightarrow{\alpha}^{\bullet} \gg 1$.*

Heuristic 3. *To avoid packet drops for lossy traffic at ingress, we should set ingress lossy threshold $\overleftarrow{\alpha}^{\circ}$ to an infinitely large value, and ensure that egress lossy pool size $\overrightarrow{b}^{\circ}$ is not larger than ingress pool size $\overleftarrow{b}$ i.e., $\overrightarrow{b}^{\circ} \leq \overleftarrow{b}$.*

2.3 Root Causes of the Buffer Issues

To systematically analyze the problems, we consider a fluid flow model with deterministic packet arrivals and analyze the steady state[3] of the buffer, similar to prior works [1, 15]. In the following, we explain our key findings intuitively. Our model and complete analysis can be found in Appendix A.

First, based on Heuristic 1, lossless traffic is allowed to use the entire buffer at the ingress i.e., $\overleftarrow{b}+b_h \leq b$. Further, lossless traffic is allowed to use the entire buffer at the egress i.e., $\overrightarrow{b}^{\bullet}=b$ based on Heuristic 2. Still, we end up with Issue 1.

[3] A steady state is achieved when the queue lengths stabilize i.e., packet arrival rate equals departure rate.

Root cause of Issue 1: To shed light on this issue, we analytically derive the aggregate buffer allocation $\overleftarrow{q}^{\bullet}$ to lossless queues and the aggregate buffer allocation $\overrightarrow{q}^{\circ}$ to lossy queues. Let $\overleftarrow{n}^{\bullet}$ and $\overrightarrow{n}^{\circ}$ denote the number of ingress lossless queues and egress lossy queues using the buffer respectively. We have:

$$\overleftarrow{q}^{\bullet} = \overleftarrow{b} \cdot \left(\frac{\overleftarrow{n}^{\bullet} \cdot \overleftarrow{\alpha}^{\bullet}}{1+\overleftarrow{n}^{\bullet} \cdot \overleftarrow{\alpha}^{\bullet}} \right) - \overrightarrow{b}^{\circ} \cdot \left(\frac{\overleftarrow{n}^{\bullet} \cdot \overleftarrow{\alpha}^{\bullet}}{1+\overleftarrow{n}^{\bullet} \cdot \overleftarrow{\alpha}^{\bullet}} \cdot \frac{\overrightarrow{n}^{\circ} \cdot \overrightarrow{\alpha}^{\circ}}{1+\overrightarrow{n}^{\circ} \cdot \overrightarrow{\alpha}^{\circ}} \right) \quad (2)$$

$$\overrightarrow{q}^{\circ} = \frac{\overrightarrow{n}^{\circ} \cdot \overrightarrow{\alpha}^{\circ} \cdot \overrightarrow{b}^{\circ}}{1+\overrightarrow{n}^{\circ} \cdot \overrightarrow{\alpha}^{\circ}} \quad (3)$$

Figure 4 illustrates the ratio of buffer allocated to lossy and lossless. Notice that for a sufficiently large number of lossless queues $\overleftarrow{n}^{\bullet}$ and lossy queues $\overrightarrow{n}^{\circ}$, the buffer allocation to lossless queues tends to $\overleftarrow{b} - \overrightarrow{b}^{\circ}$ (based on Equation 2) and the allocation for lossy queues tends to $\overrightarrow{b}^{\circ}$ (based on Equation 3). Unless $\overleftarrow{b} \geq 2 \times \overrightarrow{b}^{\circ}$, we end up with Issue 1, caused by the buffer pools. Specifically, although lossless queues are allowed to fully utilize the buffer, the egress lossy pool *effectively* overlaps with both ingress pool and egress lossless pools as shown in Figure 1, leading to Issue 1.

■ ***Takeaway.*** *The buffer is pre-fragmented in a way that makes enforcing high-level objectives through low-level configuration impossible. Current buffer sharing practices cannot prevent Issue 1 unless the ingress pool is at least twice as large as the egress lossy pool i.e., the buffer is statically partitioned.*

Second, according to Heuristic 2 and Heuristic 3, since lossless (lossy) bypasses egress (ingress) admission control, we would expect that lossless and lossy traffic are isolated in the buffer. While Issue 1 already suggests that lossy traffic may effectively get more buffer allocation than lossless traffic, we find yet another issue that lossy and lossless traffic interact in a surprisingly unfair manner: lossy traffic is effectively prioritized over lossless traffic (Issue 2) although our expert heuristics are intended otherwise.

Root cause of Issue 2: Notice that the buffer occupancy of lossy traffic at the egress equals its occupancy at the ingress i.e., $\overrightarrow{q}^{\circ} = \overleftarrow{q}^{\circ}$, since every packet is accounted both in the ingress and egress as shown in Figure 3. As a result, the overall ingress pool occupancy is the sum of egress lossy occupancy and the ingress lossless occupancy i.e., $\overleftarrow{q} = \overrightarrow{q}^{\circ} + \overleftarrow{q}^{\bullet}$. Using Equations 2 and 3, as well as the above relation, we derive the steady-state thresholds for ingress lossless (PFC thresholds) and egress lossy (drop thresholds) based on Equation 1.

Figure 5a shows how PFC thresholds for lossless queues vary depending on the number of ingress lossless queues using the buffer and the number of egress lossy queues. Interestingly, we find that the drop thresholds vary only according to the number of egress lossy queues (affected by the own buffer occupancy), but remain unchanged as the number of ingress lossless queues increases, see Figure 5b. We observe that current buffer sharing practices allow buffering lossy packets mostly independently of lossless traffic, but lossless traffic is

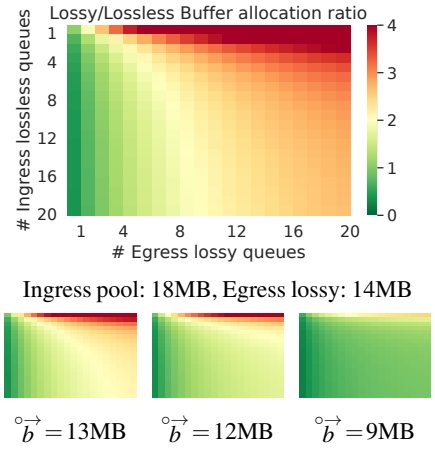
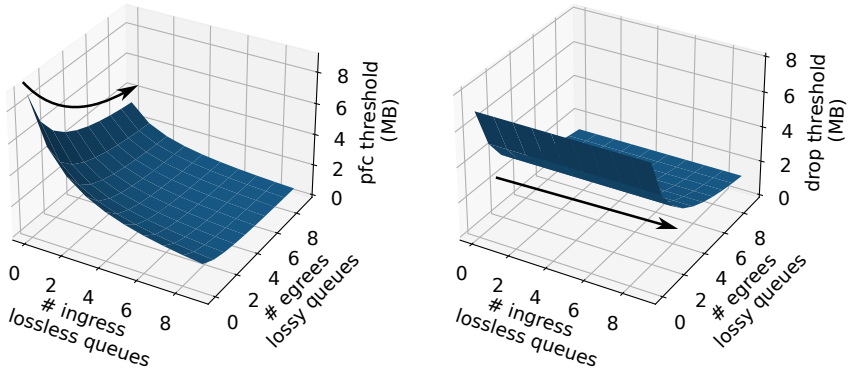


Figure 4: Isolation cannot be achieved with SONiC unless the ingress pool size $\overleftarrow{b}$ = 18MB is at least twice as large as egress lossy pool size $\overrightarrow{b}$ i.e., statically partitioning the buffer.



(a) PFC thresholds (for ingress lossless queues) are affected by both the number of ingress lossless and egress lossy queues.

(b) Drop thresholds (for egress lossy queues) are affected only by lossy queues.

Figure 5: Contrary to the expectation that lossless and lossy traffic are admitted independently in the buffer, lossless traffic is throttled due to the buffer occupancy of lossy, whereas lossy is admitted independent of the presence of lossless i.e., seemingly prioritizing lossy over lossless.

suppressed due to lossy traffic i.e., effectively prioritizing lossy over lossless. Our analysis reveals two root causes of Issue 2: *(i)* the egress lossy pool occupancy $\overrightarrow{q}$ which is used to calculate egress lossy drop thresholds (see Equation 1) does not account for lossless traffic whereas *(ii)* the ingress pool occupancy $\overleftarrow{q}$ which is used to calculate ingress lossless PFC thresholds (see Equation 1) accounts for both lossless and lossy traffic.

■ ***Takeaway.*** *Although lossless and lossy traffic are admitted seemingly independently by ingress and egress, the admission control for lossless traffic depends on both lossy and lossless occupancy, whereas the admission control for lossy traffic depends only on its own occupancy.*

It is natural to ask here whether these issues are due to the underlying buffer management scheme DT [15], and hence whether recent proposals such as ABM [1] should be able to avoid them. Although ABM can address isolation across various traffic priorities, it only works for a buffer-sharing architecture that supports lossy traffic but not lossless traffic. While, in theory, one could extend ABM to calculate the thresholds accordingly (by replacing Equations 1 within the same buffer architecture with the corresponding pools described in §2.2), both Issue 1 and Issue 2 would still hold. Indeed, if ABM operates in the default buffer architecture, it will (similar to DT) control lossless and lossy independently since lossless (lossy) bypasses egress (ingress) admission control. As a result, ABM cannot jointly impose fairness and isolate lossless and lossy traffic. Importantly, the technique used for burst absorption in ABM and other recent proposals [8, 25], i.e., prioritizing burst packets by using a high α parameter for thresholds, also results in Issue 3.

Root cause of Issue 3: Indeed, several recent works [1, 8, 25] rely on selectively taking action on burst and non-burst packets i.e., if the queue length exceeds its threshold, the buffer management scheme still accepts burst packets selectively by increasing the thresholds using a higher α parameter value only for those packets identified as burst. Figure 13 in Appendix A intuitively summarizes our key points. For example, if a non-burst (e.g., long flow) packet interleaves burst packets, the non-burst packet would be dropped since the queue length exceeds its threshold. However, a key property of PFC is that once PFC is triggered due to a non-burst packet, it affects all arriving traffic (including bursts) to the queue due to PAUSE frames. Selectively accepting packets does not apply for PFC.

■ ***Takeaway.*** *Optimally prioritizing bursts involves preferentially treating packets belonging to the burst only; this is possible for lossy but not for lossless traffic where congestion is signaled at a per-queue granularity.*

3 REVERIE

Based on the lessons learned from our analysis in §2, we design a buffer-sharing scheme REVERIE which prevents harmful interactions between lossless and lossy traffic (isolation) while absorbing bursts of both. We first describe the two pillars on which REVERIE relies: *(i)* consolidated admission control; and *(ii)* a low pass filter. Next, we explain how they fit together to form REVERIE. Finally, we discuss REVERIE's properties and its practicality.

3.1 Single Buffer Pool for Isolation

We argue that the first step towards achieving isolation is to have full visibility and control over the state of the buffer. Yet, current buffer-sharing practices maintain independent views and admissions at the ingress and egress, prohibiting global visibility and control. To address this, REVERIE uses a single shared buffer pool as shown in Figure 1, in

addition to a headroom pool dedicated to lossless queues. Further, REVERIE uses a single admission control that jointly optimizes the buffer allocation for lossless and lossy queues. The single shared buffer pool and a single admission control offer REVERIE a bird's eye view over the buffer.

Specifically, upon a packet arrival, REVERIE first determines the packet's class (lossless or lossy). If the packet belongs to lossless • class, REVERIE maps the packet to an ingress lossless queue $(s,p) \in \overleftarrow{Q}^{\bullet}$, where s is the source port that received the packet and p is the packet priority. Similarly, if the packet belongs to the lossy class, REVERIE maps the packet to an egress lossy queue $(d,p) \in \overrightarrow{Q}^{\circ}$, where d is the port to which the packet is destined and p is the packet priority.

As illustrated in Figure 6, REVERIE maintains only two types of queues (counters) i.e., ingress lossless and egress lossy, as opposed to the four types of queues in SONiC (see Figure 3). Further, REVERIE accounts for each packet only once as opposed to twice (once at ingress and once at egress) in the current buffer-sharing practices. In essence, all the lossless and lossy queues are mapped to the same (single) shared buffer pool as shown in Figure 1.

Using a single shared buffer pool and a single admission control leaves REVERIE solely responsible for fairly allocating the buffer across all queues to ensure isolation. In §3.3, we show how REVERIE's allocation achieves isolation across lossless and lossy.

3.2 Low-Pass Filter for Burst Absorption

A vast majority of prior works, including DT [15], FAB [8], ABM [1], TDT [25], calculate a threshold and compare it against *instantaneous* queue lengths in order to take buffer decisions. To improve burst absorption, prior works [1,8,25] selectively prioritize certain packets (e.g., short flows) by assigning them a larger threshold compared to the default threshold for other packets (e.g., long flows). However, as we explained in §2.3, selective packet prioritization cannot improve burst absorption for lossless traffic leading to Issue 3. It is essential to identify a queue experiencing a burst and prioritize all incoming traffic to the queue under bursty scenarios. A natural indicator for a queue that is experiencing a burst is its queue gradient, i.e., the rate of change of queue length. Thus, one could increase thresholds proportionally to the queue gradient. While intuitive, queue gradient is hard to monitor/calculate in practice in hardware, especially at microsecond granularity. To address this, we show an equivalence between an admission control based on queue gradient and an admission control based on first-order low-pass filtered queue lengths (Property 1). Our full proof can be found in Appendix B. Leveraging this equivalence, REVERIE uses an exponential weighted moving average which is an easy-to-implement first-order low-pass filter. In essence, REVERIE compares *average* queue lengths against a threshold, unlike prior works that use instantaneous queue lengths.

Property 1 (Relationship of low pass filter and gradient)**.** *Let Ψ be an admission control scheme that compares first order low pass filtered queue length $\hat{q}(t-\delta t)$ against a threshold $\Gamma(t)$ i.e., $\hat{q}(t-\delta t) \leq \Gamma(t)$, where $t-\delta t$ denotes the previous time instance. Let Φ be an admission control that compares instantaneous queue length $q(t)$ against Ψ's threshold $\Gamma(t)$ incremented proportionally based on the average queue gradient $\frac{d\hat{q}(t)}{dt}$ i.e., $q(t) \leq \Gamma(t) + K \cdot \frac{d\hat{q}(t)}{dt}$; where K is a constant and $\frac{d\hat{q}}{dt}$ is the gradient. Then, there exists a constant K such that Ψ and Φ are equivalent.*

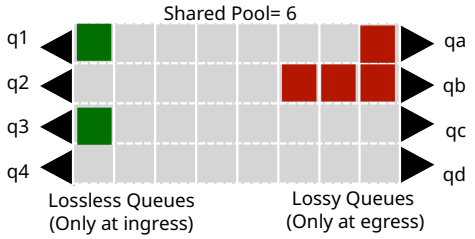


Figure 6: REVERIE's buffer bookkeeping: packets are stored once and accounted for once; lossless and lossy packets are accounted in the ingress and egress queues, respectively. All the queues are managed by a single admission control scheme.

3.3 The Workings of REVERIE

In this subsection, we put all the pieces together, to describe REVERIE's buffer-sharing architecture, admission control, and the underlying buffer management scheme.

Buffer sharing architecture: Let b be the total buffer space. REVERIE dedicates a headroom pool of size b_h for lossless traffic similar to the existing architecture. The rest of the shared buffer space denoted by $\overset{*}{b} = b - b_h$ is shared by both lossless and lossy traffic dynamically. A lossless packet is mapped to a lossless queue $(s,p) \in \overleftarrow{Q}^{\bullet}$ and a lossy packet is mapped to a lossy queue $(d,p) \in \overrightarrow{Q}^{\circ}$ based on the source port s, destination port d and the packet priority p. In total, REVERIE maintains a set $\overset{*}{Q}$ of queues, where $\overset{*}{Q} = \overleftarrow{Q}^{\bullet} \cup \overrightarrow{Q}^{\circ}$ consists of only two types of queues i.e., lossless and lossy instead of the four types maintained by SONiC.

Admission control: REVERIE calculates a threshold $\Gamma^i_p(t)$ for each queue (i,p) and compares it against the moving averaged queue length $\hat{q}^i_p(t)$ of the corresponding queue at time t i.e.,

$$\hat{q}^i_p(t) \leq \Gamma^i_p(t) \tag{4}$$

where $\hat{q}^i_p(t)$ is given by,

$$\hat{q}^i_p(t) = \overbrace{\gamma \cdot \hat{q}^i_p(t-\delta t)}^{\textit{Capture steady congestion}} + \overbrace{(1-\gamma) \cdot q^i_p(t)}^{\textit{Capture transient bursts}} \tag{5}$$

Here γ is a constant and a parameter for REVERIE, $q^i_p(t)$ is the instantaneous queue length, and $t-\delta t$ denotes the previous time instance. γ can be intuitively viewed as the degree of burst absorption. Since REVERIE's admission control (Equation 4) compares the threshold of a queue against its average queue

length (Equation 5), a higher γ masks the impact of transient bursts on the average queue lengths and, as a result, allows the admission control to absorb more transient bursts as illustrated in Figure 2. However, setting γ arbitrarily close to 1 makes admission control more oblivious to persistent or steady-state congestion. We leave it for future work to study the optimal parameter setting.

Any arriving packet that satisfies Equation 4 is buffered in the shared pool. If a lossy packet does not satisfy the above condition, the packet is simply dropped. However, if a lossless packet does not satisfy the above condition, the queue enters pause state (sends a PFC pause frame to its peer) and all the subsequent packets to the queue are buffered in the dedicated headroom irrespective of Equation 4. The queue sends out a PFC resume frame once the queue's headroom drains out completely and when Equation 4 is satisfied again.

Buffer management: The threshold $\Gamma^i_p(t)$ for each queue $(i,p) \in \overset{*}{Q}$ calculated by REVERIE at time t, depends on *(i)* a configurable parameter α_p for each priority, *(ii)* the number of congested queues n_p of priority p and the remaining shared buffer space $\overset{*}{b} - \overset{*}{q}(t)$ where $\overset{*}{b}$ is the size of shared buffer pool and $\overset{*}{q}(t)$ is the pool occupancy at time t. The threshold $\Gamma^i_p(t)$ is given by,

$$\Gamma^i_p(t) = \alpha_p \cdot \frac{1}{n_p} \cdot (\overset{*}{b} - \overset{*}{q}(t)) \quad \forall (i,p) \in \overset{*}{Q} \tag{6}$$

The thresholds used by REVERIE are similar to ABM [1] but we drop the dequeue rate factor due to the complexity of measuring it, especially when queues pause in the case of lossless traffic. However, our design does not prevent using the dequeue rate factor as well if it can be systematically measured.

3.4 The Properties of REVERIE

REVERIE inherits the steady-state isolation properties of ABM's thresholds for lossy as well as lossless traffic. Unlike ABM, though, which can only achieve isolation across priorities *within lossy* traffic, REVERIE can also achieve isolation *across lossless and lossy* priorities. In the following, for simplicity, we consider that all lossless and lossy queues are configured with the parameter value $\overset{\bullet}{\alpha}$ and $\overset{\circ}{\alpha}$ respectively. Our full proofs can be found in Appendix B. Next, we discuss our results intuitively.

In Theorem 1, we state the ratio in which buffer is allocated in the steady-state across lossless and lossy traffic in aggregate when both traffic classes compete for buffer. The ratio turns out to be the ratio of the configured α parameter. This makes it very intuitive and flexible to configure the buffer required for each traffic class, rather than the complicated pool sizes in the current practices.

Theorem 1 (Isolation)**.** *Under contention,* REVERIE *allocates buffer across lossless and lossy in the ratio of the corresponding* α *parameters i.e.,*

$$\frac{\overset{\bullet}{q}}{\overset{\circ}{q}} = \frac{\overset{\bullet}{\alpha}}{\overset{\circ}{\alpha}}$$

where $\overset{\bullet}{q}$ *and* $\overset{\circ}{q}$ *denote the steady-state shared buffer occupancy of lossless and lossy traffic respectively;* $\overset{\bullet}{\alpha}$ *and* $\overset{\circ}{\alpha}$ *denote the parameter values for lossless and lossy queues respectively.*

Based on Theorem 1, it is sufficient that the α parameter for lossless is greater than lossy in order to prevent issue 1. Further, since the thresholds are calculated with a bird's eye view of the buffer, the thresholds for both lossless and lossy depend on the overall buffer occupancy (see Equation 6). Hence, given that the α parameter for lossless is greater than lossy, REVERIE assigns a larger threshold for lossless compared to lossy i.e., prioritizing lossless over lossy. Essentially, REVERIE solves both issue 1 and issue 2 without statically partitioning the buffer.

When a single traffic class utilizes the buffer, REVERIE allocates $\frac{\alpha}{1+\alpha}$ fraction of the shared buffer, where α corresponds to the parameter value of the traffic class using the buffer. Notice that REVERIE allocates more buffer to a traffic class when it is not competing with the other class e.g., REVERIE allocates $\frac{\overset{\bullet}{\alpha} \cdot \overset{*}{b}}{1+\overset{\bullet}{\alpha}}$ amount of buffer when only lossless traffic is using the buffer compared to $\frac{\overset{\bullet}{\alpha} \cdot \overset{*}{b}}{1+\overset{\bullet}{\alpha}+\overset{\circ}{\alpha}}$ amount of buffer allocation for lossless when both traffic classes are competing for buffer space. Intuitively, REVERIE dynamically adapts the buffer allocation to lossless and lossy according to their load as opposed to the static pool sizes in the current practices. However, REVERIE keeps some buffer idle.

Theorem 2 (Buffer waste)**.** REVERIE *keeps idle a certain amount of buffer in the steady-state denoted by* b_w *given by,*

$$\frac{\overset{*}{b}}{1+\overset{\bullet}{\alpha}+\overset{\circ}{\alpha}} \leq b_w \leq \frac{\overset{*}{b}}{1+min(\overset{\bullet}{\alpha},\overset{\circ}{\alpha})}$$

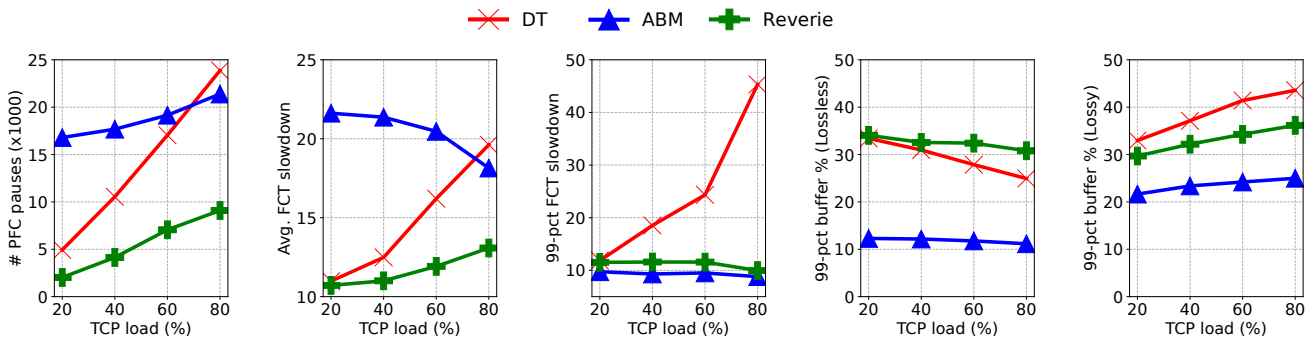
where $\overset{*}{b}$ *is the shared buffer pool size;* $\overset{\bullet}{\alpha}$ *and* $\overset{\circ}{\alpha}$ *are the parameter values for lossless and lossy queues correspondingly.*

Although REVERIE keeps a tiny buffer portion idle in the steady state, this helps in absorbing transient bursts. REVERIE effectively absorbs transient bursts even for lossless traffic since it compares *average* queue lengths against the threshold. Indeed, upon burst arrival, the average queue length hits the threshold slower than the instantaneous queue length, essentially absorbing transient bursts even for lossless queues. As a result, REVERIE finally solves issue 3.

3.5 Implementation Feasibility

A prototype implementation of REVERIE is beyond the scope of this paper and is part of our future work. Our discussions with NVIDIA already confirm that an approximation of the shared buffer pool model of REVERIE is feasible in hardware. In fact, we are currently discussing with a major Ethernet switch vendor on implementing REVERIE using their latest ASIC with programmable admission control features.

REVERIE is within reach of today's hardware because it does not significantly depart from commodity ASICs' buffer-sharing architecture and admission control mechanisms.



(a) PFC Pauses (b) RDMA incast flows (c) TCP short flows (d) RDMA buffer (e) TCP buffer

Figure 7: Buffer sharing under incast RDMA (DCQCN) workload and across various loads of websearch TCP (Cubic) workload. ABM achieves better performance for TCP but heavily penalizes RDMA, while REVERIE balances the two.

REVERIE introduces two primary changes. First, REVERIE uses a simpler buffer-sharing architecture with a single shared pool (excluding PFC headroom) and two types of queues. We have confirmed with NVIDIA that this architecture is supported by existing NVIDIA Ethernet switch ASICs[4]. Second, REVERIE uses a first-order low pass filter to obtain the moving averaged queue lengths. We believe this is practical as moving averages are used by common AQM like RED [20]. As previous works (e.g., ABM [1], [42]) have noted, various queue-length statistics are available and already used by the native buffer management of the MMU (although switch vendors do not open up the API for operators to do so on their own).

4 Evaluation

We evaluate the performance of REVERIE and compare it against the state-of-the-art approaches in the datacenter. Our evaluation aims to answer four main questions:

(Q1) *Can* REVERIE *protect RDMA from TCP?*

We find that REVERIE shields the performance of RDMA from TCP under various loads. At increased loads, REVERIE reduces the number of PFC pauses by 60% on average compared to DT and by 71.2% compared to ABM, with DCQCN as the transport protocol. When using advanced congestion control for RDMA, REVERIE reduces the number of PFC pauses by up to 100% compared to ABM.

(Q2) *Can* REVERIE *improve burst absorption of any class?*

We show that REVERIE significantly improves the burst absorption for RDMA and for TCP. With background TCP traffic (websearch), REVERIE improves the incast performance of RDMA by up to 33.3% compared to DT and by 50.4% compared to ABM. Under background RDMA traffic REVERIE improves the incast performance of TCP by up to 46.8% compared to DT and by up to 2.1% compared to ABM.

(Q3) *Does* REVERIE *penalize TCP?*

REVERIE does not penalize TCP. We find that REVERIE also improves the 99-percentile flow completion times (FCT) for short flows of TCP by 42.7% on average across various loads compared to DT. On this front, REVERIE is on par with ABM.

(Q4) *How sensitive is* REVERIE *to its parameters?*

We find an interesting characteristic of the parameter γ in REVERIE: increasing γ arbitrarily close to 1 dramatically reduces the number of PFC pauses, and improves the FCT for incast flows. However, beyond a certain value of γ, the infrequent PFC pauses negatively affect FCT. Finding the optimal γ value for a given switch remains an open question.

4.1 Setup

Our evaluation is based on network simulator NS3 [36].

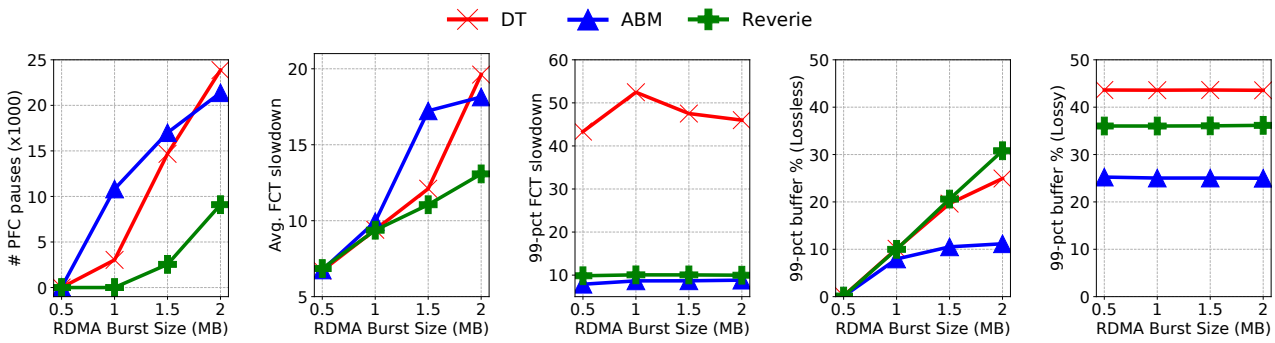
Topology: We consider a leaf-spine datacenter topology with 256 hosts organized into 4 spines and 16 leaves with 25Gbps links; link delay to 2μs (thus 17.28μs base RTT and 54KB bandwidth-delay product) and an oversubscription of 4, similarly to previous work [1, 2, 41]. All switches have 5.12KB buffer-per-port-per-Gbps similar to Broadcom Tomahawk [14][5]. All server NICs and switches are PFC enabled.

Traffic mix: We launch two types of workloads in our evaluation: *(i)* background and *(ii)* incast workloads. First, we generate background traffic across 20%-80% loads using websearch [5] flow size distribution, which is based on real-world datacenter measurements. Second, similar to prior works [1,2,4], we generate incast traffic using a synthetic workload that simulates the query-response behavior of a distributed file system. Specifically, each server in our topology sends out requests (queries) to all servers connected to a different leaf switch, chosen uniformly at random. These servers respond by sending a fraction of the file. We generate requests from each server based on a poisson process and we set the average request rate to 2 per second. We vary the file size (referred to as burst size). We use DCQCN [53] and PowerTCP [2] for RDMA congestion control; and Cubic [23] for TCP.

Baselines & metrics: We compare REVERIE with the SONiC [44] buffer model which is the state-of-the-art buffer sharing architecture widely deployed in today's datacenters.

[4]To implement this buffer-sharing architecture, we just need to map both egress lossy queues and egress lossless queues to a single egress buffer pool, and use an infinitely large egress lossless threshold.

[5]While Tomahawk splits the buffer across 4 MMUs, for simplicity, we assume a single MMU manages the entire buffer.



(a) PFC Pauses (b) RDMA incast flows (c) TCP short flows (d) RDMA buffer (e) TCP buffer

Figure 8: Buffer sharing under websearch TCP (Cubic) 80% load and across various burst sizes of incast RDMA (DCQCN) workload. As the burst size increases, the inability of DT and ABM to absorb RDMA bursts becomes more.

SONiC uses Dynamic Thresholds (DT) [15] as the buffer management scheme. The vast majority of the schemes in the literature are tailored for loss-tolerant traffic; thus it is unclear how those schemes can be evaluated in a fair manner for lossless traffic. To address this, we extend ABM [1] to support lossless traffic within the SONiC buffer model (after discussing with the authors) by accounting for the drain rate as well as the number of saturated queues in the ingress, and use it as a baseline. We report the following metrics: *(i)* number of PFC pauses triggered, *(ii)* average FCT slowdown for incast traffic, *(iii)* 99-percentile FCT slowdown for short flows of background traffic *(iv)* 99-percentile buffer occupancy of RDMA and *(v)* 99-percentile buffer occupancy of TCP, as a percentage of the total shared buffer.

Switch buffer configuration: We set the headroom pool size based on the NIC bandwidth and link delay, according to [49]. The remaining buffer is configured as ingress pool size. We set the egress lossless pool to the total switch buffer size and the egress lossy pool to 80% of the ingress pool size. For REVERIE, the headroom pool configuration is the same as stated earlier and the remaining buffer is configured as shared pool size. We set $\alpha = 1$ for all the schemes and set $\gamma = 0.999$ for REVERIE. We configure DCQCN according to [31], which is based on industry experience, and PowerTCP according to [2]. We set TCP minRTO to $1ms$.

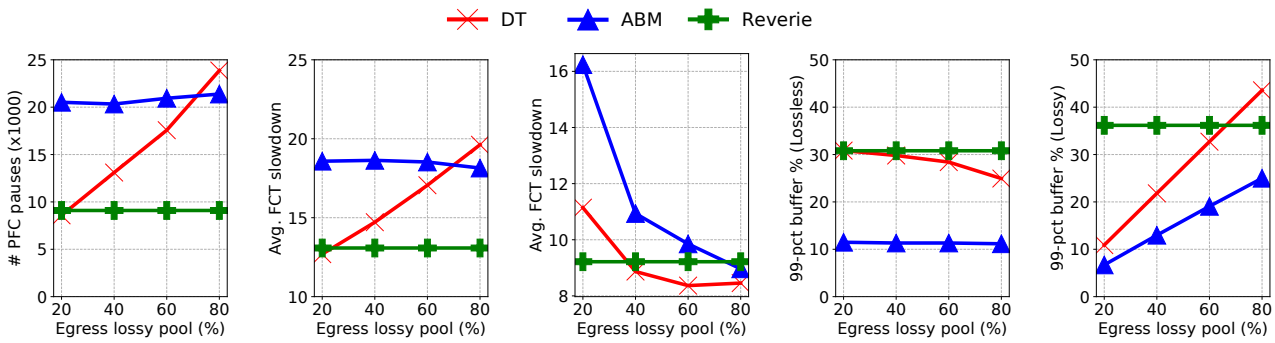
4.2 Results

REVERIE significantly reduces PFC pause rate: We generate TCP traffic using websearch workload and RDMA traffic using the incast workload in Figure 7 and Figure 8. RDMA uses DCQCN for congestion control. Across various loads of TCP and a burst size of 2MB for RDMA traffic, we observe from Figure 7a that REVERIE reduces the number of PFC pauses by 60% on average compared to DT and by 71.2% on average compared to ABM. Specifically, even at 20% TCP load, REVERIE reduces the number of PFC pauses by 58.9% compared to DT and by 87.9% compared to ABM. Further, across various burst sizes of RDMA with 80% TCP load, REVERIE reduces the number of PFC pauses by 61.8% on average compared to DT and by 57.4% on average compared to ABM, as shown in Figure 8a. In Figure 11 and Figure 12, we generate RDMA traffic using websearch workload and TCP traffic using the incast workload. We use PowerTCP as the congestion control for RDMA. From Figure 11a showing various RDMA loads at 1.5MB TCP burst size, and Figure 12a showing various TCP bursts at 80% RDMA load, we observe that REVERIE significantly reduces the PFC pauses by 100% compared to ABM, while REVERIE and DT perform similarly in this case. This confirms our observations in §2 on SONiC that lossy severely interacts with lossless traffic even though they are controlled independently by ingress and egress. REVERIE drastically reduces TCP's interference with RDMA.

REVERIE improves burst absorption for RDMA & TCP: Figure 7b, shows that across various TCP loads, REVERIE significantly reduces the average FCT for incast flows by 18.5% on average compared to DT and by 18.2% on average compared to ABM. At 80% TCP load, across various RDMA burst sizes, Figure 8b shows that REVERIE improves the average FCT for incast flows by 10% on average compared to DT and by 17% compared to ABM. This shows that REVERIE improves the overall burst absorption for RDMA. Although REVERIE's thresholds are similar to ABM's, REVERIE achieves better performance for lossless traffic due to its LPF-based admission control which favors transient bursts.

Across various loads of RDMA and 1.5MB TCP burst size, in Figure 11b, we see that REVERIE significantly reduces the average FCT for incast flows by 33.7% on average compared to DT and by 1.08% compared to ABM. From Figure 12b, across various TCP burst sizes, we observe that REVERIE reduces the average FCT for incast flows of TCP by up to 30% for large bursts compared to DT while REVERIE performs similarly to ABM. Overall, REVERIE's LPF-based admission control scheme improves burst absorption for RDMA as well as for TCP.

REVERIE protects RDMA from TCP in the buffer: Given the better burst absorption of REVERIE and significantly fewer PFC pauses even in the presence of TCP as seen in Figures 7-



(a) PFC pauses (b) RDMA incast flows (c) TCP long flows (d) RDMA buffer (e) TCP buffer

Figure 9: Buffer sharing under websearch TCP (Cubic) 80% load and incast RDMA (DCQCN) workload across various egress lossy pool sizes (% of ingress pool size) available for lossy TCP traffic. By changing the size of lossy pool, ABM and DT can only decide which traffic class will be prioritized against the other.

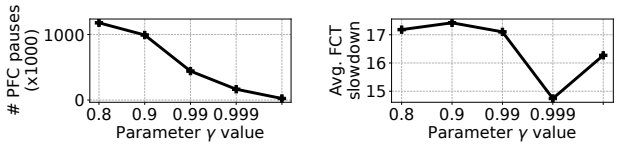


Figure 10: As the parameter γ value approaches 1, REVERIE becomes oblivious to the changes in the buffer and drastically reduces the number of PFC pauses. However, beyond a certain value, the lack of PFC pauses negatively impacts FCTs.

12, already shows that REVERIE successfully prevents TCP's interactions with RDMA. Moreover, from Figure 7d, we see that DT *reduces* its buffer allocation significantly for lossless traffic as the load of TCP increases, while *increasing* buffer allocation for TCP (Figure 7e). However, REVERIE gives lossless traffic its fair share in the buffer even at high TCP loads. Further, as the burst size of RDMA increases, from Figure 8d, we see that REVERIE opportunistically allocates increasingly buffer to RDMA whereas DT and ABM fail to allocate more buffers to RDMA, thus significantly increasing PFC pauses (Figure 8a). REVERIE's isolation properties allow RDMA to get its fair share of buffer even at high TCP loads. Reducing TCP's buffer share (egress lossy pool size) for DT and ABM improves the number of PFC pauses (Figure 9a) and flow completion times for RDMA incast flows, but TCP long flows suffer (Figure 9c) due to the reduced overall buffer available for TCP (Figure 9e). In contrast, REVERIE dynamically utilizes the entire shared buffer space in a fair manner and protects RDMA from TCP in the buffer as seen in Figures 7, 8, 9.

With advanced congestion control for RDMA (PowerTCP) under websearch workload, across various RDMA loads and TCP incasts (Figures 11d, 12d), REVERIE and DT as well as ABM occupy a significantly small portion of buffer and achieve similar FCTs for short flows of RDMA (Figures 11c, 12c). However, REVERIE triggers much lower PFC pauses than ABM as we observe in Figures 11a, 12a even with PowerTCP.

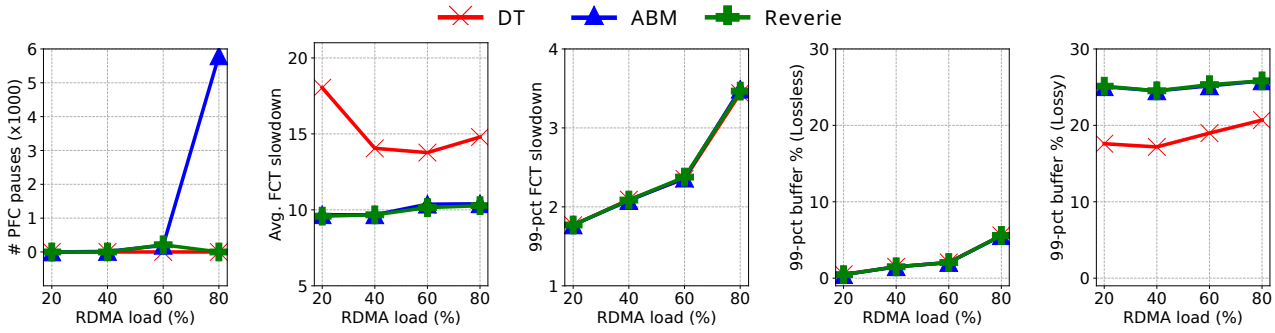
REVERIE also protects TCP in the buffer: Under websearch workload for TCP and RDMA incasts, from Figures 7c and 8c, we see that REVERIE and ABM achieve similar FCTs for short flows of TCP whereas DT severely penalizes TCP. DT penalizes TCP short flows even though it allocates more buffer to TCP compared to REVERIE as seen in Figures 7c, 8e. This excessive buffering results in increased queueing delays for DT. Further, under incast workload for TCP and websearch workload for RDMA, while REVERIE and ABM achieve similar FCTs for incast TCP flows, DT suffers from poor FCTs for TCP incasts (Figures 11b, 12b). Unlike DT, REVERIE and ABM protect TCP in the buffer.

Impact of LPF filtering: As discussed in §3.3, the parameter γ balances the capturing of steady-state (long-term) congestion against transient-state (short-term) congestion is captured by the admission control scheme. To better understand the impact of γ, we generate RDMA traffic using websearch workload at 80% load along with incast workload at 2MB burst size. In Figure 10, we show the number of PFC pauses and the average FCT for incast flows as a function of γ value. We observe that PFC pauses dramatically reduce as γ increases. Average FCT for incast flows decreases as γ increases until $\gamma = 0.999$. Yet, for $\gamma = 0.999999$ (close to 1), the average FCT increases by 9%. Naturally, a small γ value makes the admission control scheme highly sensitive to instantaneous queue lengths, which triggers PFC more frequently upon transient bursts. Similarly, a high γ value makes the admission control scheme insensitive to queue length and PFC is not triggered even when the queues *steadily* grow. In such cases, the excessive buffer occupied by steady-state traffic leaves less buffer to absorb transient bursts. Finding an optimal γ value is not required for reaping the benefits of REVERIE, as long as we avoid the extreme values that are easy to distinguish.

5 Related Work

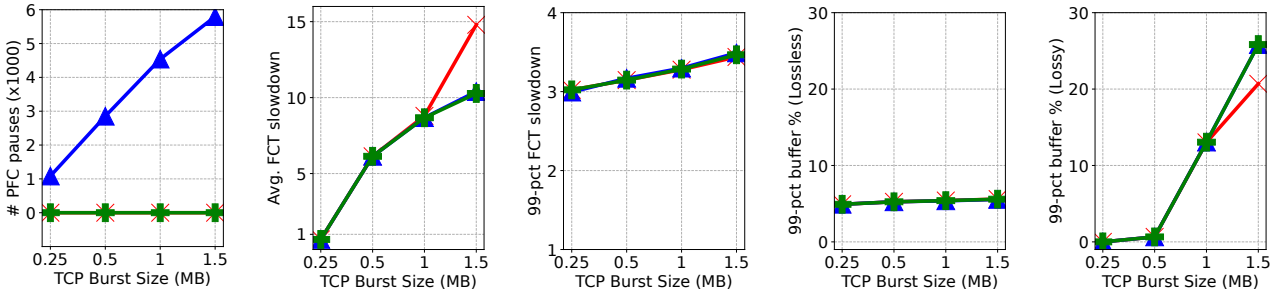
Our work relates to *(i)* buffer management; and *(ii)* RDMA.

Multiple works focus on sharing the on-chip buffer across queues of the same switch [1, 7–10, 13, 15, 18, 19, 29] and on sharing bandwidth across queues of the same port e.g.,



(a) PFC pauses (b) TCP incast flows (c) RDMA short flows (d) RDMA buffer (e) TCP buffer

Figure 11: Buffer sharing under incast TCP (with Cubic) and across various loads of websearch RDMA (with PowerTCP). ABM can only deal with low loads of RDMA traffic as it cannot distinguish or priortize it.



(a) PFC pauses (b) TCP incast flows (c) RDMA short flows (d) RDMA buffer (e) TCP buffer

Figure 12: Buffer sharing under websearch RDMA (with PowerTCP) at 80% load varying burst sizes of incast TCP (with Cubic). As TCP traffic increases, ABM further penalizes RDMA to protect TCP traffic.

AQM and scheduling [6, 20, 21, 35, 39]. In fact, there are also proposals to combine the two [16]. Further, augementing buffer sharing algorithms with machine-learned predictions has been shown to improve performance [3]. While useful, such works are designed exclusively for lossy traffic (i.e., TCP variants) and often with loss-based congestion control in mind. As a result, they are orthogonal to this work.

Many cloud providers have deployed RDMA over Ethernet to accelerate storage [11, 22], HPC, and ML [38]. To the best of our knowledge, all of these deployments [11, 22, 38] rely on PFC. Other research efforts related to RDMA include congestion control [2, 31, 53], efficient loss recovery [34], deadlock prevention [24], high performance RDMA applications [17, 26, 27, 30], testing [28], security [40, 46, 50] and performance isolation [52]. Among them, the most related topic is congestion control, but also in those works the buffer is only used by RDMA traffic [31, 53] (i.e., no TCP).

Coexistence of RDMA and TCP is an emerging new problem. Several recent parallel works proposed alternative solutions e.g., dynamically sharing the headroom buffer space under extremely shallow buffers [43]; using average occupancy time of packets to allocate buffers for each queue [32]. Yet, unlike REVERIE, these works do not address the fundamental issues that arise due to the static buffer pool configurations in today's switches. We leave it for future work to evaluate how the emerging alternative approaches fare against REVERIE's allocation.

6 Conclusion

This paper addresses the tension in buffer sharing between lossy traffic (e.g., TCP variants) and lossless traffic (e.g., RDMA) on datacenter switches. To this end, we first uncover, and explain analytically three particular unexpected buffer behaviors (issues) that today's buffer-sharing scheme can cause. Next, we find the root cause of these inefficiencies, and design a new buffer sharing scheme, REVERIE that can provide both isolation and high burst absorption to lossy and lossless traffic. In future work, we will try to closely work with a switch ASIC vendor to incorporate REVERIE into an ASIC's programmable admission control features.

Acknowledgements

We would like to thank our shepherd, Amy Ousterhout, as well as the anonymous reviewers for their useful feedback. We would like to thank Alex Shpiner, Eddy Kvetny, Matty Kadosh and Liron Mula from NVIDIA for many helpful discussions on the implementation feasibility of REVERIE. This work is part of a project that has received funding from the Austrian Science Fund (FWF), grant I 5025-N (DELTA), with Gabor Retvari, 2020-2024.

References

[1] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. Abm: Active buffer management in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 36–52, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3544216.3544252.

[2] Vamsi Addanki, Oliver Michel, and Stefan Schmid. PowerTCP: Pushing the performance limits of datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 51–70, Renton, WA, April 2022. USENIX Association. URL: https://www.usenix.org/conference/nsdi22/presentation/addanki.

[3] Vamsi Addanki, Maciej Pacut, and Stefan Schmid. Credence: Augmenting datacenter switch buffer sharing with ml predictions. In *21th USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, Santa Clara, CA, April 2024. USENIX Association. URL: https://www.usenix.org/conference/nsdi24/presentation/addanki-credence.

[4] Mohammad Alizadeh and Tom Edsall. On the data path performance of leaf-spine datacenter fabrics. In *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*, pages 71–74, 2013. doi:10.1109/HOTI.2013.23.

[5] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1851182.1851192.

[6] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. Pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 435–446, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2486001.2486031.

[7] Maria Apostolaki, Vamsi Addanki, Manya Ghobadi, and Laurent Vanbever. Fb: A flexible buffer management scheme for data center switches. *arXiv preprint arXiv:2105.10553*, 2021. URL: https://arxiv.org/abs/2105.10553.

[8] Maria Apostolaki, Laurent Vanbever, and Manya Ghobadi. Fab: Toward flow-aware buffer sharing on programmable switches. BS '19, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3375235.3375237.

[9] Mutlu Arpaci and John A. Copeland. Buffer management for shared-memory atm switches. *IEEE Communications Surveys & Tutorials*, 3(1):2–10, 2000. doi:10.1109/COMST.2000.5340716.

[10] James Aweya, Michel Ouellette, and Delfin Y Montuno. Buffer management scheme employing dynamic thresholds, September 7 2004. US Patent 6,788,697. URL: https://patents.google.com/patent/US6788697B1/en.

[11] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering azure storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, Boston, MA, April 2023. USENIX Association. URL: https://www.usenix.org/conference/nsdi23/presentation/bai.

[12] Wei Bai, Shuihai Hu, Kai Chen, Kun Tan, and Yongqiang Xiong. One more config is enough: Saving (dc)tcp for high-speed extremely shallow-buffered datacenters. *IEEE/ACM Transactions on Networking*, 29(2):489–502, 2021. doi:10.1109/TNET.2020.3032999.

[13] Andreas V Bechtolsheim and David R Cheriton. Per-flow dynamic buffer management, February 4 2003. US Patent 6,515,963. URL: https://patents.google.com/patent/US6515963B1/en.

[14] Broadcom Tomahawk. https://people.ucsc.edu/~warner/Bufs/tomahawk.

[15] A.K. Choudhury and E.L. Hahne. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Transactions on Networking*, 6(2):130–140, 1998. doi:10.1109/90.664262.

[16] Cisco. Intelligent buffer management on cisco nexus 9000 series switches. URL: https://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-738488.pdf.

[17] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014. USENIX Association. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87.

[18] F. Ertemalp, D.R. Cheriton, and A. Bechtolsheim. Using dynamic buffer limiting to protect against belligerent flows in high-speed networks. In *Proceedings Ninth International Conference on Network Protocols. ICNP 2001*, pages 230–240, 2001. doi:10.1109/ICNP.2001.992903.

[19] Ruixue Fan, A. Ishii, B. Mark, G. Ramamurthy, and Qiang Ren. An optimal buffer management scheme with dynamic thresholds. In *Seamless Interconnection for Universal Services. Global Telecommunications Conference. GLOBECOM'99. (Cat. No.99CH37042)*, volume 1B, pages 631–637 vol. 1b, 1999. doi:10.1109/GLOCOM.1999.830130.

[20] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993. doi:10.1109/90.251892.

[21] Sally Floyd, Ramakrishna Gummadi, Scott Shenker, et al. Adaptive red: An algorithm for increasing the robustness of red's active queue management, 2001.

[22] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533. USENIX Association, April 2021. URL: https://www.usenix.org/conference/nsdi21/presentation/gao.

[23] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: A new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, jul 2008. doi:10.1145/1400097.1400105.

[24] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Tagger: Practical pfc deadlock prevention in data center networks. CoNEXT '17, page 451–463, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3143361.3143382.

[25] Sijiang Huang, Mowei Wang, and Yong Cui. Traffic-aware buffer management in shared memory switches. *IEEE/ACM Transactions on Networking*, 30(6):2559–2573, 2022. doi:10.1109/TNET.2022.3173930.

[26] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, February 2019. USENIX Association. URL: https://www.usenix.org/conference/nsdi19/presentation/kalia.

[27] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia.

[28] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding performance anomalies in RDMA subsystems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 287–305, Renton, WA, April 2022. USENIX Association. URL: https://www.usenix.org/conference/nsdi22/presentation/kong.

[29] S. Krishnan, A.K. Choudhury, and F.M. Chiussi. Dynamic partitioning: a mechanism for shared memory management. In *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*, volume 1, pages 144–152 vol.1, 1999. doi:10.1109/INFCOM.1999.749262.

[30] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 90–103, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3341302.3342071.

[31] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on*

Data Communication, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3341302.3342085.

[32] Yi Liu, Jiangping Han, Kaiping Xue, Ruidong Li, and Jian Li. L2bm: Switch buffer management for hybrid traffic in data center networks. In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*, pages 1–11, 2023. doi:10.1109/ICDCS57875.2023.00076.

[33] Matt Mathis and Andrew McGregor. Buffer sizing: a position paper. URL: https://buffer-workshop.stanford.edu/papers/paper16.pdf.

[34] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 313–326, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3230543.3230557.

[35] Kathleen Nichols and Van Jacobson. Controlling queue delay: A modern aqm is just one piece of the solution to bufferbloat. *Queue*, 10(5):20–34, may 2012. doi:10.1145/2208917.2209336.

[36] ns-3. Network simulator. URL: https://www.nsnam.org/.

[37] Eugene Opsasnick. Buffer management and flow control mechanism including packet-based dynamic thresholding. *US patent US7953002B2*. URL: https://patents.google.com/patent/US7953002B2/en.

[38] Oracle Cloud Infrastructure Blog: First principles: Building a high-performance network in the public cloud. URL: https://blogs.oracle.com/cloud-infrastructure/post/building-high-performance-network-in-the-cloud.

[39] Rong Pan, Preethi Natarajan, Chiara Piglione, Mythili Suryanarayana Prabhu, Vijay Subramanian, Fred Baker, and Bill VerSteeg. Pie: A lightweight control scheme to address the bufferbloat problem. In *2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR)*, pages 148–155, 2013. doi:10.1109/HPSR.2013.6602305.

[40] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMArk: Bypassing RDMA security mechanisms. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4277–4292. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/rothenberger.

[41] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa Ammar, Ellen Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, and Amin Vahdat. Annulus: A dual congestion control loop for datacenter and wan traffic aggregates. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 735–749, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3387514.3405899.

[42] Danfeng Shan, Wanchun Jiang, and Fengyuan Ren. Absorbing micro-burst traffic by enhancing dynamic threshold policy of data center switches. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 118–126, 2015. doi:10.1109/INFOCOM.2015.7218374.

[43] Danfeng Shan, Yuqi Liu, Tong Zhang, Yifan Liu, Yazhe Tang, Hao Li, and Peng Zhang. Less is more: Dynamic and shared headroom allocation in pfc-enabled datacenter networks. In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*, pages 591–602, 2023. doi:10.1109/ICDCS57875.2023.00019.

[44] SONiC. Software for Open Networking in the Cloud. URL: https://sonic-net.github.io/SONiC/.

[45] Tim Stevenson. Nexus 9000 architecture, 2020. URL: https://www.ciscolive.com/c/dam/r/ciscolive/emea/docs/2020/pdf/BRKDCN-3222.pdf.

[46] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. sRDMA – efficient NIC-based authentication and encryption for remote direct memory access. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 691–704. USENIX Association, July 2020. URL: https://www.usenix.org/conference/atc20/presentation/taranov.

[47] NVIDIA Onyx User Manual v3.10.4302 (LTS). Shared Buffers. URL: https://docs.nvidia.com/networking/display/Onyxv3104302/Shared+Buffers.

[48] Haiyong Wang. OCPUS18 – SONiC Development and Deployment at Alibaba, 2018. URL: https://www.youtube.com/watch?v=aSd3R3gnQtw.

[49] CISCO white paper. Priority flow control: build reliable layer-2 infrastructure. 2009.

[50] Jiarong Xing, Kuo-Feng Hsu, Yiming Qiu, Ziyang Yang, Hongyi Liu, and Ang Chen. Bedrock: Programmable network support for secure RDMA systems. In *31st USENIX*

Security Symposium (USENIX Security 22), pages 2585–2600, Boston, MA, August 2022. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/xing.

[51] Zhenggen Xu. OCPSummit19 - EW: SONiC - LinkedIn Adoption of OCP SONiC, 2019. URL: https://www.youtube.com/watch?v=skUnjqPOvXs.

[52] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. Justitia: Software Multi-Tenancy in hardware Kernel-Bypass networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1307–1326, Renton, WA, April 2022. USENIX Association. URL: https://www.usenix.org/conference/nsdi22/presentation/zhang-yiwen.

[53] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2785956.2787484.

A Analysis of SONiC

In this section, we formally model the buffer sharing scheme of SONiC [44] as well as REVERIE and analyze their buffer allocation properties. Our analysis aims to understand the steady-state behavior of the buffer. We build upon the model and the notation introduced in §2. Specifically, we analyze a shared buffer switch architecture whose buffer sharing scheme supports both lossless traffic and lossy traffic. Our analysis is based on a fluid flow model with deterministic packet arrival rates, extending [1, 15].

We begin by analyzing the drop thresholds for lossy traffic. As we study the steady-state of the buffer, we drop the time variables in our notation for ease of presentation. Since every packet is accounted both in the ingress and egress, the following relations hold at all times: *(i)* lossy traffic buffer occupancy $\overset{\circ}{\underset{\rightarrow}{q}}$ at egress equals its occupancy $\overset{\circ}{\underset{\leftarrow}{q}}$ at ingress; *(ii)* lossless traffic buffer occupancy at egress $\overset{\bullet}{\underset{\rightarrow}{q}}$ equals its occupancy at the ingress pool $\overset{\bullet}{\underset{\leftarrow}{q}}$ plus headroom occupancy q_h; *(iii)* ingress pool occupancy $\underset{\leftarrow}{q}$ equals the sum of occupancy of lossless $\overset{\bullet}{\underset{\leftarrow}{q}}$ (without PFC headroom) and lossy $\overset{\circ}{\underset{\leftarrow}{q}}$ traffic occupancy at ingress.

$$\overset{\circ}{\underset{\rightarrow}{q}} = \overset{\circ}{\underset{\leftarrow}{q}} \tag{7}$$

$$\overset{\bullet}{\underset{\rightarrow}{q}} = \overset{\bullet}{\underset{\leftarrow}{q}} + q_h \tag{8}$$

$$\underset{\leftarrow}{q} = \overset{\circ}{\underset{\leftarrow}{q}} + \overset{\bullet}{\underset{\leftarrow}{q}} \tag{9}$$

Based on the egress admission control for lossy traffic i.e., Dynamic Thresholds (see Equation 1 in §2), if $\overset{\circ}{\underset{\rightarrow}{n}}$ egress lossy queues are in the steady state, then the total buffer occupancy is $\overset{\circ}{\underset{\rightarrow}{q}} = \overset{\circ}{\underset{\rightarrow}{n}} \cdot (\underset{\rightarrow}{b} - \overset{\circ}{\underset{\rightarrow}{q}})$. By rearranging the terms, we obtain the egress lossy pool occupancy:

$$\overset{\circ}{\underset{\rightarrow}{q}} = \frac{\overset{\circ}{\underset{\rightarrow}{n}} \cdot \underset{\rightarrow}{\alpha} \cdot \underset{\rightarrow}{b}}{1 + \overset{\circ}{\underset{\rightarrow}{n}} \cdot \underset{\rightarrow}{\alpha}} \tag{10}$$

Similarly, based on the ingress admission control for lossless traffic, if $\overset{\bullet}{\underset{\leftarrow}{n}}$ lossless queues are in the steady state, then the total buffer occupancy is $\overset{\bullet}{\underset{\leftarrow}{q}} = \overset{\bullet}{\underset{\leftarrow}{n}} \cdot \underset{\leftarrow}{\alpha} \cdot (\underset{\leftarrow}{b} - \overset{\bullet}{\underset{\leftarrow}{q}})$. Substituting in Equation 9 and using Equation 10 for $\overset{\circ}{\underset{\rightarrow}{q}}$, we obtain the following:

$$\underset{\leftarrow}{q} = \underset{\leftarrow}{b} \cdot \left(\frac{\overset{\bullet}{\underset{\leftarrow}{n}} \cdot \underset{\leftarrow}{\alpha}}{1 + \overset{\bullet}{\underset{\leftarrow}{n}} \cdot \underset{\leftarrow}{\alpha}} \right) + \overset{\circ}{\underset{\rightarrow}{b}} \cdot \left(\frac{1}{1 + \overset{\bullet}{\underset{\leftarrow}{n}} \cdot \underset{\leftarrow}{\alpha}} \cdot \frac{\overset{\circ}{\underset{\rightarrow}{n}} \cdot \underset{\rightarrow}{\alpha}}{1 + \overset{\circ}{\underset{\rightarrow}{n}} \cdot \underset{\rightarrow}{\alpha}} \right)$$

Finally, substituting $\underset{\leftarrow}{q}$ from above in $\overset{\bullet}{\underset{\leftarrow}{q}} = \overset{\bullet}{\underset{\leftarrow}{n}} \cdot \underset{\leftarrow}{\alpha} \cdot (\underset{\leftarrow}{b} - \underset{\leftarrow}{q})$, we obtain $\overset{\bullet}{\underset{\leftarrow}{q}}$, the buffer occupied by lossless traffic at the ingress pool.

$$\overset{\bullet}{\underset{\leftarrow}{q}} = \underset{\leftarrow}{b} \cdot \left(\frac{\overset{\bullet}{\underset{\leftarrow}{n}} \cdot \underset{\leftarrow}{\alpha}}{1 + \overset{\bullet}{\underset{\leftarrow}{n}} \cdot \underset{\leftarrow}{\alpha}} \right) - \overset{\circ}{\underset{\rightarrow}{b}} \cdot \left(\frac{\overset{\bullet}{\underset{\leftarrow}{n}} \cdot \underset{\leftarrow}{\alpha}}{1 + \overset{\bullet}{\underset{\leftarrow}{n}} \cdot \underset{\leftarrow}{\alpha}} \cdot \frac{\overset{\circ}{\underset{\rightarrow}{n}} \cdot \underset{\rightarrow}{\alpha}}{1 + \overset{\circ}{\underset{\rightarrow}{n}} \cdot \underset{\rightarrow}{\alpha}} \right) \tag{11}$$

Overall, our steady-state analysis gives the amount of buffer occupied by lossless traffic (Equation 11) and lossy traffic (Equation 10) based on the buffer configuration and the state of the buffer i.e., the number of active queues of each class. Following the admission control of Dynamic Thresholds from Equation 1, we could compute the drop thresholds for egress lossy and PFC thresholds for ingress lossless by using each pool occupancy from above.

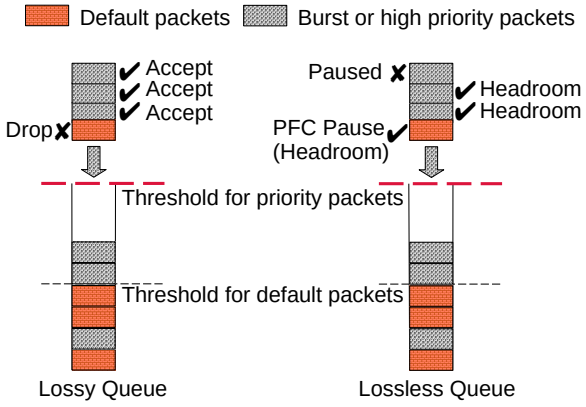


Figure 13: Per-packet prioritization cannot increase burst absorption for lossless traffic since PFC works at per-queue granularity.

B Analysis of REVERIE

In this section, we formally prove the properties of REVERIE stated in this paper. Before analyzing the steady-state behavior of REVERIE, we first show the relation between low-pass filters and gradient which builds the intuition for our low-pass filter-based approach.

Property 1 (Relationship of low pass filter and gradient)**.** *Let Ψ be an admission control scheme that compares first order low pass filtered queue length $\hat{q}(t-\delta t)$ against a threshold $\Gamma(t)$ i.e., $\hat{q}(t-\delta t) \leq \Gamma(t)$, where $t-\delta t$ denotes the previous time instance. Let Φ be an admission control that compares instantaneous queue length $q(t)$ against Ψ's threshold $\Gamma(t)$ incremented proportionally based on the average queue gradient $\frac{d\hat{q}(t)}{dt}$ i.e., $q(t) \leq \Gamma(t) + K \cdot \frac{d\hat{q}(t)}{dt}$; where K is a constant and $\frac{d\hat{q}}{dt}$ is the gradient. Then, there exists a constant K such that Ψ and Φ are equivalent.*

Proof. We consider exponentially weighted moving average for the first order low pass filter in this context. Let $q(t)$ denote the instantaneous queue length and let $\hat{q}(t)$ denote the average queue length. We denote the moving average parameter by ν. The moving average of the instantaneous queue lengths is then as follows:

$$\hat{q}(t) = \nu \cdot q(t) + (1-\nu) \cdot \hat{q}(t-\delta t)$$

where δt denotes the previous time when the average was updated. By rearranging the terms and dividing by δt,

$$\frac{\hat{q}(t) - \hat{q}(t-\delta t)}{\delta t} = \frac{\nu}{\delta t} \cdot (q(t) - \hat{q}(t-\delta t))$$

Let $K = \frac{\delta}{\nu}$. Using Euler's approximation method, we obtain the following:

$$K \cdot \frac{d\hat{q}(t)}{dt} = q(t) - \hat{q}(t-\delta t) \tag{12}$$

Using this relation, we now prove that Ψ and Φ admission control schemes are equivalent. We begin with Ψ which compares average queue lengths against a threshold $\Gamma(t)$ at time t.

$$\hat{q}(t-\delta t) \leq \Gamma(t)$$

Using Equation 12, we convert the above inequality as follows:

$$q(t) - K \cdot \frac{d\hat{q}(t)}{dt} \leq \Gamma(t)$$

By rearranging the terms, we obtain the the admission control scheme Φ. Hence Ψ and Φ are equivalent.

$$q(t) \leq \Gamma(t) + K \cdot \frac{d\hat{q}(t)}{dt}$$

□

Property 1 underlies the design of REVERIE's admission control based on LPF. We next prove the steady-state properties of REVERIE. Notice that by definition, the average queue length equals instantaneous queue length in steady-state. Due to this, REVERIE's steady-state properties are largely inherited from ABM. REVERIE differs from ABM in that, REVERIE's admission control is based on a single shared pool, where as ABM in the SONiC model would assign thresholds based on the pool size and occupancies for each priority (within a class). ABM cannot dynamically allocate buffer across RDMA and TCP since the pool sizes are fixed in the SONiC buffer model.

We next analyze the buffer allocation ratio for lossless and lossy.

Theorem 1 (Isolation)**.** *Under contention,* REVERIE *allocates buffer across lossless and lossy in the ratio of the corresponding α parameters i.e.,*

$$\frac{\overset{\bullet}{q}}{\overset{\circ}{q}} = \frac{\overset{\bullet}{\alpha}}{\overset{\circ}{\alpha}}$$

where $\overset{\bullet}{q}$ and $\overset{\circ}{q}$ denote the steady-state shared buffer occupancy of lossless and lossy traffic respectively; $\overset{\bullet}{\alpha}$ and $\overset{\circ}{\alpha}$ denote the parameter values for lossless and lossy queues respectively.

Proof. Since under steady-state, average and instantaneous converge, we simply use instantaneous values to prove our claim. Let $\overset{\bullet}{n}$ and $\overset{\circ}{n}$ be the number of congested queues of lossless and lossy. REVERIE allocates a total of $\overset{\bullet}{q} = \overset{\bullet}{n} \cdot \overset{\bullet}{\alpha} \cdot \frac{1}{\overset{\bullet}{n}} \cdot (\overset{*}{b} - \overset{*}{q}) = \overset{\bullet}{\alpha} \cdot (\overset{*}{b} - \overset{*}{q})$ to $\overset{\bullet}{n}$ lossless and a total of $\overset{\circ}{q} = \overset{\circ}{n} \cdot \overset{\circ}{\alpha} \cdot \frac{1}{\overset{\circ}{n}} \cdot (\overset{*}{b} - \overset{*}{q}) = \overset{\circ}{\alpha} \cdot (\overset{*}{b} - \overset{*}{q})$ to $\overset{\circ}{n}$ lossy queues. Since both lossy and lossless are mapped to the shared pool, we have that $\overset{\bullet}{q} + \overset{\circ}{q} = \overset{*}{q}$. But substituting the previous relations, we obtain:

$$\overset{*}{q} = \frac{(\overset{\bullet}{\alpha} + \overset{\circ}{\alpha}) \cdot \overset{*}{b}}{1 + (\overset{\bullet}{\alpha} + \overset{\circ}{\alpha})} \tag{13}$$

$$\overset{\bullet}{q} = \frac{\overset{\bullet}{\alpha} \cdot \overset{*}{b}}{1 + (\overset{\bullet}{\alpha} + \overset{\circ}{\alpha})} \tag{14}$$

$$\overset{\circ}{q} = \frac{\overset{\circ}{\alpha} \cdot \overset{*}{b}}{1 + (\overset{\bullet}{\alpha} + \overset{\circ}{\alpha})} \tag{15}$$

From the above relations, it is easy to see that the ratio $\frac{\overset{\bullet}{q}}{\overset{\circ}{q}} = \frac{\overset{\bullet}{\alpha}}{\overset{\circ}{\alpha}}$. □

Theorem 2 (Buffer waste)**.** REVERIE *keeps idle a certain amount of buffer in the steady-state denoted by b_w given by,*

$$\frac{\overset{*}{b}}{1 + \overset{\bullet}{\alpha} + \overset{\circ}{\alpha}} \leq b_w \leq \frac{\overset{*}{b}}{1 + min(\overset{\bullet}{\alpha}, \overset{\circ}{\alpha})}$$

where $\overset{*}{b}$ is the shared buffer pool size; $\overset{\bullet}{\alpha}$ and $\overset{\circ}{\alpha}$ are the parameter values for lossless and lossy queues correspondingly.

Proof. Our proof follows from the proof of Theorem 2. Specifically, for $\overset{\bullet}{n}$ lossless queues and $\overset{\circ}{n}$ lossy queues, REVERIE allocates $\overset{*}{q}$ in aggregate given by Equation 13. The remaining buffer $\overset{*}{b} - \overset{*}{q}$ which is wasted in the steady state is then given by,

$$b_w \geq \overset{*}{b} - \overset{*}{q} = \frac{\overset{*}{b}}{1 + (\overset{\bullet}{\alpha} + \overset{\circ}{\alpha})}$$

However, if a traffic class eg., lossless does not use the buffer, we can derive the remaining shared pool buffer similar to above. Depending on the smallest α value across all traffic

classes, when such a class uses the buffer alone, then in this case the buffer waste is given by,

$$b_w \leq \overset{*}{b} - \overset{*}{q} = \frac{\overset{*}{b}}{1 + (min(\overset{\bullet}{\alpha}, \overset{\circ}{\alpha}))}$$

□

We believe that REVERIE not only has interesting steady-state properties but its low pass filter based admission control find its best benefit under transient state analysis. We plan to analyze more such properties in the future.

Precise Data Center Traffic Engineering with Constrained Hardware Resources

Shawn Shuoshuo Chen ♠ Keqiang He ♦ Rui Wang ♣ Srinivasan Seshan ♠ Peter Steenkiste ♠

♠ *Carnegie Mellon University* ♦ *Airbnb* ♣ *Google*

Abstract

Data center traffic engineering (TE) routes flows over a set of available paths following custom weight distributions to achieve optimal load balancing or flow throughput. However, as a result of hardware constraints, it is challenging, and often impossible for larger data center networks, to precisely implement the TE weight distributions on the data plane switches. The resulting precision loss in the TE implementation causes load imbalances that can result in congestion and traffic loss.

Instead of treating all flows equally, we adapt the hardware resource allocation to a flow's traffic volume and its contribution to the overall precision loss. We intelligently prune select ports in weight distributions and merge identical distributions to free up hardware resources. Evaluation using realistic traffic loads shows that our techniques approximate ideal TE solutions under various scenarios within 7% error, compared to a 67% error for today's state-of-the-art approach. In addition, our design avoids traffic loss triggered by switch rule overflow. Finally, the execution time is 10× faster than the current approach.

1 Introduction

Data center networks (DCNs) have resorted to using links with abundant bandwidth, topologies with rich connectivity, and operation at enormous scale to meet growing application needs. However, these additional resources come at a significant cost and DCN operation must ensure efficient utilization of the network infrastructure. Traffic engineering (TE) plays a critical role in addressing this efficiency need by routing traffic over carefully chosen paths. Existing TE designs consist of two parts: (1) calculating the optimal plan for mapping of flows to links, which is called the *TE solution*; (2) implementing the TE solution on data plane switches, which we call *TE implementation*. Past TE research [3–5, 9, 22, 31, 38] has mostly focused on the TE solution, but the TE implementation impacts operational efficiency greatly and, despite this, has received much less attention.

Equal-Cost Multi-Path (ECMP) [30] is the most widely adopted method to implement TE solutions because of its universal hardware support. Weighted-Cost Multi-Path (WCMP) [73] is an extension of ECMP that supports a weighted split of traffic among paths by using forwarding table entry replication. This enables finer-grained traffic control but can consume much more space in the switch hardware table. As a result, two problems occur. First, some groups (the data structure that implements weight distributions in the switch) used by the TE solution cannot be installed in the switch after the switch group table becomes full, leading to traffic loss. Second, if the weights are adjusted to use less space in the group table, the resulting TE implementation may have poor TE performance. We refer to the difference between link utilization of the TE solution and of the TE implementation as the *precision loss*.

Recently, spine-free DCN designs [7, 45, 46] propose to replace the traditional Clos topology with clusters connected as a full mesh. Spine-free design reduces the total cost of ownership and accelerates DCN evolution, but we found that they also increase usage on the group table, exacerbating WCMP's performance issues.

We seek to answer this question: *how to implement TE solutions with high precision given constrained hardware space?* Adjusting weight distributions of groups proves effective in lowering table space requirement. However, when we compared the groups retrieved from production switches with the TE solution, we found that the precision loss is often high. We analyzed the differences, which led to several insights on how to reduce the group table usage with lower precision loss: (1) Groups that serve significant traffic contribute disproportionally to the precision loss. They should be given more group table space. (2) Some groups have very skewed weight distributions and consume many table entries. Normal weight adjustment that preserves all paths struggles to avoid high precision loss. However, certain paths can be pruned from the distribution to significantly lower the table space requirement, often with little impact on precision loss. (3) After adjusting weights, some groups can become identical. Deduplicating such groups frees up group table space without incurring precision loss.

We develop three heuristics based on the insights: (1) traffic-aware resource allocation, (2) prune paths, and (3) deduplica-

♦ This work was done when Keqiang He was at Google.

tion. We then implement two algorithms that map TE solutions to switch hardware, applying the above heuristics. They both aim to minimize precision loss while adjusting group weights to meet the hardware resource constraint. Heuristic 2 is invoked conditionally when the resource constraint is otherwise impossible to satisfy. The two algorithms make different tradeoffs: one has very low precision loss, while the other runs faster but with higher precision loss in some cases.

Since evaluation in production DCNs could impact user traffic, we built a data center TE evaluation framework named FabricEval that allows us to run controlled experiments using production-like DCN topologies and traffic loads. Our implementations of the algorithms are compared to the state-of-the-art approach [73] under both common and extreme DCN configurations. Results show that our design implements TE solutions with no more than 7% precision loss, while using the state-of-the-art approach can have up to 67% precision loss. Furthermore, our design achieves a 10× execution speed improvement over the existing approach and successfully avoids traffic loss triggered by exhausted hardware resources.

In summary, this paper makes the following contributions:

- We quantify the TE precision loss problem for a variety of DCN topologies, scale, and traffic patterns. Our study identifies five root causes of precision loss (§3).
- We present two algorithms with different speed and optimality tradeoffs that can generate high precision TE implementations, at the same time complying with hardware resource constraints (§4).
- We develop a TE evaluation framework that mirrors Google's production DCNs and TE system and use it to evaluate our design (§5).
- We release the source code of our algorithm implementation and the evaluation framework FabricEval [1].

2 Background

We provide an overview of how data center traffic engineering systems are designed, as well as the hardware resource constraint these systems face. We then describe how the emerging spine-free DCN architecture differs from the traditional Clos-based design from a traffic engineering perspective.

2.1 Traffic engineering in data center networks

End hosts in DCNs are densely connected by many paths traversing different switches. The TE system acts as a centralized controller that manages all links/switches and routes traffic end-to-end via available paths. It looks for an optimal plan to fulfill a list of goals, including serving all traffic `demands` (bytes to send between each src-dst pair), maximizing flow throughput, balancing link loads etc. The TE problem is typically formulated as a multi-commodity flow (MCF) optimization problem. Though it is theoretically possible to obtain a TE solution by solving an MCF with comprehensive constraints, e.g., switch group table size limits, the scale of today's DCNs makes this approach prohibitively

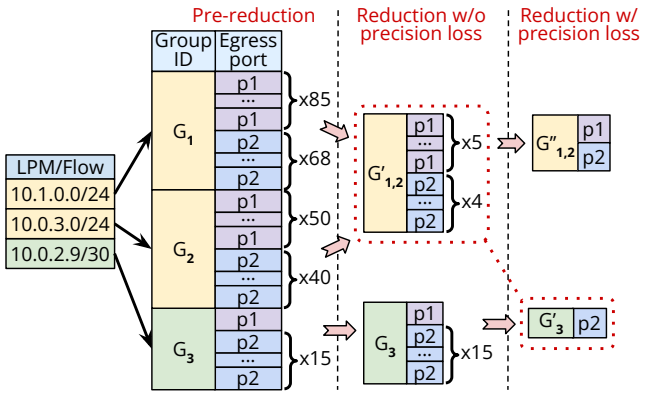


Figure 1: Hardware representation of flows/groups, and a demonstration of group reduction.

expensive. Common TE systems break the task into two steps: first generate a *TE solution* to the MCF problem without hardware constraints, and then map the TE solution to a data plane *TE implementation*. Some TE systems generate solutions hierarchically [18] for better scalability, but this does not impact the TE implementation step—the focus of this paper.

The TE solution specifies for each demand which paths to use and how much traffic to place on each path. When paths diverge at a switch, the demand is split by a ratio (weight distribution) according to the routing decision. The TE system updates each switch along the paths to reflect the routing decision. To map the TE solution to a TE implementation, the end-to-end TE solution first needs to be broken down into switch-local routing decisions. Next, switch-local decisions are translated to switch rules, namely longest prefix match (LPM) entries and groups, and installed on switches using switch control plane APIs (e.g., OpenFlow/SDN [44]).

An LPM/flow entry points to a group that is used for packet forwarding (see the pre-reduction phase in Figure 1). A group consists of a set of ports, each with an integer weight reflecting the rounded fractional traffic volumes assigned to each path. Weight distributions are achieved via port/entry replication since all entries in a group are selected by uniform hashing with equal probability at runtime. When ECMP is used, each port in a group uses exactly one entry in the group table, which is very efficient. With WCMP, the entries a port uses is proportional to its weight in the group. It is easy to see that the number of entries needed depends strongly on the weight ratios, e.g., G_1=$\{p1{:}85; p2{:}68\}$ will require a lot of entries.

The group table space is a scarce resource. Table 1 lists the group table capacity of a few types of switches found in production DCNs (also called fabric). Given the large number of flows, and the use of WCMP in today's DCNs, it is hard to avoid exhausting the group table. A group reduction algorithm is needed to reduce group sizes—the sum of entries used by each group—so the TE solution fits in the group tables. The group reduction algorithm should generate a new weight distribution that closely approximates the original one. For example in Figure 1, groups G_1-G_3 can be reduced to $G'_{1,2}$ and G'_3 with

Table 1: Switch hardware profile of an example deployment.

Switch generation	Port speed	Group table size
Gen. 1	40 Gbps	4096 entries
Gen. 2	100 Gbps	16384 entries
Gen. 3	200 Gbps	32768 entries

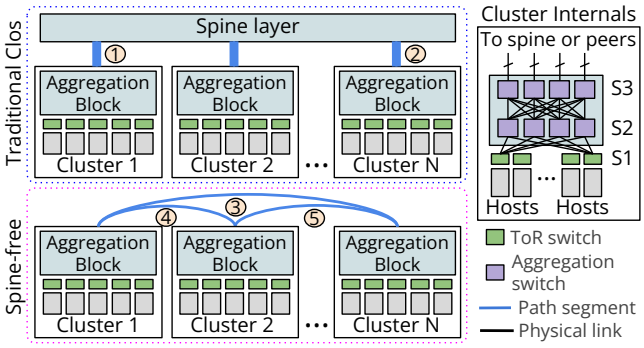


Figure 2: Traditional Clos and spine-free DCN topology. A path segment is an abstraction of parallel physical links.

minimal precision loss, while lowering table usage from 259 to 10 entries (more details given in §3 and §4). The state-of-the-art group reduction algorithm is called *WCMP TableFitting* [73].

2.2 Spine-free data center networks

Until recently, DCNs have used a traditional Clos topology that connects clusters to a spine layer, as illustrated in Figure 2 (top left). Inside each cluster, a non-blocking tree topology connects a large number of hosts. In this paper, we consider a 3-stage cluster layout, as illustrated in Figure 2 (right). Stage 1 (S1) switches are also called ToR switches. Stage 2 (S2) and stage 3 (S3) switches are aggregation switches. The S2 and S3 switches in each cluster also form an aggregation block.

Clos DCNs can use all links leaving a cluster to reach another cluster (assuming no failure) because they belong to the same shortest path. For example, for cluster 1 to reach cluster N in the traditional Clos DCN in Figure 2, there is only one shortest path: ①→② through the spine. The spine is often the link speed bottleneck of all paths because of infrequent upgrades. TE in Clos DCNs is thus straightforward, with all links running at the same speed after speed auto-negotiation, flows are spread across links in the shortest path in the form of either ECMP or very simple weight distributions (e.g., due to asymmetric link striping or some out-of-service links). In a cluster with egress demands, each switch has one flow entry matching the aggregated IP prefix of each destination cluster. Every flow references an individual group that contains ports used for reaching the next-stage switches, e.g., S1→S2 and so on[1]. Groups on S3 switches contain ports to the spine.

The spine layer is expensive and requires frequent upgrades to keep up with the growing traffic demand. To overcome this cost disadvantage, DCNs are moving to a spine-free topology enabled by optical circuit switching [64] that directly connects the clusters using a full mesh, as illustrated in Figure 2 (bottom left). Note that the intra-cluster topology and number of links coming out of a cluster remain the same.

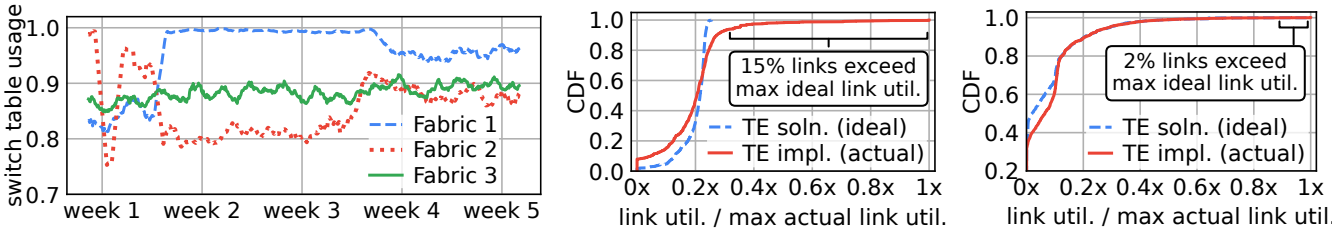
TE in spine-free DCNs is different and more challenging to implement than in Clos DCNs in two ways. First, link speed differences hidden by the slow spine layer are now exposed. This translates to more skewed weight distributions in groups, which consume more hardware resources. Second, spine-free DCNs have far fewer direct-connect links in the shortest path between clusters. In order to support high traffic volumes, non-shortest-path forwarding must be adopted to utilize all links. In the spine-free DCN of Figure 2, cluster 1 can reach cluster N via the direct path ③, as well as multiple indirect paths such as ④→⑤ through cluster 2. However, graph theory shows that longer paths lead to higher bandwidth overhead and lower flow throughput [61]. Therefore, the length of non-shortest paths is limited to two hops, i.e., a flow can transit through at most one intermediate cluster to reach the destination. We term these two types of paths *direct* and *indirect* paths. In practice, TE avoids using all indirect paths simultaneously since this leads to a Valiant Load Balancing (VLB) scheme [72] with higher link loads, although VLB does reduce the group table footprint. The benefits of combining VLB with WCMP are left for future work.

For both direct and indirect paths, switches of the source cluster need to install flows and groups the same way as in Clos DCNs. We call these `src`-typed flows/groups. In addition, the intermediate cluster in an indirect path uses separate `transit`-typed flows/groups to forward traffic from source clusters to the final destination. Indirect-path traffic only traverses the aggregation block. `transit` flows are installed on all S2/S3 (but S1) switches to match on the source and destination prefixes. Corresponding `transit` groups reflect indirect-path traffic in the direction of S3→S2→S3.

3 TE precision loss challenges

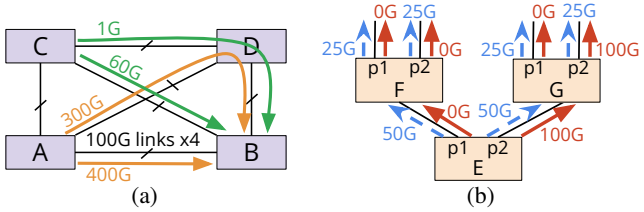
To illustrate the pressure on group tables, Figure 3a shows the group table utilization on Google's production spine-free fabrics. We see that group table usage can be as high as 90-100%. However, high table usage does not directly represent the negative impact on traffic. The key network-level metrics are the traffic load and loss on each link. We observed that high table usage strongly correlates with high link utilization. For example, Figure 3b shows the actual link utilization in a spine-free fabric in which the TE implementation produces a max table utilization of 90%. We see that the TE implementation results in a much more imbalanced link utilization distribution compared to the (ideal) TE solution. Moreover, 15% of the links exceed the max link utilization. The worst few links see an actual utilization 5 times higher than that of the TE solution, resulting in congestion loss. Figure 3c shows that a Clos fabric of similar scale also experiences

[1] A small, constant number of static flows/groups are installed on each switch to handle intra-cluster and ingress (destination) traffic. They are not an important factor in the discussion of this paper, hence ignored.



(a) (b) (c)

Figure 3: (a) 30-day time series of maximum group table usage in 3 arbitrary production fabrics. Usage is normalized to the table size of each switch. (b) Normalized link utilization CDF plotted from the TE solution and TE implementation in one of the spine-free fabrics. (c) Normalized link utilization CDF in a traditional Clos fabric of scale similar to the fabric in (b).



(a) (b)

Figure 4: (a) Abstract view of a small spine-free network. (b) Cascading precision loss in a 2-stage aggregation block. Desired traffic is in dashed blue, actual traffic is in solid red.

precision loss due to group table constraints, although the problem is less severe. We focus our discussion on a spine-free TE setting, though our findings also apply to Clos DCNs.

We compared a large number of TE solutions and their result and identified five key challenges that exacerbate precision loss: (1) Large network scale leads to more groups sharing hardware resources; (2) TE solutions are generated without considering hardware constraints and often contain skewed weights that are difficult to reduce; (3) Heterogeneous DCN hardware subjects parts of the network to higher precision loss; (4) Diverse paths in spine-free TE result in larger group sizes; and (5) The cascading impact of errors in one stage to the next in multi-stage DCNs increases precision loss multiplicatively. We now explain these challenges in detail.

① **Scalability challenge.** The total number of flows/groups to install on a switch increases with the number of clusters because each cluster has demands for virtually all clusters. As the number of groups in a table increases, each group has to use fewer entries, resulting in higher precision loss overall.

Considering the Clos and spine-free DCNs in Figure 2, hosts in each cluster are usually assigned continuous IP addresses that aggregate into a single prefix. The number of distinct prefixes equals the number of clusters N. In Clos DCNs, each switch needs to support N-1 `src` flows/groups, one for each destination cluster. In spine-free DCNs, each switch not only has N-1 `src` groups, but also one `transit` group for each indirect path traversing it. In the worst case, a cluster can be used by all (N-2) source clusters to reach (N-1) destination clusters indirectly. This sums up to (N-1)(N-2) `transit` groups on each aggregation switch.

② **Skewed weights challenge.** Since the TE system is not aware of any hardware resource constraint, the TE solution it generates may include highly skewed traffic distributions. Consider the example in Figure 4a, demand C→B has a total volume of 61G. The TE solution assigns 1G to path C→D→B and 60G to path C→B—a rather skewed split because path C→A→B has no slack capacity left and at least two paths must be used for diversity. Aggregation blocks A, B, C, and D each has four physical S3 switches. The four switches in each block are connected pair-wise (via four links) to the four switches in each of the other blocks. As a result, the `src` group for demand C→B on the S3 switches in C has a weight distribution of 0.25 : 15, which is rounded to 1 : 15.

Demand C→B experiences precision loss for two reasons. First, rounding weights to integer as required by the hardware triggers precision loss. We could also use 1 : 60, but this increases the group size. Second, if we cannot afford 16 entries, we can reduce the larger weight while retaining all member ports in the group, which is a common group reduction behavior. However, this further increases precision loss and leads to oversub(scription) on link C→D. In the worst case, 1 : 15 is reduced to 1 : 1 (ECMP), resulting in an oversub of $\frac{\text{actual volume}}{\text{desired volume}} = \frac{1/2 \cdot (0.25+15)}{0.25} = 30.5$ times more traffic on link C→D. Alternatively, precision loss can be lowered if certain ports are removed from the group. We discuss this option as a heuristic in §4.

③ **Heterogeneity challenge.** Cloud data centers are expanded incrementally to deal with growing traffic demand. Newly installed clusters are always equipped with the latest generation switch hardware, which means different generations of hardware co-exist in the DCN. Switches differ in link speeds and group table sizes, as demonstrated in Table 1. This heterogeneity aggravates precision loss in two ways: (1) The smaller table size of older switches forces groups to be reduced more heavily; (2) Mixed link speeds make group sizes larger on newer generation switches.

Gen. 3 switches have bandwidth 5× higher than Gen. 1 switches, but group table size is 8× larger. This means newer switches can support more groups with less group reduction if flow/group counts scale with bandwidth; however, older switches may will struggle to store the necessary state.

In spine-free DCNs, a switch can see multiple speeds across its ports since they may peer with switches of different

generations. The TE system seeks to balance link loads by placing more traffic on links with higher speed. A switch with mixed link speeds (e.g. Gen 3 switches with connections to slower hardware) increases the skew of group weights. This is defined as the max-to-min weight ratio of a group. The larger the ratio, the more skewed a group is. Our experiments (§5.4) show that in a baseline spine-free topology, the average skew ratio on Gen. 1 switches, which don't suffer from this issue, is 484 and the max ratio is 1176. In contrast, for Gen. 3 switches, the average ratio is 1187 and the max ratio is 2676.

④ **Path diversity challenge.** Group pruning can result in a significant reduction of path diversity which may result in high link utilization when there are traffic shifts or link failures. [52] proposes a technique that accounts for potential demand shifts by imposing a minimum level of path diversity for each flow. We incorporate this technique as a path diversity constraint into our MCF formulation[2] that generates TE solutions. With this constraint, TE solutions must use at least a certain percentage of the available paths to serve each demand. This percentage is referred to as the *spread*. While this constraint makes the TE solution more resilient, it degrades the solution optimality.

We compared the TE solutions generated by the MCF without and with a path diversity constraint for the same traffic load. Findings suggest that the standard formulation tends to use as few paths as possible. Many groups in the standard solution contain only one member port because the demand is small enough to fit onto one path. These groups use only one entry in the group table. Solution to the modified formulation uses several paths as the diversity constraint mandates. With a *spread*=50%, flows use more paths, which increases the pressure on the group table. As mentioned in the Heterogeneity challenge, larger groups will be reduced more heavily to fit into the same table space, hence causing high precision loss. We present a quantitative study in §5.3.

⑤ **Cascading precision loss challenge.** The multi-stage DCN topology has a cascading effect that amplifies precision loss multiplicatively as traffic flows from upstream switches to downstream switches.

To understand the cascading effect, consider the example in Figure 4b. Switches E, F and G form a 2-stage topology inside an aggregation block. A demand of 100G needs to be sent out from switch F and G. The TE solution (illustrated in dashed blue) divides the demand uniformly on upstream (S2) switch E, i.e., it sends 50G to both F and G. Next, downstream (S3) switches F and G each uniformly split the traffic between their two ports. However, to conserve group table entries, the TE implementation places all 100G on link E→G (shown in solid red) while switch F receives zero traffic. The oversub of link E→G is 2, while the undersub of link E→F is 0. This oversub-/undersub is inherited by the downstream links on switches F and G. The two ports on switch F are undersubscribed with zero traffic, regardless of the groups installed on F. On the other hand, even if switch G implements a perfect ECMP group, port G-p1 and G-p2 will each carry 50G, which yields an oversub of 2. This is the baseline precision loss inherited from switch E.

This example only shows the cascading precision loss inside one aggregation block. For indirect paths, the aggregation block of intermediate clusters inherits the baseline precision loss from the upstream aggregation block in source clusters.

[2] §A.1 details the formulation. Equation 2f is the path diversity constraint.

4 Design

We now present two algorithms that lower the TE precision loss compared to current group reduction solutions. Our algorithms, called *Direct Mixed-Integer Reduction (DMIR)* and *Iterative Greedy Reduction (IGR)*, aim to convert a set of original input groups to weight-reduced output groups so that the sum of the group sizes does not exceed hardware limit. DMIR builds on top of mixed-integer programming (MIP) to approximate groups to the TE solution. Specifically, it uses an MIP solver (e.g., Gurobi [24]) to directly find the optimal weight assignment for each group. Since MIP-based solutions have exponential time complexity, we also develop IGR which greedily searches for the smallest-sized groups that best approximate the TE solution in polynomial time. IGR is similar to WCMP TableFitting but uses additional heuristics.

Both algorithms achieve group reduction by decreasing group weights relatively. They also use three heuristics to deal with challenging scenarios (§3) in the reduction process. We first describe the heuristics in §4.1 and then introduce the algorithms in §4.2 and §4.3.

4.1 Shared heuristics

The heuristics described next are used in both algorithms. The first two, Group Sharing and Group Pruning, directly reduce the number of entries used in the group table. The third heuristic, Table Carving, ensures that high-volume flows have sufficient group entries to accurately distribute traffic.

Group Sharing. Group table usage depends on both the number of groups and the number of entries per group. As the network size increases, the number of groups to install on a switch increases, requiring more hardware resources (see §3 Scalability challenge). This heuristic reduces the number of groups by eliminating duplicates. Two groups are identical if they use the same ports and weights for each port. While `src` and `transit` flows/groups must be kept isolated by a design requirement, flows of the same type can share the same group entries to distribute their traffic. This optimization does not incur any penalty, since it does not change the traffic distribution. For example, G_1 and G_2 in Figure 1 can be reduced to the same $G'_{1,2}$, which is shared by the two flows.

We find that while `src` groups generally differ, `transit` groups often become identical after group reduction. For instance, the TE solution in Figure 4a leverages both direct and indirect paths to fulfill demands A→B and C→B. Two `transit` groups are installed on switches of aggregation block D, one for path C→D→B and one for path A→D→B. Since

D→B is a common segment, the same ports are used in both `transit` groups. The two groups only differ in pre-reduction port weights. Considering any S3 switch in D, it only has one port connected to B, as explained in the Skewed Weights challenge. The `transit` group for C→D→B places 0.25G on this port while the `transit` group for A→D→B puts 75G on it. Both groups can be reduced to single-port ECMP groups with weight 1, so they become identical. Generally speaking, `transit` groups for indirect paths with a common segment are identical post-reduction.

In production DCNs, groups are reduced batch by batch as the TE system generates new groups in response to the ever-changing network conditions. Since Group Sharing itself does not incur TE precision loss, it should always be invoked to conclude a round of reduction. The saved table space can subsequently benefit the next batch.

Group Pruning. Sometimes not all groups in the TE solution fit in the table, even if they are reduced to ECMP. In this case, some groups are not installed, and their corresponding traffic is dropped by the switch. Barring this heuristic, group reduction algorithms do not proactively reduce groups beyond ECMP because they aim to preserve the path diversity. But as a last resort to avoid dropping traffic, our algorithms invoke this Group Pruning heuristic that prunes ports from groups.

The key question is which port to prune. There exist multiple victim selection policies, e.g., prune the first port or a random port. The strategies are easy to implement, but they are not necessarily good choices. For example, a port carrying 90% of the group traffic might be pruned, impacting precision loss significantly. Our pruning policy considers both group size reduction (which impacts the path diversity of the group) and the increase in precision loss. Similar to WCMP TableFitting, our goal is to limit the max port oversub in a group. For IGR, we prune the port that results in the least increase in max oversub across all member ports, i.e., the port with the smallest weight (traffic volume). For example, p1 in G_3 of Figure 1 can be pruned—the size of pruned G'_3 becomes one, while the remaining p2 is merely oversubscribed by 1.016×. As discussed later in §4.2, the best port to prune in DMIR is not always the one with the smallest weight, but rather the port that yields the optimal objective.

We expect Group Pruning to be especially useful in addressing the Skewed Weights and Heterogeneity challenges. The reason is that both challenges lead to small weights on some ports, which can be pruned with minimal impact on precision loss. Our evaluation in §5.8 shows that Group Pruning has very little impact on path diversity.

Table Carving. This heuristic addresses the Path Diversity and Cascading Precision Loss challenges. Groups are forced to use more egress ports to satisfy the path diversity constraint. This means groups used by flows with a low traffic volume may have a lot of non-zero weights and large sizes, even though their contribution to the overall precision loss is more limited than high-volume groups. We develop Table Carving to ensure high-volume groups are prioritized over others and can receive sufficient resources during group reduction.

Algorithm 1 *DMIR*($\{G_i\}, T$), $1 \le i \le n$. $\{G_i\}$ is a set of `src` or `transit` groups. T is the available table space.

1: *// Step 1: Table Carving.*
2: **for** $i = 1$ **to** n **by** 1 **do**
3: $\quad T_i = \text{MAX}(len(G_i), \left\lfloor \frac{\text{SUM}(G_i)}{\sum_i \text{SUM}(G_i)} \cdot (T - \sum_i len(G_i)) \right\rfloor)$
4: *// Step 2: single-group optimization.*
5: $\{G'_i\} = \{G_i\}$
6: **for** $i = 1$ **to** n **by** 1 **do**
7: $\quad G'_i = \text{SINGLEGROUPMIP}(G_i, T_i)$ // Equation 1.
8: *// Step 3: reclaim and redistribution.*
9: **for** $G'_i = G'_1, \ldots, G'_n \in \text{SORT}(\{G'_i\})$ **do**
10: $\quad unused = \text{RECLAIMUNUSED}(\{G'_i\})$
11: $\quad$ **if** $unused > 0$ **then**
12: $\quad\quad G'_i = \text{SINGLEGROUPMIP}(G_i, T_i + unused)$
13: *// Step 4: Group Sharing.*
14: **return** DEDUP($\{G'_i\}$)

Table Carving allocates table entries exclusively to each group. A minimum number of entries are allocated per-group to avoid that low-volume groups end up with too few entries and little path diversity. Remaining entries are allocated to groups in proportion to their traffic volume. Note, a group will receive an allocation at least equal to the number of entries required by its ECMP form. Some low-volume groups are likely to be reduced to ECMP (or even get pruned) if they are only allocated ECMP-sized entries. Nevertheless, the minimum allocation is a tradeoff between protecting high-volume groups and avoiding starving low-volume groups.

While $G'_{1,2}$ in Figure 1 can be pruned to $G''_{1,2}$ to further reduce table usage without hurting path diversity, reducing G_3 to G'_3 is preferred because it carries less traffic. In addition to the Path Diversity challenge, this heuristic also helps address Cascading Precision Loss. Protecting the high-volume groups in the upstream switches through Table Carving effectively limits the baseline precision loss on the downstream switches.

4.2 Direct mixed-integer reduction

DMIR is a parallel group reduction algorithm. For a given set of input groups, DMIR performs a reduction on each group individually while ensuring the overall resource constraint is not violated. Algorithm 1 presents the structure of DMIR. Reduction happens in four steps: (1) The Table Carving heuristic allocates table space T to each group (line 2-3). (2) DMIR instantiates a number of single-group MIP optimization problems (described below) and solves them in parallel to get final reduced groups (line 5-7). (3) Table entries unused by reduced groups are reclaimed (line 10) and redistributed to other groups, potentially allowing precision improvements over that of the original allocation (line 12). (4) Final groups are deduplicated using the Group Sharing heuristic (line 14).

Algorithm 2 *IGR*($\{G_i\}, T$), $1 \le i \le n$.

```
1:  // Step 1: Table Carving, see Algorithm 1
2:  // Step 2: single-group reduction.
3:  θ = 1.00 // oversub limit.
4:  {G'_i} = SORT({G_i}) // sort by descending group size.
5:  while SUM({G'_i}) > T or SUM(G'_i) > T_i, ∃i do
6:      for i = 1 to n by 1 do
7:          G'_i = REDUCESINGLEGROUP(SORT(G_i), T_i, θ)
8:          if SUM({G'_i}) ≤ T and SUM(G'_i) ≤ T_i, ∀i then
9:              return DEDUP({G'_i})
10:     θ += 0.05
11:     PRUNEGROUPIFSTUCK(G_i_not_fit)
12: // Step 3: Group Sharing.
13: return DEDUP({G'_i})
14: function REDUCESINGLEGROUP(G = {w_i}, T_i, θ)
15:     G'* = G' = {w'_i} = (1...1) // G'*: optimal G'.
16:     oversub_min = ∞
17:     for w'_1 = 1 to wmax = ⌈ w_1·T_i / SUM(G) ⌉ by ⌈ wmax / ITER ⌉ do
18:         for i = 2 to p by 1 do
19:             w'_i = ⌈ (w'_1 / w_1) · w_i ⌉
20:         oversub = MAXPORTOVERSUB(G, G')
21:         if oversub < oversub_min then
22:             UPDATE(oversub_min = oversub, G'* = G')
23:         if SUM(G') > T_i or oversub_min ≤ θ then
24:             break
25:     return G'*
```

The objective of single-group MIP optimization is to minimize the difference between an original group G and the reduced group G' such that G' uses no more than its allocated resources. Equation 1 is the single-group MIP formulation. w_i and w'_i are port weights of groups G and G'. p is the number of ports on the switch. T_G is the allocated table entries for group G'. The formulation treats G and G' as two p-dimensional vectors such that $G = (w_1, w_2, \cdots, w_p)$ and $G' = (w'_1, w'_2, \cdots, w'_p)$. This brings G and G' into the same vector space $\mathbb{R}^p$ for convenient comparison. Note that this formulation also organically integrates Group Pruning with the common path-diversity-preserving reduction behavior by allowing w'_i to be 0.

$$\begin{aligned} \textit{minimize} \quad & \sum_{i=1}^{p} \left| \frac{w_i}{\sum_{i=1}^{p} w_i} - \frac{w'_i}{\sum_{i=1}^{p} w'_i} \right| \\ s.t. \quad & \sum_{i=1}^{p} w'_i \le T_G \\ & w'_i \in \mathbb{Z}^+, \forall i \in \{1,\ldots,p\} \end{aligned} \tag{1}$$

The difference between G and G' can be measured by various metrics, such as cosine similarity, L1-norm, Kullback-Leibler divergence etc. We choose L1-norm $||G - G'||_1$ since finding the best G' is a combinatorial optimization problem and L1-norm's linearity makes it computationally tractable. This metric is different from that used by WCMP TableFitting and IGR: restricting the max port oversub of a group. While restricting max port oversub is an intuitive objective—as long as none of the ports exceeds the oversub limit, the overall precision loss is bounded—subjecting all ports to the same oversub limit does not always produce the optimal group reduction outcome (as seen in the Skewed Weights challenge).

An alternative to single-group MIP is directly solving all groups in a monolithic formulation[3]. This is nevertheless infeasible because with so many decision variables and constraints, it takes the solver more than days to find a solution.

In the third step, DMIR collects the unused entries from each group after reduction. This is in order to make further improvements on some groups by rerunning reduction with extra hardware resources. Some groups have unused entries because these entries do not significantly improve the solution quality, so we reclaim them for other groups as follows. After the initial reduction, we sort groups by their objective metric (line 9). The group with the largest difference from its original group is ranked on top. All the unused entries are allocated to the top group, and group reduction is rerun with the larger space limit. It is possible that the top group only consumes a few or none of the unused entries. If so, unused entries are reclaimed again and redistributed to the next group in line. This process repeats until either all groups are reduced again or all previously unused entries have been used.

4.3 Iterative greedy reduction

IGR has two objectives: a primary one to keep reduced group sizes under resource limit, and a secondary one to restrict the max port oversub in each group to an upper bound. Algorithm 2 describes the three-step structure of IGR. Step 1 is the same Table Carving heuristic of Algorithm 1. In step 2, IGR iteratively reduces all groups (line 5-11) until both conditions are met: (1) sum of all group sizes meets the table limit T, and (2) each group size meets its allocated space limit T_i. Each group is reduced individually by the REDUCESINGLEGROUP function. IGR is greedy because reduction could terminate in the middle of an iteration once the two conditions are met (line 8-9), leaving some groups reduced more than the rest. Some groups may fail to meet the upper bound θ on port oversub, especially when θ is very tight in the initial iterations. IGR relaxes θ in each iteration (line 10) by a constant step size. A more relaxed θ enables REDUCESINGLEGROUP to reduce the group size more. If the reduced group size stops decreasing after θ has been relaxed for *thresh*=3 consecutive times, we consider the reduction "stuck". *thresh* is a tunable parameter that allows us to balance between aggressive reduction and more port oversub tolerance. When the reduction is stuck, the Group Pruning heuristic is invoked (line 11).

REDUCESINGLEGROUP starts the search for a final group from the original group's ECMP form (line 15). It locks the relative weight ratio between ports in the group under reduction G'. In each iteration, ports in G' are sorted by their

[3] See §A.2 for a comprehensive monolithic formulation.

weights in descending order (line 7). Once the assigned weight of the first port w'_1 is determined, all the remaining port weights are determined based on the relative weight ratio (line 18-19). The number of iterations performed on each group is limited to a small constant $ITER$. REDUCESINGLEGROUP conducts the search in increments of $\lceil \frac{wmax}{ITER} \rceil$ instead of the smallest increment of one (line 17), to ensure the search process finishes within $ITER$ iterations. This effectively mitigates the long convergence time (see §5.6) that iterative-reduction-style algorithms suffer from—there are many groups (Scalability challenge) and many ports per group (Path Diversity challenge) to iterate through. REDUCESINGLEGROUP makes the best effort to meet the oversub limit and returns the smallest group it has found when it hits the group size limit (line 21- 24).

5 Evaluation

In §5.1, we describe FabricEval, a data center TE evaluation framework that we created. The DMIR and IGR algorithms are compared to WCMP TableFitting under a wide range of network conditions in §5.3-§5.5. We also present group reduction speed as another metric in §5.6. §5.7 investigates the application layer impact. §5.8 inspects the side effect of Group Pruning. §5.9 discusses the contribution of each heuristic.

5.1 FabricEval evaluation framework

FabricEval is a network-level data center TE evaluation framework. It models the entire data center TE pipeline end-to-end: from taking snapshots on network states and traffic demands to generating TE solutions, group reduction and finally implementing the solution on data plane. FabricEval employs a modular design that allows us to plug in different TE solving algorithms, group reduction algorithms, and switch hardware models with different configurations. This enables us to compute both the ideal link utilization expected by the TE solution and the actual link utilization from different group reduction algorithms, on all links. It also tracks table usage and traffic loss due to missing forwarding entries from switches.

In FabricEval, topology inputs are represented as Protobuf [49]. Traffic demands are represented as plain matrices. Inputs are passed to a TE solver module, which by default runs the same TE algorithm used in our production fabrics. The generated TE solution is organized by cluster and forwarded to a group reduction module of each cluster. Finally, reduced groups are installed on the corresponding switches. FabricEval is implemented in 15,820 lines of Python code. Since FabricEval is not a packet-level simulator, we use *ns-3* [51] to study application layer metrics.

Open source releases. The source code of FabricEval, DMIR, IGR and our implementation of WCMP TableFitting is released online [1].

5.2 Experimental setup

All configurations used in the evaluation are developed from a *baseline* production-like configuration that includes (1) a 65-cluster heterogeneous spine-free fabric, (2) a path diversity spread of 50%, and (3) a traffic matrix (TM) representing the total egress/ingress demands between ToR pairs. Due to security and privacy reasons, we cannot publish most results from production fabrics and traffic traces. However, we carefully tuned FabricEval to match production results. Using FabricEval also avoids the difficulty of running controlled experiments on live fabrics. All experiments in this paper are conducted on a Dell PowerEdge R720 server, with two 20-core Intel Xeon E5-2680 CPUs, 128 GB of memory and 1.2 TB of disk.

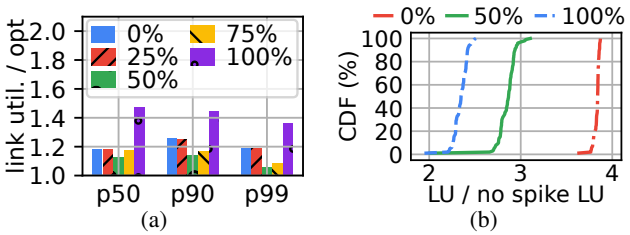


Figure 5: (a) 50% spread achieves best link utilization using production traffic trace and routing settings. (b) larger spreads are more robust against traffic spikes.

Network topology. The *baseline* fabric features 2.6K switches and 83.2K links. Each cluster in the fabric has 32 S1 switches. Each S1 has 32 ports—8 facing the S2 switches and 24 facing the host machines. The aggregation block in each cluster has 4 S2 and 4 S3 switches interconnected as a nonblocking FatTree; each has 128 ports—64 facing up and 64 facing down. The 64 up-facing ports on S3 are evenly spread to 64 peer S3 switches. The S2 switches connect to each S1 with 2 links.

Traffic matrix. The *baseline* TM captures properties found in production traffic traces. Clusters have unequal egress and ingress demands—storage clusters, for example, typically have a small ingress but a large egress because applications often read more data than they write. The demands follow an exponential distribution. The traffic volume between a cluster pair is characterized by the Gravity model [52, 55, 63, 70]. A cluster's total egress/ingress volume is distributed proportionally among peer clusters, weighed by each peer's total volume. Incremental data center expansion could leave a few clusters deployed but having zero demand. They can still be used by other clusters for transit.

5.3 Spread and path diversity

The Path Diversity challenge in §3 focuses on balancing resilience to traffic changes and optimal efficiency. This tradeoff depends on the spread. To determine an appropriate spread for our evaluation, we run production trace using realistic routing settings (i.e., TM fed into TE comprises peak demands of the past 24 hours and TE is recomputed every hour) on a large spine-free production fabric[4]. Figure 5a shows the link utilization for this realistic scenario, normalized relative to the optimal TE solution computed against the instantaneous

[4] §D.3 includes link utilization data for more production fabrics.

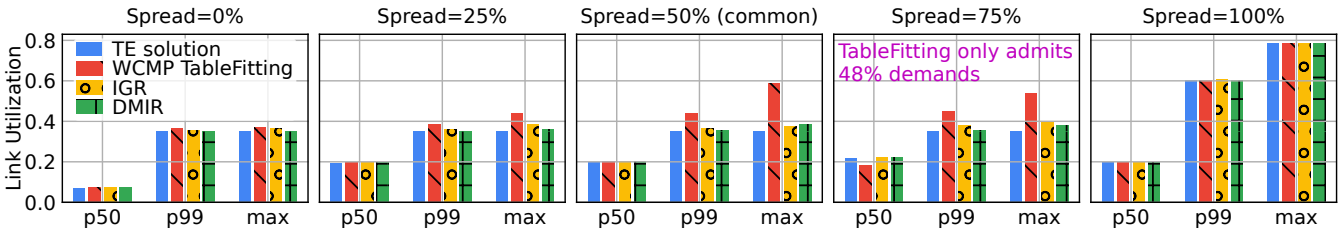


Figure 6: Median and tail link utilization with different path diversity.

Table 2: Summary of various fabrics used in FabricEval.

name	topology	# Gen1	# Gen2	# Gen3	link failure
Fabric 1	spine-free	22	22	21	none
Fabric 2	spine-free	11	11	11	none
Fabric 3	spine-free	65	0	0	none
Fabric 4	spine-free	22	22	21	1%
Fabric 5	Clos	21	21	22	1%
Fabric 6	random graph	22	22	21	none

demand (i.e., assuming traffic measurement and TE solution programming have zero delay). We find that 50% spread achieves link utilization closest to OPT (1.14× OPT) because higher spreads rely on costly indirect paths and lower spreads are fragile to TM changes. To understand how spread values affect TE solution's *robustness*, we further study more extreme cases, such as when 25% of the original demands spike by 4×. We define robustness as the link utilization post-spike divided by the pre-spike link utilization for the same spread. Figure 5b shows this ratio for each link in the same production fabric for 0% (no) spread, 50% and 100% (full) spread. As expected, 100% spread is most robust since spikes are spread throughout the network. 50% spread is significantly more robust than 0%. Overall, 50% spread proves to be a reasonable tradeoff between TE optimality and robustness.

Figure 6 compares the link utilization of IGR and DMIR with that of the TE solution (assuming no table size constraint) and WCMP TableFitting (TF) for five spread values. The results are collected on the *baseline* fabric and TM. In each graph, the y-axis shows the link utilization of the 50th (p50) and 99th (p99) percentile, and the maximum (max) link in the network. We see that with 0% spread, all algorithms come close to the link utilization of the TE solution, which is not surprising since most flows have a group size of one. As the spread increases, the link utilization difference between the TE solution and the group reduction algorithms increases. With a 50% spread, WCMP TF's link utilization is 25.4% higher at p99 and 67.1% higher at max. DMIR sees link utilization 1.6% higher at p99 and 9.1% higher at max. IGR increases over the TE solution by 4% and 7.4%, respectively. The result at 75% spread largely resembles that at 50% spread, except for a seemingly counter-intuitive link utilization improvement from WCMP TF [5].

[5] WCMP TF's result at 75% spread does not reflect a true improvement. The reason is that the groups generated by WCMP TF do not fit in the switch table. The overflow groups cause the associated traffic being dropped. This benefits the groups that do fit.

As the spread further increases, link utilization continues to deteriorate due to increased pressure on the group tables. At 100% spread, all algorithms achieve a link utilization equal to the TE solution. Since 100% spread forces demands to spread across all paths, groups become identical and after deduplication, they easily fit in the table with minimum precision loss.

5.4 Heterogeneous fabrics

In order to understand how different fabric configurations affect TE precision, we construct a list of fabrics that differ in scale, hardware and/or topology, as described by Table 2. Fabric 1 is the *baseline* fabric. Fabric 6 is a Jellyfish-like fabric with random graph topology [61] using the same hardware as fabric 1.

Figure 7 shows how results differ across these fabrics. Compared to fabric 1, all three algorithms in fabric 2 and 3 operate close to the TE solution in terms of both median and tail utilization. This confirms our analysis in §3 that Scalability and Heterogeneity both have an impact on precision loss. The tail link utilization in fabric 3 is noticeably higher than other fabrics of same scale because the same TM is applied to a fabric with lower capacity. Note that DMIR's max link utilization in fabric 3 is 10% higher than the TE solution, while WCMP TF and IGR are 12.4% and 3% higher, respectively. This is caused by a 120-second timeout on DMIR. Due to the nature of MIP problems, we use a timeout to ensure DMIR terminates within a duration comparable to the other two algorithms, which can lead to a suboptimal solution. Without this timeout, DMIR can still find an optimal solution within 10 minutes in all the experiments we run.

Looking at fabric 1 and 4, we find that the impact of failure is mostly on tail link utilization. The max link utilization under failure has increased by up to 63% over that without failure. Among the three algorithms, DMIR and IGR are more affected than WCMP TF. Compared to fabric 1, precision loss in fabric 5 is moderate—WCMP TF is 9% higher than the TE solution, yet IGR and DMIR are still effective in bringing this down to 1.3%. This aligns with our production observation that precision loss in Clos fabrics is not as grave as spine-free fabrics. Precision loss in fabric 6 is high. This is due to the asymmetry and imbalance introduced from random link assignment. WCMP TF has a max link utilization 55.1% higher than the TE solution, while IGR and DMIR lower this

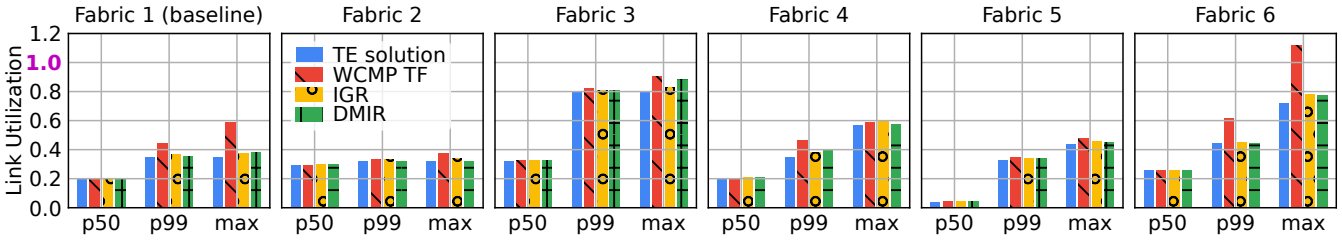


Figure 7: Link utilization distribution in various network fabrics.

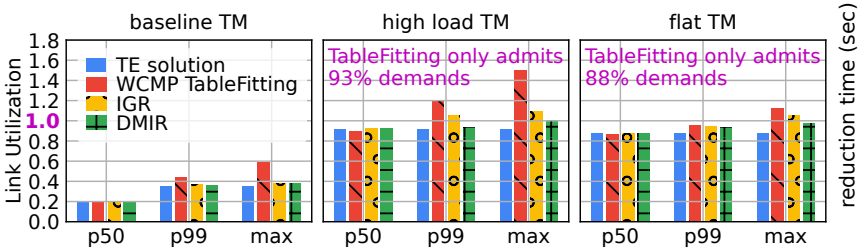


Figure 8: Link utilization distribution with different traffic patterns.

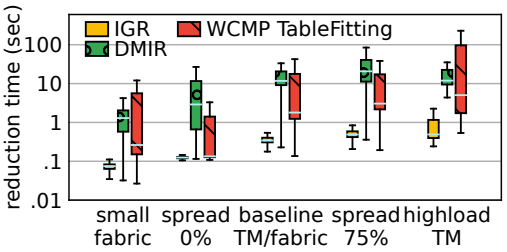


Figure 9: Comparison of group reduction speed.

gap to 8.4% and 7.5%. Generally speaking, IGR and DMIR are more effective when the topology is irregular.

5.5 Traffic pattern

The *baseline* TM has an average cluster ingress/egress volume that is about 40% of the cluster's bisection bandwidth. 15% of the latest generation clusters are configured to be in expansion mode with empty demands. For comparison, we created two more TMs: a high load TM and a flat TM. The high load TM is generated in the same way as the *baseline* TM, except that its average cluster volume is raised to 70% of the bisection bandwidth and all clusters have non-empty demands. The flat TM does not follow the Gravity model. Instead, each cluster has an equal ingress and egress demand of 7.68 Tbps (240 Gbps per ToR) that is uniformly spread across all peer clusters. There are no empty clusters in the flat TM. We run all three TMs on the *baseline* fabric with 50% spread.

With the high load and flat TMs, WCMP TF results in traffic loss due to group table overflow, as shown in Figure 8. As much as 7% and 12% of the total demands are dropped, respectively. In the high load TM scenario, even with a fraction of groups rejected, WCMP TF increases the actual link utilization over the TE solution by up to 62.1%. On the other hand, DMIR increases actual link utilization by only 9.4% over the TE solution at max, and IGR increases by 19%. This result is consistent with the increase found in the *baseline* TM scenario. The flat TM results are slightly better: WCMP TF has a max link utilization 28.4% higher than the TE solution, while it is 11.1% for DMIR and 20.2% for IGR.

All three algorithms will result in congestion loss due to oversub on some links. For example, in the high load TM scenario, WCMP TF's actual max link utilization is 1.49, almost 50% over the physical link capacity, while that of DMIR and IGR are 1.007, and 1.096. This is however expected given that the max link utilization of the TE solution is over 0.9. Any error in group reduction could lead to traffic exceeding the link capacity. While the average cluster volume is configured to 70% of the bisection bandwidth, link utilization of the TE solution ends up much higher. We attribute this to the blocking property of the spine-free topology.

5.6 Execution speed

The execution speed of the IGR and DMIR algorithms is important since they are used online. Each switch runs group reduction twice, first on the `transit` groups and then on the `src` groups. We measure the total time to complete both reductions on each switch. A number of representative configurations are selected to cover the common and corner cases. They include: (1) fabric 2 (small-scale) in §5.4, (2) 0% spread in §5.3, (3) the *baseline* fabric/TM in §5.5, (4) 75% spread in §5.3, and (5) the high load TM in §5.5.

Figure 9 plots the distribution of per-switch group reduction time as standard boxplots with max/min, first/third quartile and median. As can be seen, the reduction time on each switch spans a wide range: there is about two orders of magnitude difference. The reduction time differs significantly across switches as well as between different group types. On switches with smaller tables it take longer to fit the groups since there is a tighter table size constraint. The `transit` groups are easier to reduce than `src` groups because they are mainly ECMP-like groups. If a cluster has a zero demand, there is no `src` group to be installed on the switches in this cluster. The execution time is then solely up to `transit` reduction.

DMIR and WCMP TF have overlapping ranges. In some scenarios, such as 0% and 75% spread, DMIR is slower than WCMP TF, while in other cases, they are on par. IGR is the fastest among all three. It generally completes group reduction in under a second. Overall, IGR is 17-42× faster than DMIR and 10× faster than WCMP TF by comparing the median reduction time.

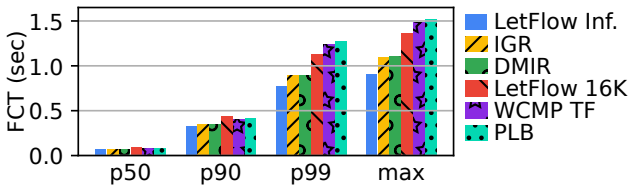


Figure 10: *ns-3* flow completion time.

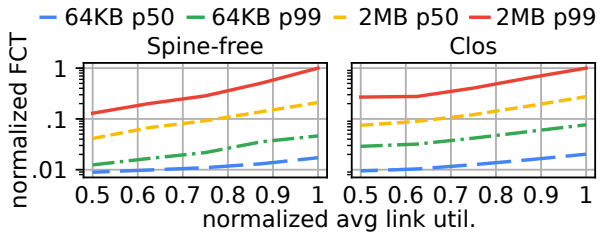


Figure 11: Production flow completion time vs. link load.

5.7 Impact on flow completion time

Thus far, we have shown our approach is effective at meeting TE's link utilization objectives. We now explore the impact our approach has on flow completion times. We extend our FCT comparison to include flowlet-based and host-based schemes. The traffic trace is constructed by sampling the flow sizes and start times from published production measurement studies [8, 35, 41, 56, 69] while ensuring that the total flow size of each cluster adds up to its demand in the TM.

In contrast to ECMP/WCMP, TE approaches such as [5, 37, 38, 65] balance load at the flowlet level [34]. While this finer grain load balancing could improve FCT, most large-scale DCN operators adopt some form of ECMP/WCMP [2, 22, 47, 58, 60] instead of a flowlet-based approach. We attribute this to the universal vendor support for ECMP/WCMP and the simplicity of deployment, e.g., no parameter tuning is required. We focus on LetFlow [65], a recent flowlet proposal that provides significant FCT performance gains.

We implemented IGR, DMIR, WCMP TF and LetFlow in *ns-3*, and use the *baseline* fabric/TM. As per [65], each switch uses a flowlet table that maps a 5-tuple hash to an egress port id. ECMP groups are still required to track ports that can reach the destination for each flow aggregate. When the inactive interval in a flow exceeds the pre-configured timeout, the flowlet entry expires and a new egress is randomly selected from the corresponding group. The best timeout value is picked by sweeping through a range of timeout values. We assume the flowlet table is subject to hardware resource constraints similar to WCMP. A 32K-entry memory is split in half—16K for groups and 16K for flowlets[6].

Figure 10 shows that IGR and DMIR reduce the p99 FCT of WCMP TF by 28.1% and 27.8%, respectively. LetFlow with an unlimited table size outperforms all other candidates, shortening the p99 FCT of WCMP TF by 37.8%. With a 16K table size constraint, LetFlow underperforms IGR and DMIR but remains better than WCMP TF by 8.7% in p99 FCT. To put this improvement into perspective, we run flows of two sizes in two production fabrics, then we steadily increase average link utilization of the fabric, the p50/p99 FCTs at every link load are collected. Figure 11 illustrates how FCT normalized to each flow's min RTT correlates with link utilization (more data in §D.4). On the more loaded side of the spectrum, a 10% reduction in link utilization can translate to a 20-40% reduction in p99 FCT.

Another class of designs that has reported significant FCT gains is host-based TE (e.g., PLB [53]). [53] shows that PLB can fully correct load imbalance caused by ECMP between two links with 1:2 capacity difference. However, with capacity difference of 1:100, PLB can only correct it to 1:3.5, and p99 FCT is 2.3× higher than that of a load balanced scenario. Figure 10 indicates that PLB alone achieves FCT on par with WCMP TF and we hold the same view as [53] that in-network WCMP and host-based PLB are complementary in performance benefits.

5.8 Group Pruning's impact on path diversity

A potential side effect of Group Pruning is that heavily pruned groups might lose a lot of paths. This reduced path diversity could impact the network's resilience to link failures. To understand the impact of pruning, we record the paths/ports used by all groups in the *baseline* fabric/TM experiment. Figure 12 shows that IGR prunes less than 0.01% of the total groups, while DMIR prunes 0.07% of them. DMIR's pruning decision is more aggressive than IGR: 48 groups (out of 1350624) are pruned down to a single port. In contrast, groups pruned by IGR only lose 1-3 ports. More results can be found in §D.5.

Figure 12 also shows the fraction of traffic impacted by Group Pruning. Impacted means that the traffic is steered away from pruned paths, but is not dropped. Despite DMIR's more aggressive pruning, the total traffic impacted (area under curve) is lower than IGR. We attribute this to DMIR's tradeoff between path diversity and lower traffic impact.

Overall, the small number of aggressively pruned groups and tiny fraction of traffic impacted suggests that the negative impact of Group Pruning is minor.

5.9 Ablation study

Now, we evaluate how much each of the heuristics in §4 contributes to the overall precision improvements. We use the *baseline* fabric/TM and 50% spread. First, the contribution of Group Sharing is evaluated by disabling this heuristic in both DMIR and IGR. Figure 13 illustrates the significant impact of Group Sharing on IGR: 5× decrease in max link utilization. DMIR is improved by 1.7×. Additionally, Group Sharing reduces the O(N^2) `transit` group entries (e.g., 162K) to no more than 64 (see §D.2 for details). Next, the contribution of Table Carving is demonstrated in Figure 14. Instead of standard Table Carving, the algorithms are modified to carve the table in constant sizes: each group receives an equal share of the table space. IGR and DMIR with standard Table Carving perform

[6] Cisco ACI fabric with Nexus 9300 switches supports flowlets under the feature named Dynamic Load Balancing [13]. A total of 4096×8-way ECMP entries (see Table 7 in [14]) are available on chip.

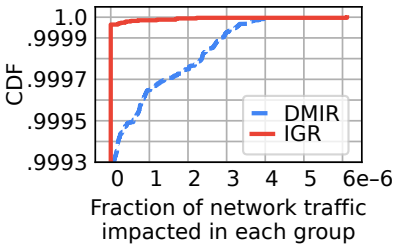


Figure 12: Impact of reduced path diversity on network traffic.

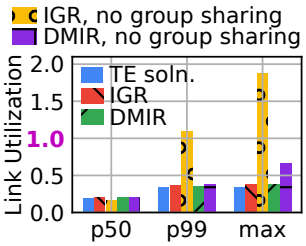


Figure 13: Group Sharing's contribution to TE precision.

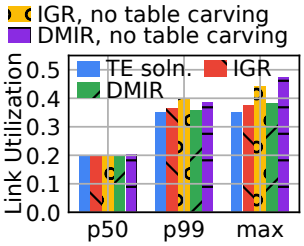


Figure 14: Table Carving's contribution to TE precision.

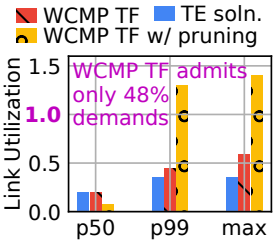


Figure 15: Impact of Group Pruning.

1.2× and 1.24× better than the constant version. Finally, we look at the contribution of Group Pruning in Figure 15. 75% spread is used to reflect how Group Pruning avoids traffic loss. The pruning logic is hard to completely disable in both algorithms, so instead, we add Group Pruning to WCMP TF. With Group Pruning, WCMP TF can avoid group installation failures and the consequent traffic loss entirely, at the cost of a 2.4× increase in link utilization. However, this increase is specific to certain scenarios and can be offset by other improvements.

5.10 Recommendation

While both our algorithms perform well, DMIR generally has lower precision loss than IGR. Though its advantage is usually small, some extreme scenarios can benefit from DMIR. For example, in the high load TM scenario (§5.5), DMIR is the only option that avoids link utilization above 1.0, except for a slight violation on the worst link. In contrast, IGR is always faster than DMIR, so it responds better to traffic load changes. Because of that, we recommend generally using IGR, except for difficult scenarios where low precision loss matters.

6 Related work

Data center TE. Many data center network designs [2–4, 22] leverage flow-level ECMP [30] to achieve failure resilience and efficient bandwidth utilization. However, ECMP is known to perform poorly with asymmetry and heterogeneity. WCMP [73] improves over ECMP. Niagara [36] distributes traffic using flow rules instead of group rules. DASH [32] splits traffic via comparison-based hash space partition. It avoids WCMP-style entry replication but requires arithmetic operation and P4 [11] support. Our work assumes fixed-function switches but also applies to programmable switches.

Flare [34], CONGA [5], HULA [38], Clove [37], LetFlow [65], and Contra [31] leverage the elasticity of flowlets to balance traffic load across paths. In networks with different path latencies [12, 17, 42, 45, 50, 66, 67], one has to carefully select the flowlet detection timeout to avoid reordering. Flowlets are also susceptible to traffic characteristics, e.g., [43, 62] find that RDMA traffic [6, 19, 23, 27, 39, 47, 74] exhibits weak flowlet pattern. As described in LetFlow [65], flowlet switching requires switch support [13], which is not as widely available as ECMP. There are also works like DRILL [20] and RPS [16] that operate at packet-level.

Host-based TE approaches like Presto [26], Hermes [68], MPTCP [54], FlowBender [33], RePaC [71] and PLB [53] assume either no in-network support or merely ECMP. PLB [53] shows that our work complements host-based TE.

Spine-free data center. Dragonfly [40], Dragonfly+ [59], Slim Fly [10], Slingshot [15], Aquila [21] have studied direct-connect topology networks. However, they primarily focus on HPC networks. Meanwhile, most deployed DCNs [2, 22] employ a Clos topology with a spine. Harsh et al. [25] discusses ECMP-based TE in spine-free data centers. Sirius [7] and RotorNet [46] are spine-free but spread traffic using extended VLB [72]. Jupiter [52] explores advanced TE in spine-free DCNs. We believe our work is among the first to improve TE precision loss in spine-free DCNs.

Wide area network TE. Wide area network TE faces similar switch hardware constraints. SWAN [28] allocates traffic demands to a set of tunnels, which are limited by the number of supported switch rules. B4 TE [29] generates groups that consume 14× the switch table space. It has to decouple traffic splitting rules across two stages in its Clos fabric to reduce per-switch table consumption. Meta's Express Backbone [48] and Edge Fabric [57] run centralized TE on BGP and ECMP. We found no available data on their TE precision loss.

7 Conclusion

Precision loss is an inherent problem when implementing traffic engineering with limited switch hardware resources and the shift towards spine-free topologies exacerbates the problem. We introduce two group reduction algorithms that offer different tradeoffs in terms of precision and execution time. Both algorithms use heuristics to address challenges, such as the hardware-agnostic nature of TE algorithms, switch heterogeneity, and path diversity. Our evaluation shows they achieve significant improvements in various network scenarios compared to the current solution.

Acknowledgments. We thank Adithya Abraham Philip, Zico Kolter, Akshitha Sriraman, Miguel Ferreira, Wei Bai, Brian Chang, Weiyang Wang, Min Yee Teh, Nandita Dukkipati and the anonymous NSDI reviewers for their feedback. We also thank engineers from the Google NetInfra team for their support. This research is sponsored by the U.S. Army Contracting Command under award number W911QX20D0008.

References

[1] FabricEval TE Evaluation Framework, 2024. https://github.com/shuoshuc/FabricEval.

[2] Anubhavnidhi Abhashkumar, Kausik Subramanian, Alexey Andreyev, Hyojeong Kim, Nanda Kishore Salem, Jingyi Yang, Petr Lapukhov, Aditya Akella, and Hongyi Zeng. Running bgp in data centers at scale. In *NSDI*, pages 65–81, 2021.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 38(4):63–74, 2008.

[4] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. *Proceedings of NSDI 2010: 7th USENIX Symposium on Networked Systems Design and Implementation*, pages 281–295, 2010.

[5] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 503–514, 2014.

[6] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering azure storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, Boston, MA, April 2023. USENIX Association.

[7] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. Sirius: A flat datacenter network with nanosecond optical switching. *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 782–797, 2020.

[8] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, page 267–280, New York, NY, USA, 2010. Association for Computing Machinery.

[9] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the seventh conference on emerging networking experiments and technologies*, pages 1–12, 2011.

[10] Maciej Besta and Torsten Hoefler. Slim fly: A cost effective low-diameter network topology. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 348–359, 2014.

[11] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, jul 2014.

[12] Shawn Shuoshuo Chen, Weiyang Wang, Christopher Canel, Srinivasan Seshan, Alex C. Snoeren, and Peter Steenkiste. Time-division tcp for reconfigurable data center networks. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 19–35, New York, NY, USA, 2022. Association for Computing Machinery.

[13] Cisco. Cisco application centric infrastructure fundamentals, 2015. https://www.cisco.com/c/en/us/td/docs/switches/datacenter/aci/apic/sw/1-x/aci-fundamentals/b_ACI-Fundamentals/m_fundamentals.html#concept_F280C079790A451ABA76BC5C6427D746.

[14] Cisco. Cisco nexus 9000 series nx-os verified scalability guide, release 9.3(12), 2023. https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus9000/sw/93x/scalability/guide-9312/cisco-nexus-9000-series-nx-os-verified-scalability-guide-9312.html.

[15] Daniele De Sensi, Salvatore Di Girolamo, Kim H. McMahon, Duncan Roweth, and Torsten Hoefler. An in-depth analysis of the slingshot interconnect. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[16] Advait Dixit, Pawan Prakash, Y. Charlie Hu, and Ramana Rao Kompella. On the impact of packet spraying in data center networks. In *2013 Proceedings IEEE INFOCOM*, pages 2130–2138, 2013.

[17] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 339–350, New York, NY, USA, 2010. Association for Computing Machinery.

[18] Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. Orion: Google's software-defined networking control plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.

[19] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533. USENIX Association, April 2021.

[20] Soudeh Ghorbani, Zibin Yang, P. Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Drill: Micro load balancing for low-latency data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 225–238, New York, NY, USA, 2017. Association for Computing Machinery.

[21] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M G Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M K Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, and Amin Vahdat. Aquila: A unified, low-latency fabric for datacenter networks. pages 1249–1266. USENIX Association, 4 2022.

[22] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 51–62, 2009.

[23] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[24] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023.

[25] Vipul Harsh, Sangeetha Abdu Jyothi, and P. Brighten Godfrey. Spineless data centers. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 67–73, New York, NY, USA, 2020. Association for Computing Machinery.

[26] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based load balancing for fast datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 465–478, New York, NY, USA, 2015. Association for Computing Machinery.

[27] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. Masq: Rdma for virtual private cloud. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 1–14, New York, NY, USA, 2020. Association for Computing Machinery.

[28] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):15–26, aug 2013.

[29] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. B4 and

after: Managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined wan. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 74–87, New York, NY, USA, 2018. Association for Computing Machinery.

[30] C. Hopps. Rfc2992: Analysis of an equal-cost multi-path algorithm, 2000.

[31] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Contra: A programmable system for performance-aware routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 701–721, Santa Clara, CA, February 2020. USENIX Association.

[32] Kuo-Feng Hsu, Praveen Tammana, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Adaptive weighted traffic splitting in programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 103–109, 2020.

[33] Abdul Kabbani, Balajee Vamanan, Jahangir Hasan, and Fabien Duchene. Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, page 149–160, New York, NY, USA, 2014. Association for Computing Machinery.

[34] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. Dynamic load balancing without packet reordering. *SIGCOMM Comput. Commun. Rev.*, 37(2):51–62, mar 2007.

[35] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: Measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, page 202–208, New York, NY, USA, 2009. Association for Computing Machinery.

[36] Nanxi Kang, Monia Ghobadi, John Reumann, Alexander Shraer, and Jennifer Rexford. Efficient traffic splitting on commodity switches. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–13, 2015.

[37] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. Clove: Congestion-aware load balancing at the virtual edge. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, page 323–335, New York, NY, USA, 2017. Association for Computing Machinery.

[38] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.

[39] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. FreeFlow: Software-based virtual RDMA networking for containerized clouds. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 113–126, Boston, MA, February 2019. USENIX Association.

[40] John Kim, William J. Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. *Proceedings - International Symposium on Computer Architecture*, pages 77–88, 2008.

[41] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.

[42] He Liu, Matthew K. Mukerjee, Conglong Li, Nicolas Feltman, George Papen, Stefan Savage, Srinivasan Seshan, Geoffrey M. Voelker, David G. Andersen, Michael Kaminsky, George Porter, and Alex C. Snoeren. Scheduling techniques for hybrid circuit/packet networks. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery.

[43] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-Path transport for RDMA in datacenters. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 357–371, Renton, WA, April 2018. USENIX Association.

[44] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, mar 2008.

[45] William M. Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C. Snoeren, and George Porter. Expanding across time to deliver bandwidth efficiency and low latency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1–18, Santa Clara, CA, February 2020. USENIX Association.

[46] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. Rotornet: A scalable, low-complexity, optical datacenter network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 267–280, New York, NY, USA, 2017. Association for Computing Machinery.

[47] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From luna to solar: The evolutions of the compute-to-storage networks in alibaba cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 753–766, New York, NY, USA, 2022. Association for Computing Machinery.

[48] Henry Kwok Mikel Jimenez Fernandez. Building express backbone: Facebook's new long-haul network, 2017.

[49] Jeffrey C. Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with modeling network topologies at multiple levels of abstraction. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, page 403–418, USA, 2020. USENIX Association.

[50] Matthew K. Mukerjee, Christopher Canel, Weiyang Wang, Daehyeok Kim, Srinivasan Seshan, and Alex C. Snoeren. Adapting TCP for reconfigurable datacenter networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 651–666, Santa Clara, CA, February 2020. USENIX Association.

[51] NS-3 Consortium. ns-3 network simulator, 2023. https://www.nsnam.org/.

[52] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter evolving: Transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 66–85, New York, NY, USA, 2022. Association for Computing Machinery.

[53] Mubashir Adnan Qureshi, Yuchung Cheng, Qianwen Yin, Qiaobin Fu, Gautam Kumar, Masoud Moshref, Junhua Yan, Van Jacobson, David Wetherall, and Abdul Kabbani. Plb: Congestion signals are simple and effective for network load balancing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 207–218, New York, NY, USA, 2022. Association for Computing Machinery.

[54] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. *SIGCOMM Comput. Commun. Rev.*, 41(4):266–277, aug 2011.

[55] Matthew Roughan. Simplifying the synthesis of internet traffic matrices. *SIGCOMM Comput. Commun. Rev.*, 35:93–96, 10 2005.

[56] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. *SIGCOMM Comput. Commun. Rev.*, 45(4):123–137, aug 2015.

[57] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V. Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 418–431, New York, NY, USA, 2017. Association for Computing Machinery.

[58] Hua Shao, Xiaoliang Wang, Yuanwei Lu, Yanbo Yu, Shengli Zheng, and Youjian Zhao. Accessing cloud with disaggregated Software-Defined router. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 1–14. USENIX Association, April 2021.

[59] Alexander Shpiner, Zachy Haramaty, Saar Eliad, Vladimir Zdornov, Barak Gafni, and Eitan Zahavi. Dragonfly+: Low cost topology for scaling datacenters. In *2017 IEEE 3rd International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*, pages 1–8, 2017.

[60] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 183–197, New York, NY, USA, 2015. Association for Computing Machinery.

[61] Ankit Singla, Chi Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. *Proceedings of NSDI 2012: 9th USENIX Symposium on Networked Systems Design and Implementation*, pages 225–238, 2012.

[62] Cha Hwan Song, Xin Zhe Khooi, Raj Joshi, Inho Choi, Jialin Li, and Mun Choon Chan. Network load balancing with in-network reordering support for rdma. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 816–831, New York, NY, USA, 2023. Association for Computing Machinery.

[63] Paul Tune and Matthew Roughan. Spatiotemporal traffic matrix synthesis. *SIGCOMM Comput. Commun. Rev.*, 45:579–592, 8 2015.

[64] Ryohei Urata, Hong Liu, Kevin Yasumura, Erji Mao, Jill Berger, Xiang Zhou, Cedric Lam, Roy Bannon, Darren Hutchinson, Daniel Nelson, Leon Poutievski, Arjun Singh, Joon Ong, and Amin Vahdat. Mission apollo: Landing optical circuit switching at datacenter scale, 2022.

[65] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 407–420, Boston, MA, March 2017. USENIX Association.

[66] Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T.S. Eugene Ng, Michael Kozuch, and Michael Ryan. C-through: Part-time optics in data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 327–338, New York, NY, USA, 2010. Association for Computing Machinery.

[67] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 739–767, Boston, MA, April 2023. USENIX Association.

[68] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. Resilient datacenter load balancing in the wild. In *Proceedings of the 2017 SIGCOMM, Los Angeles, CA, USA*, August 2017.

[69] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, page 78–85, New York, NY, USA, 2017. Association for Computing Machinery.

[70] Yin Zhang, Matthew Roughan, Nick Duffield, and Albert Greenberg. Fast accurate computation of large-scale ip traffic matrices from link loads. *SIGMETRICS Perform. Eval. Rev.*, 31:206–217, 6 2003.

[71] Zhehui Zhang, Haiyang Zheng, Jiayao Hu, Xiangning Yu, Chenchen Qi, Xuemei Shi, and Guohui Wang. Hashing linearity enables relative path control in data centers. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 855–862. USENIX Association, July 2021.

[72] Rui Zhang-Shen and Nick McKeown. Designing a predictable internet backbone with valiant load-balancing. In *Proceedings of the 13th International Conference on Quality of Service*, IWQoS'05, page 178–192, Berlin, Heidelberg, 2005. Springer-Verlag.

[73] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. Wcmp: Weighted cost multipathing for improved fairness in data centers. *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[74] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery.

Appendix

A Formulations

A.1 Data center TE formulation

The TE system solves a multi-commodity flow (MCF) problem on a network $G=(V,E)$ with a set of vertices/switches V and edges/links E. Assume that there are k commodities between sources and destinations, denoted as $(s_i,t_i,d_i), 0 \le i \le k$. Each commodity has demand d_i between source s_i and destination t_i. The objective of the TE system is to balance load (i.e., link utilization) across all links. This objective can be linearized as minimizing the maximum link utilization. Below is the path-based MCF formulation.

$$
\begin{aligned}
\textit{minimize} \quad & u_{max} && \text{(2a)}\\
s.t. \quad & u(x,y) \le u_{max}, \forall (x,y) \in E && \text{(2b)}\\
& u(x,y) = \frac{\sum_{i=1}^{k} \sum_{p \in \mathbb{P}_i^{(x,y)}} f_{ip}}{c(x,y)}, \ \forall (x,y) \in E && \text{(2c)}\\
& \sum_{i=1}^{k} \sum_{p \in \mathbb{P}_i^{(x,y)}} f_{ip} \le c(x,y), \ \forall (x,y) \in E && \text{(2d)}\\
& \sum_{p \in \mathbb{P}_i} f_{ip} = d_i, \ \forall i \in \{1,...,k\} && \text{(2e)}\\
& f_{ip} \le d_i \cdot \frac{c_p}{S \cdot \sum_{p \in \mathbb{P}_i} c_p}, \ \forall p \in \mathbb{P}_i && \text{(2f)}\\
& f_{ip} >= 0, \ \forall p,i && \text{(2g)}
\end{aligned}
$$

u_{max} is the maximum link utilization of the network, $u(x,y)$ is the link utilization of link $(x,y) \in E$. $c(x,y)$ represents the capacity of link (x,y). f_{ip} is a portion of commodity i (i.e., flow) assigned on path p, where p includes multiple links between the source and destination. $\mathbb{P}_i$ is the set of all paths between s_i and t_i of commodity i. $\mathbb{P}_i^{(x,y)}$ is the subset of $\mathbb{P}_i$ where all paths contain link (x,y). c_p represents the capacity of path p, which is the bottleneck capacity across all links in this path.

Equation 2d is the (optional) link capacity constraint. If a commodity has a high demand such that no valid assignment exists to meet this constraint, the problem is considered infeasible. However, if we are willing to overload a link, this constraint can be removed. Equation 2e is the flow conservation constraint. It indicates that all demands must be fully served, the network cannot hold or drop any demand. Equation 2f is the path diversity constraint mentioned in §3. It enforces that no more than a certain fraction of demand d_i is assigned to flow f_{ip}. The fraction is proportional to the total capacity of the paths used to serve d_i, which is controlled by a spread parameter $S \in (0,1]$. When S is close to 0, f_{ip} becomes essentially unbounded. When S approaches 1, flows for commodity i are forced to use all available paths. Note that S cannot be set to 0, but a small enough value is sufficient to completely remove this constraint, as f_{ip} is upper bounded by Equation 2e instead.

A.2 Multi-group monolithic reduction

To extend the discussion in §4.2, we now consider another group reduction formulation. Unlike Equation 1, this formulation attempts to combine all the single-group reduction problems into one monolithic formulation that can be directly solved by calling the MIP solver. Since this monolithic formulation needs a single optimization objective, we construct a weighted sum of L1-norm based on each group's individual L1-norm, where the weights are the total traffic volume carried by each group. The intuition is that groups with more traffic contribute more to the overall precision loss metric. Therefore, the MIP solver should spend more effort in optimizing those groups. Equation 3 shows the comprehensive formulation.

$$
\begin{aligned}
\textit{minimize} \quad & \sum_{i=1}^{n} \Big(\sum_{j=1}^{p} w_{ij} \Big) \cdot \sum_{j=1}^{p} \left| \frac{w_{ij}}{\sum_{j=1}^{p} w_{ij}} - \frac{w'_{ij}}{\sum_{j=1}^{p} w'_{ij}} \right| \\
s.t. \quad & \sum_{i=1}^{n} \sum_{j=1}^{p} w'_{ij} \le T \\
& w'_{ij} \in \mathbb{Z}^{+}, \forall i \in \{1,...,n\}, j \in \{1,...,p\}
\end{aligned}
\qquad \text{(3)}
$$

$G_i = (w_{i1}, w_{i2}, ..., w_{ip})$ is the original group i, w_{ij} is the original weight on port j of group i. Similarly, w'_{ij} is the reduced weight on port j of group i. T is the total available table space on the target switch. This formulation only enforces the sum of group sizes across all reduced groups to meet the table space limit, there is no per-group space limit and the Table Carving heuristic is not invoked.

Multi-group monolithic formulation scales poorly as mentioned in §4.2. For a large-scale production DCN, we fail to find a solution to the monolithic formulation after running the latest version of Gurobi solver for days. On the other hand, with a one-hour timeout set on Gurobi, the resulting solution is considerably worse compared to that obtained from single-group MIP.

B Example snippet of `TEIntent`

The TE system in large-scale DCNs usually divides the TE solution into batches, where each batch contains instructions of traffic split for all demands in a cluster. In FabricEval, such a batch is referred to as a `TEIntent`. The instruction of traffic split for one demand in a cluster is referred to as a `PrefixIntent`. A `TEIntent` can contain multiple `PrefixIntents`. Each `PrefixIntent` needs to specify its type (`src` or `transit`). This information is required to later translate the `PrefixIntent` to groups. The `PrefixIntent` also specifies the name of the destination aggregation block it wants to reach. Here the name of the destination cluster is not used instead because the `PrefixIntent` indicates the ports to use in the aggregation block. It should be consistent to use names of aggregation blocks for source, transit, and destination. Finally, the `PrefixIntent` lists all the northbound ports on S3 used to forward the demand, and the amount of traffic to be placed on each port in Mbps. The Protobuf format of `TEIntent` and `PrefixIntent` is as follows.

```
message PrefixIntent {
  enum PrefixType {
    UNKNOWN = 0;
    SRC = 1;
    TRANSIT = 2;
  }

  message NexthopEntry {
    // Name of next-hop port.
    string nexthop_port = 1;
    // Traffic volume in Mbps.
    double weight = 2;
  }

  string dst_prefix = 1;
  uint32 mask = 2;
  string dst_name = 3;
  PrefixType type = 4;
  repeated NexthopEntry nexthop_entries =5;
}

message TEIntent {
  // Name of cluster.
  string target_cluster = 1;
  // A list of PrefixIntents.
  repeated PrefixIntent prefix_intents = 2;
}
```

We also capture an actual `TEIntent` forwarded to one of the clusters in an experiment in §5.4. Due to the large Protobuf size, only a small snippet is demonstrated below.

```
te_intents {
  target_cluster: "toy3-c1"
  prefix_intents {
    dst_name: "toy3-c2-ab1"
    type: SRC
    nexthop_entries {
      nexthop_port: "toy3-c1-ab1-s3i1-p1"
      weight: 25.5
    }
    nexthop_entries {
      nexthop_port: "toy3-c1-ab1-s3i2-p1"
      weight: 25.5
    }
    nexthop_entries {
      nexthop_port: "toy3-c1-ab1-s3i3-p1"
      weight: 25.5
    }
    nexthop_entries {
      nexthop_port: "toy3-c1-ab1-s3i4-p1"
      weight: 25.5
    }
    nexthop_entries {
      nexthop_port: "toy3-c1-ab1-s3i1-p3"
      weight: 0.765625
    }
    nexthop_entries {
      nexthop_port: "toy3-c1-ab1-s3i2-p3"
      weight: 0.765625
    }
    nexthop_entries {
      nexthop_port: "toy3-c1-ab1-s3i3-p3"
      weight: 0.765625
    }
    nexthop_entries {
      nexthop_port: "toy3-c1-ab1-s3i4-p3"
      weight: 0.765625
    }
  }
  prefix_intents {
    dst_name: "toy3-c3-ab1"
    type: SRC
    ...
  }
  prefix_intents {
    dst_name: "toy3-c2-ab1"
    type: TRANSIT
    nexthop_entries {
      nexthop_port: "toy3-c1-ab1-s3i1-p9"
      weight: 3095.288
    }
    ...
  }
}
```

The network entities follow a naming scheme of [network name]-[cluster name]-[aggregation block name]-[switch stage][switch name]-[port name]. Each entity is indexed in the topology, hence its name is always the entity type + index. For example, in next-hop port "toy3-c1-ab1-s3i1-p1", "toy3" is the network name, "c1" identifies cluster 1, "ab1" is the only aggregation block in cluster 1, "s3i1" refers to the first switch in stage 3, and "p1" is the first port on this switch.

C Time complexity of IGR

Step 1 of IGR (Table Carving, line 2-3 of Algorithm 1) computes $len(G_i)$, $\text{SUM}(G_i)$ and $\sum_i \text{SUM}(G_i)$. This requires iterating over all p port weights in all n groups, so it takes O(pn). Function REDUCESINGLEGROUP runs at most $ITER$ iterations in the outer loop. The inner loop iterates over each port. So REDUCESINGLEGROUP takes O(p·$ITER$). Each iteration of oversub relaxation (line 5-11) takes O(pn·$ITER$). In the worst case, oversub relaxation stops when all groups are pruned until one port is left (zero port left means traffic loss). Here we make a reasonable assumption that a set of singleton groups must fit. This means the while loop (line 5) runs at most O(pn) times, hence making step 2 O($p^2n^2 \cdot ITER$). Step 3 Group Sharing requires scanning all groups and their ports

Table 3: Group table space usage on sampled S1/S2/S3 switches by different group types.

stage	group type	original usage	post-reduction
S1	`src`	8 entries	8 entries
S2	`src`	1746624 entries	64 entries
S3	`src`	2241913 entries	18285 entries
S1	`transit`	0 entry	0 entry
S2	`transit`	162112 entries	64 entries
S3	`transit`	792738 entries	60 entries

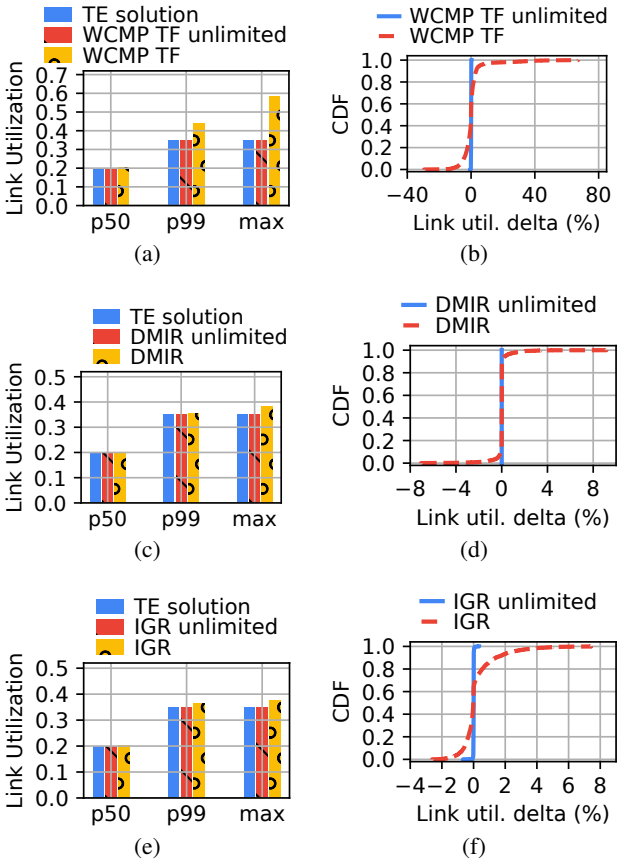


Figure 16: (a, c, e) Median and tail link utilization if running original group reduction algorithm vs. running it with infinite group table size, so that there is only precision loss caused by rounding weights. (b, d, f) Per-link utilization delta over the TE solution.

to mark duplicates, so it is O(pn). Overall, the time complexity of IGR is $O(pn+p^2n^2 \cdot ITER+pn)$, which is polynomial.

D Extra evaluation

D.1 Quantization error

Rounding fractional weights in groups to integers introduces a minor yet acceptable quantization error. We choose to express weights in Mbps so that the quantization error of rounding is at most $\pm$1Mbps. Weights in Kbps or bps can further reduce quantization error, but it comes at a cost where the Protobuf is bloated with larger integers. Figure 16 shows that the quantization error of Mbps granularity is sufficiently small.

We repeat the *baseline* fabric/TM experiment using three group reduction algorithms, WCMP TableFitting, DMIR and IGR, with unlimited group table space on each switch. In other words, the original fractional weights in each group only need to be rounded to integers, but will not be reduced in size—as there is infinite group table space to store any arbitrary number of entries. The precision loss over the TE solution from each group reduction algorithm with unlimited table space reflects the pure impact of rounding.

For instance, WCMP TableFitting is significantly improved with unlimited table space. The TE implementation is almost identical to the TE solution. Per-link utilization delta over the TE solution is also reduced to close to zero, which means there is very minimum oversub and the implementation overall is accurate. Similarly, for DMIR and IGR, the implementation is also improved, although the improvement is not as large as WCMP TableFitting. The TE precision loss (i.e., quantization error) achieved by all three group reduction algorithms is generally under 0.01%.

D.2 Table usage

Table 3 lists the group table usage statistics from sampled S1/S2/S3 switches in the *baseline* fabric from §5.4. The original usage column indicates the number of entries required if the table space is unlimited (pre-group-reduction). The post-reduction column indicates the actual number of entries consumed found on the switch (after running group reduction). We can see that table usage on S1 switches is low and group reduction is irrelevant here. This is because the groups are always ECMP on S1 switches and a shared group is used for serving all demands. As mentioned in §4, the Group Sharing heuristic significantly reduces the number of entries required by `transit` groups to no more than 64, on both S2 and S3 switches. The `src` groups on S3 switches see a table usage reduction from 2241913 to 18285—a 122× reduction!

D.3 Spread in TE solution

Figure 17 demonstrates the link utilization relative to optimal link utilization under shifting traffic demands for 7 different production spine-free fabrics. As can be seen, 50% spread generally yields the best link utilization comparing to other spreads. In some fabrics, e.g., Fabric A and Fabric D, 25% or 75% spread can achieve a link utilization similar to 50% spread due to the traffic demands on these fabrics are either more stable or more uncertain.

Figure 18 plots the CDF of per-link utilization delta under different spreads. Some links are severely under-utilized and some are severely over-utilized by up to 65%. However, the relative percentage change in link utilization does not always translate into high absolute link utilization because the traffic volume carried on such links can be low. We are more interested in the absolute link utilization than per-link delta.

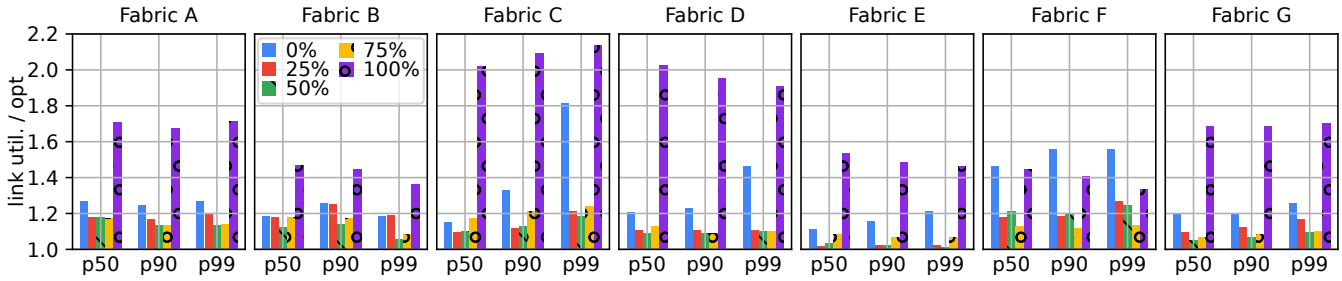


Figure 17: Production link util. of various TE spreads for 7 different fabrics.

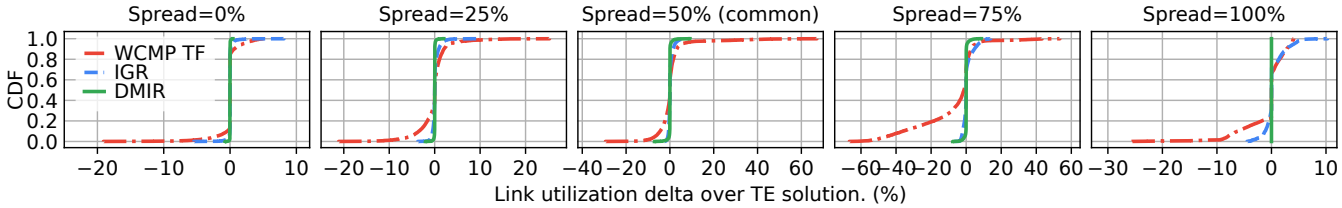


Figure 18: CDF of each link's delta over TE solution

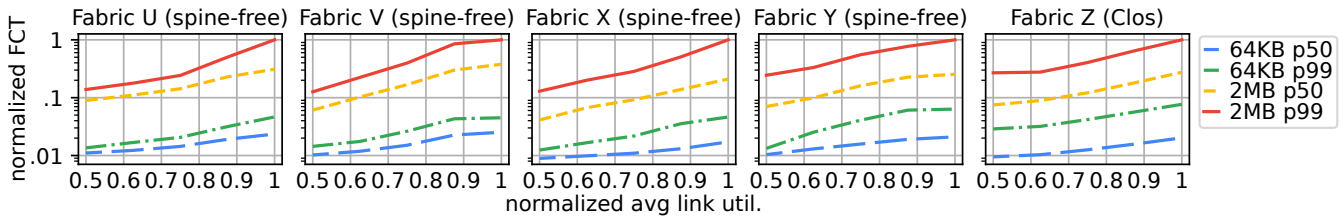


Figure 19: Production flow completion time vs link util. for 5 fabrics.

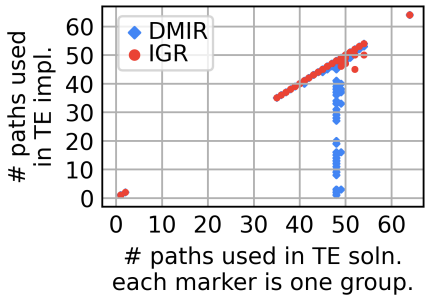


Figure 20: Original path diversity (x-axis) vs. reduced path diversity (y-axis) for each group of 64 ports.

D.4 Flow completion time

Figure 19 extends Figure 11 in §5.7 and shows a total of 5 production fabrics. The first 4 fabrics are spine-free fabrics of different sizes and the last one is a large traditional Clos fabric. As we can see, these fabrics share the same general trend—tail flow completion time of large flows could increase drastically as the average link load increases. In terms of network topology, spine-free fabrics see no difference than the Clos fabric, which confirms that the tail FCT increase is a universal trend.

D.5 Reduced path diversity

Group Pruning is a mechanism to further reduce group sizes while sacrificing path diversity. With the pruning policy introduced in §4.1, we try to minimize the impact on path diversity by pruning the small weights in the groups. Figure 12 and Figure 20 confirm that the chosen pruning policy indeed has very minor impact on path diversity. We obtain all the groups in the *baseline* fabric and plot a scatter plot of their used paths in the TE solution vs. TE implementation. As Figure 20 shows, groups (markers) on the $y = x$ line are not pruned and see no reduction in path diversity. Those not on the line have reduced path diversity. The closer a group is to the x-axis, the more reduction in path diversity it has. DMIR is more aggressive in pruning, some groups could have only a few paths left. However, such groups only constitute less than 0.003% of the total groups (48 out of 1350624) and their traffic impact is insignificant as discussed in §5.8.

One might argue that even 48 groups (of one single port left) would be a blast radius too large. If link failure does happen, traffic on these groups will be dropped. However, these 48 groups are distributed across different switches, they do not share the same port/link. The probability for all of them to fail simultaneously is extremely low. On the other hand, we have implemented a protective lower bound in the Group Pruning

heuristic. If users prefer not to have single-port groups, they can set this lower bound to a number greater than 1 (e.g., 2). In which case, Group Pruning will stop when there are only two ports left in the group. The larger this lower bound, the less likely the group is going to drop its traffic (unless all ports in the group fail). But this also comes with a disadvantage: groups may no longer fit in the table if the lower bound is set too large.

Multitenant In-Network Acceleration with SwitchVM

Sajy Khashab
Technion

Alon Rashelbach
Technion

Mark Silberstein
Technion

Abstract

We propose a practical approach to implementing multitenancy on programmable network switches to make in-network acceleration accessible to cloud users. We introduce a *Switch Virtual Machine (SwitchVM)*, that is deployed on the switches and offers an expressive instruction set and program state abstractions. Tenant programs, called *Data-Plane Filters (*DPF*s)*, are executed on top of SwitchVM in a sandbox with memory, network and state isolation policies controlled by network operators. The packets that trigger DPF execution include the code to execute or a reference to the DPFs deployed in the switch. DPFs are Turing-complete, may maintain state in the packet and in switch virtual memory, may form a dynamic chain, and may steer packets to desired destinations, all while enforcing the operator's policies.

We demonstrate that this idea is practical by prototyping SwitchVM in P4 on Intel Tofino switches. We describe a variety of use cases that SwitchVM supports, and implement three complex applications from prior works – key-value store cache, load balancer and Paxos accelerator. We also show that SwitchVM provides strong performance isolation, zero-overhead runtime programmability, may hold two orders of magnitude more in-switch programs than existing techniques, and may support thousands of concurrent tenants each with its private state.

1 Introduction

Data-plane programmable PISA switches[1] transformed the field of in-network computing from dream into reality. By enabling stateful custom packet processing at a switch line-rate, they offer dramatic performance boost in infrastructure services such as telemetry [26, 33, 35] and congestion control [18, 34], and accelerate networking applications such as load balancers [24, 37, 59], distributed protocols [10, 11, 31], concurrency control [30, 54], aggregation [42, 43], storage [22, 23, 32], and more [25].

Unfortunately, the benefits of in-network application acceleration have only been accessible to data center operators. The vision of in-network programs being deployed in switches by *tenants* [28, 50] has so far remained quite far from materialization. The main obstacle is well-known: existing switches, such as Intel Tofino [19], lack the essential support for multitenancy as they do not guarantee fault, resource, and performance isolation of individual programs. Furthermore, they cannot enforce the program compliance with operator security policies, and require expensive reconfiguration each time a new program is installed, causing traffic disruption for all tenants. These limitations are further aggravated by the severe scarcity of on-switch hardware resources partitioned among co-resident data-plane applications.

Motivated by the need for in-network acceleration of tenants' applications, recent proposals tackle the challenge of switch virtualization. These solutions, however, still have some practical limitations. Novel switch hardware [15, 28, 44, 46, 49] offers isolation, but the perspectives of its adoption are unclear. Others propose to run P4-based hypervisors [17, 56] to guard the execution of application code, but high resource demands limit their application to FPGAs and software targets rather than ASIC switches such as Tofino. Another approach is to merge multiple P4 programs into a single one while enforcing isolation among them in software [41, 50, 58]. However, neither does this approach guarantee fault isolation nor does it scale beyond a handful of co-resident programs because the switch resources must be provisioned at compile-time for the aggregate of their hardware demands. Further, deploying a new program requires recompilation and reconfiguration that disrupts the traffic through the switch. Runtime programmability for partial reconfiguration alleviates the traffic disruption problem [53] but still requires an isolation-aware source code merging mechanism to allow multitenancy.

In summary, multitenancy poses two fundamental challenges: (1) how to execute code on physical network switches while constraining it to a *stringent security sandbox* guarded by network operators; (2) how to enable a concurrent deployment, execution and update of thousands of switch programs while assuring *strong isolation* among tenants.

We advocate for a *Language-Level Virtualization approach* to overcome these challenges. The switch runs a *Switch Virtual Machine, SwitchVM,* which executes *Data-Plane Filters, (*DPF*s),* written using a specialized, Turing-complete Instruction Set. SwitchVM dynamically loads a DPF specified by the tenant on a *per-packet basis*, either from pre-deployed in-switch libraries or from code located in the packets themselves. SwitchVM allows every packet to be processed by a different DPF, allocating switch hardware resources on demand per DPF while reusing the VM runtime. Thus, SwitchVM introduces a runtime interpreter for DPF

[1] We only discuss programmable switches, so we omit *programmable* in the rest for brevity.

code that translates it into switch operations using a shared dataplane runtime. This approach enables time-sharing of switch hardware across multiple DPFs, as opposed to static resource partitioning in alternative solutions, and is key to achieving superior multi-tenant scalability.

SwitchVM is inspired by software sandboxing techniques such as Berkeley Packet Filters (BPFs) [36]. DPFs are Turing-complete, yet they are less flexible than P4. Notably, DPFs cannot parse new protocol headers or define match-action tables with arbitrary keys and actions. Nevertheless, DPFs are powerful enough to implement a variety of sophisticated in-network applications. A DPF may perform arithmetic operations or hashes on packet header fields, update the in-switch program state, and modify the packet forwarding depending on the execution output, e.g., by choosing from a pre-defined set of possible destinations or by performing a multicast. We discuss many applications (§4), and fully implement a Key-Value cache [23], count-min sketch [9], Paxos accelerator [10], and several load balancers [40, 59]. Furthermore, SwitchVM architecture is modular and can be easily extended to implement additional functionality, e.g., as new instructions.

DPFs of the same tenant may share a state in a switch. At the same time, SwitchVM ensures strict resource isolation between DPFs across tenants. A DPF uses virtual registers with load/store instructions to access the state in the packet, in-switch meta-data, or access *virtual* per-tenant space in the switch memory. In RMT switches, DPFs invoked by different packets will only observe consistent updates to the shared state thanks to the per-packet atomicity guarantees of the pipelined architecture. Proper placement of DPFs in multi-pipeline switches is crucial for achieving this consistency (§3.1). Furthermore, cross-tenant performance isolation is achieved thanks to the RMT line-rate throughput guarantees.

Crucially, SwitchVM restricts DPFs to a sandbox, giving fine-grain control over the data-plane functionality to operators on a per-tenant basis. SwitchVM enforces the sandbox policy at runtime because DPFs originate from untrusted tenants. Deployment of DPFs is performed via a control plane. Runtime reconfiguration by a tenant, i.e., code deployment and resource allocation, has no impact on the traffic of other tenants, as it is equivalent to adding/removing entries of match-action tables. Operators may deploy a per-tenant security policy, such as disallowing access to switch state, use of packet steering, or disabling DPF execution. The switches that are not authorized or unable to execute DPFs forward the packets as usual as they are compatible with network encapsulation protocols.

DPFs may reside in packets, similar to capsule-based active networks [13, 21, 45, 47, 48], and also pre-stored in a switch and invoked on-demand. In-switch deployment option is important because it eliminates the non-negligible bandwidth overheads of the DPF code header in a packet, and enables *code sharing* among tenants thus saving in-switch resources, yet without compromising inter-tenant state isolation.

Under the hood, SwitchVM implements a pipeline of generic Execution Units (EUs) that enable Multiple Instruction Multiple Data instruction execution, with several virtual registers and an extensive instruction set. Each DPF is spatially mapped onto the EUs. DPF chaining is supported by allowing a DPF to choose and invoke another DPF in the same or different switch to implement more complex logic.

We prototype SwitchVM on Intel Tofino-1. Implementing such a complex mechanism in P4 under tight hardware constraints is a formidable challenge. The key goal has been to fit as much logic as possible in a single hardware stage to increase the maximum instruction count of a single DPF. Higher instruction count allows DPF execution in a single pass through the switch, thereby eliminating bandwidth overheads associated with recirculation, and guaranteeing atomicity of accesses to a shared switch state, as well as strict performance isolation among DPFs.

We evaluate SwitchVM on a range of microbenchmarks and real-world applications of in-network acceleration which were originally developed in P4 in prior works. We demonstrate that performance overheads of SwitchVM compared to the P4 analogous code are negligible or none, whereas it allows concurrent execution of thousands of DPFs from different tenants, each with its own private state. We also show zero-overhead reconfiguration without packet loss, and strong performance isolation among different DPFs.

2 Motivation

The demand for in-network data-plane acceleration of network applications is steadily growing, and there is an abundance of proposals to use data-plane programmable switches for that purpose [10, 22, 23, 25, 40, 59]. Such switches have become a commodity, with offerings from multiple vendors, most notably Intel [19], Broadcomm [5], Juniper [38], and NVIDIA [39].

In practice, this acceleration has been accessible only to the data center's operators, not its tenants. Predeploying in-network applications as services by the data center is possible, but these applications might require fine-tuning for the needs of the specific tenant, i.e., custom-specific policies for the RPC scheduler [27], or non-standard key-value sizes and cache admission policies for KVS [23]. Such fine-tuning is impractical on a per-tenant basis.

Building general in-switch acceleration services that can suit multiple use cases of different tenants is notoriously difficult due to the switch's hardware constraints. Satisfying these constraints is fundamental to achieving high performance on any architecture, as both Reconfigurable Match Table switches (RMT), e.g., Intel Tofino, and disaggregated RMT (dRMT) switches, e.g., NVIDIA Spectrum, cannot guarantee line-rate throughput if attempted to execute a program that does not fit their hardware constraints.

Multitenancy in the switch would enable tenants to accelerate their own virtual networking infrastructure with custom network stacks and functions. Similarly, tenants might need to deploy custom in-network telemetry to debug their network performance. These services are difficult to offer as a general service since the internals of tenant virtual networks are usually not visible to the data center operators.

In summary, *switch multitenancy is not supported today, but could have significantly extended the range of applications accelerated on switches in data centers and clouds*.

2.1 Challenges

Present ASIC-based programmable switches lack the necessary mechanisms for multitenancy. Their programs expect to run assuming unmediated access to all hardware resources, and existing platforms do not provide any kind of privilege separation necessary to isolate tenant applications from the infrastructure packet processing logic. Any bug in a program may disrupt the stability of all the network traffic. Therefore, network operators might be reluctant to use the solutions that merge P4 programs of multiple clients into a single one to be executed on the switch [7, 41, 50, 58].

Another challenge stems from the need to support thousands of tenants with their own in-switch programs. Existing compile-based solutions that merge multiple P4 programs, or allow runtime reconfigurability in hardware [15, 49, 53], allow co-residency of about a dozen of programs, which is a few orders of magnitude less than needed.

Last, allowing a program to be dynamically updated without disrupting the rest of the network traffic is difficult. Recent works have introduced runtime reconfigurability by extending the dRMT [53] and RMT [49] architectures. The dRMT extensions are indeed feasible on existing CPU-based switches, but the RMT modifications are more invasive as they require hardware changes. Our goal is to achieve runtime reconfigurability without architectural changes.

2.2 Packet Filters for Switches

The primary tenet of multi-tenancy is per-tenant isolation of state, faults, and performance. Language Virtual Machines (LVM), such as the Java Virtual Machine, can achieve the first two. Such virtual machines enable full sandboxing of the untrusted tenant code, with fine-grain control over the accessible hardware resources. Furthermore, it may help solve the current scalability issue by enabling *switch compute resource sharing* among the programs, by loading and unloading the relevant bytecode at runtime, without provisioning the actual hardware resources at compile time for all co-resident programs as has been done till now. Last, if such an LVM could run at line-rate on an RMT switch, then it would be possible to provide perfect performance isolation among the tenants.

Implementing a general-purpose high-performance VM in RMT systems is clearly unrealistic. Fortunately, however, many data-plane programs do not require such a VM, and involve relatively simple, more restricted logic. This observation is not new, as it served the designers of the popular Packet Filters [36] (BPF) virtual machine to allow the execution of untrusted code in the OS kernel. Drawing inspiration from BPFs, we seek to build an LVM to be executed on a switch, while potentially restricting the scope of programs that it can run efficiently and deviating from the standard P4 programming model.

Last, by implementing our design in P4, we intend it to be modular and flexible, serving as the framework for implementing a broader set of in-switch functions tailored to a particular network environment.

2.3 Target Switch Architectures

Switch architectures with data-plane programmability range from ASIC pipelines (i.e., Intel Tofino, RMT) and FPGAs to manycore CPU processors (i.e., NVIDIA Spectrum-3, dRMT), as well as a variety of hybrid designs. Among these, Tofino switches have been prominent in in-network computing research for the past few years. This success can be attributed to the ability to execute stateful packet processing logic, defined using the P4 language, while maintaining line-rate performance on a multi-Tbps switch. These capabilities lead us to focus on the Tofino RMT architecture as the target for SwitchVM design.

We believe, however, that the concept of running in-network programs in a Language Virtual Machine sandbox to offer scalable multi-tenancy in resource-constrained switches is not limited to RMT architecture. Yet, using other architectures may involve different design considerations. For instance, CPU-based switches may benefit from Just-in-Time compilation to reduce virtualization overhead. We leave the exploration of other architectures for future research.

3 Design

SwitchVM enables secure and isolated per-tenant execution of short programs, called *Data Plane Filters* (DPFs). DPFs are invoked for each packet arriving at the switch. The packets include the input and the reference/code of the DPF to invoke. DPFs are executed inside a sandboxed environment controlled by the operator. We first discuss the DPF programming model, and then explain the SwitchVM design.

3.1 Programming Model

A DPF comprises three sections. A *prolog* specifies the location of the input parameters and initializes the DPF execution *registers*. A *body* specifies the actual data-plane logic using the SwitchVM instruction set. An *epilog* determines what to

Instructions	Description	Instructions	Description
Data Movement		ALU Operations	
MOV LOAD_IMM LOAD_CONST HASH	R1=R2 B=imm B=const_tbl[imm] A=hash(A)	ADD SUB AND OR XOR LSH RSH NEG MIN MAX	A = A op B A = op(A) A = op(A, B)
Memory Operations		Control Flow	
LOAD STORE	B = mem[A] mem[A] = B	HALT JMP	Goto END Goto pc
FAADD FAOR FAAND FAMAX	t=Mem[A] Mem[A]=op(t, B) B=t	BEQ BSET BLT BGT BLTS BGTS	Goto A==B ? pc_taken : pc_not_taken
Prolog		Epilog	
POP PEEK LOAD_MD RAND	R=stack.pop() R=stack.peek() R=MD[md_idx] R=rng.get()	PUSH FWD NEXT_PC HDR_MOD	stack.push(R) fwd according to R pkt.next_pc = R drop DPF from pkt

Table 1: SwitchVM Instruction Set with representative instructions of each type. A/B are the register types, R can be either, other values are immediates.

do with the packet after the DPF completes, e.g., send it to a new destination based on the DPF output. It may also store the DPF outputs in the packet.

In-packet state. The data inside the packet is handled as a stack, similar to Tiny Packet Programs [21]. It can be read in the prolog and written in the epilog. This is convenient: when a packet traverses multiple switches, each pops the inputs and pushes the output back to the stack, simplifying the implementation of multi-switch applications. A variable-size stack allows each application to allocate precisely the required packet space, potentially growing or shrinking a packet during processing, using a single unified SwitchVM parser.

Body. The code is organized into a sequence of *Execution Stages* (ES), each comprising several instructions arranged in *lanes*. The instructions in an ES are executed in parallel, effectively implementing a Multiple-Instructions-Multiple-Data (MIMD) program. Intra-lane parallelism is also available: each lane may concurrently execute instructions of different types. There are three types of instructions: ALU operations, in-switch memory access and control flow branches.

Instructions use *virtual registers* shared across the execution stages. Thus, DPF computation can be seen as the processing of information in the registers while it is flowing through the pipeline of DPF ESes. There are two sets of registers, *A* and *B*, and certain restrictions apply to their usage, e.g., only *A* registers may be used as addresses for memory access instructions. Memory is *virtualized* to provide inter-tenant isolation.

DPFs inherit the constraints of the RMT pipeline. Therefore, a memory buffer is accessible only to a specific ES and lane, since a packet may access each memory location only once.

Instruction set. The SwitchVM instruction set is summarized in Table 1. Some instructions have restrictions. First, they might support specific register types (either *A*, *B*, or both). Second, some instructions are limited to the prolog/epilog DPF sections only: register initialization is restricted to the prolog, while packet steering, program-counter modification, and stack enlargement must be performed at the epilog. Other instructions are not limited to specific DPF sections.

Turing completeness. Turing completeness of a computational system implies that it is universal, i.e., it can perform arbitrary computations. A system is Turing-complete if it can access arbitrary memory locations and read/write any amount of data, and perform conditional jumps [16]. SwitchVM instruction set satisfies these requirements.

DPF size. The number of instructions in a DPF is constrained by the availability of hardware resources. If the application code does not fit in a single DPF, dynamic *chaining* can be used whereby one DPF may choose to invoke any other DPF, enabling them to be automatically invoked one after the other. The sequence of DPFs may run either on the same switch (via recirculation) or on another one. We note that recirculation might affect the performance isolation, so we strive to enable the execution of complex DPFs in a single switch pass. For non-RMT switches, DPFs can be sized for line-rate processing to achieve performance isolation in the common case.

Atomicity execution semantics. DPFs naturally inherit the per-packet atomicity execution semantics offered by the RMT architecture. Specifically, the intermediate state updates done by a packet are not visible to other packets that invoke the same DPF on the same switch pipeline. Moreover, the state shared among multiple DPFs of the same tenant is also updated atomically, significantly simplifying the program development. Multi-pipe switches, such as Tofino, comprise multiple packet processing pipelines, each designated to serve a subset of ports. Switch allocation to run DPFs in a data center (§3.8) must consider the switch configuration to guarantee atomicity for a particular DPF.

Example: in-switch counter. Assume that several clients mark their outgoing packets with a unique application ID $\in \{1,\ldots,N\}$, and wish to count the total number of packets per ID. One possible implementation of this logic in a DPF is as follows. The DPF maintains one counter per ID allocated to lane 0 of EU 0. The ID is used as a memory address of the counter. The prolog POP-s the *ID* from the stack head in the packet into register A[0]. Then, EU 0 loads constant 1 (LOAD_IMM) into register B[0], and invokes a *fetch-and-add* (FAADD) operation in that lane to update the counter. The epilog then can PUSH back the ID back onto the stack along with the counter value. DPF placement and memory allocation for the counters are executed by the control plane in the setup stage, as explained in §3.8.

3.2 SwitchVM Design

Figure 1 shows the high-level design of the SwitchVM processing pipeline. It comprises several fixed function modules: Filtering and Permissions, Prolog, and Epilog, as well as a variable number of *Execution Units, (EUs)*. Each EU has *L* lanes which is analogous to an execution unit of a Very Large Instruction Word (VLIW) processor. Each Execution Stage

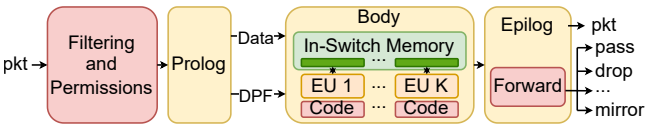


Figure 1: SwitchVM data-plane processing pipeline.

of a DPF is mapped onto its own EU. The instruction scheduling is static, i.e., a DPF specifies which instructions must be scheduled on which EU and each lane. The maximum number of EUs determines the maximum number of instructions in the DPF, and depends on the number of available hardware stages in the switch. The maximum number of lanes is determined by the number of match-action tables that can applied in parallel at a single hardware stage. For example, we can place 4 and 8 EUs in Tofino-1 and Tofino-2 respectively.

Packet processing flow. The packet is first passed to the Permissions module which determines whether a tenant may invoke a DPF. The Prolog then prepares the inputs of the first EU. The EUs then execute the DPF body. Finally, the Epilog prepares the output packet, and the Steering executes the forwarding logic.

3.3 Filtering and Permissions

The permission module is not visible to an executing DPF. It is managed by a privileged operator.

When a packet first enters the pipeline, its trusted `tenant_id` is matched against the permissions table. This table is programmed by the operator for each tenant that wishes to run a DPF. The table contains the action to perform on a packet that attempts to invoke a DPF without authorization (i.e., drop or pass), and also may specify the default DPF to invoke on any packet from that tenant. The table also stores a bit vector that controls tenant's access to certain functions such as access to the switch metadata.

In addition, a user may want to invoke only a subset of DPFs on a subset of switches. She specifies a special `app_id` token when asking for the DPF authorization. This token is stored in the Permission table as well. It must match the `app_id` token in the packet to execute a DPF. We use this functionality in our KVS cache application (See §4).

3.4 Program Loading and Initialization

The prolog module is responsible for initializing the virtual registers ($2L$, where L denotes the number of lanes) used as inputs for the first EU. The initial values can come from three sources: the in-packet stack (`POP`,`PEEK`), switch metadata (`LOAD_MD`), or from a random number generator (`RAND`). These instructions are the only ones available in Prolog.

Initializing registers from the packet stack cannot lead to underflow since the data stack size is known. The switch metadata may include ingress/egress ports, timestamps, or a switch identifier. Execution of `LOAD_MD` can be restricted to specific tenants to keep the network topology hidden for the rest. Access to predefined protocol fields, e.g., TCP/IP headers of inner packet, could also be supported in the same manner, assuming it aligns with the operator's sandboxing policy.

The code for each segment (prolog, execution units, or epilog) may reside in a packet, or in the SwitchVM code memory in the switch. A *pointer* to the prolog is taken from the packet header, or is passed from the Permission module, effectively enforcing invocation of a default DPF for the current tenant on *all* the packets even without the SwitchVM header. The prolog transfers the execution to the first EU, by specifying the pointer to the first EU's instruction.

3.5 Epilog and Steering

The epilog pushes the results of the computations into the in-packet stack (up to $2L$ entries). It works similarly to Prolog. Specifically, the only operation that can be executed is `PUSH`, using any register as its input.

The epilog code additionally defines how to steer the packet. This is determined by a dedicated immediate value in the epilog, or from any of the $2L$ registers. This value is then used to fetch a matching entry from a forwarding table defined for the specific `tenant_id`. We currently support the following actions (but more can be added): `pass`, `drop`, `return-to-sender`, `forward`, `multicast`, `set_port`, `recirculate`.

The default action is `pass`. The `set_port` and `recirculate` actions are reserved only for privileged tenants. Since the network uses encapsulation, some of these actions can also affect the headers of the encapsulated packet. Updates to the steering table require the use of privileged control-plane requests to allow secure handling of virtual network address translations without disclosing the physical IPs to the tenant as we explain below.

A DPF may *drop* the code header after execution, saving bandwidth if there is no need to keep it.

3.6 Execution Unit

The architecture of a single EU is shown in Figure 2. An EU comprises L lanes. For each lane, each EU holds several fixed-function modules: Selectors that determine how to move data among registers, an ALU (one per lane), an in-switch memory access (one per lane), and a branching unit (one per EU) that determines the instruction pointer for the next EU. These modules are divided between two switch pipeline stages, and all modules in the same stage are executed in parallel.

Registers. An EU operates on $2L$ 32-bit registers, two for each lane. These are divided into 2 groups: $A[1..L]$ and $B[1..L]$. Due to resource constraints, the registers from different groups

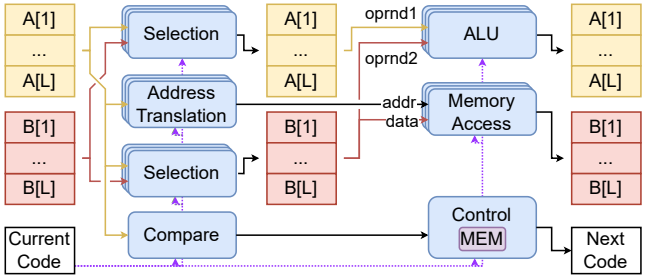


Figure 2: Execution Unit (EU) architecture.

have minor usage restrictions. Specifically, only *A* registers are allowed to compute a hash of their previous value, and only *B* registers may store an immediate or a constant. These constraints are insignificant in practice but allow reducing required switch resources per EU, specifically the action instruction memory and hashing logic, as shown in §5.1.

The Selector module allows transferring data among arbitrary registers via `MOV` instruction. This is crucial because ALUs are constrained to execute operations only on the registers belonging to the same lane, as explained next.

ALU. Each ALU only performs 32-bit operations on registers from its lane, i.e., ALU i operates on $A[i]$ and $B[i]$. The supported set of operations can be easily extended (§3.7).

3.6.1 Memory Access

A DPF invokes `LOAD`, `STORE` and a few atomic operations to access in-switch memory buffers. These buffers are separately allocated for each EU, and each lane in the EU, without the ability to share them among EU instructions. As mentioned earlier, this limitation is due to Tofino RMT implementation (see §3.1).

DPFs may only access *virtual memory* to isolate memory between tenants. The memory is allocated by the control-plane API. Memory accesses involve virtual-to-physical address translation as we explain next. Once translated, the respective memory instruction is executed. The address operand in lane i is provided in $A[i]$, and $B[i]$ is used for the stored or loaded data.

Virtual address translation. SwitchVM supports two types of translation mechanisms. A segment-based memory translation allows allocating power-of-two-sized contiguous buffers. The allocation request specifies the requested virtual address and the buffer size. The control plane then allocates a contiguous range in physical memory, and uses TCAM to store the virtual address range as a key, and the numerical offset between the virtual and physical addresses as a value. The virtual address needs to be aligned to the segment's size.

When attempting to translate any address in this buffer, that address will match the allocated range and TCAM will retrieve the offset, allowing one to compute the physical address by adding that offset to the virtual address. For example, given the virtual address `0x4` and the request to allocate 4 bytes, the control plane may allocate a physical buffer of that size at physical `0x2`. The TCAM will store the range `0x4-0x7` as the key (using the mask 0xC), and the offset `-2` as the value. When accessing virtual `0x6`, the respective physical is computed as `0x6-2`. Note that if a virtual address is not mapped, it will not be found in the translation table, so no buffer overflows are possible.

However, for the cases where we need to store sparse data structures such as hash tables, the segmented allocation is wasteful. Using many small segments depletes the TCAM storage (about 1500 ranges per lane), whereas large ranges waste too much memory due to internal fragmentation.

In applications where the data to store is known in advance, one can create one-to-one virtual addresses mappings we call *direct*. These mappings are stored in a table with up-to 45K entries per lane. One notable example of using direct mappings is a KVS cache (see §4).

Memory Isolation. The entries in the translation tables are all amended with `tenant_id` as another key. Thus, each tenant may access only its own translation table.

3.6.2 Control Flow

Branching is essential to support realistic DPFs. Each EU contains a single Control Flow Unit (CFU) that implements different types of branches. Thus, DPFs specify the addresses of the instructions in *code memory*, or from the packet.

Each stage in a DPF, including prolog, EUs and epilog, may alter the control flow and choose the instructions invoked in the next stage. The CFU may use any register to compute the predicate and determine whether a branch is taken or not. The code specifies the instruction pointers for both taken and not-taken branches. SwitchVM supports multiple types of conditional and unconditional branches.

Code memory. To allow code sharing among DPFs, code memory is not virtualized and provides no memory protection. The security is not compromised, because the DPF memory isolation, permissions, and steering are enforced at runtime. Thus, even though a DPF code is shared, each DPF instance operates on a separate, per-tenant state.

The code addresses used in each stage point to code in the following stage. For example, the addresses in the Prolog are interpreted as addresses for the first EU, addresses in first EU point to instructions in the second EU, and so on.

The use of instruction pointers per stage is useful for other purposes too. First, it allows reusing code in switch memory by storing an `init_pc` that points to the prolog code that needs to be executed. This pointer is stored either in the packet (to invoke a specific DPF), or in the Permission table of a switch to serve as a default DPF. This enables atomic reconfiguration when replacing an in-switch DPF in-place, by replacing the prolog only after all other stages of a DPF have been stored in the switch. We demonstrate in-place replacement in §6.2. Second, in the epilog, the code has the ability

to change the `init_pc` field in the SwitchVM header, which allows *service chaining* whereby one DPF invokes the next one in the next supporting switch. We show several examples of such applications in §4.

3.7 P4 Extensions

We envision SwitchVM to serve as an extensible framework that can be tailored to the needs of the specific environment by minor modifications to its P4 implementation. Thanks to its modular design, the modifications are localized. These additions will extend SwitchVM's capabilities and can be invoked from DPFs.

SwitchVM parser can be easily modified to expose new packet fields to a DPF and use them in the prolog to initialize DPF registers. Similarly, it is possible to expose additional switch metadata, such as port queue length. The steering unit can also be updated with new steering policies. Further, new instructions can be easily added to ALUs and memory units if the hardware support is available. For example, one may add support for bit slicing, or new Read-Modify-Write operations. More elaborate changes may involve new fixed-function modules added to an EU. The code for these modules can be integrated into an EU, and contain the control signals needed to invoke them. For example, one may add a match-action table that can match on multiple registers and access memory from different lanes. Last, one may replace individual EUs with fix-function units, e.g., complex data-plane sketch.

3.8 Control Plane

Control plane is used for managing authorization, deployment, configuration, and memory allocation in switches. Our current design assumes that this functionality is implemented in a centralized controller managed by data center operators.

Authorization and Scheduling. DPF placement for in-network computing must be taken into account by the data center's resource management and scheduling system [2–4]. A tenant may ask to invoke specific DPFs (or a DPF chain) on certain switches along the data path of its virtual machines. To prevent exposing the physical network topology to tenants, authorization requests are expressed relative to a tenant's virtual network structure. Resource allocation should consider both resource utilization and the network topology in order to ensure consistent processing of tenant's DPFs across all packets, in accordance with the packet routing policies. While these aspects are beyond the scope of this paper, SwitchVM offers flexible *mechanisms* to control the DPF execution (§3.3).

In-switch memory management. The mechanisms are similar to the OS memory allocation. We implement a simple first-fit allocation policy. Unlike the traditional allocators, however, the tenant chooses the virtual addresses.

In-switch code deployment. Deploying a DPF involves several steps. When a DPF does not invoke the code from other DPFs (simplest case), the branch addresses it uses internally can be automatically inferred during the deployment. The physical address of the prolog is then reported to the user to be specified in the packets. However, reusing in-switch code requires knowledge of the addresses of the deployed code, requiring a process analogous to linking, which currently is performed manually.

Steering rules. The tenants use virtual networks with their private IP addresses, thus they are unaware of the physical addresses used to forward packets in a physical network. Thus, updates to DPF steering rules require translation from the virtual to physical IP addresses. The physical addresses are not visible to the DPF. DPF chaining is already considered at the authorization time, thus its effect on routing is acceptable to the operator.

Switch state migration and replication. In-switch state becomes an integral part of the tenant's application logic. Thus, migration of a VM to a different location in the network topology might require one to also migrate the state in the respective switches. Multi-path forwarding in modern networks might require maintaining consistent program state across multiple switches. Data-plane inter-switch replication solutions [55] can be deployed by data center operators to automatically synchronize the tenant's state across switches.

3.9 Security

Tenant identifier. Ensuring isolation among tenants relies on a trusted `tenant_id`, as it is used to distinguish tenants in all the security-critical units in SwitchVM, such as steering and virtual memory mappings. We use virtual network identifier fields, e.g., VID or VXLAN ID, common to encapsulation protocols, similar to prior research [28, 46, 49, 50].

DPF injection. Adding the DPF code or data to the packets requires privileged access to the encapsulation protocol headers via a hypervisor, which is expensive. One option is to add the DPF invocation requests in the control path (e.g., by attaching a function to an OS socket [14]), while adding the respective headers during encapsulation. Another option that we leave for future work is to modify the hypervisor networking interface to specify DPF invocation requests per packet. The in-packet data stack size needs to be validated at this point to prevent under-/overflow during packet parsing and deparsing.

Steering, chaining and routing. By default, DPFs cannot change packet steering nor routing, unless specifically authorized via dedicated steering rules installed by the control plane. These rules are tenant-specific, and they ensure that the tenant packets cannot escape the virtual network. The same mechanism applies to DPF chaining. In summary, neither steering nor chaining can affect the packet's original network path without control plane authorization, which only installs rules compliant with the network routing policies.

Unauthorized operations. Unauthorized access to memory and the attempt to invoke non-installed steering rules is iden-

Application	DPF	Logic	Memory Mapping	Packet State	Steering	Chaining
KVS Cache [23]	Cache-put/get/update	Math	Direct,Segment	In/Out	Return-to-sender	✓
	Count-min sketch [9]	Hash,Math	Segment	In/Out	Optional	
Load Balancer	LB1 (Beamer [40])	Hash	Segment	Out	Dest. array	
	LB2 (Batch)	Math	Direct		Dest. array	
	LB3 (RackSched [59])	Rand	Segment		Dest. array	
Paxos [10]	Leader	Math	Direct	Out	Mulitcast	✓
	Acceptor	Math	Segment	In/Out	Mulitcast,Drop	✓

Table 2: Applications implemented using DPFs and the features they use.

tified by SwitchVM and can be reported to a control plane for further handling.

Denial-of-service with recirculation. Packet recirculation is not allowed by default, and can be limited to specific applications. Further, per-tenant rate limiting can be employed to guarantee performance isolation between recirculated packets of different tenants. In theory, packets can be recirculated indefinitely, as expected from a Turing-complete system. This can be prevented using per-packet recirculation counters, akin to the concept of *gas* in blockchain smart contracts [6, 52], or by re-purposing in-packet TTL counters.

4 Applications

We explain several complex applications we implement using DPFs (Table 2). More use-cases supported by SwitchVM are described in Appendix C.

4.1 Key-Value Store Cache

We implement an in-switch KVS cache, similar to NetCache [23] with keys and values of 4B and 12B, respectively.

Frequently accessed keys are cached and returned upon GET requests, while PUT requests invalidate the respective cached entries. The installation and eviction of the entries are performed by the control plane and orchestrated by the auxiliary mapping data stored in the server using UPDATE operations. The cache evictions are guided by hit-counters and a count-min sketch [9] for frequently accessed keys.

Figure 3 summarizes the GET, PUT, and UPDATE operations. There are four DPFs (in boxes) in the application.

We implement a chain of two DPFs on the miss path. Specifically, cache-get invokes count-min in case of a miss. The DPFs are invoked in two pipes of the same switch, as our prototype only runs on the ingress pipe. These DPFs make use of most SwitchVM functions as explained next.

Data structures. Denote the cache size as k. We use direct mapping for k memory elements in EU0 for mapping keys to <valid-bit, index> tuples. The valid bit is reset when an entry is invalidated. The index represents the virtual address where the value for that key is stored. Additionally, a *null* segment mapping maps all misses to a single invalid tuple. Values are

Figure 3: In-swtich key-value store cache implemented using SwitchVM. ① and ② show the flow of cache hit and miss respectively. Cache-get/put/update and count-min represent DPFs and the pipeline they are executed on.

stored in 3 k-large segment-mapped arrays in EU2 using 4B per entry, and a k-large array in EU3 for the hit counters.

Cache-get. The prolog `POP`s the requested key from the stack and EU0 fetches the corresponding <valid-bit, index> tuple. A `BSET` (branch-if-set) in EU1 computes `AND` between an MSB one-hot mask (MSB set) and the tuple's valid-bit and accesses the tuple's index upon a hit. On a hit, EU2 `LOAD`s the 12B value, and EU3 atomically increments the hit counter using a `FA-ADD` instruction. The epilog `PUSH`s the key and value onto the stack, executes a *return-to-sender* steering policy, and drops the code section. On a miss, the epilog `PUSH`s the key onto the stack and changes the DPF pointer in the packet code header to point to the Count-Min DPF, and the packet continues to the next switch pipe.

Cache-put. The prolog `POP`s the requested key from the stack and EU0 performs a `FA-AND` operation with an MSB one-cold (all but LSB set) mask to mark the entry as invalid. Invalidating a non-cached key is okay, since it is mapped to an invalid entry anyway. The epilog `PUSH`s the key onto the packet stack, and changes the code pointer to a default *empty program* entry. The packet then continues to the next pipe which passes it as is.

Cache-update. The server maintains the mappings of valid keys and their corresponding tuple indices in the switch cache. Upon the installation of a new key, the server sends the index of a free tuple to the switch, alongside the key and the 12B value. The prolog `POP`s these values from the stack, EU0 maps the key to the new tuple, and marks the tuple as valid using an `FA-OR` operation with an MSB one-hot mask. EU2 `STORE`s the 12B value and the epilog executes a *return-to-sender* operation and returns the packet as an ACK to the server.

Count-min sketch. The prolog extracts the key into first three lanes using two `PEEK`s and a single `POP` operations. EU0 `HASH`es these lanes using three different hashing polynomials,

and ANDs the results to acquire only the $\log k$ least significant bits. EU1 use the results as indices for atomically incrementing three counters using FA-ADD instructions. EU2 and EU3 extract the MIN value from all three counters. The epilog PUSHes the key and the frequency back into the packet.

Selective execution. This application demonstrates the use of app_id token for selective DPF execution. The server uses the **cache-update** DPF, which executes at the cache switch. The DPF needs to pass through the sketch switch, but it should not be executed there. However, the *tenant_id* is already configured there for allowing the *count-min* DPF to execute. Thus, we use a separate app_id token for the cache-update DPF and configure it only in the caching switch. We use a different app_id for all other DPFs.

Comparison to NetCache. The key/value bitwidth is limited in SwitchVM, due to the small number of lanes. Therefore, the amount of data that can be read from the packet is limited too. The number of entries in the cache is also limited because we divide the memory between VLIW lanes equally. Consequently, while NetCache supports up to 64K entries, 16B keys, and 128B values, SwitchVM can only support up to 45K entries, 4B keys, and 12B values.

4.2 Load Balancer

We implement three switch-accelerated load balancers.

Load-agnostic LBs. We consider two policies. LB1 splits the traffic by hashing a user-defined value, as previously suggested by Beamer [40]. LB2 schedules batches of 1024 packets to each server using round-robin.

Load-aware LB (LB3). Traffic is split according to load on the servers. Similar to RackSched [59], requests are forwarded to the least loaded server using an in-switch scheduler that implements a power-of-two-choices selection. Servers send their loads by piggybacking on existing traffic (Figure 4).

For lack of space, we only explain the details of LB3 DPFs.

LB3 setup. Denote the number of servers as 2^m. We store two copies of the server loads in two segment-mapped arrays of size 2^m, each assigned to a different lane of EU1. Updates are assured to be consistent between the two arrays due to the RMT per-packet atomicity guarantees.

Client requests. The prolog RAND-omizes two registers. EU0 AND-s each register with a mask to achieve the m least significant bits per register. The results indicate the indices of two servers. Next, EU1 LOAD-s the values of the server-reported loads, and performs a conditional branch using BGT. The selected packet steering policy is based on the branch result.

Server responses. The prolog extracts the server index and updates two registers using two PEEKs and two POPs. EU1 uses these indices to STORE the new load into the corresponding array entries. The packets are not modified.

Comparison to RackSched/Beamer. LB1 and LB3 functionalities are identical to those of Beamer [40] and RackSched [59]. RackSched uses a different memory for each server load, allowing it to use a single copy. This, however, limits the number of supported servers to a few dozens.

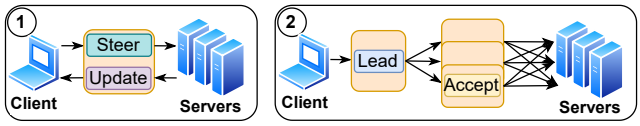


Figure 4: ① Load-aware load balancing. ② In-network Paxos acceleration with one leader switch and three acceptors.

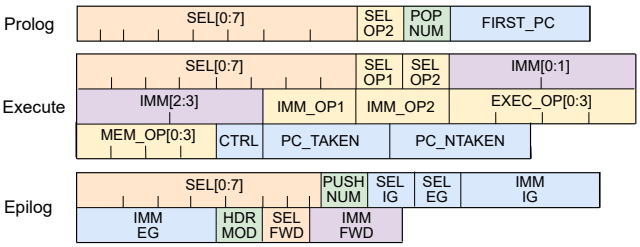


Figure 5: Program encoding for prologs, EUs, and epilogs.

4.3 Paxos

Paxos [29] is a consensus protocol in which clients agree on one *proposed value*. Figure 4 shows how we use DPFs for an in-network acceleration of Paxos, similar to P4xos [10]. The leader is similar to NoPaxos [31]. Due to space limitations, we provide the DPF descriptions in Appendix A.

5 Implementation

Our prototype targets Intel Tofino [19] and has four EUs, each with four lanes. Each lane has up to 45K 32-bit memory entries and 1.5K segment mappings. The first lane of each EU has additional direct mappings, possibly to all registers. Overall, each DPF can execute up-to 16 ALU and 16 Memory operations. Permissions and forwarding tables have up to 32K entries each. It is possible to store up-to 4K in-switch instructions of each type. Assuming each DPF uses at most a single two-way branch, as is the case with our applications, the switch can store up-to 2K entirely distinct DPFs. These can be shared among tenants thanks to virtual memory and instruction pointers. We highlight only the most interesting aspects of the implementation for the lack of space.

Packet structure. DPFs are stored in an option field of the Geneve encapsulation protocol [20]. It includes parser hints, data used by SwitchVM, optional ingress and/or egress DPFs and a data stack. If the stack is small (up to 16 entries) it can be fully parsed with the encapsulated packet (up to layer 4).

Code header. The code header encodes the DPF program (Figure 5). All fields have a default *nop* operation to be ignored by the relevant unit. The *SEL* field determines where to read register values from. The *EXEC_OP* and *MEM_OP* fields dictate which operation should be executed by the ALU

or Memory access. The *CTRL* field determines the DPF control flow. All SwitchVM components are implemented using match-action tables. These tables effectively map the opcodes to the actual execution logic.

We use the prolog unit as an example to show how code in the header is translated into the actual execution. A unit with four lanes is built with 8 match-action tables we call selectors, and each initializes the value of a respective virtual register. The value *SEL[i]* in the prolog code section determines the source for initialization of the i-th register, i.e., the stack (`POP`,`PEEK`) the metadata (`LOAD_MD`) or `RAND`. These opcodes are used by the selector match-action table to initialize the respective register. Semantically, the A registers are initialized first, and the registers with lower indices are initialized before those with higher ones.

Execution Unit. Each EU occupies two hardware pipeline stages (Figure 2). In the first it executes register selection, memory translation, and comparisons. The second runs ALU operations, memory access and branching. Selection is implemented similarly to the Prolog selection above, and it is used to shuffle the registers between different lanes, use them for hashing, and for loading immediate values. The ALUs are per-lane match-action tables with the key being the operation opcode, and the action simply performs the operation on the respective register pair.

Memory Access. Direct and segmented mappings are done in first stage of the EU, for each lane. We match on both tables, but only at most one action is executed, giving priority to the direct mapping. In addition to the address at the lane's A registers, each table accepts the `tenant_id` as an additional matching key for isolation. The resulting physical address is used to access the per-lane Tofino register in the next stage, using the RegisterAction specified by the opcode.

Control Unit. In each EU, we can perform a single branch operation based on evaluating a predicate on two registers. In the first stage, we use comparison to compare two registers. In the second stage we use the result to select code for execution in the next EU. More details about the implementation of all branching operations are found in Appendix D.

5.1 Resource Usage

The key optimization goal is to reduce the number of hardware stages, hence our decision to use VLIW EUs. Permissions, prolog, epilog and steering each requires one hardware stage, whereas each execution unit takes up two. Since we have four execution units, SwitchVM uses all 12 stages of Tofino-1.

Appendix B reports total resource consumption of our SwitchVM prototype, which we summarize here. Our prototype utilizes 57% of available SRAM, due to having the memory access only at the second stage of the two-stage EU. We use the 41% of the TCAM for virtual memory mappings and the control flow units. We utilize 64% of the PHV containers, used to mainly to carry the in-packet parsed DPF code through the pipeline.

The primary constraint on expanding the EU size is the number of logical tables (73%) and available action instruction memory (85%) within each stage. Tofino's design is geared towards a small number of large tables capable of executing wide actions, i.e., ones that can modify many header fields simultaneously. In contrast, SwitchVM would benefit from a large number of small tables with narrow actions. Tofino-2 has 20 stages, which can be utilized for doubling the number of EUs from 4 to 8, implementing P4 extensions, or for handling standard data-center networking operations. We note that we were able to compile eight 2-way VLIW EUs for Tofino-2, but increasing the number of ways failed to compile due to a compiler bug.

As mentioned earlier, the switch can store a maximum of 2K distinct DPFs. These can be safely shared among tenants because they use virtual memory. However, certain resources that cannot be shared may limit the number of co-located tenants. These include the number of direct and segmented mappings in each EU, total memory per EU, and the overall number of forwarding rules.

Appendix B offers more details on the resource usage by the concrete DPFs we implemented in our prototype. The number of distinct co-resident DPFs of the same type depends on the resource usage of each DPF. For example, a single key-value cache can accommodate up to 45K keys, but the number of isolated key-value cache instances is capped at 2K, regardless of the cache size, because of the limit on the number segmented mappings. Similarly, the total memory capacity directly impacts the number of co-resident DPFs. On the other hand, some of the scalability limits can be increased via a more sophisticated implementation of SwitchVM, e.g., by implementing direct mappings on additional lanes.

This factor should be carefully considered, as highlighted in previous research [60].

Deployment. It is often desirable to combine the execution of SwitchVM with traditional networking functions, such as a regular switch. Indeed, we implement a simple L3 switch on the same pipeline. However, relatively high resource usage of SwitchVM may complicate colocation with resource-intensive P4 programs. To this end, we propose three practical deployment options: (1) Leveraging additional pipeline stages of Tofino-2, where 12 stages can be used for SwitchVM and the rest 8 stages for other data-plane functions. (2) In a multi-pipe switch, dedicating a separate pipe for SwitchVM at the cost of reduced bandwidth. (3) Employing SwitchVM as a discrete in-network computing appliance, in line with the existing proposals for stateful network function disaggregation [1].

5.2 Limitations

Design limitations. First, by choosing the language-level virtualization approach, SwitchVM introduces non-negligible

resource overheads which increase the resource consumption and constrain the generality of the functions that can be implemented on top. We believe that this is a viable design point, however, as many in-network computing functions, e.g., eBPFs, require a fairly limited functionality. Second, SwitchVM is tailored for in-network computing and not for general data-plane programmability. This specialization facilitates implementing the security sandbox, which restricts the packet headers exposed to DPFs and constrains the effect of the DPFs on the packet network behavior. The downside of this design choice is that DPFs cannot be used to implement new network protocols, for example.

RMT-related constraints. First, to guarantee performance isolation, RMT-based switches require a DPF to fit within a single pipeline pass. This requirement places a hard upper limit on the DPF size. In contrast, CPU-based dRMT architectures could allow longer DPFs at the expense of gradual performance degradation. Second, RMT switches parse the entire packet prior to processing, necessitating the use of a uniform packet structure for all tenants. Since in SwitchVM the access to packet headers is virtualized, DPFs cannot access the headers, unless dedicated P4 extensions are incorporated during compile time, as discussed in §3.7.

Prototype. We do not yet have a compiler for DPFs, so they are implemented in assembly, using a Python framework for code construction. Further, SwitchVM cannot run on both egress and ingress pipelines due to the limitations of the egress parser in Tofino-1. Last, our control-plane APIs and the client API for using DPFs in applications are rather immature, and the Recirculation rate-limiting logic is not implemented.

6 Evaluation

We aim to highlight the following SwitchVM characteristics:

1. Strong performance isolation among tenants;
2. Runtime programmability behavior without interference when adding/removing tenants;
3. Low latency and bandwidth overheads;
4. End-to-end performance in applications equivalent to P4 baselines.

We emphasize that *SwitchVM was compiled once and never modified throughout all the experiments*.

6.1 Methodology

Setup. We use a dual-socket machine with Intel Xeon Silver 4216@2.1 GHz CPU with 188 GB of RAM and connect it to a 3.2Tbps Intel Tofino switch (EdgeCore Wedge 100BF-32X) via two two-port 100G NICS (Intel E810-C). Unless stated otherwise, the packet generator and the receiver server use different NIC ports and run on different sockets. Hyper-threading and power saving are disabled for consistent results.

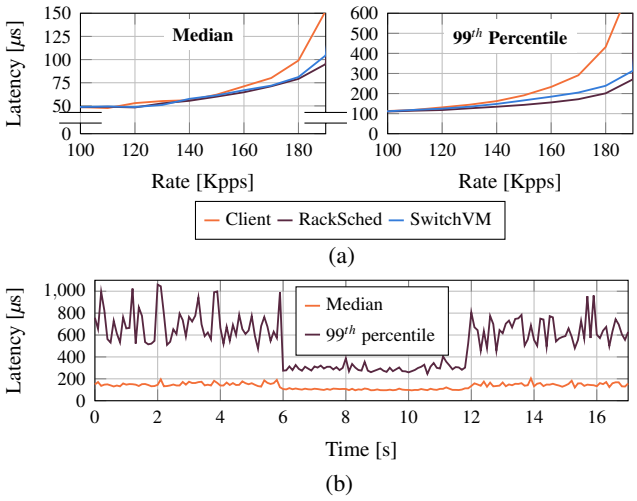


Figure 6: A load-aware load balancer using SwitchVM. (a) SwitchVM demonstrates negligible end-to-end latency overheads compared to RackSched in P4 [59]. (b) SwitchVM changes the load-balancer scheduling policy at times $t = 6, 12$ without any request delays or packet drops.

Packet generator and receiver. Microbenchmarks were performed using Cisco TRex traffic generator [8] (DPDK-based). End-to-end applications are implemented using DPDK. Unless stated otherwise, packets are encapsulated using the Geneve protocol [20] for simulating data center environments.

6.2 End-to-End Applications

We show SwitchVM performance using the Load Balancer and KV store cache applications from §4.

6.2.1 Load-Aware Load Balancer

Our environment consists of two servers that execute the requests of a single client. We compare our implementation to the RackSched [59], a load-aware load balancer written in P4. We use the original RackSched implementation for clients, servers and the P4 scheduler. To run with DPFs we slightly modify the client and servers. Additionally, we add a dummy payload to RachSched original packets to mimic the bandwidth overheads of Geneve encapsulation protocol in SwitchVM, but without performing encapsulation itself to avoid intrusive P4 code changes.

As in RackSched, we run a synthetic workload with a bimodal request processing distribution (5μs and 50μs, for 90% and 10% of the requests respectively). We measure the median and the 99-percentile latency as a function of the load for three systems: the original RackSched, SwitchVM using in-switch DPF, and a client-based implementation that randomly selects a server for each request.

Figure 6a shows no latency difference compared to the P4-only baseline for up to 180K requests/s for the median latency, and up to 15% overheads in the 99% latency for higher rates. We believe, however, that the overhead is not related to the in-switch processing (which shows no sensitivity to load as we see in microbenchmarks), but is an artifact of the increased server load due to the Geneve encapsulation used in SwitchVM. Regardless of these artifacts, the use of DPFs improves the client-only implementation latency by 1.4× and 2.4× at maximum throughput.

In-place policy replacement. Unlike RackSched, SwitchVM can easily replace its scheduling policy without suffering from packet loss due to switch reconfiguration. We show this by dynamically modifying the scheduling policy from batch-round robin to the load-aware power-of-two-choices policy and back (§4). Figure 6b demonstrates the end-to-end median and the 99-th percentile latency as observed by the client. At $t = 6$ and $t = 12$ the policy is altered without the client having to stop. These results clearly demonstrate the power of SwitchVM to react to different network conditions with full application transparency.

6.2.2 In-Switch Cache for Key-Value Stores

We connect one client to a server and use two switch pipes, i.e., each pipe acts as an independent switch. SwitchVM runs a chain of two DPFs that act similarly to NetCache [23].

We demonstrate SwitchVM end-to-end latency for GET operations under abrupt changes in the key distribution (hot-in in [23]) and a constant TX throughput. The client sends 10K requests per-second (10 Kpps) while targeting a set of 128 keys that repeatedly changed every 10 seconds. The server periodically collects the most frequently accessed keys from a count-min sketch [9] implemented in DPF, then caches the 128 most frequent keys in the in-switch cache. Cache hits and sketch counters are cleared every 5 seconds.

Figure 7 reports the median and the 99th percentile of the requests' end-to-end latency. The periodic key changes induces cache misses that redirect the requests to the server, as evident from the momentary latency spikes.

This experiment shows the power of DPFs to implement complex in-network accelerated applications.

6.3 Microbenchmarks

Similar to previous works [23], we simulate a fully loaded switch using a snake configuration, where two ports are connected to the traffic generator and to the receiver server while all remaining ports are connected to each other in pairs. Unless states otherwise, we use MTU-size packets (1500B).

Throughput vs. latency. We compare SwitchVM performance to a standard L3 forwarding application implemented in P4. SwitchVM invokes a *private counter* DPF that increments a counter stored in switch memory, similar to the leader

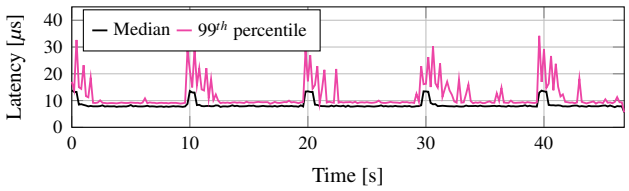


Figure 7: NetCache-like cache for key-value store implemented with DPFs. Latency spikes reflect cache misses due to an abrupt change in the hot-key working set. As the cache is re-populated with new hot keys the latency drops.

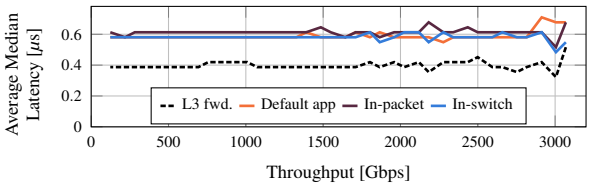


Figure 8: Throughput vs. per-port median latency using 1.5KB MTU-size packets. L3 fwd shows the smallest latency of a functional switch.

DPF used in Paxos (§4).

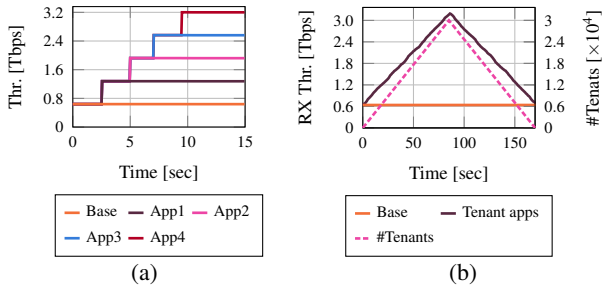
We compare the performance of three ways to invoke a DPF: in-switch, in-packet, and in-switch default DPF that has no bandwidth overheads.

Direct measurements of per-port latency were too noisy due to sub-μs values. Instead, we calculate the average per-port latency as follows. First, we measure the median latency from the generator to the receiver after passing through the switch in the snake configuration. Next, we obtain the median latency from the generator to the receiver using a single pass, and subtract the two to cancel the latency of the server and client machines. We then divide the result by 31, which is the number of additional times the packet traverses the switch in the snake configuration.

Figure 8 shows that the switch approaches its maximum throughput (3.14Tbps) at about the same latency for all invocation techniques. As expected, L3 forwarding has lower latency, all three ways to invoke DPFs perform the same.

Performance isolation. We demonstrate the performance isolation among tenants by running five network flows which together achieve the maximum aggregate switch bandwidth. Among these, there are four flows, each with its own DPF, and a base flow that does not invoke any. We measure the throughput per DPF at the receiver. We start with a single flow and gradually add more to see if there is any interference between them. No measurable interference was observed; neither among the DPFs nor with the other traffic (Figure 9a).

Scalability. We measure SwitchVM tenant reconfiguration performance by scaling the number of served tenants. We use the private counter DPF again, with in-packet code. Note that for a given amount of required resources, the DPF code has

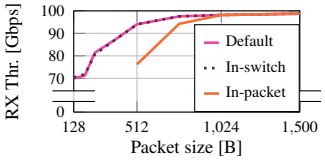


(a) (b)

Figure 9: (a) SwitchVM demonstrates strong performance isolation between different DPFs and a base flow without DPF invocation. (b) Aggregate receive throughput vs. the number of tenants, each having its own DPF and a private switch state. Packets of unauthorized tenants are dropped. Base represents the background traffic not using DPFs.

App	Header Size [B]
Default	0
In-switch	12
In-packet	192

(a)



(b)

Figure 10: (a) Packet overheads for DPFs. (b) Receive side throughput as a function of the packet size.

no effect on scalability, therefore this experiment is representative for the case when each tenant executes her own DPF (Figure 9a).

We start by dynamically adding tenants in batches of 350 tenants per second until reaching 30K tenants, then removing them at the same rate. The process invokes the control plane API to authorize execution of DPFs for the respective tenants, and allocates the private state in the switch (or deauthorizes and deallocates state upon tenant removal). For the purpose of this experiment, SwitchVM is configured to drop packets that attempt to run a DPF that has not been authorized.

The packet generator constantly sends packets for *all* the tenants at 80% line rate. The remaining bandwidth is occupied by packets that do not invoke any DPF. We measure the total bandwidth at the receiver. Initially, tenants are not authorized to run any DPF, so all the packets but those not invoking DPFs are dropped, as expected (Figure 9b). Once the tenants are authorized to run, the aggregate bandwidth grows until all the 30K tenants are authorized. Symmetrically, the tenants are gradually removed in the second part.

This experiment shows that SwitchVM scales to up to 30K tenants with strong performance isolation and without measurable interference. A similar experiment with in-switch code, instead of in-packet, shows the same results. Note however that in this case the amount of *different* DPFs (shared across tenants) is limited to 2K, assuming each DPF has at most a single branch instruction. For comparison, compiler-based approaches for merging P4 programs [41,50,58], and runtime reconfigurable switches [53] can only scale to a handful of co-resident in-switch applications.

DPF bandwidth overheads. Figure 10a reports the additional header size required for a representative DPF application. This overhead translates to a reduction in the effective bandwidth of the system (goodput) as a function of the packet size.

Figure 10b shows the maximal achievable throughput per packet size for different DPF invocation mechanisms. Naturally, the throughput is lower for smaller packets but eventually converges to 99% of the line-rate for all types, at MTU-size packets. Small in-packet DPFs impose packet parsing overheads that reduce the receive-side throughput.

7 Related Work

P4 virtualization. Several prior works focus on virtualization of the data-plane at the P4-level. HyPer4 [17] and HyperVDP [56] use a hypervisor P4 program that emulates other P4 programs, thus incurring significant overheads that limit these approaches to software and FPGA targets. SwitchVM offers a more efficient virtualized ISA that is optimized for in-network computing.

In-switch programming abstractions. Several works implement application-specific in-switch primitives that can be composed at runtime to implement certain in-switch applications. DIP [51] proposes in-switch primitives combined using in-packet recipes for implementing network layer protocols. NetRPC [57] offers primitives that can be composed via an in-switch recipe to implement a few popular in-network applications. Their primitives, however, are tailored to specific tasks. Moreover, their multitenancy is limited to a few dozens of concurrent applications of a few types. SwitchVM is more general and flexible, and scales to thousands of tenants executing arbitrary DPFs.

Compile-time merge. P4Visor [58], PRIME [41], SPEED [7] and [50] propose merging p4 programs before compilation. This has the potential of providing multitenancy, but is limited to only a few applications due to logic partitioning. In addition, changing the set of applications requires recompilation and switch reconfiguration and disrupts the switch traffic.

New architectures. Menshen [49] proposes an extension of the RMT architecture for data-plane multitenancy at the P4 level. Their solution adds a hardware indirection layer to allow running per-tenant packet processing logic on each packet, ensuring cross-tenant isolation using a specialized trusted compiler. SwitchVM also adds an indirection layer with its virtual ISA, with the key difference that it allows secure execution of untrusted DPFs thanks to runtime checks. P4VBox [44], MTPSA [46] and [28] propose new hardware architectures for data-plane virtualization. These architectures allow spatial partitioning of the resources between different

hot-pluggable applications. However, they are currently limited to FPGA targets as they require hardware modifications, and they cannot scale beyond tens of programs. In contrast, SwitchVM runs on commodity switches and can scale to thousands of DPFs.

Runtime programmability. FlexCore [53] and IPSA [15], propose a runtime programmable dRMT architecture that allows changing functionality without causing traffic disruptions. These works do not discuss inter-tenant isolation. SwitchVM naturally offers runtime programmability since changing programs is equivalent to installing match-action entries. HW modification proposals [44,46,49] naturally integrate runtime programmability, a necessity for ensuring performance isolation across tenants.

In-network computing as a service. Several works discuss the isolation requirements [28,46,49,50] for allowing multitenancy of programmable switches in data centers. Runtime memory allocation was proposed in [50,60] to allow changing memory allocation between tenants. SwitchVM allows implementing such allocation policies due to virtual memory.

Multi-core switches. Multicore switches, e.g., Juniper's Trio [38] and NVIDIA's Spectrum [39], follow a more conventional Von Neumann hardware architecture, rather than the dataflow type architecture of RMT switches. SwitchVM can be seen as a compatibility layer that allows architecture-independent network function development, but we leave adaptation of SwitchVM for such targets for future work. These architectures differ substantially from RMT, and therefore might require different design considerations.

RMT switches feature a simple programming abstraction, where packets execute a sequence of match-action tables, such that each packet appears to execute to completion before the next packet processing starts. CPU-based switches on the other hand are more versatile, which potentially makes them better suited for complex in-network applications. In particular, they offer gradual performance degradation for longer programs, as opposed to the RMT pipelines that either fail to run or require coarse-grained packet recirculation. Additionally, CPU-based switches can potentially achieve higher scalability by utilizing a memory hierarchy, in contrast to the use of stage-local memory in RMT pipelines. However, achieving line-rate processing and performance isolation between co-resident programs on CPU-based switches is much more challenging, and cannot be generally guaranteed.

Deployment. Harmony [2] and HIRE [3] discuss the management implications and scheduling requirements of multitenant in-network computing. SwitchVM's scalability reduces the complexity of these management tasks as it makes migrating programs between switches more efficient.

Active Networks. The idea of embedding code into packets for in-network execution dates back to Active Networks [45,47,48]. Active networks allow users of a *shared* infrastructure to inject customized packet processing programs into network nodes, primarily driven by the desire to introduce new networking services on a per-user basis. SwitchVM can be viewed as a version of active networks for multi-tenant in-network computing within a data center. A key difference, however, is that SwitchVM establishes clear *privilege separation* between the network's operator and its users, sidestepping the security and management issues associated with active networks. The in-packet state management we implement in SwitchVM is inspired by tiny packet programs [21]. TPPs do not consider multitenancy and isolation and are focused primarily on telemtery applications.

In a concurrent work, ActiveRMT [12, 13] proposes a capsule-based active networking approach with a shared runtime for interpreting small in-packet programs. Similarly to SwitchVM, ActiveRMT also enables per-packet reprogrammability, with the primary goal of improving in-switch memory utilization for multiple programs. Despite this similarity, ActiveRMT pursues different goals which further dictate different design choices, and make it less suitable for our purposes. First, the in-switch memory in ActiveRMT is segmented, rather than virtualized as in SwitchVM. As a result, in-switch program deployment and sharing available in SwitchVM cannot be easily supported, incurring bandwidth overheads due to in-packet code header and preventing service chaining natively supported in DPFs. Another implication is that, unlike in SwitchVM, in-switch memory reallocation requires updating all the tenants affected by it, impeding scalability in a multi-tenant system. Further, ActiveRMT design relies heavily on packet recirculations, which SwitchVM explicitly strives to minimize in order to enjoy atomic in-switch state update, reduce bandwidth overheads, and achieve strict performance isolation. Last, ActiveRMT does not support variable-size in-packet state, so it complicates its use in multi-switch applications such as in-network telemetry.

8 Conclusion

SwitchVM enables data center tenants to build in-network accelerated applications by securely deploying and executing Data Plane Filters on programmable switches. SwitchVM supports concurrent execution of thousands of DPFs while offering strong performance, state and fault isolation. We show SwitchVM's ability to build complex applications and experimentally demonstrate its performance and scalability. We envision that SwitchVM will open new opportunities for in-network accelerated applications in data centers.

Acknowledgments

We thank our shepherd Mina Tahmasbi Arashloo and the reviewers for their helpful comments and feedback. We also thank Ron Marcus and Assaf Klein for their help with shaping this paper. We gratefully acknowledge generous support from Israel Science Foundation (Grant 1998/22).

References

[1] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, Balakrishnan Raman, Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, and Srikanth Kandula. Disaggregating stateful network functions. In *USENIX NSDI*, 2023.

[2] Theophilus A. Benson. In-network compute: Considered armed and dangerous. In *ACM HotOS*, 2019.

[3] Marcel Blöcher, Lin Wang, Patrick Eugster, and Max Schmidt. Switches for hire: Resource scheduling for data center in-network computing. In *ASPLOS*, 2021.

[4] Marcel Blöcher, Lin Wang, Patrick Eugster, and Max Schmidt. Holistic resource scheduling for data center in-network computing. *IEEE/ACM TON*, 2022.

[5] Broadcom. Trident4 / BCM56880 series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series, 2023.

[6] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[7] Xiang Chen, Hongyan Liu, Qun Huang, Peiqiao Wang, Dong Zhang, Haifeng Zhou, and Chunming Wu. SPEED: Resource-efficient and high-performance deployment for data plane programs. In *IEEE ICNP*, 2020.

[8] Cisco. TREX: Realistic traffic generator. https://trex-tgn.cisco.com/, 2023.

[9] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 2005.

[10] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM TON*, 2020.

[11] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at network speed. In *ACM SOSR*, 2015.

[12] Rajdeep Das and Alex C. Snoeren. Enabling active networking on RMT hardware. In *ACM HotNets*, 2020.

[13] Rajdeep Das and Alex C Snoeren. Memory management in activermt: Towards runtime-programmable switches. In *ACM SIGCOMM*, 2023.

[14] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An infrastructure for inline acceleration of network applications. In *USENIX ATC*, 2019.

[15] Yong Feng, Zhikang Chen, Haoyu Song, Wenquan Xu, Jiahao Li, Zijian Zhang, Tong Yun, Ying Wan, and Bin Liu. Enabling in-situ programmability in network data plane: From architecture to language. In *USENIX NSDI*, 2022.

[16] Maurizio Gabbrielli and Simone Martini. *Programming Languages: Principles and Paradigms*. Undergraduate Topics in Computer Science. Springer, 2010.

[17] David Hancock and Jacobus van der Merwe. HyPer4: Using P4 to virtualize the programmable data plane. In *ACM CoNEXT*, 2016.

[18] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM SIGCOMM*, 2017.

[19] Intel. Intel® tofino™ programmable ethernet switch asic. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html, 2023.

[20] T. Sridhar J. Gross, I. Ganga. Geneve: Generic network virtualization encapsulation. https://www.rfc-editor.org/rfc/rfc8926.pdf, 2020.

[21] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of little minions: Using packets for low latency network programming and visibility. In *ACM SIGCOMM*, 2014.

[22] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-RTT coordination. USENIX NSDI, 2018.

[23] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. ACM SOSP, 2017.

[24] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable load balancing using programmable data planes. In *ACM SOSR*, 2016.

[25] Elie F. Kfoury, Jorge Crichigno, and Elias Bou-Harb. An exhaustive survey on P4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *IEEE Access*, 2021.

[26] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, 2015.

[27] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: making RPCs first-class datacenter citizens. In *USENIX ATC*, 2019.

[28] Johannes Krude, Jaco Hofmann, Matthias Eichholz, Klaus Wehrle, Andreas Koch, and Mira Mezini. Online reprogrammable multi tenant switches. In *ACM ENCP*, 2019.

[29] Leslie Lamport. Paxos made simple, fast, and byzantine. In *OPODIS*, 2002.

[30] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-free consistent transactions using network multi-sequencing. In *ACM SOSP*, 2017.

[31] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say no to Paxos overhead: Replacing consensus with network ordering. In *USENIX OSDI*, 2016.

[32] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. USENIX OSDI, 2020.

[33] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A better netflow for data centers. In *ACM NSDI*, 2016.

[34] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High precision congestion control. In *ACM SIGCOMM*, 2019.

[35] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM SIGCOMM*, 2016.

[36] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, 1993.

[37] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *ACM SIGCOMM*, 2017.

[38] Juniper Networks. MX series universal routing platforms. https://www.juniper.net/us/en/products/routers/mx-series.html, 2023.

[39] NVIDIA. NVIDIA spectrum-4. https://www.nvidia.com/en-us/networking/ethernet-switching, 2023.

[40] Vladimir Andrei Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *USENIX NSDI*, 2018.

[41] Ricardo Parizotto, Lucas Castanheira, Fernanda Bonetti, Anderson Santos, and Alberto Schaeffer-Filho. PRIME: Programming in-network modular extensions. In *IEEE NOMS*, 2020.

[42] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. ACM HotNets, 2017.

[43] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *USENIX NSDI*, 2021.

[44] Mateus Saquetti, Guilherme Bueno, Weverton Cordeiro, and Jose Rodrigo Azambuja. P4VBox: Enabling P4-based switch virtualization. *IEEE Communications Letters*, 2020.

[45] B. Schwartz, A.W. Jackson, W.T. Strayer, Wenyi Zhou, R.D. Rockwell, and C. Partridge. Smart packets for active networks. In *OPENARCH*, 1999.

[46] Radostin Stoyanov and Noa Zilberman. MTPSA: Multi-tenant programmable switches. In *ACM EuroP4*, 2020.

[47] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *SIGCOMM CCR*, 2007.

[48] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden. A survey of active network research. *IEEE Communications Magazine*, 1997.

[49] Tao Wang, Xiangrui Yang, Gianni Antichi, Anirudh Sivaraman, and Aurojit Panda. Isolation mechanisms for high-speed packet-processing pipelines. In *USENIX NSDI*, 2022.

[50] Tao Wang, Hang Zhu, Fabian Ruffy, Xin Jin, Anirudh Sivaraman, Dan RK Ports, and Aurojit Panda. Multitenancy for fast and programmable networks in the cloud. In *USENIX HotCloud*, 2020.

[51] Ziqiang Wang, Zhuotao Liu, Xiaoliang Wang, Songtao Fu, and Ke Xu. DIP: Unifying network layer innovations using shared L3 core functions. In *ACM HotNets*, 2022.

[52] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.

[53] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. Runtime programmable switches. In *USENIX NSDI*, 2022.

[54] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, centralized lock management using programmable switches. ACM SIGCOMM, 2020.

[55] Lior Zeno, Dan RK Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. SwiSh: Distributed shared state abstractions for programmable switches. In *USENIX NSDI*, 2022.

[56] Cheng Zhang, Jun Bi, Yu Zhou, and Jianping Wu. HyperVDP: High-performance virtualization of the programmable data plane. *IEEE JSAC*, 2019.

[57] Bohan Zhao, Wenfei Wu, and Wei Xu. NetRPC: Enabling in-network computation in remote procedure calls. In *USENIX NSDI*, 2023.

[58] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4Visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *ACM CoNEXT*, 2018.

[59] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. RackSched: A microsecond-scale scheduler for rack-scale computers. In *USENIX OSDI*, 2020.

[60] Hang Zhu, Tao Wang, Yi Hong, Dan R. K. Ports, Anirudh Sivaraman, and Xin Jin. NetVRM: Virtual register memory for programmable networks. In *USENIX NSDI*, 2022.

A The Paxos DPFs

We focus on two DPFs that correspond to Paxos's second phase messages.

Leader. This DPF requires a single directly mapped counter that can be placed in either lane of any EU. Upon packet arrival, the EU increments the counter using a `FA-ADD` instruction. Next, the epilog `PUSH`es the register value onto the packet stack alongside the Paxos round number, which is a constant zero for the fast-path phase of Paxos. The epilog also changes the program counter to the acceptor DPF and performs *multicast* steering action to send the packet to all acceptors.

Resource	Utilization
Exact match Xbar	24%
Ternary match Xbar	20%
TCAM	41%
SRAM	57%
Hash bit units	49%
Hash distribution units	72%
Gateways	17%
Action instruction memory	85%
Logical tables	73%
Normal PHV containers	64%

Table 3: SwitchVM resource utilization on Tofino-1 with 4 EUs and 4 lanes per EU.

Acceptor. Three n-sized arrays are allocated; one in EU0, two in EU2. The first array holds a list of Paxos round values per proposition. Similarly, the second and third arrays hold the most recent value and the Paxos *vround* value for each proposition, respectively.

The prolog `POP`-s the value proposition, its index i ($i < n$), and the round number from the packet stack. EU0 performs an atomic *fetch and max* (`FA-MAX`) between the locally saved i^{th} instance's round number and the one extracted from the packet. Next, EU1 compares these two values. If the local round number is larger than that of the packet, the packet is dropped. Otherwise, EU2 updates the ith vround entry to hold the new round value and alters the ith value to the one suggested by the packet. Then, the epilog *multicast*-s the packet to all learners.

Comparison to P4xos and NoPaxos. The leader is functionally identical to NoPaxos. Constraints similar to the ones presented for the KVS application (§4.1) limit the values to 4B, compared to 32B in P4xos.

B Resource Usage

Table 3 reports total resource consumption of the SwitchVM prototype. Table 4 shows the resource consumption of various DPFs in terms of SwitchVM resources.

C Additional Applications

Tiny Packet Programs. SwitchVM extends Tiny Packet Programs (TPP) [21] with the ability to do complex computations instead of a handful of simple instructions. This allows, for example, aggregation of the maximal hop latency along the path instead of collecting the samples into the packet.

Distributed coordination. A small amount of stateful memory can be used for implementing several distributed latency-sensitive coordination tasks, including leader election, aggregation, and distributed barriers.

NetChain. NetChain [22] performs in-network coordination between switches. It requires similar functionalities as in

App	EU0			EU1			EU2			EU3			FWD
	Dir	Seg	Mem	Dir	Seg	Mem	Dir	Seg	Mem	Dir	Seg	Mem	
Cache	k		k		3	$3k$		1	k				1
Sketch					3	$3k$							
LB1					3	$3k$				1		1	k
LB3					2	$2k$							k
Leader	1		1										1
Acceptor		1	k				1	2	$2k+1$				1
Available total	45K	6K	180K	45K	6K	180K	45K	6K	180K	45K	6K	180K	32K

Table 4: SwitchVM resource consumption for various DPFs. k stands for the size of each DPF (top to bottom): number of cached keys, number of counters in sketch, number of servers, number of servers, N/A, number of value propositions.

Paxos and NetCache, both already implemented. Source routing can be performed using the in-packet data stack.

Complex functions. Turing-completeness allows building arbitrary functions using packet recirculation and the code-pointer changing mechanism. For example, one can implement bubble sort in a few DPFs, i.e., element swapping DPF, comparison DPF, and an iterator DPF.

Fast-path/slow-path. By adding mirroring capabilities to the steering module, complex data-plane operations can split the traffic to fast-/slow-paths. For example, all packets execute a DPF that selects which packets to mirror at egress while changing their code pointer to a slow-path DPF, which may be a complex Turing-complete program.

D Control Unit Implementation

We now describe how the control logic parses the DPF fields and changes the execution flow. The DPF fields that are taken into account in the control unit are `CTRL_OP`, `PC_TAKEN`, `PC_NTAKEN`, `SEL_OP1`, `SEL_OP2`, `IMM_OP1` and `IMM_OP2`. We refer the reader to Figure 5 for a complete list of the DPF fields. `PC_TAKEN` and `PC_NTAKEN` contain pointers to the next execution unit's instruction, or for the epilog code in the last EU. It can point to either an in-packet instruction, or an in-switch one.

Figure 11 provides an overview of the control logic. At first, the operands are acquired based on the `SEL_OP1` and `SEL_OP2` fields. Each of these fields may either represent a register index or denote that an immediate value should be loaded from one of the `IMM_OP` fields. After obtaining the operands they get compared using five operators (CMP component in Figure 11): bitwise-AND, bitwise-XOR, saturated-subtraction, sign-operand1, and sign-operand2. The outcome is a vector that holds the results of all operations.

Next, a branch resolution unit takes the `CTRL_OP` data-field for determining the next program-counter value. This is performed using a TCAM selection that finds the *first matching result* and acts according to the output action, effectively implementing a long if-else-if operation. Depending on the result, the code for the next stage is loaded from either the packet or the stage local instruction memory. The table is configured as follows (underscores represent don't-cares):

```
table branch_tbl {
    key = {
        ctrl_op : ternary;
        bitwise_and : ternary;
        bitwise_xor : ternary;
        sat_sub: ternary;
        oprnd1_sign : ternary;
        oprnd2_sign : ternary;
    }
    entries = {
        (HALT, _, _, _, _, _) : halt();
        (JMP,  _, _, _, _, _) : taken();
        (BSET, 0, _, _, _, _) : not_taken();
        (BSET, _, _, _, _, _) : taken();
        (BEQ,  _, 0, _, _, _) : taken();
        (BEQ,  _, _, _, _, _) : not_taken();
        (_,    _, 0, _, _, _) : not_taken();
        (BLT,  _, _, 0, _, _) : taken();
        (BGT,  _, _, 0, _, _) : not_taken();
        (BGT,  _, _, _, _, _) : taken();
        (BSLT, _, _, 0, 0, 0) : taken();
        (BSLT, _, _, 0, 1, 1) : taken();
        (BSLT, _, _, _, 1, 0) : taken();
        (BSGT, _, _, _, 0, 1) : taken();
        (BSGT, _, _, 0, 0, 0) : not_taken();
        (BSGT, _, _, _, 0, 0) : taken();
        (BSGT, _, _, 0, 1, 1) : not_taken();
        (BSGT, _, _, _, 1, 1) : taken();
        (_,    _, _, _, _, _) : not_taken();
    }
}
```

`HALT` stops the execution, `JMP` unconditionally choose the instruction at `PC_TAKEN` for the next stage, all others opcodes implement conditional branches by choosing between `PC_TAKEN` and `PC_NTAKEN` based on the comparison result. `BEQ` checks for equality, `BLT` and `BGT` check of less-than and great-than inequalities of unsigned integers, `BSLT` and `BSGT` are their signed counterparts, `BSET` performs a bitwise-and between the operands and compares it to zero. Due to the use of two-way branches, we can achieve more functionality by swapping `PC_TAKEN` and `PC_NTAKEN`. For example, to implement a less-than-equal branch we can use a greater-than branch with swapped PC targets.

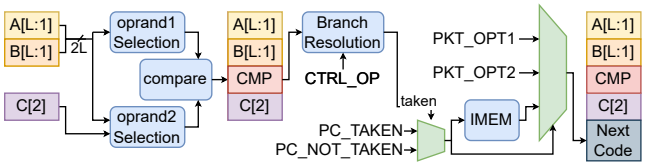


Figure 11: Overview of SwitchVM control unit.

Characterization of Large Language Model Development in the Datacenter

Qinghao Hu[*☆1], Zhisheng Ye[*☆3], Zerui Wang[*☆4], Guoteng Wang[☆], Meng Zhang[☆1], Qiaoling Chen[☆1]
Peng Sun[☆5], Dahua Lin[☆6], Xiaolin Wang[3], Yingwei Luo[3], Yonggang Wen[2], Tianwei Zhang[2]

[☆]*Shanghai AI Laboratory* [1]*S-Lab, Nanyang Technological University* [2]*NTU*
[3]*Peking University* [4]*Shanghai Jiao Tong University* [5]*SenseTime Research* [6]*CUHK*

Abstract

Large Language Models (LLMs) have presented impressive performance across several transformative tasks. However, it is non-trivial to efficiently utilize large-scale cluster resources to develop LLMs, often riddled with numerous challenges such as frequent hardware failures, intricate parallelization strategies, and imbalanced resource utilization. In this paper, we present an in-depth characterization study of a six-month LLM development workload trace collected from our GPU datacenter `Acme`. Specifically, we investigate discrepancies between LLMs and prior task-specific Deep Learning (DL) workloads, explore resource utilization patterns, and identify the impact of various job failures. Our analysis summarizes hurdles we encountered and uncovers potential opportunities to optimize systems tailored for LLMs. Furthermore, we introduce our system efforts: (1) *fault-tolerant pretraining*, which enhances fault tolerance through LLM-involved failure diagnosis and automatic recovery. (2) *decoupled scheduling for evaluation*, which achieves timely performance feedback via trial decomposition and scheduling optimization.

1 Introduction

Over the years, advances in LLMs have attracted significant attention from both academia and industry owing to their impressive performance and capabilities, such as ChatGPT [2] and GitHub Copilot [3]. However, due to their immense model sizes and extensive data demands, training such models necessitates a substantial computational infrastructure with thousands of accelerators [27, 68]. Hence, it is a common practice for tech companies and cloud providers to build large-scale GPU clusters to facilitate LLM development, especially after the popularity of ChatGPT. Nevertheless, it is non-trivial to perform efficient LLM development on such high-cost infrastructure. Developers often confront numerous issues and challenges, including frequent hardware failures [64, 96], intricate parallelization strategies [68, 113], unstable training progress [1, 110], long queuing delay [104], etc.

Developing LLMs is closely intertwined with the support of GPU clusters in various aspects. A thorough analysis of cluster workloads is essential for comprehending challenges and uncovering opportunities in designing systems tailored for LLMs. However, many conclusions and implications from existing DL workloads analysis works [38, 45, 97], conducted before the rise of LLMs, are not applicable to LLM development. This is primarily due to the divergent characteristics and requirements of LLMs:

(1) *Paradigm Transition*. DL workloads generally follow a *task-specific* paradigm that trains the model on domain-specific data to tackle a particular task (e.g., translation [18]). In contrast, LLMs follow an emerging paradigm that performs self-supervised training on broad data to generate a *foundation model* [19] and further adapts to a wide range of downstream tasks. This shift signifies a substantial divergence in the model development pipeline (e.g., pretraining [85], alignment [37]) and workload characteristics from prior DL workloads (§2.1).

(2) *Tailored Software Stack*. To accommodate the enormous model size of LLMs, a series of systems implement advanced techniques to optimize the execution of LLMs. For instance, Deepspeed [79], Megatron [68] and Alpa [113] accelerate the training via hybrid parallelism or state-sharding optimizer. As for model serving, Orca [104] and vLLM [51] improve throughput via iteration scheduling or memory management.

(3) *Unified Architecture*. Prior DL workloads usually employ various model architectures (e.g., CNN [54], RNN [18]) to address diverse tasks. In contrast, LLMs commonly embrace the Transformer [93] architecture, like BERT [31], GPT-3 [20], LLaMA [91] and PaLM [27]. The architectural homogeneity suggests a high level of uniformity in the LLM development pipeline and similarity across different datacenters.

To bridge this gap, we present an in-depth study of our operational experiences in the datacenter `Acme` of Shanghai AI Laboratory. It houses two distinct clusters, `Seren` and `Kalos`, dedicated to LLM development and equipped with 4,704 A100 GPUs in total. Our analysis draws upon traces collected over a six-month period from March to August 2023, encompassing scheduler logs, infrastructure monitoring data, failure logs, and fine-grained profiling data. Our key findings and identified challenges can be summarized as follows:

• **Shorter Job Duration and Unfair Queuing Delay**. In contrast to the common stereotype that LLM workloads are usually long-term, the workloads in our datacenter exhibit 2.7∼12.8× shorter average job duration compared to the DL workloads in previous traces [38, 45, 97]. This can be

[*]Equal Contribution.

attributed to the presence of numerous short-term tasks such as evaluation. In terms of job queuing delay, our findings also diverge from previous DL traces that larger-scale jobs experience longer wait times. We observe that evaluation jobs, despite being short-term and small-scale, have the longest queuing delay. This discrepancy stems from reserving the majority of resources for pretraining jobs to minimize their queuing delays. Evaluation jobs are scheduled with a lower priority, utilizing the limited spare resources.

• **Imbalanced Resource Usage**. The imbalance is manifested in two aspects. Firstly, in terms of workload distribution, pretraining jobs only account for 3.2% of the total job count but consume 94.0% of the whole compute resource (i.e., GPU time) in `Kalos`. Conversely, evaluation jobs, despite constituting 92.9% of all jobs, only utilize a meager 0.8% of resources. Secondly, when looking at infrastructure utilization, we find that associated resources including CPU, host memory, and network, are frequently underutilized. In contrast, the GPU, as the primary resource, shows high utilization rates. Both GPU memory and GPU utilization exhibit substantially higher median values at 75% (60GB) and 99% respectively in `Kalos`, as opposed to the 10% and 4% observed in PAI [97]. These observations corroborate that LLMs are computationally and memory intensive. It also implies that GPU-sharing-based techniques [40, 98, 99, 106] may not be suitable for LLMs.

• **Long GPU Idle Time in Evaluation Workload**. Our profiling of evaluation workloads reveals substantial underutilization of GPU resources at various stages. For example, the evaluation job on HumanEval consumes 29.5% of its time for model loading and data preprocessing, and an additional 19.0% is spent conducting synthesized program correctness tests. As a result, only half of the time is dedicated to GPU inference, leading to long queuing delays in evaluation trials.

• **Frequent Job Failures**. We find various errors primarily occur at the beginning of LLM workloads, leading to fast job termination. However, infrastructure failures, which are common in long-term pretraining jobs, significantly impede training efficiency. Therefore, prompt diagnosis and recovery from these failures are crucial to enhance training efficiency.

Based on our characterization study, we identify several challenges encountered during the LLM development, such as unstable training progress, remote storage bottleneck and delayed feedback on model performance. To tackle these issues, we consolidate the insights gained from our operational experience and build two systems that are integrated into our LLM framework to improve development robustness and efficiency. *Firstly*, to mitigate the frequent failure problem, we establish a system to achieve ***fault-tolerant pretraining***. It incorporates three key designs: (1) achieving frequent model saving through asynchronous checkpointing, (2) identifying the root causes of various failures through a combination of heuristic rules and LLM, (3) employing a holistic detection toolkit to pinpoint fault nodes and automatically restart training from properly saved checkpoint. It accelerates checkpointing by 3.6∼58.7× and significantly reduces manual intervention. *Secondly*, we develop a system that performs ***decoupled scheduling for evaluation*** to provide developers with timely feedback on model quality. It not only resolves the remote model loading contention issue via decoupled model retrieval but also minimizes GPU idle time via decoupling the metric computation process. It further leverages the prior knowledge and flexibility of datasets to balance workload across all GPUs. Our experiment shows that it can reduce the evaluation makespan by up to 1.8×.

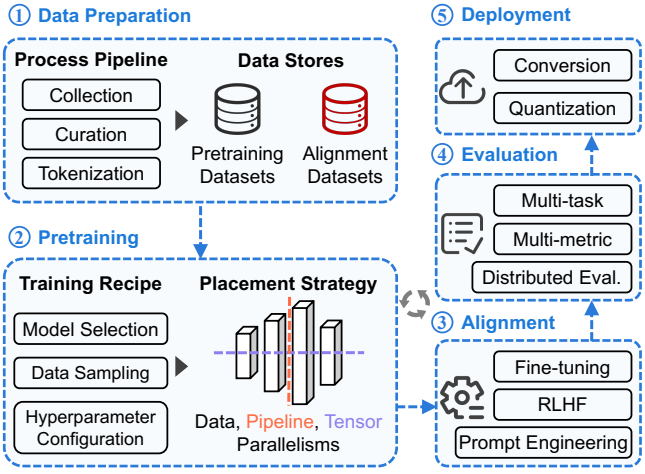


Figure 1: Overview of the LLM development pipeline.

We believe the observations and insights derived from our datacenter do not stand in isolation. Our traces are publicly available at `https://github.com/InternLM/AcmeTrace`. We also release our system code and models (Appendix C). We hope these resources and lessons can benefit researchers in optimizing LLM systems as well as GPU cluster management.

2 Background

2.1 LLM Development Pipeline

Distinguished from task-specific DL models, LLMs follow an emerging paradigm that performs self-supervised training on broad data and further adapts to a wide range of downstream tasks [19]. The development of LLMs necessitates the use of extensive computational infrastructure due to their substantial model size (comprising billions of parameters) and the vast amount of training data involved. Figure 1 depicts the comprehensive LLM development pipeline, encompassing five distinct stages (blue blocks) that span from scratch to service (follow blue arrows). The grey circular arrow indicates that the pretraining stage enables periodical alignment and evaluation to assess intermediate models and adjust configuration on the fly. We explain each stage as follows:

Data Preparation. The initial stage involves gathering and preprocessing the training data, which can be categorized into two parts: (1) *pretraining data*, consisting of extensive unlabeled corpora obtained from public or private sources and curated through processes like detoxification and dedu-

Cluster	#CPUs	#GPUs	Mem(GB)	Network	#Nodes
Seren	128	8	1,024	1×200Gb/s	286
Kalos			2,048	5×200Gb/s	302

Table 1: Summary of per-node specification and cluster scale for two independent LLM clusters in Acme.

plication; (2) *alignment data*, comprising a smaller set of high-quality labeled corpora used to align the model with specific tasks. This data is typically acquired through expensive human annotation or labeling [71]. Besides, all the data must be tokenized to ensure compatibility with the model's input.

Pretraining. It involves self-supervised training on large-scale curated data, demanding a majority of resources within the overall development workflow. Training LLMs efficiently at scale necessitates various system innovations, such as state-sharding optimizers [79], meticulous model placement using data, pipeline, and tensor parallelisms [67, 68, 113].

Alignment. This stage aims to adapt LLMs with user intent on a wide range of downstream tasks. Two primary aligning paradigms are commonly used: (1) *prompt engineering*, specifying prompts (i.e., inputs) without modifying model parameters. For example, in text summarization, appending a prompt *TL; DR* to the input article can improve model performance [78]; (2) *fine-tuning*, updating model parameters on a task-specific dataset to improve performance in a particular domain. Additionally, reinforcement learning from human feedback (RLHF) [71] further enhances the alignment effect, and parameter-efficient techniques like LoRA [37] have been proposed to reduce the cost of fine-tuning.

Evaluation. Given the vast application scenarios of LLM, it may be inaccurate to assess model quality solely based on a single metric like training loss. There are numerous factors to consider, such as accuracy, fairness, and toxicity [58]. Consequently, it is crucial to account for a diverse set of criteria and measure performance across multiple tasks [22]. Furthermore, regular evaluation is essential during the pretraining stage to provide timely feedback on model quality.

Deployment. To meet the strict cost and latency constraints of LLM applications, several advanced techniques have been developed to achieve efficient model serving, including quantization [30], distillation [83], CUDA kernel optimization [29,43], model parallelism [57, 104] and memory management [51].

2.2 Acme Overview

Acme is our private GPU datacenter that empowers researchers and engineers to develop DL models across diverse domains. In this work, we focus on analyzing workloads within two clusters dedicated to developing LLMs: Seren and Kalos. We collect and analyze all jobs in these two clusters. Note that there are additional clusters within Acme that are designated for different fields, such as autonomous driving, and AI for scientific research. However, these clusters are excluded in this work as they are unrelated.

Cluster Architecture. Table 1 summarizes configurations of these two homogeneous LLM clusters. Seren and Kalos have 2,288 and 2,416 GPUs respectively. Each node is equipped with 8× NVIDIA A100-SXM 80GB GPUs [6] and 2× Intel Xeon Platinum 8358P CPUs (128 threads in total). GPUs are interconnected to each other by NVLink and NVSwitch [9], and inter-node communication is achieved via NVIDIA Mellanox 200Gbps HDR InfiniBand [4]. Compared to Seren, Kalos is a relatively newer cluster with an improved network configuration. Each node in the Kalos has a larger host memory (2TB) and is equipped with four InfiniBand HCAs specifically for application communication, along with an extra HCA dedicated to storage.

Besides, the distributed storage system is also critical for workload performance. Acme adopts an all-NVMe shared parallel file system for fast data access and storage. Moreover, as time has advanced, our resource scheduling system has evolved to support diverse cluster environments. Specifically, our scheduler on Seren and Kalos is built atop Slurm [102] and Kubernetes [21] respectively. In order to provide resource guarantees for large-scale pretraining jobs, our scheduler enables resource isolation and quota reservation. It further incorporates a best-effort job mechanism for higher utilization.

LLM Workloads. We develop a collection of LLMs[1] ranging from 7B to over 123B parameters. All of these models follow the transformer-based decoder-only architecture, similar to the GPT [20, 77, 78] and LLaMA [91, 92] series. Acme encompasses tasks in the aforementioned general LLM development pipeline (§2.1). Note that Acme does not involve any serving jobs, as our LLMs are deployed on a separate cluster specifically for serving purposes.

Software Stack. To support the training of billion-scale models across thousands of GPUs, we built a system InternEvo[2], which integrates various system optimization techniques, such as FlashAttention [28, 29], 3D parallelism [68], zero redundancy optimization [79], mixed precision training [10], selective activation recomputation [50] and fine-grained communication overlap. Moreover, it accommodates additional tasks such as model fine-tuning and evaluation.

2.3 Traces from Acme

The optimization of LLM-tailored systems and datacenter management can significantly boost development efficiency and yield substantial financial benefits. Achieving this goal requires a profound understanding of the intrinsic characteristics of LLM workloads. Many insights in existing DL workloads analysis works [38, 45, 97] are not applicable to LLM workloads due to the unique attributes of LLMs. To fill this gap, we collected and analyzed workload traces from our datacenter Acme. Table 2 compares the specifications and trace information of Acme with prior trace analysis works conducted by Microsoft, SenseTime, and Alibaba. Unlike

[1]Model: https://huggingface.co/internlm
[2]System: https://github.com/InternLM/InternEvo

	For Task-Specific DL Models			For LLMs
Datacenter	**Philly** [45]	**Helios** [38]	**PAI** [97]	**Acme**
Year	2017	2020	2020	2023
Duration	3 months	6 months	2 months	6 months
#Jobs	113K	3.36M	1.26M	1.09M
Avg. #GPUs	1.9	3.7	0.7	6.3
GPU Model	12GB/24GB	1080Ti/V100	T4/P100/V100	A100
Total #GPUs	2,490	6,416	6,742	4,704

Table 2: Comparison between our datacenter Acme and GPU datacenters in prior trace analysis works: Microsoft Philly [45], SenseTime Helios [38], Alibaba PAI [97]. Philly only provides GPU memory sizes without clarifying GPU models. The average number of requested GPUs in PAI can be less than 1 (0.7), as it supports fractional (<1) GPU requests.

Acme, which is solely dedicated to LLM development, these datacenters encompass a mixture of general DL workloads from various domains. For instance, Helios [38] consists of 4 clusters dedicated to training models in computer vision and reinforcement learning, while PAI [97] includes a diverse range of servers for training and serving jobs.

Trace Source. Our characterization study is based on traces collected from two LLM clusters in Acme. The traces span 6 months from March to August 2023. Seren contains 368K CPU jobs and 664K GPU jobs, while Kalos job trace consists of 42K CPU jobs and 20K GPU jobs. Additionally, we provide a summary of the data sources for the traces used in our study: (1) *Job Log*. We collect the job logs from our scheduler database, which contains detailed information for each job. This includes the job's execution time (submission, start, and end), final status (completed, canceled, failed), requested resources (CPU, GPU, memory), work directory, and other relevant data. (2) *Hardware Monitor Data*. This encompasses long-term, multi-dimensional data obtained from various sources. We collect CPU, memory, and network usage data from Prometheus [75] database, GPU-related metrics from NVIDIA DCGM [7], and power-related data from IPMI [12]. The sampling interval for this data is set at 15 seconds. (3) *Runtime Log*. To conduct a precise job failure analysis, we capture stdout and stderr logs from LLM frameworks during job execution. (4) *Profiling Data*. For a subset of representative jobs, we delve deeper by performing fine-grained profiling using tools like DCGM. The synergy of these trace dimensions allows us to gain a holistic understanding of LLM job characteristics in datacenters.

3 Datacenter Characterization

In this section, we perform a thorough analysis of Acme, including comparing workload distribution between LLMs and previous DL workloads (§3.1), investigating different LLM workload types (§3.2), exploring resource utilization patterns (§3.3) and assessing environmental impacts (§3.4).

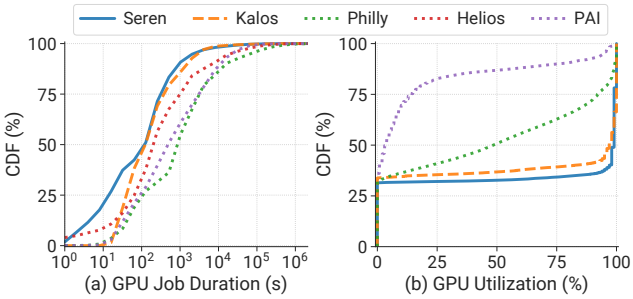


Figure 2: Overview of different datacenter characteristics. (a) *Workload*: CDF of the GPU job duration. (b) *Infrastructure*: CDF of GPU utilization, where Helios' data is not available.

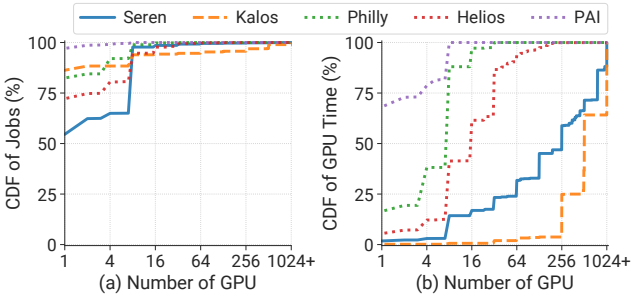


Figure 3: Comparison of workload distribution based on the number of requested GPUs. (a) CDF of job count. (b) CDF of GPU time (i.e., requested GPU number × duration).

3.1 LLMs versus Prior DL Workloads

Shorter Job Duration. As shown in Figure 2 (a), contrary to the prevailing stereotype that LLM-related jobs are typically long-running, we find the workloads in our clusters (blue and orange lines) exhibit shorter GPU job durations (i.e., job runtime, excluding queuing delay) compared to the DL workloads observed in previous job traces (dotted lines). Specifically, both the Seren and Kalos have a median job duration of 2 minutes, which is 1.7∼7.2× shorter than the median job durations of other clusters. Furthermore, it is evident that the more recent trace demonstrates a shorter job duration distribution. In particular, when considering the average job duration in the Philly cluster (collected in 2017), it is 2.7∼3.8× longer than Helios (2020) and PAI (2020), and 12.8× longer than Acme (2023). To provide an explanation for this observation, we outline three potential factors: (1) *Hardware upgrade*. The iteration of GPU and networking delivers substantial efficiency improvement. (2) *Abundant resources*. Users usually request more resources (as shown in Table 2), averaging 5.7 GPUs in the Seren and 26.8 GPUs in the Kalos. This can significantly accelerate the training process. (3) *Extensive associated workloads*: LLM development pipeline involves numerous small-scale associated jobs, such as evaluation. We will delve into this in §3.2. (4) *High incompletion rate*: Approximately 40% of jobs fail, with completed jobs consuming only 20∼30% of GPU resources. This highlights the urgent need for a fault-tolerant system. Further details can be found in Figure 17 and Appendix A.1.

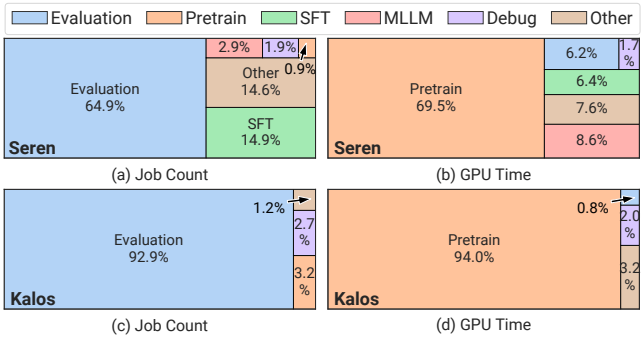


Figure 4: Distribution of different workload types in `Seren` (a, b) and `Kalos` (c, d). Note that CPU jobs are excluded. *SFT*: Supervised Fine-Tuning for model alignment. *MLLM*: Multimodal Large Language Model. *Other*: Unclassified jobs.

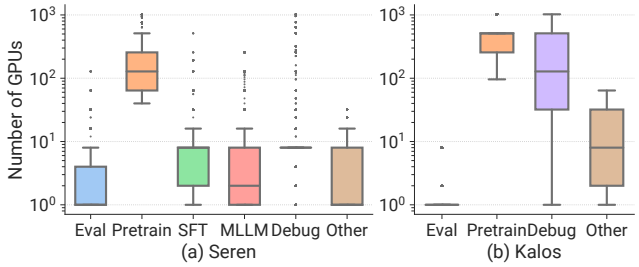


Figure 5: The boxplot of the distribution of GPU demand across different workload types in `Seren` (a) and `Kalos` (b).

Polarized GPU Utilization Figure 2 (b) shows cluster-wide GPU utilization distributions across various clusters. It is evident that the GPU utilization in our two clusters exhibits a polarized pattern, primarily concentrated in two distinct states: 0% and 100%. This polarization mainly stems from the fact that the workloads in our clusters share similar model architectures, i.e., transformer-based LLMs. In contrast, Philly and PAI encompass a broader range of utilization. Besides, when comparing the median GPU utilization, `Seren` and `Kalos` exhibit significantly higher values at 97% and 99%, respectively, in contrast to 48% and 4% observed in Philly and PAI. This observation aligns with the common understanding that LLMs are computationally intensive. It also implies that GPU-sharing-based scheduling techniques [40, 98, 99, 106] may not be suitable for LLM development. Note that 'GPU utilization' may sometimes be a weak utilization indicator [8, 94]. We provide a more precise utilization analysis in §3.3.

High-skewed Workload Distribution. We further investigate the CDF of GPU demands in relation to the number of jobs (Figure 3 (a)) and GPU time (Figure 3 (b)). For the number of jobs, all the clusters share a similar pattern in that the majority of jobs are single-GPU jobs and less than 7% of jobs request over 8 GPUs. However, when examining GPU time, single-GPU jobs only account for less than 2% resources in our two clusters, while taking over 68% GPU time in PAI. In stark contrast, large-scale jobs ($\geq$ 256 GPUs) dominated the GPU time in `Kalos`, occupying more than 96% of resources. The much steeper distribution poses substantial challenges for the design of cluster schedulers. A majority of resources

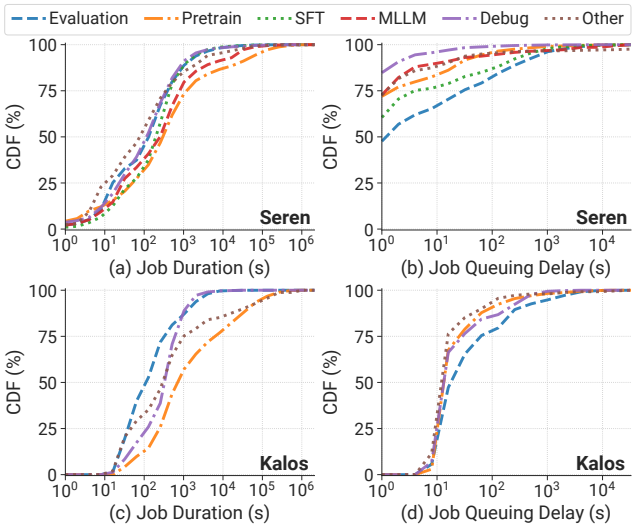


Figure 6: CDF of GPU job duration and queuing delay for different workload types in `Seren` (a, b) and `Kalos` (c, d).

are allocated to a few pretraining jobs, potentially causing head-of-line blocking issue and resulting in severe queuing delay. Existing DL cluster schedulers [35, 61, 74, 98, 101] typically depend on preemption mechanism, however, the considerable recovery overhead makes them not applicable to LLM workloads. This highlights the critical need for a scheduling system tailored for LLM clusters, considering the workload features of the entire pipeline.

3.2 Workload Categories

To strive for a deeper understanding of the characteristics of different workloads in the LLM development pipeline (§2.1), we further categorize jobs into specific types according to their production division and metadata (e.g., job names).

Irrelevance of Job Count and Resource Usage. Figure 4 presents the distribution of job counts and GPU time across various workload types, where only `Seren` contains SFT and MLLM workloads. Besides, MLLM jobs incorporate their own development pipeline (e.g., pretraining) and adopt smaller model scales for exploration purposes. Our analysis primarily focuses on LLM jobs. It is obvious that evaluation jobs constitute the majority of the total job count in both clusters, yet they consume a relatively small portion of resources (0.8% in `Kalos`). In contrast, pretraining jobs only account for 0.9% and 3.2% of the total job count but consume 69.5% and 94.0% of the total GPU time in `Seren` and `Kalos` respectively.

Job Type Correlates with GPU Demand. We further depict GPU demand distribution across various workload types in Figure 5. Each box is framed by the first and third quartiles, while the median value is indicated by the black line within the box. Both whiskers are defined at 1.5$\times$ the InterQuartile Range (IQR). Compared to evaluation jobs, which typically require less than 4 GPUs, pretraining jobs often require over 100 GPUs. This observation partially explains why evaluation jobs in `Kalos` consume only minimal resources in Figure 4(d). Additionally, we notice that debugging jobs have a wide range

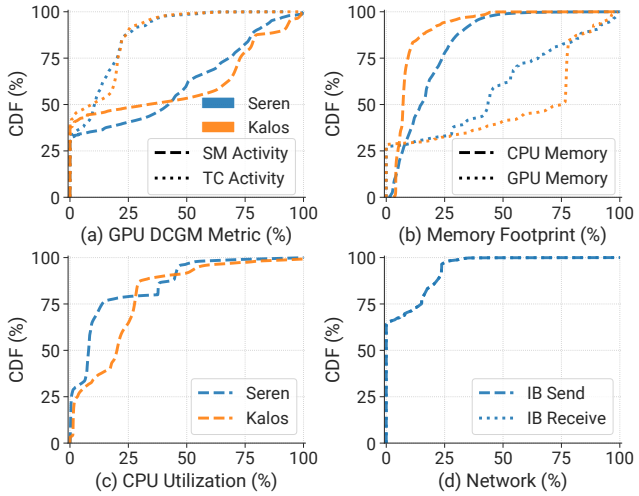


Figure 7: Infrastructure utilization. CDF of various metrics: (a) active fractions of Streaming Multiprocessor (SM) and Tensor Core (TC), (b) memory footprints of host and GPU, (c) CPU utilization, (d) normalized InfiniBand (IB) HCA send and receive bandwidths (`Seren` only). `Seren` and `Kalos` are represented by blue and orange lines respectively.

of GPU requests, which aligns with the fact that testing jobs are typically needed for various types of tasks.

Similar Temporal Distribution. Figure 6 shows the distribution of job duration and queuing delay across different workloads. In terms of job duration, although pretraining jobs have the longest duration, they surpass other workloads within an order of magnitude in the median, and less than 5% jobs last for over 1 day in both clusters. This can be attributed to frequent failures during pretraining, which will be further explored in §5. Regarding job queuing delay, contrary to previous reports [38, 45, 97] suggesting that larger-scale jobs experience longer wait times, we observe that evaluation jobs have the longest queuing delay despite having the lowest GPU demands and shortest job duration. This discrepancy is due to the majority of resources being reserved for pretraining jobs to minimize their queuing delays. Evaluation jobs are typically submitted as a batch simultaneously with lower priority, utilizing the limited spare resources.

3.3 Infrastructure

Beyond the workload characterization, we further conduct a comprehensive analysis of our infrastructure utilization.

Higher GPU Utilization. Given the critical role of GPUs in LLM development, as shown in Figure 7 (a, b), we collect fine-grained performance counter metrics from DCGM [7], including SM Activity (`PROF_SM_ACTIVE`), TC Activity (`PROF_PIPE_TENSOR_ACTIVE`), and GPU memory footprint (`DEV_FB_USED`). In contrast to PAI [97], where a significant portion of GPU memory is underutilized (less than 25% memory), our observations in `Kalos` indicate that 50% of GPUs consume over 75% of GPU memory (60 GB). Furthermore, we observe that the median SM activity in both clusters is approximately 40%, which is twice the reported 20% in PAI.

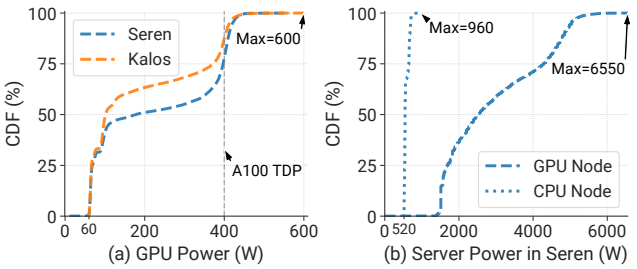


Figure 8: Power consumption. CDF of (a) A100 GPU power. (b) server power in `Seren`. *TDP*: Thermal Design Power.

Figure 9: Average power distribution of hardware modules in `Seren` GPU servers, gathered from IPMI and DCGM.

These findings align with the memory-intensive and compute-intensive natures of LLMs.

Underutilized Associated Resources. We also delve into the aspects of CPU, host memory, and network that are closely associated with LLM development. In Figure 7 (b), we compare the memory footprint on the host side and GPU side. It is evident that CPU memory utilization remains below 50%. Note that `Kalos` boasts twice the memory capacity (2TB) compared to `Seren` (Table 1). This demonstrates the significant underutilization of CPU memory. More detailed analysis is provided in Appendix A.2. Although the GPU memory offloading technique [80, 81] improves CPU memory utilization and alleviates GPU memory limitations, it also impedes training throughput due to limited PCIe bandwidth. Therefore, we do not employ the offloading mechanism. Additionally, due to a high CPU-to-GPU ratio (16 CPUs per GPU), CPUs are typically underutilized, as depicted in Figure 7 (c). Moreover, in Figure 7 (d), we measure the network send and receive bandwidths of IB in `Seren`. Two lines are well overlapped, as IB serves for symmetrical communication during LLM execution. We observe that NICs remain idle for over 60% of the time, and the active bandwidth rarely surpasses 25% of the maximum bandwidth provided by IB.

3.4 Environmental Impact

LLM development leads to substantial energy consumption and carbon emissions [72, 103]. We report our analysis of infrastructure power consumption patterns to inspire future datacenter designs that minimize environmental impact.

GPUs Dominate Power Consumption. Figure 8 (a) depicts the distribution of GPU power consumption. We observe that around 30% of GPUs are in an idle state and still need to consume 60W. Besides, due to intensive computation demand, we find that 22.1% and 12.5% of GPUs consume over 400W (TDP) in `Seren` and `Kalos` respectively, with some even reaching 600W. This may cause the risk of some metastable issues [41]. Figure 8 (b) presents the power consumption distribution of all GPU servers, along with an additional 6 CPU-

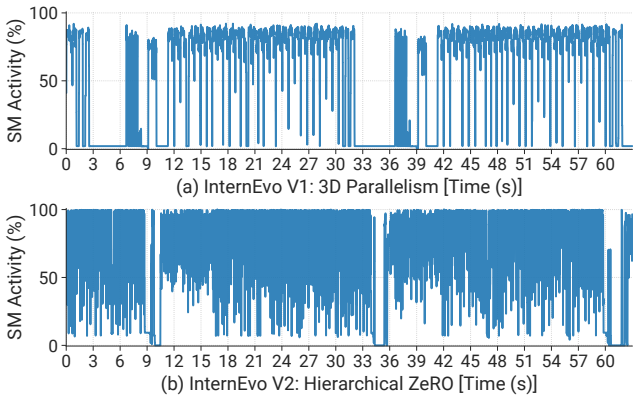


Figure 10: GPU SM utilization of pretraining a 123B LLM using different strategies of InternEvo [25] over 2048 GPUs.

only servers, in Seren. We find GPU servers consume 5× power than CPU servers on average. Additionally, Figure 9 demonstrates that GPUs account for approximately 2/3 of the total power consumption in GPU servers, while CPUs only contribute 11.2% and power supply units (PSUs) consume 9.6% of the energy during voltage conversion. These observations align with the understanding that GPUs are the primary power consumers in LLM development. We also provide the estimation of datacenter carbon emission in Appendix A.3.

4 Workload Profiling

In this section, we conduct fine-grained analyses of resource utilization for representative tasks. Specifically, we focus on pretraining and evaluation jobs, as they are the most resource-intensive or quantity-intensive workloads.

4.1 Pretraining Workload

As aforementioned, pretraining LLMs requires substantial computational resources. To enhance training efficiency, our pretraining framework, InternEvo [25], undergoes continuous refinement and iteration in its system design. As presented in Figure 10, the initial version of InternEvo (adopted by our early jobs) is denoted as (a) primarily utilizes 3D parallelism akin to that of MegatronLM [68], and (b) employs a hierarchical ZeRO mechanism [25] that implements selective redundant sharding of model states. To provide a detailed example, we profile an LLM with 123 billion parameters across 2048 GPUs. We also provide the profiling results of 1024 GPUs in Appendix A.4. For (a) 3D parallelism approach, we adopt a configuration with `pipeline parallelism`= 4, `tensor parallelism`= 8. We sample the first GPU of the first pipeline rank for profiling. For (b) hierarchical ZeRO approach, we limit parameter sharding to subgroups of 64 GPUs each and enable recomputation. We collect GPU performance counters like DCGM metrics at 1 ms intervals.

GPU SM Utilization. Figure 10 illustrates the GPU SM utilization for the same LLM under various training strategies. Both versions maintain the same global batch size and are optimized according to their respective configurations. It is evident that InternEvo V2 presents superior peak SM utilization and exhibits reduced idle periods compared to InternEvo V1, achieving around 16% acceleration. The relatively low utilization of 3D parallelism is mainly due to the impact of communication introduced by hybrid parallelism on the critical path, such as bubbles in pipeline parallelism. Note that the different inter- and intra-node communication hardware settings may lead to different optimal configurations.

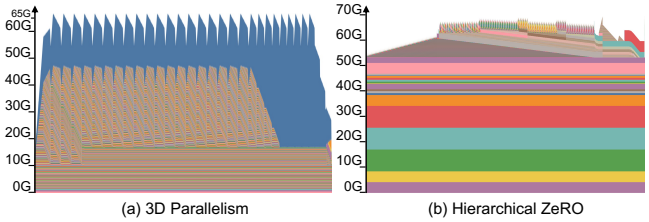


Figure 11: Memory snapshot under different pretraining strategies. Note that the extensive blue segment at the top of (a) is simplified and can be further broken down into massive fragments (memory allocations), similar to the lower part.

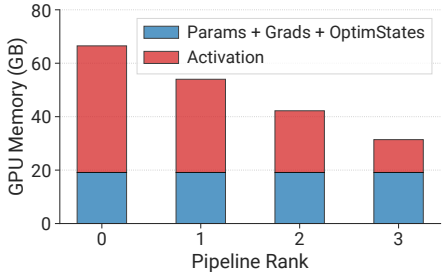


Figure 12: GPU memory consumption of different pipeline ranks employing the 1F1B [67] strategy in InternEvo V1.

GPU Memory Footprint. For a model comprising Ψ parameters, in the mainstream mixed precision training using Adam [48] optimizer, the memory footprint of the parameters, gradients, and optimizer states are 2Ψ, 2Ψ, and 12Ψ, respectively. To reduce memory cost, ZeRO [79] effectively shards redundant memory of these elements across global GPU workers. Figure 11 illustrates the actual GPU memory usage over time captured by the Pytorch memory snapshot tool [11]. The upper dynamic part represents activations and gradients, while the lower static part represents the memory occupied by parameters and optimizer states. Note that only allocated memory is depicted, while reserved memory is not presented. Our analysis reveals that, in comparison to hierarchical ZeRO, the memory requirement for activations in 3D parallelism is substantially higher. This observation underscores the importance of efficient activation memory management as a key factor for enhancing batch size and throughput in 3D parallelism.

Imbalance in Activation Sizes. When employing pipeline parallelism, each rank needs to hold a different quantity of activations since the diverse number of micro-batches pending backward computation across various pipeline ranks. Figure 12 illustrates this imbalance issue on different pipeline ranks. It suggests that we should employ a specialized partitioning mechanism to address the unbalanced memory usage among

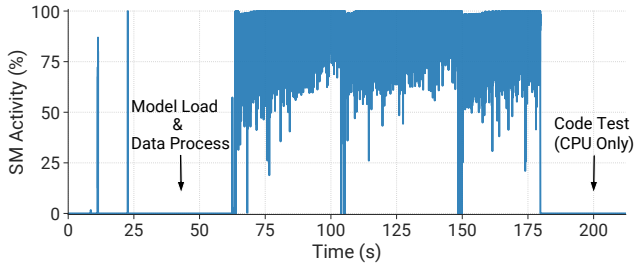


Figure 13: GPU SM utilization for the entire evaluation workload on HumanEval [24] dataset using a 7B LLM.

different ranks in pipeline parallelism, in order to achieve higher efficiency, such as recomputing activations.

4.2 Evaluation Workload

It is necessary to regularly evaluate the checkpoints produced during pretraining to guide the evolution of LLM pretraining. Therefore, the LLM evaluation jobs take the majority of jobs, each performing metric computation on different LLM benchmark datasets. We analyze the workflow of the entire evaluation and combine it with fine-grained resource usage information collection quantitatively, demonstrating two upcoming resource utilization issues. We will also discuss the corresponding solutions in §6.2.

High Model Loading and Data Preprocessing Overhead. During the initiation phase of evaluation jobs, it is imperative to load model checkpoints for each task. Additionally, the data preprocessing stage, particularly for tokenization, constitutes a significant time expenditure. These factors contribute to the underutilization of allocated GPU resources for a relatively long period. As illustrated in Figure 13, the evaluation task consumes over 1 minute prior to the actual GPU inference, accounting for 29.5% of the evaluation duration. This overhead is likely to increase with larger models or datasets. To address the preprocessing overhead, one effective strategy is to cache the tokenized data. Moreover, evaluation jobs are flexible, allowing for the consolidation of multiple evaluation tasks (datasets) into a single job. This consolidation can effectively reduce the relative time overhead of the model loading phase within the evaluation process.

High Metric Computation Overhead. The evaluation process can often involve complex and time-consuming metric computation. For example, synthesized program correctness tests need to be performed on coding datasets like HumanEval [24] and MBPP [17]. Moreover, the OpenAI GPT-4 API is invoked to assess the performance of model conversations (e.g., Chatbot Arena [112]). These procedures can take up to 30 minutes, during which the GPU remains idle. Therefore, we can observe distinct stages of GPU usage, including stages that require GPU for inference and generation, and stages that do not require GPU for metric computation and verification. Taking the HumanEval benchmark as an example, as shown in Figure 13, the GPU is idle for the last 42 seconds, wasting about 19.0% of the total GPU time.

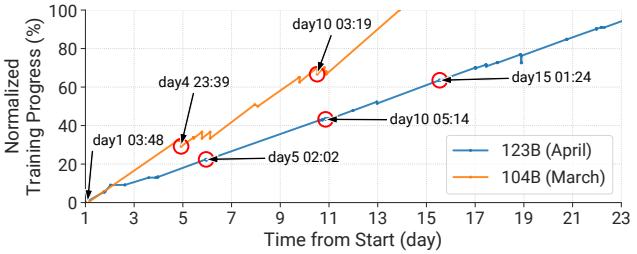


Figure 14: The training progress of two LLMs, with special emphasis on the manual recovery at night.

5 Failure Analysis

In this section, we conduct a comprehensive analysis of job failures, primarily relying on *runtime logs* and *hardware monitor data* from our two clusters. In `Kalos`, we gather logs from 32,500 tasks, which include 31,293 (96.3%) inference tasks, 647 (2.0%) pretraining tasks, and debugging tasks (1.7%). In `Seren`, we only collected logs from 675 pretraining tasks. Additionally, for pretraining tasks, we extract all pertinent information and metadata recorded in the logs, including actual training steps, cold-start overhead, recovery timestamp, etc. We hope our analysis can provide insights for future fault-tolerance research in the development of LLMs.

5.1 Failure Category

We employ a failure diagnosis system, leveraging a combination of rule-based and LLM techniques, to extract error information from the runtime logs. We provide detailed explanations of this system in §6.1. Besides, to ensure the accurate identification of the types and root causes of failures, manual checks and corrections are conducted. Table 3 provides a summary of common failures in `Acme`, including their occurrence frequency and restart time. Basically, they can be classified into three categories as follows. Note that these classifications may overlap, and the primary criterion for classifying a specific type of error is its most frequent occurrence.

- **Infrastructure.** Infrastructure-related failures arise from issues within the underlying computational platform or remote storage. These failures mainly occur midway through the job execution process, especially in pretraining tasks. They severely impact the training progress due to laborious and time-consuming recovery process.
- **Framework.** Several types of runtime errors, such as *RuntimeError*, *ValueError*, and *AttributeError*, can be associated with tensor operations, shapes, data types, or unexpected behaviors. They are often observed in the initial phases of jobs and are typically resolved by fixing the configurations.
- **Script.** Script errors typically stem from programming errors or user oversights. They constitute the majority of failures and are often addressed by revising codes.

5.2 Failure Characterization

We highlight several key observations from our analysis:
Infrastructure Failures Cause Most Severe Impact. As shown in Table 3, jobs that fail because of infrastructure is-

Category	Reason	Num	GPU Demand		Time to Failure (mins)		GPU Time (mins)		Time to Restart (mins)			Cluster
			Average	Median	Average	Median	Average	Total%	Average	Median	TR/TF%	
Infrastructure	NVLink Error	54	800	896	868.1	155.3	585683	30.25%	95.6	0.2	11.02%	S, K
	CUDA Error	21	847	1024	923.2	586.0	785099	15.77%	78.3	2.0	8.48%	S, K
	Node Failure	16	712	768	1288.8	535.8	934394	14.30%	102.8	21.5	7.98%	S
	ECC Error	12	680	512	1303.4	1192.3	958404	11.00%	2.8	1.8	0.21%	S, K
	Network Error	12	758	768	549.6	310.1	394821	4.53%	592.1	7.4	107.74%	S, K
	Connection Error	147	29	1	51.9	0.5	24492	3.44%	0.8	0.0	1.51%	S, K
	S3 Storage Error	10	422	256	2317.8	202.2	222151	2.12%	6.2	0.2	0.27%	S
	NCCL Timeout Error	6	596	512	159.7	48.1	86856	0.50%	66.7	43.6	41.78%	K
	NCCL Remote Error	3	1152	1024	50.5	22.6	52419	0.15%	0.0	0.7	0.09%	K
Framework	Dataloader Killed	6	445	508	1580.6	961.4	764170	4.38%	115.1	0.9	7.28%	K
	Attribute Error	67	228	8	67.8	1.2	60914	3.90%	2.4	0.0	3.58%	S, K
	Out of Memory Error	14	572	640	323.8	14.5	245278	3.28%	122.7	1.2	37.89%	S, K
	Runtime Error	65	441	352	66.4	3.9	27667	1.72%	10.9	1.5	16.41%	S, K
	Assertion Error	105	413	256	41.7	3.0	12315	1.24%	185.9	1.6	445.87%	S, K
	Value Error	33	387	256	9.9	3.7	5049	0.16%	27.4	0.6	276.74%	S, K
	Zero Division Error	5	499	256	14.5	15.6	5363	0.03%	2.5	1.1	17.31%	S, K
	Model Loading Error	104	8	8	2.6	2.6	20	0.00%	0.0	0.0	0.00%	K
	Dataset Loading Error	5	1	1	1.6	1.6	1	0.00%	0.0	0.0	0.00%	K
Script	File Not Found Error	568	21	1	14.2	0.4	5210	2.83%	0.4	0.0	2.58%	S, K
	OS Error	266	8	1	9.6	0.8	1098	0.28%	0.3	0.0	3.17%	S, K
	Type Error	620	18	4	0.9	0.3	97	0.06%	0.2	0.0	28.27%	S, K
	Name Error	18	247	24	3.2	0.5	947	0.02%	2.9	2.4	90.92%	S, K
	Permission Error	7	438	512	4.3	0.8	2131	0.01%	2.4	2.2	56.38%	S
	Import Error	111	93	8	1.1	0.4	74	0.01%	0.7	0.0	63.68%	S, K
	Key Error	260	7	0	3.0	1.6	55	0.01%	0.1	0.0	2.10%	S, K
	Syntax Error	10	391	384	0.7	0.6	348	0.00%	1.7	1.7	261.73%	S, K
	Argument Error	3	344	512	0.7	0.7	288	0.00%	2.7	0.7	408.47%	S
	Called Process Error	4	256	256	0.2	0.2	52	0.00%	11.7	10.9	5714.29%	S
	Index Error	23	6	1	1.6	0.9	21	0.00%	0.8	0.0	49.73%	S, K

Table 3: Job failure statistics. It is sorted based on **Total%** (i.e., the percentage of GPU time summation in different categories). **Num**: Number of Occurrence. **TF**: Time to Failure. **TR**: Time to Restart (i.e., Restart Timestamp − Failure Timestamp). **GPU Time**: TF×GPU Demand. **S/K**: Occurrence of errors in `Seren`/`Kalos` respectively.

sues often use a substantial number of GPUs (*GPU Demand*) and require considerable effort to restart (*Time to Restart*). They take over 82% GPU resources (*GPU Time*) with only 11% failed job quantity (*Num*). Most of these jobs are long-term pretraining tasks that can experience hardware failures multiple times, such as issues with GPU (e.g., *CUDAError*, *ECCError*), NVLink (*NVLinkError*), and network system (*NCCLRemoteError*, *S3StorageError*). Note that *NodeFailure* indicates uncategorized errors caused by unclear hardware issues. Addressing these infrastructure failures requires meticulous diagnostic efforts to pinpoint the source of the problems, often leading to the maintenance or replacement of defective hardware, which results in significant restart costs.

Failures Caused by High Temperature. Another noteworthy observation is that training 7B models in `Kalos` tend to result in GPU overheating, which can cause *NVLinkError* or *ECCError*. This phenomenon is largely due to the highly optimized communication cost, resulting in an exceptionally low GPU idle rate. We observe that the overall temperature in the cluster server room increased by approximately 5°C when training these models. Besides, we find most of these jobs occurred in July 2023, which is the hottest month on record [63]. This anomalous climate may be a potential cause of these failures, which is aligned with the finding recently reported by Microsoft [100]. We provide more detailed data on GPU temperature in Appendix A.5. Subsequently, our team enhanced the cooling capabilities of the cluster, leading to a significant reduction in the frequency of such failures.

Many Failures Induced by Auxiliary Services. In our pretraining framework, we connect to external components or services for metric reporting, logging, monitoring and alerting. These auxiliary services are vulnerable to network instabilities, potentially resulting in timeouts or failures that can decelerate or disrupt the training process. A significant number of *ConnectionError* and *NetworkError* incidents stem from these auxiliary services.

Evaluation Jobs Rarely Encounter Errors. In `Kalos`, only 6.7% of evaluation tasks encounter errors, and notably, there are no recorded instances of GPU or NVLink failures. The low error rate may be attributed to their short duration and the resultant decreased stress on GPUs and NVLink connections. Consequently, this diminishes the chance of hardware and operational failures that are more frequent in pretraining jobs.

5.3 Failure Recovery

There are three scenarios where we should restart a job: (1) when an error occurs within the job, (2) when there are anomalies in training metrics such as a loss spike, and (3) when the training process is stuck. A 'loss spike' refers to a sudden

increase in the loss that was previously decreasing normally, and does not recover over a certain period. Upon restarting, jobs revert to the last checkpoint, resulting in loss of training progress. Since existing LLM frameworks lack automatic recovery support, developers usually manually restart interrupted training jobs. Developers often need to be on call in turn to ensure timely completion of the pretraining model.

As shown in Figure 14, we select two pretraining jobs in the early stage (March to April) when we handle all failures manually. We extract information from the logs of two clusters' large-scale model training processes, including the runtime duration of each submission, start and end times, and the initial and final iteration numbers of training. The 104B model is an early attempt when the framework is still under development. Consequently, the process of loading previous model checkpoints led to a substantial loss in the overall training process. Conversely, in the training of the 123B model a month later, we improved the framework and adopted smaller checkpoint save intervals. Moreover, we added a feature to gracefully terminate jobs, allowing for the preservation of current training results before ending the job. It is evident that the training process of the 123B model is more stable, with fewer losses incurred due to rollbacks. However, this progress came at a cost, as jobs that were interrupted at various times had to be rapidly restarted.

6 Deployed LLM Systems

As highlighted earlier, the development process of LLMs presents significant obstacles yet unveils viable strategies for overcoming these issues. This section will introduce our efforts in two stages: (1) *Pretraining*: enhancing fault tolerance through LLM-involved failure diagnosis and automatic recovery. (2) *Evaluation*: achieving prompt performance response via task decomposition.

6.1 Fault-tolerant Pretraining

Motivation. During LLM pretraining, failures are inevitable and frequently occur due to the substantial number of GPUs involved and the extensive duration of the training process [15, 44, 88, 96]. These failures can dramatically impede the training progress and lead to severe resource inefficiency (§5). Consequently, to minimize infrastructure downtime, it is common practice to assign on-call duties to address failures manually. This places a significant burden on engineers and researchers, as expressed in the complaints raised by the Meta OPT [110] and BigScience BLOOM [1] teams. Our team also faces this problem. To alleviate this burden and enhance hardware efficiency, we develop a system that automatically detects the root causes of faults and facilitates recovery.

System Design. Our fault tolerance system is seamlessly integrated into our LLM pretraining framework. It comprises three essential modules: (1) *Checkpointing*, achieving more frequent model saving to minimize the loss of training progress; (2) *Diagnosis*, using a combination of heuristic rules and LLM to identify the root cause of different failures accurately; (3) *Recovery*, employing a holistic detection toolkit to pinpoint fault nodes and automatically restart training from the properly saved checkpoint. We delve into them in detail.

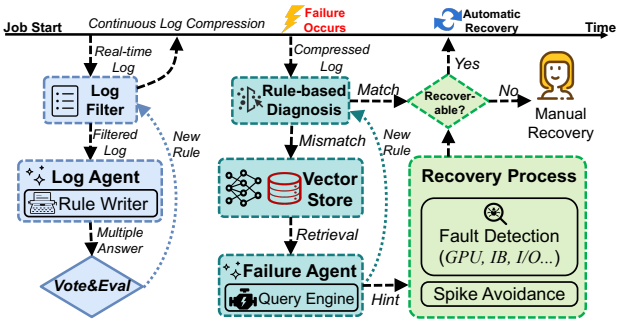


Figure 15: Workflow of failure diagnosis and model recovery.

1. Asynchronous Checkpointing. Frequent checkpointing efficiently mitigates the wasted time caused by unexpected faults [32]. However, as LLMs can produce TB-scale model states (referring to total model states across all GPUs), the process of saving checkpoints itself can introduce substantial overhead, resulting in training time slowdown up to 43% [60]. To tackle this problem, we adopt the asynchronous checkpointing strategy [64, 69], which effectively separates the checkpointing process from the training process. Our observations indicate that the CPU memory (refer to Figure 7 (b)) is capable of accommodating several checkpoints. By taking advantage of this, we can store the model state in memory and utilize a separate thread to regularly save these states to remote persistent storage. This simple strategy significantly reduces checkpointing overhead.

2. Failure Diagnosis. As we discussed in §5, failures can arise from numerous intricate factors, including errors from user script or framework, as well as issues with hardware subjected to high-stress conditions. It is crucial to determine whether a failure is recoverable for the purpose of automatic recovery. A common approach is to use a combination of heuristic rules to filter and conduct regular expression matching on the logs of faulty jobs [23, 45, 52, 53, 66]. However, this approach often proves inaccurate due to the wide-ranging diversity and complexity of error logs. There might not be a specific error statement in many cases, but multiple errors could coexist simultaneously. For example, a job might fail with messages that include *NCCLTimeoutError*, *CUDAError*, and multiple kinds of *RuntimeError*, whereas the root cause is *CUDAError*. Trying to match every error scenario with a specific rule set can become impractical.

To address this challenge, we utilize the exceptional text understanding ability and extensive knowledge base of LLMs to identify the root causes of different failures automatically. As depicted in Figure 15, we incorporate an LLM with rule-based diagnosis to achieve efficient and accurate failure diagnosis. It mainly contains the following two steps:

➤**Real-time Log Compression**. The extensive log files generated by pretraining jobs, primarily consisting of training

metric records, can reach sizes of hundreds of MBs. To accelerate diagnosing and meet the context length limit of LLMs, log compression is conducted first. The system continuously updates a collection of regular expressions, termed as *Filter Rules*. These rules efficiently remove regular log outputs, such as initialization information, training metric records, framework outputs, and debug information. A vital component of the system, the LLM-based *Log Agent*, is responsible for analyzing real-time generated log segments and identifying lines that follow fixed patterns. By doing so, the LLM-based *Log Agent* dynamically writes regular expressions to update the *Filter Rules*, effectively minimizing the size of the log files. Additionally, the *Log Agent* forwards identified error messages to subsequent modules for diagnosis.

Furthermore, we employ the self-consistency [95] approach to ensure the robustness of the *Log Agent*'s results and to guarantee the formatting of these results. This involves processing each log segment multiple times and having another LLM vote on multiple results from the *Log Agent*, ensuring the accuracy of matches through regular expressions. Over time, the *Filter Rules* become more comprehensive for the current task, making the log filtering process more efficient. Furthermore, the system can utilize metadata from tasks to identify repetitive or similar tasks, directly applying existing *Filter Rules* for log filtering, thereby avoiding redundant work. This feature is particularly beneficial in large model cluster environments, where fewer tenants and task resubmissions are common.

➤**LLM-assisted Automated Diagnosis**. The *Log Agent* efficiently compresses runtime logs, isolating critical error logs like *CUDAError*s or runtime exceptions. Though logs are already compressed upon arrival at this module, error logs may still be lengthy. We apply a two-step approach to tackle this issue. First, the error logs are compared against a rule set that has been defined over time through the diagnosis of errors from past failed jobs. If the pre-defined rules fail to diagnose the issue, the compressed log is vectorized through an embedding model and stored in a vector store, serving as a retrieval repository. Then, the *Failure Agent* intervenes. It utilizes a *Query Engine* [55] to search through the vector store. Through this search, the *Failure Agent* can identify log lines that reflect the root cause of job interruption, extract the type of error, and indicate whether the error originated from user mistakes or infrastructure failures, providing a hint for the recovery process. In addition, it also generates a mitigation suggestion for users or the operation team.

The *Failure Agent* also contributes to the continuous learning of the failure diagnosis system. For each new failure, once diagnosed and resolved, the *Failure Agent* writes a corresponding regular expression and adds it to the *Rule-based Diagnosis* module. This process is iterative and ensures that the *Failure Diagnosis* system evolves, becoming more adept at diagnosing and suggesting mitigation methods for failures. To achieve more robust performance, we currently utilize the GPT-4 [2] for diagnosis, with plans to transition to our LLMs.

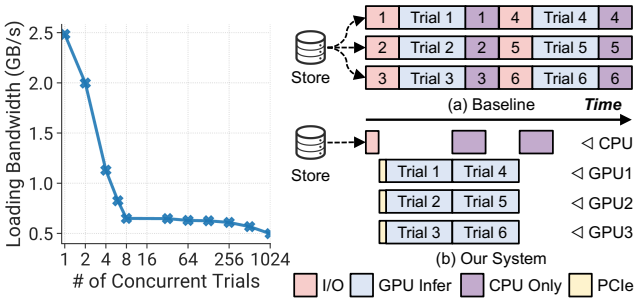


Figure 16: *Left*: Stress testing of model loading from remote storage in `Seren`. Each trial involves one GPU. *Right*: Scheduling evaluation trials. (a) Baseline: each dataset is treated as a trial. (b) Our system: decoupled scheduling.

3. Fast Fault Detection and Recovery. Based on the failure diagnosis result, if it belongs to one kind of infrastructure failure, we conduct a corresponding detection test to identify the problematic nodes. For instance, to promptly resolve the frequent *NVLinkError*, we employ a two-round NCCL test [5] approach. First, we divide all nodes into multiple two-node worlds and execute `allgather` task in each pair. If the total number of servers is odd, we leave one world size as three. If `allgather` task fails in a world, the nodes in that world are potentially faulty nodes. Then, in the second round, we pair potential faulty nodes with normal nodes to form new worlds. The nodes in each world continue to execute the `allgather` task, thus identifying the faulty nodes and then cordoning them off. On the other hand, if the failure is attributed to a sudden increase in loss (i.e., 'loss spike' [27, 110]), which is automatically triggered by our pretraining framework, we opt to an earlier healthy restart checkpoint and bypass subsequent data batches. This approach effectively maintains model quality.

System Performance. Our asynchronous checkpointing strategy offers a substantial reduction in checkpointing overheads, as the checkpointing process does not block the training process. The checkpoint time and overhead percentage of 7B and 123B size models are reduced by 3.6∼58.7× (interval=30 mins), respectively. Note that the time taken for persisting to storage is not included in asynchronous checkpointing measurement. Moreover, our failure diagnosis system significantly reduces manual intervention by around 90% and thus reduces developers' burden. Note this is not a rigorous assessment since components of our system are still under improvement.

6.2 Decoupled Scheduling for Evaluation

Motivation. Evaluating the quality of LLMs based solely on a single metric, such as training loss, may not provide an accurate assessment [58]. Therefore, it is vital to incorporate a variety of criteria and evaluate performance across an array of tasks [22]. Our LLM framework conducts regular evaluations for every checkpoint during the pretraining phase in our datacenter. This allows developers to track the progress of model training and identify the optimal model checkpoint. We aim

for swift feedback to facilitate timely adjustment. However, as shown in Figure 6, evaluation jobs experience the longest queuing delay due to limited resources and concurrent submission of numerous trials. Despite these challenges, we identify several opportunities to expedite the evaluation process.

System Design. We develop a *trial coordinator* to harmonize the operations of the cluster scheduler and LLM framework. This design incorporates the following three key techniques aimed at enhancing the efficiency of the evaluation process.

1. Decoupling Remote Model Loading. Given the substantial size of LLMs, retrieving and loading them from remote storage can be a lengthy process. Furthermore, the concurrent execution of numerous evaluation tasks (around 60 datasets) can exacerbate this loading process due to increased contention. Figure 16 (Left) shows the average model loading speed on a range of concurrent evaluation trials within `Seren`. It reveals a huge decline in loading speed when increasing the number of single-GPU trials from 1 to 8 on a single node, due to the bandwidth limitation (25Gb/s) of our storage NIC. On the other hand, the loading speed stabilizes when the number of trials ranges from 8 to 256 GPUs. This observation inspires us to take a strategic approach. Rather than submitting each evaluation dataset as a separate trial, we separated the model loading process from the evaluation process, as depicted in Figure 16 (Right). Specifically, the trial coordinator initially retrieves the available node list from the cluster scheduler and then generates a series of precursor jobs for each node. These jobs load the model from remote storage to local shared memory. Following this, the coordinator submits the evaluation jobs to the scheduler, which loads the model via the high-bandwidth PCIe. This method effectively utilizes spare host memory. After the evaluation finishes, the coordinator clears the files.

2. Decoupling Metric Computation. As shown in Figure 13, the evaluation process can often involve complex and time-consuming metric computation. For example, synthesized program correctness tests must be performed on coding datasets like HumanEval [24] and MBPP [17]. To address this issue, we decouple the metric computation process from the evaluation trial. After the model inference is performed on the GPU, its output is quickly saved into files, terminating the inference workload. Given that the outputs are typically text-based and thus small in size, this file-dumping process is swift. We then generate CPU jobs to carry out the metric computations. This approach effectively minimizes GPU idle time and accelerates the evaluation.

3. Prior-based Elastic Scheduling. In addition to the decoupling approach, we notice that our prior knowledge regarding the approximate trial runtime for each evaluation dataset is quite robust. Furthermore, these datasets are flexible, allowing us to batch multiple sets into one trial to circumvent model loading. We can also break down large datasets and decouple metric computation. As a result, the trial coordinator can maximize GPU occupancy through decomposition, balance each GPU's workload using prior information, and employ a round-robin allocation strategy on sorted job queues. Moreover, we prioritize evaluation trials with lengthy CPU metric computations in the job queue to better overlap its computation. This approach not only enhances workload balance but also minimizes trial switch overhead.

System Performance. We conducted a representative test of the trial coordinator using a typical evaluation job on a 7B size LLM, which involved evaluating the workload across 63 datasets. We measured the makespan necessary to complete all evaluation trials under two different conditions: a single node (representing limited resources) and four nodes (representing relatively ample resources). The trial coordinator can reduce the makespan by 1.3× and 1.8× respectively.

7 Discussion

Related Work. Due to the page limit, we provide a detailed discussion of our related work in Appendix B.

Scope Limitations. Despite our best efforts to analyze the workloads in `Acme`, it is an inescapable reality that we cannot cover all types of workloads. Limitations include: (1) Our analysis focuses on the developmental process preceding model serving and `Acme` does not encompass any serving jobs (i.e., workload in the deployment stage). (2) We concentrate our analysis predominantly on GPU jobs, providing limited room for CPU jobs. (3) We mainly characterize transformer-based, decoder-only architecture models (GPT-3 [20] and LLaMA 2 [92]). For newer model architecture, we provide a simple characterization of the Mixture of Experts (MoE) model [84] in Appendix A.6. Other model architectures like the Multimodal LLM [76] fall outside our scope of analysis.

Continuous System Enhancement. With the rapid advancement of large models, the systems described in this work may not suffice for the demands of future workloads. In response, we are actively refining our system to accommodate advanced training workloads, including long sequence pretraining, MoE pretraining, and efficient RLHF. Additionally, we are upgrading our infrastructure, with a particular focus on NIC, and expanding our computing cluster to facilitate larger-scale pretraining. Furthermore, we are exploring promising directions, such as improving the quality of LLMs through hyperparameter optimization using Hydro [39], and providing efficient system support for emerging model architectures like Diffusion [82] and Mamba [34].

8 Conclusion

In summary, we analyze LLM workloads and resource utilization in our datacenter `Acme`, revealing unique features and challenges of LLM development, such as resource inefficiencies and failure impacts. We also uncover potential opportunities to optimize systems tailored for LLMs and introduce our efforts for pretraining and evaluation workloads. We believe that our lessons and insights have broad applicability and can well benefit subsequent research.

Acknowledgments

We sincerely thank our shepherd, Guyue Liu, and anonymous NSDI reviewers for their valuable comments on this paper. We thank Penglong Jiao, Wenwei Zhang, Xingpu Li, and the broader InternLM team for their support throughout this project. The research is supported under the National Key R&D Program of China (2022ZD0160201) and the RIE2020 Industry Alignment Fund - Industry Collaboration Projects (IAF-ICP) Funding Initiative, as well as cash and in-kind contributions from the industry partner(s).

References

[1] Bloom. https://bigscience.huggingface.co/blog/bloom, 2024.

[2] Chatgpt. https://openai.com/blog/chatgpt, 2024.

[3] Github copilot. https://github.com/features/copilot/, 2024.

[4] Infiniband networking. https://www.nvidia.com/en-us/networking/products/infiniband/, 2024.

[5] Nccl tests. https://github.com/NVIDIA/nccl-tests, 2024.

[6] Nvidia a100 tensor core gpu. https://www.nvidia.com/en-us/data-center/a100/, 2024.

[7] Nvidia data center gpu manager. https://developer.nvidia.com/dcgm, 2024.

[8] Nvidia-smi. https://developer.nvidia.com/nvidia-system-management-interface, 2024.

[9] Nvlink and nvswitch. https://www.nvidia.com/en-us/data-center/nvlink/, 2024.

[10] Pytorch automatic mixed precision training. https://pytorch.org/docs/stable/amp, 2024.

[11] Pytorch memory snapshottool. https://pytorch.org/blog/understanding-gpu-memory-1, 2024.

[12] Supermicro ipmi. https://www.supermicro.com/en/solutions/management-software/ipmi-utilities, 2024.

[13] Tensorboard. https://www.tensorflow.org/tensorboard, 2024.

[14] Bilge Acun, Benjamin Lee, Fiodar Kazhamiaka, Kiwan Maeng, Udit Gupta, Manoj Chakkaravarthy, David Brooks, and Carole-Jean Wu. Carbon explorer: A holistic framework for designing carbon aware datacenters. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '23, 2023.

[15] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, 2022.

[16] Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiří Šimša, and Chandramohan A. Thekkath. tf.data service: A case for disaggregating ml input data processing. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '23, 2023.

[17] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, 2021.

[18] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations*, ICLR '15, 2015.

[19] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael

Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. On the opportunities and risks of foundation models. *CoRR*, 2021.

[20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, NeurIPS '20, 2020.

[21] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, 2016.

[22] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. A survey on evaluation of large language models. *CoRR*, 2023.

[23] An Ran Chen. An empirical study on leveraging logs for debugging production failures. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ICSE '19, 2019.

[24] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, 2021.

[25] Qiaoling Chen, Diandian Gu, Guoteng Wang, Xun Chen, YingTong Xiong, Ting Huang, Qinghao Hu, Xin Jin, Yonggang Wen, Tianwei Zhang, and Peng Sun. Internevo: Efficient long-sequence large language model training via hybrid parallelism and redundant sharding. *CoRR*, abs/2401.09149, 2024.

[26] Sangjin Choi, Inhoe Koo, Jeongseob Ahn, Myeongjae Jeon, and Youngjin Kwon. Envpipe: Performance-preserving dnn training framework for saving energy. In *2023 USENIX Annual Technical Conference*, USENIX ATC '23, 2023.

[27] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *CoRR*, 2022.

[28] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *CoRR*, 2023.

[29] Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Re. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems*, NeurIPS '22, 2022.

[30] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems*, NeurIPS '22, 2022.

[31] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, NAACL '19, 2019.

[32] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-N-Run: a checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '22, 2022.

[33] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '11, 2011.

[34] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *CoRR*, abs/2312.00752, 2023.

[35] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, 2019.

[36] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '15, 2015.

[37] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, ICLR '22, 2022.

[38] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, 2021.

[39] Qinghao Hu, Zhisheng Ye, Meng Zhang, Qiaoling Chen, Peng Sun, Yonggang Wen, and Tianwei Zhang. Hydro: Surrogate-Based hyperparameter tuning service in datacenters. In *17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '23, 2023.

[40] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. Lucid: A non-intrusive, scalable and interpretable scheduler for deep learning training jobs. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '23, 2023.

[41] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, 2022.

[42] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. Elastic resource sharing for distributed deep learning. In *18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '21, 2021.

[43] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers. In *Proceedings of Machine Learning and Systems*, MLSys '21, 2021.

[44] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, SOSP '23, 2023.

[45] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference*, USENIX ATC '19, 2019.

[46] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b. *CoRR*, abs/2310.06825, 2023.

[47] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. Megascale: Scaling large language model training to more than 10,000 gpus. *CoRR*, abs/2402.15627, 2024.

[48] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, ICLR '15, 2015.

[49] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding performance anomalies in RDMA subsystems. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '22, 2022.

[50] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *CoRR*, 2022.

[51] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, SOSP '23, 2023.

[52] Van-Hoang Le and Hongyu Zhang. Log parsing: How far can chatgpt go? In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, ASE '23, 2023.

[53] Van-Hoang Le and Hongyu Zhang. Log parsing with prompt-based few-shot learning. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, 2023.

[54] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[55] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, NeurIPS '20, 2020.

[56] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Lyra: Elastic scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, 2023.

[57] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '23, 2023.

[58] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher Ré, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav Santhanam, Laurel Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. Holistic evaluation of language models. *CoRR*, 2022.

[59] Kefei Liu, Zhuo Jiang, Jiao Zhang, Haoran Wei, Xiaolong Zhong, Lizhuang Tan, Tian Pan, and Tao Huang. Hostping: Diagnosing intra-host network bottlenecks in RDMA servers. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '23, 2023.

[60] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, and Carole-Jean Wu. Understanding and improving failure tolerant training for deep learning recommendation with partial recovery. In *Proceedings of Machine Learning and Systems*, MLSys '21, 2021.

[61] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, 2020.

[62] Maxime Martinasso, Grzegorz Kwasniewski, Sadaf R. Alam, Thomas C. Schulthess, and Torsten Hoefler. A pcie congestion-aware performance model for densely populated accelerator servers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016.

[63] Jackie McGuinness and Katherine Rohloff. Nasa clocks july 2023 as hottest month on record ever since 1880. https://www.nasa.gov/news-release/nasa-clocks-july-2023-as-hottest-month-on-record-ever-since-1880/, 2024.

[64] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. CheckFreq: Frequent, Fine-Grained DNN checkpointing. In *19th USENIX Conference on File and Storage Technologies*, FAST '21, 2021.

[65] Derek Gordon Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf.data: A machine learning data processing framework. *Proceedings of the VLDB Endowment*, 2021.

[66] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, 2012.

[67] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, 2019.

[68] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, 2021.

[69] Bogdan Nicolae, Jiali Li, Justin M Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, CCGRID '20, 2020.

[70] George Ostrouchov, Don Maxwell, Rizwan A. Ashraf, Christian Engelmann, Mallikarjun Shankar, and James H. Rogers. Gpu lifetimes on titan supercomputer: Survival analysis and reliability. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20, 2020.

[71] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Gray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, NeurIPS '22, 2022.

[72] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *CoRR*, 2021.

[73] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, 2018.

[74] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '21, 2021.

[75] Björn Rabenstein and Julius Volz. Prometheus: A Next-Generation monitoring system (talk). Dublin, 2015. USENIX Association.

[76] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In *Proceedings of the 38th International Conference on Machine Learning*, ICML '21, 2021.

[77] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[78] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[79] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20, 2020.

[80] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21. Association for Computing Machinery, 2021.

[81] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. In *2021*

USENIX Annual Technical Conference, USENIX ATC '21, 2021.

[82] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, CVPR '22, 2022.

[83] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *CoRR*, 2019.

[84] Noam Shazeer, *Azalia Mirhoseini, *Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations*, ICLR '17, 2017.

[85] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, 2020.

[86] Amir Taherin, Tirthak Patel, Giorgis Georgakoudis, Ignacio Laguna, and Devesh Tiwari. Examining failures and repairs on supercomputers with multi-gpu compute nodes. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '21, 2021.

[87] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. NetBouncer: Active device and link failure localization in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, 2019.

[88] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large dnns. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '23, 2023.

[89] Devesh Tiwari, Saurabh Gupta, George Gallarno, Jim Rogers, and Don Maxwell. Reliability lessons learned from gpu experience with the titan supercomputer at oak ridge leadership computing facility. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, 2015.

[90] Devesh Tiwari, Saurabh Gupta, James Rogers, Don Maxwell, Paolo Rech, Sudharshan Vazhkudai, Daniel Oliveira, Dave Londo, Nathan DeBardeleben, Philippe Navaux, Luigi Carro, and Arthur Bland. Understanding gpu errors on large-scale hpc systems and the implications for system design and operation. In *International Symposium on High Performance Computer Architecture*, HPCA '15, 2015.

[91] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *CoRR*, 2023.

[92] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, 2023.

[93] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, NeurIPS '17, 2017.

[94] Shang Wang, Peiming Yang, Yuxuan Zheng, Xin Li, and Gennady Pekhimenko. Horizontally fused training array: An effective hardware utilization squeezer for training novel deep learning models. In *Proceedings of Machine Learning and Systems*, MLSys '21, 2021.

[95] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations*, ICLR '23, 2023.

[96] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. Gem-

ini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, SOSP '23, 2023.

[97] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '22, 2022.

[98] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, 2018.

[99] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, 2020.

[100] Yifan Xiong, Yuting Jiang, Ziyue Yang, Lei Qu, Guoshuai Zhao, Shuguang Liu, Dong Zhong, Boris Pinzur, Jie Zhang, Yang Wang, Jithin Jose, Hossein Pourreza, Jeff Baxter, Kushal Datta, Prabhat Ram, Luke Melton, Joe Chau, Peng Cheng, Yongqiang Xiong, and Lidong Zhou. Anubis: Towards reliable cloud ai infrastructure via proactive validation. *CoRR*, abs/2402.06194, 2024.

[101] Zhisheng Ye, Peng Sun, Wei Gao, Tianwei Zhang, Xiaolin Wang, Shengen Yan, and Yingwei Luo. Astraea: A fair deep learning scheduler for multi-tenant gpu clusters. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[102] Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, 2003.

[103] Jie You, Jae-Won Chung, and Mosharaf Chowdhury. Zeus: Understanding and optimizing GPU energy consumption of DNN training. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '23, 2023.

[104] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, 2022.

[105] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. Profiling network performance for multi-tier data center applications. In *8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '11, 2011.

[106] Peifeng Yu and Mosharaf Chowdhury. Fine-grained gpu sharing primitives for deep learning applications. In *Proceedings of Machine Learning and Systems*, MLSys '20, 2020.

[107] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '23, 2023.

[108] Qiao Zhang, Guo Yu, Chuanxiong Guo, Yingnong Dang, Nick Swanson, Xinsheng Yang, Randolph Yao, Murali Chintalapati, Arvind Krishnamurthy, and Thomas Anderson. Deepview: Virtual disk failure diagnosis and pattern detection for azure. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, 2018.

[109] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. An empirical study on program failures of deep learning jobs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, 2020.

[110] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models. *CoRR*, 2022.

[111] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. Justitia: Software Multi-Tenancy in hardware Kernel-Bypass networks. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '22, 2022.

[112] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. *CoRR*, 2023.

[113] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, 2022.

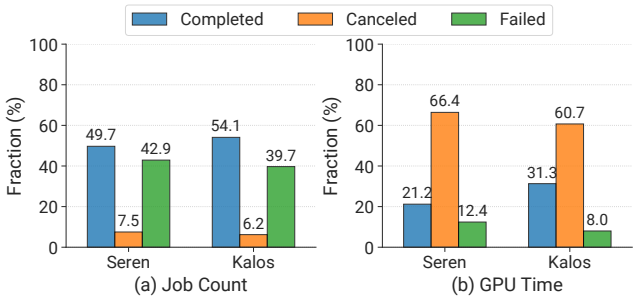


Figure 17: Final statuses of jobs in terms of (a) quantity and (b) utilized GPU resources.

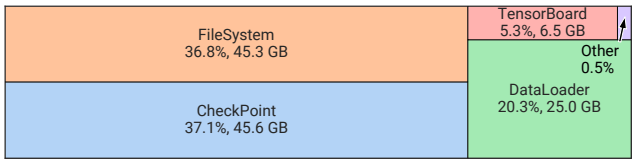


Figure 18: Distribution of host memory on a server in `Seren` during executing a pretraining job. Idle memory is not shown.

A Supplementary Characterization

In this section, we provide additional analysis to further characterize the workload features during our LLM development.

A.1 Job Final Statuses

High Incompletion Rate. Figure 17 summarizes the distribution of three key final statuses across our two clusters, revealing a similar pattern. It is obvious that only approximately 20∼30% of resources are consumed by jobs that finally complete. Besides, about 40% of jobs fail, utilizing 10% of GPU resources. This suggests that failures predominantly occur in the early stages of execution, aligning with the statistics presented in Table 3. Canceled jobs, while constituting only around 7% of the total job count, command over 60% of GPU resources. This pattern suggests a prevalence of large-scale pretraining jobs being canceled by users. Beyond the common cancellation motives cited in prior studies (e.g., achieving desired model performance sooner than expected, early recognition of poor hyperparameter configuration) [38, 73, 98], our experience with LLM pretraining has identified two additional frequent causes: (1) Users pausing jobs to adjust parameters in response to performance anomalies, such as loss spikes. (2) Jobs stalling due to infrastructure issues without throwing error messages, only to be addressed upon manual inspection by users, leading to significant resource wastage. These observations underscore the necessity for a failure-handling system that can autonomously detect and rectify faulty jobs, which is further elaborated in §5.3 and §6.1.

A.2 Host Memory

Memory Footprint Breakdown. As depicted in Figure 18, we illustrate the distribution of active physical host memory within a compute node, which utilizes 123GB of the total 1TB available. This showcases a typical pattern of CPU memory

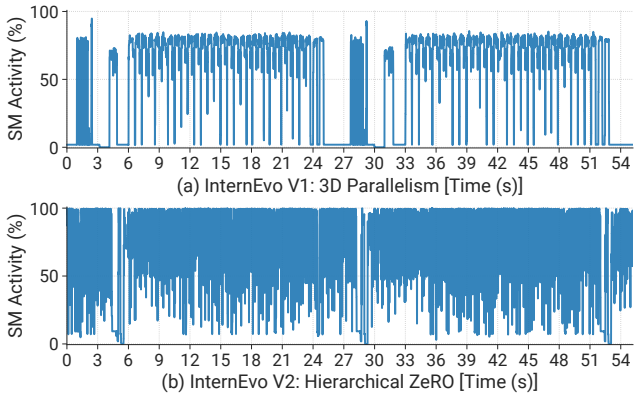


Figure 19: GPU SM utilization of pretraining a 123B LLM using different strategies of InternEvo [25] over 1024 GPUs.

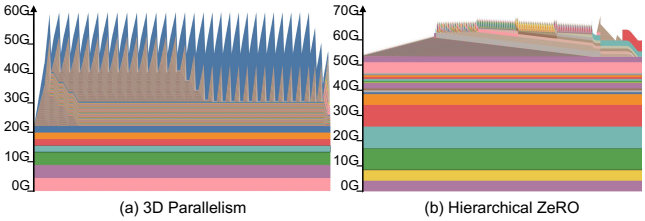


Figure 20: Memory snapshot under different pretraining strategies. Note that the extensive blue segment at the top of (a) is simplified and can be further broken down into massive fragments (memory allocations), similar to the lower part.

consumption for pretraining jobs. However, it is important to note that memory usage can significantly vary across different tasks. Specifically, the memory footprint of the dataloader can be considerably larger when employing Megatron-LM [68], which requires loading the metadata of the entire dataset. In contrast, our approach of loading data on-the-fly proves to be more memory-efficient without obviously impacting throughput. Furthermore, the memory requirements for asynchronous checkpointing (§6.1) largely depend on the model size and training configurations. The memory footprint depicted in this figure corresponds to the configuration outlined in Figure 10(a). In addition to the training processes, memory-intensive operations include TensorBoard [13] (6.5GB), the client daemon along the critical components (e.g., data and metadata caching) of the distributed file system (45.3GB). The remainder (0.6GB) encompasses Prometheus monitoring components, NVIDIA drivers, the Slurm scheduler daemon, and other system processes primarily related to sensor monitoring and system management. In general, there is a substantial amount of memory available, which can be leveraged for various purposes. Previous efforts [16, 65] have shown the potential for disaggregating CPU and memory usage from GPU allocations, suggesting that there may be additional optimization opportunities for LLMs, such as enhancing fault tolerance [96].

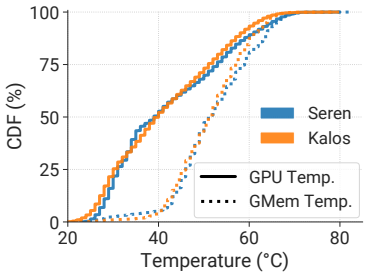


Figure 21: CDF of GPU core and GPU memory temperature.

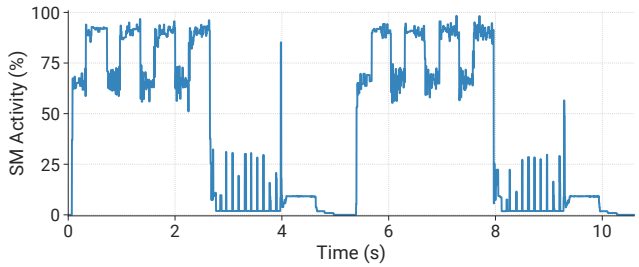


Figure 22: GPU SM utilization of pretraining a MoE model Mistral 7B [46] with 1024 GPUs in Seren.

A.3 Carbon Emission

Our datacenter Acme has a PUE (Power Usage Effectiveness) of 1.25. Moreover, it operates on approximately 30.61% of carbon-free energy (statistics for the year 2022), which includes renewable sources like solar and wind power and achieves a carbon emissions footprint (CO_2e) rate of 0.478 tCO_2e/MWh. Based on our node-level power consumption data, we calculate that Seren consumed approximately 673 MWh electricity in May 2023, which leads to total effective emissions of 321.7 tCO_2e. We believe that implementing advanced approaches like [14, 26, 103] can effectively reduce carbon emissions.

A.4 Pretraining under Different Scale

Figures 19 and 20 provide supplementary pretraining profiling results for a 123B LLM across 1024 GPUs. These figures complement the profiling observations depicted in Figures 10 and 11. It is evident that they present very similar patterns to the 2048 GPUs, demonstrating the generalizability of our characterization.

A.5 GPU Temperature

Figure 21 depicts the temperature distributions of GPU core and GPU memory. The GPU memory temperature is generally higher than the GPU core temperature. As corresponding to power consumption distribution shown in Figure 8, temperature presents a similar pattern in that some GPUs are under heavy load and have higher temperature (over 65°C). These suggest a need for enhancements in our cluster's cooling system to address the issue of elevated temperatures.

A.6 MoE Model

As shown in Figure 22, the MoE model presents much lower GPU utilization compared with the dense model shown in Figure 10. This is mainly due to the fact that the MoE model requires frequent all-to-all communication and necessitates high-quality internode communication, however, our single IB NIC server cannot efficiently handle such job. Here we directly use the official training configuration released by Mistral. On the other hand, InternEvo is still under development and we are working on performing tailored optimizations for MoE models.

B Related Work

DL Workload Characterization. Prior works conduct DL workloads analysis from different companies. Philly [45] provides insights on the impact of job locality on queuing delay and resource utilization from Microsoft, in addition to identifying different failure reasons. Helios [38] from SenseTime illustrates the nature of cluster resource utilization and user disparity, evaluated from the perspectives of the cluster, job, and user. Alibaba's PAI [97] contributes to this discourse by analyzing the challenges encountered within their clusters from both temporal and spatial viewpoints. Different from them, we focus on the characteristics of LLM workloads. Concurrently, MegaScale [47] presents ByteDance's experience in training LLMs using a formidable array of over 10,000 GPUs, complementing our focus with their practical insights.
Fault Tolerance Systems. Fault tolerance is a crucial consideration across various disciplines. In the context of LLM systems, Varuna [15], Bamboo [88], and Oobleck [44] focus on the fast recovery from failures in cloud spot instance scenarios. Gemini [96] facilitates swift recovery through CPU memory checkpointing. In addition, a body of research works dedicated to GPU-related failure analysis [33, 45, 60, 70, 86, 89, 90, 109], while several deep learning schedulers [42, 56, 107] consider fault tolerance. Furthermore, a series of studies have profiled [49, 62, 111] or diagnosed [36, 47, 59, 87, 105, 108] potential performance bottlenecks within RDMA or intra-host network communication. We provide a thorough analysis of the failure events in LLM workloads and propose a LLM-involved diagnosis system.

C Resource Links

InternLM is a project focusing on LLM research in Shanghai AI Laboratory. InternLM team keeps open-sourcing high-quality LLMs as well as a full-stack toolchain for LLM development. More resources can be accessed via the following links:

InternLM Links

Project: https://github.com/InternLM
Trace: https://github.com/InternLM/AcmeTrace
System: https://github.com/InternLM/InternEvo
Model: https://huggingface.co/internlm

QuickUpdate: a Real-Time Personalization System for Large-Scale Recommendation Models

Kiran Kumar Matam, Hani Ramezani, Fan Wang, Zeliang Chen, Yue Dong,
Maomao Ding, Zhiwei Zhao, Zhengyu Zhang, Ellie Wen, and Assaf Eisenman

Meta, Inc.

Abstract

Deep learning recommendation models play an important role in online companies and consume a major part of the AI infrastructure dedicated to training and inference. The accuracy of these models highly depends on how quickly they are published on the serving side. One of the main challenges in improving the model update latency and frequency is the model size, which has reached the order of Terabytes and is expected to further increase in the future. The large model size causes large latency (and write bandwidth) to update the model in geo-distributed servers. We present QuickUpdate, a system for real-time personalization of large-scale recommendation models, that publishes the model in high frequency as part of online training, providing serving accuracy that is comparable to that of a fully fresh model. The system employs novel techniques to minimize the required write bandwidth, including prioritized parameter updates, intermittent full model updates, model transformations, and relaxed consistency. We evaluate QuickUpdate using real-world data, on one of the largest production models in Meta. The results show that QuickUpdate provides a serving accuracy that is comparable to a fully fresh model, while reducing the average published update size and the required bandwidth by over 13x. It provides a scalable solution for serving production models in real-time fashion, which is otherwise not feasible at scale due to the limited network and storage bandwidth.

1 Introduction

Deep Learning Recommendation Models (DLRM) are widely used in many online companies. These models are trained using data at scale to learn user and product characteristics, providing personalized recommendations in a variety of contexts. For instance, Netflix [7] and YouTube [4] provide lists of movies for customers; Amazon [19] and Alibaba [20] recommend relevant products based on user search queries, and Google [3] and Meta [23] display ads and contents according to user interests. DLRMs consume a major part of the AI infrastructure in these companies. In Meta, for example, DLRMs consume more than 80% of the machine learning inference cycles [8] and more than 50% of the training cycles.

Recommendation models help the business grow. For instance, they contribute to 35% of the entire purchase in Amazon [8, 14]. As a result of such extensive business impact, accuracy becomes an important performance metrics for recommendation models at scale. In particular, Meta business required accuracy loss to be less than 0.01% in designing checkpoint and quantization algorithms [5]. This is a very narrow margin, indicating the importance of recommendation models and their accuracy.

Model freshness is a key contributor to the accuracy of personalized recommendation models [4, 6, 9, 22, 25]. The accuracy can deteriorate rapidly because the models run inferences in very highly dynamic environments. For instance, every day new users and items are registered in the system and user interests may be impacted by recent events. If the model is not updated frequently, it would not incorporate the changes in users and products, leading to a gradual deterioration in accuracy. To further highlight the impact of freshness, Figure 3 displays the significant accuracy loss when the model is not refreshed for hours. Thus, in order to keep accuracy at an acceptable level, recommendation models need to be retrained using the most recent data, and the updated models should be used to serve real time inferences.

A common technique to keep inference models fresh is online training. Instead of re-training the model from scratch each time, it is continuously trained and refined using real-time streaming data. Periodically, a snapshot of the model is created and published to hundreds of servers, located across different geographical regions. These servers then utilize the model to perform real-time predictions for online queries. However, updating the serving model incurs latency between the training cluster and the distributed serving hosts, resulting in a delay in refreshing the model, primarily due to the large size of modern models.

Over the years, model sizes have grown rapidly, reaching the scale of terabytes and containing trillions of param-

eters [5, 10, 15] to capture millions of sparse features and improve the model accuracy. The limited write bandwidth poses a challenge when transferring such large models to distributed servers and storage. As a consequence, the update latency can extend to the order of hours. This prolonged latency can adversely affect accuracy, as discussed in more detail in section 3.

To address the above challenges posed by the large model sizes and their consequent update latency, we present QuickUpdate. QuickUpdate employs the following design elements to provide real-time personalization of large-scale DLRM:

1. **Prioritized parameter update:** Performing a complete update of all parameters in each of the serving models across hundreds of geo-distributed nodes, would require substantial network and storage bandwidths, which constitute a bottleneck.
   QuickUpdate minimizes the update size by performing prioritized parameter selection. It ranks and selects specific parameters to be updated in the serving model while pruning the remaining ones from the update. This approach significantly reduces the overall update size and mitigates the bandwidth demands.
   The parameter ranking algorithm is important to avoid accuracy degradation when minimizing the update size.
2. **Intermittent full model update:** Intermittent full model updates occur when a series of consecutive partial updates are followed by a complete model update. The primary purpose of these full updates is to maintain long-term accuracy in the serving model. After each partial update, the serving model deviates from the training model, since the former still utilizes outdated parameter values. This deviation grows larger with more partial updates, leading to potential accuracy implications over time. To improve accuracy, a full model update is published intermittently to limit the gap between the serving model and the training model.
3. **Model transformations for real-time update:** QuickUpdate employs several model transformations to reduce the published model size, including inference pruning and quantization.
   Quantization has been successfully implemented in some studies [11, 24, 27] to reduce floating-point precision without sacrificing accuracy. It helps to reduce the required size of storage in the inference cluster and the required communication cost. Inference pruning is implemented on very large look-up tables. Entities, such as users or videos, and their corresponding vectors are stored as look-up embedding tables in DLRMs. The entity indices (or IDs) that are practically inactive are pruned from the serving platform to significantly reduce the size of the served model.
4. **Simplified serving design and relaxed consistency requirements:** In traditional serving designs, the model is fully loaded in the serving platform before starting to serve queries, to maintain strong consistency. In such designs, each inference request is executed based on a specific version of the model weights, ensuring consistent and reliable results. However, this approach incurs considerable infrastructure overhead due to the use of extra buffer nodes.
   In QuickUpdate, we introduce a more efficient serving design by relaxing the consistency requirements. Instead of using buffer nodes, the weights are directly updated in the serving nodes. This eliminates the need for extra infrastructure and reduces overhead. However, this relaxed design may result in some inconsistency in the embedding tables, as they may contain a mixture of fresh and stale weights.
   Despite the potential inconsistency in the embedding tables, our evaluation demonstrates that the accuracy of the serving model is not compromised, but rather it leads to accuracy gains.

We evaluate QuickUpdate using real-world data and one of the largest models that is deployed in production at Meta. Overall, our results demonstrate that QuickUpdate is able to provide a serving accuracy that is comparable to a fully fresh model, while minimizing the required write bandwidth by over 13x. It provides a scalable solution for serving production DLRM in real-time fashion, which is otherwise not feasible at scale due to the limited network and storage bandwidth. QuickUpdate achieves this by leveraging novel techniques, including selectively publishing the most important parts of each update, while still incorporating low-frequent intermittent full model updates that ensure long term accuracy.

2 Background

2.1 Deep Learning Recommendation Models (DLRM)

Typically, deep learning recommendation models are composed of sparse and dense layers, as displayed in Figure 1 [5, 10, 26]. Sparse layers are practically the embedding tables, where each embedding table represents a categorical feature and each row of this table represents a specific ID (e.g., user ID or video ID). The embedding table transforms each ID into a fixed size vector with float values that are trainable. The remaining trainable parts of the model are called dense layers.

Figure 1 displays how data flows in a DLRM. Sparse features are transformed by embedding tables; and dense features are transformed by the bottom dense layer. The transformed features are then concatenated and further transformed in the top dense layer to compute a likelihood for the input data.

2.1.1 Training DLRM

Parallelization is the main approach to train recommendation models at scale [5, 8]. Different parallelization logic can be implemented for the sparse and dense layers. Sparse layers

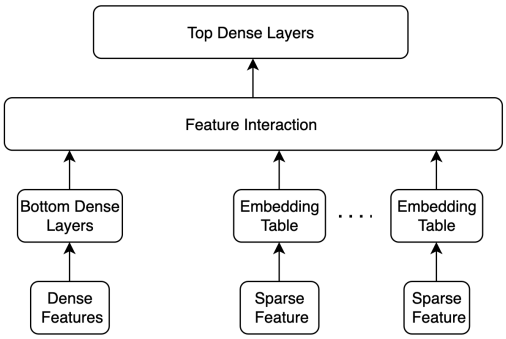


Figure 1: A DLRM architecture

contribute to $> 99\%$ of the entire model size and can be in the order of several terabytes. Because storing all the sparse layers in a single node is not feasible, a model parallelism approach is used to shard the tables on several nodes. The dense layer size, on the other hand, is small enough to be accommodated by each node, thus they are replicated across the nodes to leverage data parallelism.

At Meta, a typical training cluster includes 16 nodes, where each node contains a multi-socket CPU and 8 GPUs. Sparse layers are sharded across all the GPUs, with "all-to-all" communication during both forward and backward computations. Dense layers are replicated across all the GPUs, with "all-reduce" communication to aggregate gradients computed in multiple GPUs during the backward propagation [16]. During training, the new weights are computed and updated for sparse and dense layers synchronously to avoid accuracy degradation.

2.1.2 Serving DLRM

To efficiently serve batches of requests in a high-throughput manner, GPUs are typically used for model serving. In Meta, serving nodes are located in dedicated serving clusters. A serving node consists of a host CPU and GPUs attached to it. The serving model is replicated across the serving nodes, and data parallelization is used for model serving at scale. Ads embedding tables are stored in a single GPU because they require higher read throughput. The other embedding tables are stored in the CPU, which typically has much larger memory capacity (e.g., 1.5 TB DRAM). For storing the embedding tables, a compact data structure is used to minimize the size and store it in a GPU access friendly manner. In particular, the embedding tables are stored consecutively, and each embedding table is stored with row-major order in an array data structure.

To refresh the serving model, additional buffer nodes are utilized to avoid pausing the currently serving nodes. After the newly published model is loaded into a buffer node, request traffic is switched to the buffer node.

2.2 Online versus Offline Training

Online training is implemented when the serving model needs to be continually trained and updated using a real-time stream of data. In online training, the serving model provides predictions while being updated in regular time intervals (in the order of minutes to hours). The training continues to operate in the background (typically in a separate cluster) to fine tune the model. The rate at which the trained model is published to the serving platform can have a significant impact on the accuracy of the predictions it generates.

In contrast, offline training does not use a real-time stream of data for training and does not have a tight time constraint to train and publish the model to the serving platform. Instead, it usually uses a bulk of data that is already stored in a data storage. The model is trained using the entire available data, and the training stops when certain optimality conditions are satisfied, after which it is published to the serving platform.

Deciding between online and offline training depends on the use case. Online training systems are implemented when the model needs to be updated in a timely fashion. A typical use case could be DLRMs for ads, search and videos (e.g., [13, 18, 21]) when the environment is highly dynamic and requires the model to be updated almost in real time (in the order of minutes). The online training helps these models to incorporate the most recent data and to avoid accuracy degradation. When the business requirements do not justify real time model update, one can use an offline training approach to update models.

2.3 Optimizer State as Feature Importance Measure

In general, large DLRMs can use hundreds of thousands of features for training; however, some of these features and corresponding weights do not impact the accuracy. These features may belong to inactive users or content IDs, or some other features may not provide training signals. Maintaining all these parameters would consume some extra bandwidth when the model is published, or some extra computation and GPU storage when the inference is run in the serving platform. To mitigate these adverse impacts, we can compute feature importance and accordingly prune the features and weights that practically do not impact the accuracy.

QuickUpdate uses optimizer state (or gradient momentum) to compute feature importance. It belongs to the family of gradient-based feature importance metrics (e.g., [2, 12, 18]). The optimizer state is the historical average of gradient values and is more stable than gradient values, which sometimes oscillate between positive and negative values. The optimizer state can give us the following indications:

1) Impact on accuracy: it practically shows how often and how much a specific parameter has been updated during the course of training. A high value of gradient value shows a

high impact on improving accuracy; and if this impact persists historically, we can more confidently deduce that the parameter is important for accuracy; thus the optimizer state would be a stable indication of feature impacts on accuracy.

2) Access rate: The optimizer state of zero (or near zero) indicates that the parameter has not been practically updated during the course of training. This has twofold implications. First, it can imply that the parameter has not been accessed. This can happen for some inactive user IDs or outdated videos. Second, if the entity is active, near-zero values can imply that there may not be important signals to learn from the relevant data. For instance, it can happen for a specific user ID that is not using the platform to click on ads.

Based on the information above, QuickUpdate uses optimizer state for two different tasks: 1- to perform inference pruning when a complete model is published to reduce the size of the complete model update. In this stage, QuickUpdate focuses on low tail of optimizer state values and prunes the parameters that do not have impact on accuracy 2- to perform prioritized parameter selection when a partial update is published. In this stage, the QuickUpdate focuses on the high tail of the optimizer state values to update more important parameters.

2.4 Inference Pruning

Inference pruning is performed to reduce the size of the model when a complete snapshot is published to the serving platform. Pruning is particularly implemented on the look-up embedding tables and reduces their size by a significant amount (e.g. 50%). Since look-up tables compose more than 99% of the size of DLRMs, the pruning considerably reduces the size of the model without compromising the accuracy. Reducing the size of the model helps to consume smaller bandwidth to publish the model updates; and as a result, they can be published to hundreds of geo-distributed clusters with shorter latency. Additionally, it helps to perform inference faster because fewer number of rows are involved in the computations.

Figure 2 displays an example of a look-up table which includes indices and the corresponding row. Each index represents a unique ID which is assigned to users, videos, etc. Each row can be considered as a vector of trainable float values used by the model to generate personalized recommendations for users.

Intuitively, the inference pruning algorithm identifies the rows that represent entities that are inactive or do not provide training signals to improve accuracy. Mathematically, this is accomplished using the optimizer state vector. In particular, the trainer can provide a vector of optimizer state for each row. Each element in the optimizer state vector demonstrates the gradient momentum of the corresponding element in the row. The average of the elements in the optimizer state vector is used to quantify the row importance value. If the row importance value is close to zero, it implies that the row elements

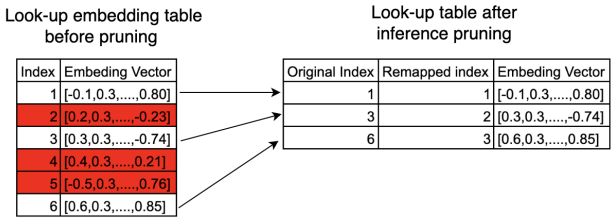


Figure 2: Inference pruning of embedding tables

have not been practically updated during the training; and as a result that row can be pruned.

Figure 2 also shows how the lookup table changes before and after pruning. Using the row importance values, the inference pruning algorithm determines the least important indices, and prunes them before publishing the complete update to servers. For operational purposes, the original index values are remapped to new indices. The new indices are simply incremented as they appear in the original table.

It should be noted that in this paper, a complete model update refers to the model snapshot after performing inference pruning.

3 Motivation

In this section, we present real-world data that highlights the motivation behind the development of QuickUpdate. First, we demonstrate how accuracy considerably drops when the model is not updated for an hour or longer. Then, we discuss the implications of scaling up the model size. Without modifying the model publishing approach, we have to either accept prolonged update latency and diminishing accuracy, or invest heavily in infrastructure to keep the update latency consistent. Finally, we highlight the limitation of lossless model updates, emphasizing the need for prioritized updates.

Accuracy gain: Updating a full serving model is a time-intensive process that may span several hours. As a result, recent updates like user actions and interests (e.g., posting new stories or engaging with specific content) would not be reflected in the serving model for several hours, potentially reducing the model accuracy.

Figure 3 demonstrates how accuracy drops as we postpone the model update for one of the large scale models at Meta. It compares the accuracy loss of a stale serving model with different model update latencies (1 to 7 hours), to a fully fresh model. It shows that accuracy loss significantly increases when model updates are delayed, reaching a loss of more than 0.6% after 7 hours.

Reducing the update size can help accelerate the model updates and improve the accuracy of the serving model.

Model size: DLRM sizes have seen a marked increase over the past years. These models utilize a large amount of data and parameters to better understand user interests and

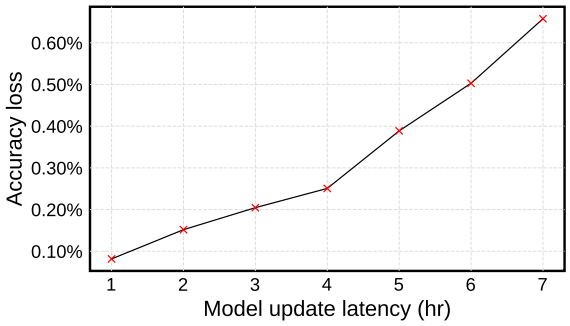


Figure 3: Accuracy loss in the case of different update latencies, when compared to a fully fresh model

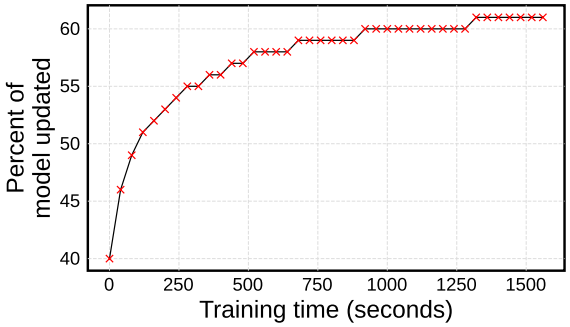


Figure 4: The percentage of the model updated over time

product characteristics, leading to improved accuracy. This progress has resulted in the development of complex models with trillions of parameters [15] and several terabyte model size [5]. Moreover, this trend is expected to continue in the future.

With the growth in model size comes the challenge of extended model update latency due to the increased bandwidth required for transfers. If left unaddressed, this is expected to lead to model freshness degradation in the future. Simply augmenting the infrastructure is not a viable long-term solution, given the relentless expansion of model sizes. Therefore, performing partial updates for large DLRM appears to be a promising strategy, aiming to reduce update latency without the demand for more infrastructure.

Lossless model updates: To better understand the proportion of the model that is modified over time, we monitored the updated embedding rows and accordingly computed the average fraction of the model touched. Figure 4 shows the percentage of the model updated over time. It is evident that a large part of the model is updated in a short span of time. For example, in just a 10-minute interval, 58% of the model gets updated. Updating 58% of the model is resource-intensive, requiring considerably more infrastructure than a full model update every hour. This leads us to explore an approach of prioritized updates to significantly reduce the update footprint.

4 System Overview

Figure 5 provides an overview of QuickUpdate architecture. The DLRM system consists of training nodes, serving nodes, and remote storage to save the model snapshots. The publishing logic of QuickUpdate has been mainly implemented in the UpdateSelector and UpdatePatcher agents, which are respectively implemented in the trainer node and serving node. The UpdateSelector is responsible to decide which portion of the model should be updated and quantize it before saving in the remote storage. UpdatePatcher implements different patching strategies depending on the type of update being performed. Additional information is provided in the following sections.

4.1 What to Update

QuickUpdate focuses on performing partial updates specifically for the embedding tables, which typically constitute the big majority of deep learning recommendation models (more than 99% in our workload). In such models, each table represents a categorical feature (e.g., users, videos), and each row within a table corresponds to a specific ID associated with that feature.

During our exploration, we considered two options for updating the embedding tables: 1. Updating all rows for selected tables. 2. Updating selected rows for all tables (the selected row indices may vary from one table to another).

We found that updating at the row level granularity resulted in improved accuracy while minimizing the overall update size. Consequently, QuickUpdate determines which specific rows within the tables need to be updated on the serving side. This approach enables QuickUpdate to prioritize the updates of certain content or user IDs that are more likely to contribute to accuracy gains, ensuring an efficient and effective update strategy. For dense layers in the model, QuickUpdate performs a full update. This is because the size of the update for these layers is relatively small, and any optimization specific to these layers would not have a significant impact on the overall update process.

4.2 UpdateSelector

The UpdateSelector component of QuickUpdate is implemented within the training cluster. This is because it requires certain model information, such as parameter values, from the trainer in order to prepare the model update.

During online training, the trainer operates in batch intervals. At the end of each training interval, the trainer shares both the model state and the optimizer state with the UpdateSelector. The model state includes sharded embedding tables and dense parameter values, while the optimizer state includes gradient values and their momentum. These states are copied from GPU memory to the host CPU memory.

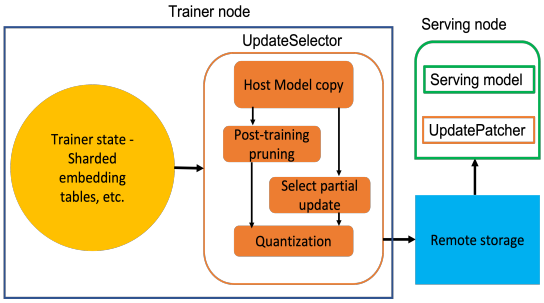


Figure 5: System architecture

UpdateSelector uses optimizer state to perform the following two tasks on the copied model in CPU:

1. Prioritized parameter selection: The primary objective of this task is to update only a small percentage of the model parameters, while minimizing any degradation in accuracy (compared to a full update). In this stage, QuickUpdate selects the embedding rows based on their optimizer state values, prioritizing those that are likely to result in a larger improvement in accuracy.
2. Inference pruning: This task is implemented when a full model is published. Inference pruning focuses on the sparse embedding tables and aims to reduce the size of the full model update. During this stage, QuickUpdate identifies the low-tail optimizer state values and prunes the embedding rows with values close to zero. These rows have negligible impact on the accuracy of the model.

Once the updates (whether full or partial) are prepared, they undergo quantization to reduce their size. The quantization is used as a compression method and has negligible impact on the model accuracy. The quantized updates are then stored in remote storage, ready to be utilized for the update process.

4.3 UpdatePatcher

UpdatePatcher is responsible for loading the published snapshots and updating the serving model. It utilizes an efficient non-atomic update approach for both partial and full model updates. In the non-atomic update process, multiple threads have access to the model parameters and gradually patch the parameters to the servers. This allows for concurrent parameter patching without the need to lock the servers or models. As a result, the servers can continue to run inference on incoming traffic simultaneously while the updates are being applied. This approach ensures efficient and uninterrupted serving of real-time traffic during the update process.

4.4 Workflow

Figure 6 demonstrates QuickUpdate workflow. For simplicity, we only show the timescale in the trainer, UpdateSelector, and a serving node. The evolution of the model is a repeatable pattern, thus we focus on one cycle, which is further divided into intervals. At the beginning of each interval i in the cycle c, UpdateSelector has access to a full model $F_{c,i}$ to determine which portion of the model should be updated. In particular, first, a full snapshot (i.e., $F_{c,1}$) will be published and loaded in the server, and then consecutive partial updates ($P_{c,i}$ for $i > 1$) will be published and patched to the full snapshot to create the serving snapshot $S_{c,i}$.

Merging more partial updates with the serving model may cause more deviation between the serving model $S_{c,i}$ and the current trainer state $F_{c,i}$. This deviation may result in accuracy degradation. As a result, another full fresh snapshot (i.e., $F_{c+1,1}$) will be published to the serving cluster, marking the end of the current cycle. This model evolution in the serving side can be represented as follows:

$S_{c,1} = F_{c,1}$

$S_{c,i} = M(S_{c,i-1}, P_{c,i})$ for $1 < i \leq I$

where I is the number of intervals in a cycle, and M is the merge operator. The merge operator simply copies the parameter values of $P_{c,i}$ and updates them in $S_{c,i-1}$.

5 Design

In this section, we discuss the design options and their impact on accuracy metrics. We begin by defining the specific accuracy metrics that guide our design and evaluation. By prioritizing accuracy throughout the design process, we aimed to create an effective system that provides high serving accuracy while addressing the network and storage bandwidth bottleneck. Note that QuickUpdate is configurable and monitored in production for the unlikely event of accuracy degradation.

5.1 Accuracy Metrics

Binary Cross Entropy or Entropy [17] is a well known aggregate metric to evaluate accuracy of ads models. In this study, we use Normalized Entropy (NE) which is defined as the Binary Cross Entropy divided by a constant. The variations of NE are computed as follows to understand how the partial update would perform with respect to the fully fresh snapshot and stale model. For simplicity, we drop cycle subscript c from the notation.

1- NE loss: It determines the accuracy reduction when model S_i is used instead of the corresponding fully fresh model F_i to run inferences.

$$NEloss(S_i) = \frac{NE_{S_i} - NE_{F_i}}{NE_{F_i}} * 100 \quad (1)$$

where NE_{S_i} and NE_{F_i} respectively denote Normalized Entropy for models S_i and F_i.

2- NE gain: It represents how much accuracy improvement to expect if S_i is used for inference instead of the stale model:

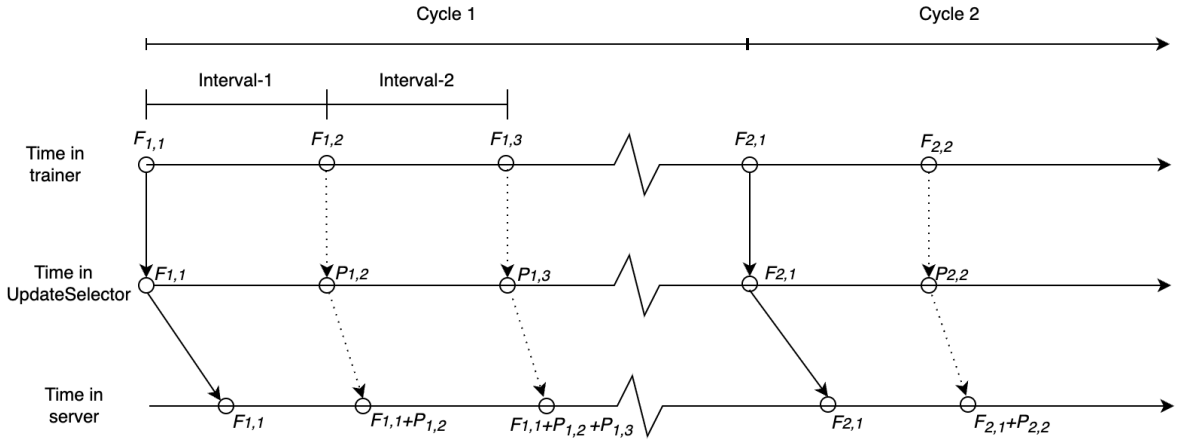


Figure 6: QuickUpdate workflow

$$NEgain(S_i) = \frac{NE_{S_i} - NE_{stale}}{NE_{stale}} * 100 \quad (2)$$

The stale model is considered to be the latest published full model F_1.

3- NE recovery: It demonstrates the percentage of maximum NE gain that has been attained by model S_i. We assume that the maximum NE gain is achieved if the fully trained model, F_i, could have been used for inference. Thus, the NE recovery is defined as:

$$NErecovery(S_i) = \frac{NEgain(S_i)}{NEgain(F_i)} * 100 \quad (3)$$

5.2 Selection Criteria

To prioritize updating rows that yield larger accuracy gains, we need a reliable indicator that remains stable throughout the training process. While the gradient vector could serve as a criterion, its oscillation between positive and negative values introduces numerical instability. Instead, we can utilize the optimizer state vector, also known as momentum, which provides a more stable measure. The optimizer state vector represents the averaged squared sum of historical gradients for a specific row. By denoting $OS^r_{c,i}$ as the optimizer state vector for a row r in the model at interval i, and $\overline{OS}^r_{c,i}$ as the average of its elements, we can leverage this measure as an indication of row importance.

Intuitively, the rows with larger values of $\overline{OS}^r_{c,i}$ would be more likely to improve accuracy. For example, these rows can represent a specific user who frequently uses the platform to click on ads, or it can represent a specific video with a high access rate. Other than the magnitude of $\overline{OS}^r_{c,i}$ for a given interval, it might be important to track how it changes over time. This can be potentially informative for situations when we prefer to prioritize parameters that have changed with respect to older versions. Based on these intuitions, we evaluate the following selection criteria:

Table 1: NE recovery for different selection criteria

Selection criteria	NE recovery
Absolute optimizer state	70%
Delta optimizer state	100%

1- Absolute optimizer state:

$$abs(\overline{OS}^r_{c,i}) \quad for \ \ i > 1 \quad (4)$$

2- Delta optimizer state:

$$abs(\overline{OS}^r_{c,i} - \overline{OS}^r_{c,i-1}) \quad for \ \ i > 1 \quad (5)$$

The choice between using absolute optimizer state or delta optimizer state as selection criteria depends on their respective advantages and trade-offs. While absolute optimizer state provides a stable and aggregate measure of how a row impacts accuracy, delta optimizer state captures the change in impact compared to the previous interval. However, using delta optimizer state requires additional memory to store the optimizer state from the previous interval. To evaluate the impact of these criteria, we conducted experiments with a 30-minute interval length and a 10% update size. After publishing a full snapshot and training for another hour, we published a partial snapshot with a 10% update size based on the two criteria. We then evaluated the serving accuracy compared to a fully fresh model. The results in Table 1 (which were consistent across multiple such experiments) indicate that delta optimizer state achieves 100% NE recovery, while absolute optimizer state achieves 70% NE recovery. This implies that selecting rows based on delta optimizer state reduces the discrepancy between the serving model and the corresponding full snapshot.

Table 2: NE recovery and NE gain for different baselines

Baseline	NE recovery
Previous full update	95.94%
Previous update	99.05%

5.3 Baseline for Delta Selection

Delta optimizer state is computed with respect to a baseline. We considered two options for choosing a baseline when computing the delta optimizer state:

1- Model state at the previous update: This option considers the model state at the time of the previous update as the baseline. It is the same as the delta optimizer definition in the previous section.

2- Model state at the previous full update: As elaborated in section 5.6, QuickUpdate also leverages intermittent full updates. In this baseline option, the model state at the time of the last intermittent full update is used as delta. The delta optimizer is defined in this case as the following:

$$abs(\overline{OS}^r_{c,i} - \overline{OS}^r_{c,1}) \quad for\ \ i > 1 \tag{6}$$

For the first option, the baseline needs to be saved at the end of every training interval, while for the second option, only one baseline is saved in the first interval of the cycle. Thus, the first option provides a more recent and fresh baseline at the cost of extra compute resources for saving it in the memory.

We examine the impact of the different baselines on the serving accuracy by conducting an experiment where each intermittent full update is followed by four partial updates. We evaluate the serving accuracy (NE recovery) compared to a fully fresh model. The results in Table 2 show that previous interval baseline would deliver 3.11% (95.94% versus 99.05%) more NE recovery. This suggests that using the model state of the previous update as a baseline can better reflect recent user interests, as it is refreshed in every update interval. In addition, the use of full updates as baseline may prioritize parameters that were important in previous intervals but are no longer contributing to accuracy. Over time, the optimizer states of these parameters may reach a plateau, yet the full update baseline may still consider them due to their large delta optimizer state values. Refreshing the baseline more frequently helps eliminate the prioritization of such parameters and instead prioritize recently changed parameters that are more likely to contribute to accuracy improvements.

5.4 Real-time Inference Pruning

Inference pruning, described in subsection 2.4, helps reduce the size of the serving model and the number of required GPUs. It practically prunes rows (or IDs) that are not active anymore or have a negligible impact on accuracy to reduce the size of embedding tables. Then, the pruned tables are stored compactly in a GPU access-friendly manner to further reduce their size in the serving cluster.

The pruning is only implemented when a full model is published to the serving cluster. For the subsequent intervals, we like partial updates to be compatible with pruned tables in full model updates. Ideally, the row IDs in partial updates should be present in the pruned table. This helps us to simply update the values of existing rows without re-structuring the tables in GPUs. However, this is not always the case. Since the training data for partial updates is different from the full model update, it is possible that some row IDs become important in partial updates while those IDs are not present in the serving-side pruned table. When such cases occur, a naive implementation would involve the insertion of the missing rows into the serving-side pruned table, which can be resource-intensive and may require reshuffling of all the embedding tables across all GPUs to ensure accessibility and efficiency (e.g., avoid memory fragmentation).

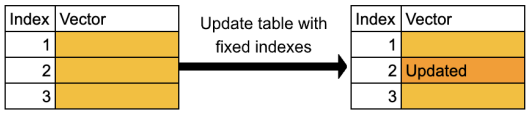
To avoid intensive reshuffling of embedding tables, we explored two alternative inference pruning strategies that are compatible with the partial updates:

1. Fixed indices pruning (see figure 7a): In this strategy, QuickUpdate performs prioritized parameter selection to choose candidate row indices for updating. However, only the rows that are already present in the embedding table are updated, while the pruned rows remain unchanged.
2. Fixed pruning ratio (see figure 7b): In this strategy, a fixed ratio of rows is pruned from the embedding table at each full update. When QuickUpdate performs prioritized parameter selection, it selects a maximum of X indices to update, where X is the total number of rows in the given table on the serving platform. This ensures that the number of rows in a table is consistent.

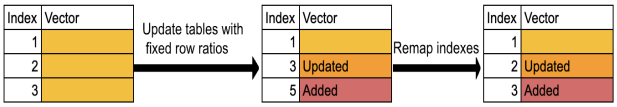
The first strategy avoids reshuffling, as there would be only row update operations and no row insert operations into the embedding tables. The second strategy avoids reshuffling by using both row update and remapping index operations. As the sizes of the embedding tables don't change in the second strategy, it would also avoid the need to reshard the embedding tables across the GPUs.

To evaluate the two pruning strategies, we consider three training scenarios: 1-No pruning, 2-fixed pruned indexes and 3-fixed pruned ratio per table.

Our experiments showed that NE loss due to pruning is practically negligible (<0.001%), with no accuracy difference between the pruning strategies. Considering implementation requirements, we opted for the fixed pruned indexes strategy due to its simpler implementation. Unlike the fixed pruned ratio strategy, it does not necessitate updating the index map with each new update.



(a) Fixed indexes pruning



(b) Fixed ratio pruning

Figure 7: Inference pruning strategies

5.5 Granularity

The granularity in QuickUpdate refers to the percentage of rows to include in a partial update. This parameter determines the amount of bandwidth required to transfer the update to the serving nodes. While a smaller granularity allows for faster and more efficient transfer, it may also limit the inclusion of important rows, potentially leading to a reduction in accuracy.

The granularity setting is configurable and can be adjusted based on various factors such as available infrastructure resources, desired accuracy, and the overall size of the model. In section 6 we evaluate the trade-off between update granularity and accuracy on our production workload.

5.6 Intermittent Full Model Update

After each partial update, the serving model deviates from the most recent trained model. It is because only a small percentage of parameters in the serving model are based on the most recent trained model, and the rest of the parameters are based on the older versions. As we perform more partial updates, the deviation becomes larger, leading to accuracy degradation. Thus, intermittent full model update is required to maintain the accuracy of the serving model and keep its long term accuracy at a desired level.

The main design parameter for the intermittent full model update is its frequency. Determining the desired frequency of full model updates depends on different factors such as granularity and the required accuracy. As we show in section 6, there exists a trade-off between the granularity of partial updates and the required frequency of full model updates. A system that applies larger granularity in partial updates can delay the need for a full model update, at the cost of higher average write bandwidth.

5.7 Relaxed Model Update Consistency

Traditionally, in order to ensure consistency during a full model update, the updated model is first loaded into buffer nodes. Only when loading is completed, the user request traffic is re-routed to the buffer nodes, which then serve as the new inference nodes. In QuickUpdate, we relaxed this design because it uses considerable infrastructure resources. Instead, whenever updates are available, they are directly patched to the serving model, which continues to serve real-time traffic concurrently without the need for a separate buffer node. This relaxation of the design allows for more efficient utilization of infrastructure resources.

The relaxed consistency in QuickUpdate is specifically related to the loading duration of parameters in the serving node. During this loading process, incoming inference requests may encounter inconsistent views of the embedding tables, leading to three possible cases for a particular query:

1. None of the parameters have been updated yet, and the inference is performed using the stale parameters.
2. Some parameters have been updated, and inference is performed using a mixture of stale and fresh parameters.
3. All parameters have been updated and inference is based on the fully fresh model.

Our experiments have shown that this relaxed consistency approach does not result in a negative impact on the serving accuracy when compared to a fully consistent policy, while also eliminating the need for additional buffer nodes. In section 6, we further evaluate the effect of these inconsistent embedding tables on the serving accuracy and demonstrate that they actually lead to positive accuracy gains, providing an additional benefit of the relaxed consistency approach.

6 Evaluation

We evaluate QuickUpdate on one of the largest recommendation models deployed in Meta, using real-world data, and trained on our production training cluster in a setup similar to [15]. The model is an extension of the DLRM model proposed in [16], but it is substantially larger, in the order of Terabytes. We used the same pre-recorded data stream for all the experiments, making the experiments reproducible and comparable, and eliminating potential result skew due to temporal data variations. The model was initially trained using several weeks of real-world data as the warm-up period, in order to reach a steady state. For accuracy evaluations, we evaluate the serving predictions on data stream that comes after the training data in time (i.e., the data evaluated during inference was not used in previous training).

6.1 Accuracy

In this section, we compare the accuracy implications of different update granularities, and derive the minimal full snapshot frequency such that the NE loss does not fall below 0.01%. In these experiments, we publish a full snapshot in the beginning and continue to publish partial updates with different granularities. These partial updates are applied on top of the full

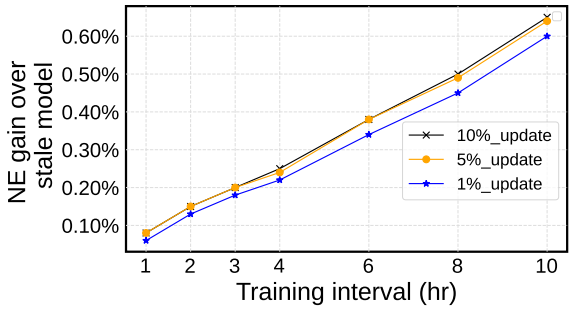


Figure 8: NE gain for partial updates with varying training periods

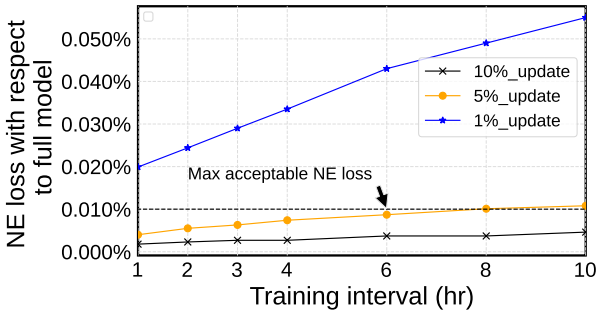


Figure 9: NE loss for partial updates with varying training periods

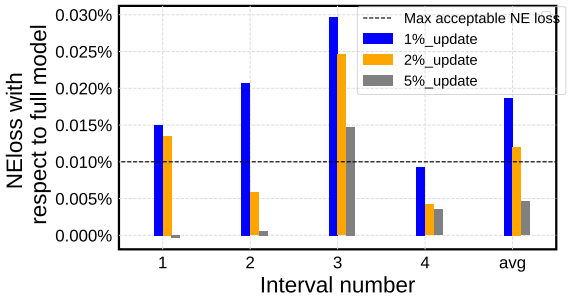


Figure 10: NE loss of partial updates with different granularities, in 4 consecutive 10 minute intervals

served snapshot and evaluated for accuracy using the same recorded dataset.

6.1.1 NE gain compared to a stale model

We begin by comparing the accuracy of QuickUpdate with the accuracy of a stale model, to quantify the accuracy gains and to validate that applying partial updates on top of a full snapshot does not negatively affect accuracy. Here, the stale model refers to the initially published full snapshot. Figure 8 shows the NE gain with respect to the stale model, for different update granularities (and without intermittent full model updates). All the update sizes result in higher NE gain than the stale model, and the NE gain increases over time. The 5% and 10% updates provide very similar NE gains, but the 1% update returns less NE gain, indicating some important rows are not included in the 1% updates. Overall, these trends indicate that even after applying partial updates for over 10 hours, there is no adverse impact and accuracy improves by 0.7% compared with the stale model.

6.1.2 NE loss compared to a fully fresh model

In this section, we investigate the NE loss of a model published by QuickUpdate using partial updates, compared to an ideal fully fresh serving mode. The results presented in figure 9 show that with 10% update, the NE loss is below 0.005% during the entire 10 hours. With 5% update, the NE loss is consistently higher than 10% update, but is still less than 0.01% during over six hours. The NE loss increases as training period increases, since the discrepancy between the serving model and the corresponding trained model increases.

The results also demonstrate the impact of employing different update granularities on the delay of full model publication, while ensuring the NE loss remains below the acceptable threshold of 0.01%. By adopting a 10% granularity, we can effectively delay the need for full model publication by over 10 hours. Similarly, when utilizing a 5% granularity, we can postpone the full model publication by 6 hours while still keeping the NE loss within the acceptable range. This highlights the efficacy of partial updates at the 5% granularity in capturing important updates and maintaining the model's accuracy and freshness over a considerable period.

6.1.3 NE loss over short time periods

In order to analyze the NE loss over short time periods, we conducted an evaluation involving four consecutive 10-minute updates. The update granularities examined were 5%, 3% and 1%. The NE loss was measured after each update in comparison to a fully fresh model using unseen data. Figure 10 shows the variance across different 10 minute intervals, emphasizing the fluctuating nature of the streaming data. However, the NE loss stabilizes when averaged over multiple short time intervals. As expected, the results show that the NE loss reduces as we increase the granularity. The average NE loss presented at the last column confirms that the 5% granularity would return an acceptable NE loss (on average) in our workload.

6.1.4 Conclusion

The accuracy results demonstrate the effectiveness of QuickUpdate in employing a 5% update granularity for up to 6 hours, while maintaining accuracy levels comparable to a fully fresh model and ensuring NE loss below the threshold of 0.01%.

Additionally, the use of QuickUpdate with 5% update granularity allows for a delay of 6 hours before the necessity of publishing a full model arises. This delay is possible due to the successful incorporation of partial updates, which capture and integrate important changes, resulting in accurate and up-to-date models.

Based on these findings, QuickUpdate triggers intermittent full model publishing every 6 hours by default, optimizing the balance between accuracy and update frequency.

6.2 Analyzing Long Term Row Convergence

In the previous analysis, our focus was on minimizing partial update granularity and determining the appropriate frequency for intermittent full updates, based on the accuracy metrics. The results indicated that utilizing partial updates with granularity of 5% for 6 hours achieved satisfactory accuracy.

In this experiment, our objective is to explore what percentage of the important rows in the model is updated by partial updates.

In order to determine a proxy of the important rows, we train the model for a duration of 6 hours (i.e., the same duration with satisfactory accuracy). We determine important rows to be the top percentage of rows in the trained model such that publishing them once at the end of the 6 hours duration, instead of the entire model, would have returned the satisfactory accuracy (i.e., below 0.01% difference compared to a fully fresh model).

Figure 11 shows the NE loss (compared with a fully fresh model) with different sizes of a single update after 6 hours of training. As can be seen, a single 5% partial update is not sufficient to achieve an NE loss below the acceptable threshold of 0.01%. However, a 10% partial update proves to be adequate in reducing the NE loss to an acceptable level. This indicates that the top 10% of the ranked embedding rows are a good proxy of the important rows in this time window.

To understand the percentage of those important rows that is covered by multiple smaller 5% updates, we ran QuickUpdate for 6 hours with multiple partial updates of 5% granularity. Upon combining all these updates into a union set, we observe that this set encompasses 70% of the important rows mentioned above, and overall covers 7.3% of all the rows in the model. Thus, a large portion of the important rows are covered by consecutive, smaller partial updates.

6.3 Bandwidth Usage

In QuickUpdate, the update size is a proxy for the bandwidth usage. The amount of bandwidth usage depends on the granularity, update interval, and frequency of the intermittent full model update. In general, these parameters are configurable and may change according to the type of DLRM and the desired accuracy. In this section, we evaluate bandwidth usage for different policies based on the percentage of the model

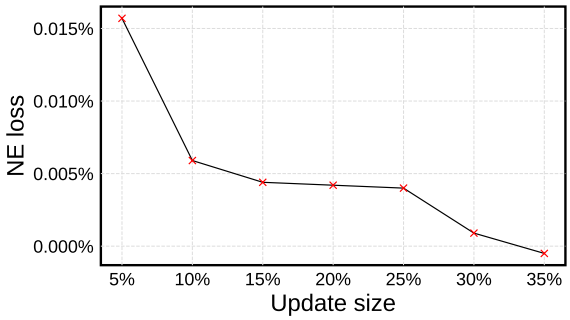


Figure 11: NE loss versus update size for a 6-hour training period

that is published. The details of them are elaborated below and in Figure 12:

1. Baseline 1: Full model is published every hour
2. Baseline 2: Full model is published every 10 minutes (not shown in the figure)
3. 5% update (default policy): A partial update is published every 10 minutes with a granularity of 5%, with intermittent full update every 6 hours (as discussed in 6.1).
4. 10% update: Similar to the previous policy, a partial update is published every 10 minutes, but with a granularity of 10%. An intermittent full update is published every 6 hours.

In order to compare these policies, we average the consumed bandwidth. The results show that the default policy of 5% update granularity with 6-hour intermittent full update interval writes on average 43.6% of the model size per hour, compared with 68.2% in policy 3 (10% update), and 100% in the baseline 1 case. Baseline 2, which provides a comparable accuracy to policies 2 and 3, would require publishing 600% of the model size per hour.

Overall, with default policy, QuickUpdate is able to reduce the consumed bandwidth by 2.3x compared to baseline 1, while providing a better accuracy that is comparable to a fully fresh model. Compared to baseline 2 (which is not feasible at scale due to the network and storage bandwidth limitation), QuickUpdate is able to reduce the required bandwidth by over 13x, while still providing comparable accuracy.

6.4 Relaxed Consistency

Traditionally, serving models are updated atomically to maintain consistent inference. This involves loading all model weights into buffer nodes, which later become the serving nodes for computing inferences. However, this approach is resource intensive due to the use of buffer nodes. To address this, QuickUpdate relaxes the consistency requirement and parameters are directly updated in the serving nodes while simultaneously performing inference queries.

We evaluate the NE recovery (compared to a fully fresh model) during an intermittent full model update in QuickUp-

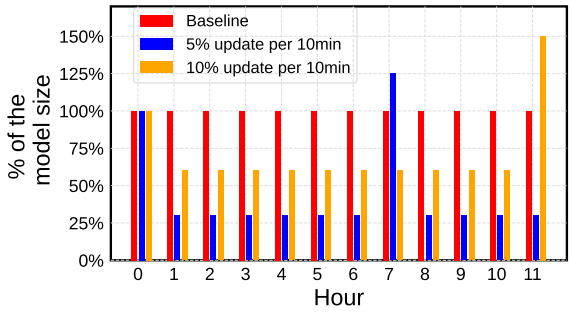


Figure 12: Bandwidth measure: update size per hour for different update scenarios

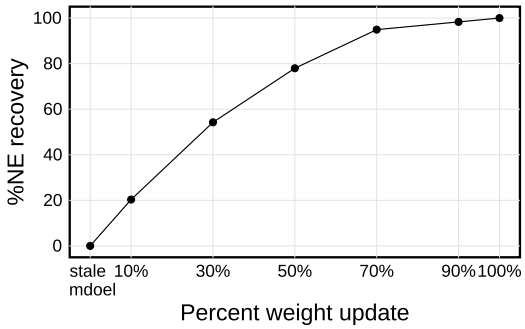


Figure 13: NE recovery during model update (relaxed consistency)

date, as a function of the percentage of the weights that have been updated. As shown in figure 13, relaxing the consistency can improve the accuracy in production during loading. The NE recovery increases as more parameters are loaded. Our data shows that we can capture about 54% of the NE recovery by patching 30% of the parameters. The NE recovery reaches around 94% after patching just 70% of the parameters.

Relaxed consistency allows for the early serving of fresh rows (rather than waiting for the entire model to update), leading to overall higher accuracy. Although there is an inconsistent view of tables during loading (implying that different rows may belong to different states), serving a subset of fresh rows already leads to in increase in accuracy. The NE recovery continues to grow over time until the entire model have been updated.

7 Related Work

Asynchronous or partial update strategy has been implemented for few real-time DLRMs [13, 18, 21]. In Kraken [21], dense parameters are updated every few seconds in a batch, while the sparse parameters are updated whenever their values change in the trainer. This is a lossless parameter update that can produce a significant amount of traffic for large models with 100-1000 billion parameters [15] and geo-distributed servers. Monolith [13] mainly focused on developing a system with collision-less embedding tables for sparse features. A sparse parameter can be updated in minute-level granularity when it is trained and its value changes from the last synchronization. Similar to Kraken, this is a lossless update which can create huge traffic. Overall, the lossless model update can be very resource intensive, as discussed in section 3. To overcome this issue, QuickUpdate can perform prioritized parameter selection that results in about $78\% - 92\%$ bandwidth reduction and a negligible accuracy loss ($< 0.01\%$). In another study, Ekko [18] is designed as an efficient system to broadcast the updates from the trainer model to all the serving inference nodes. To quickly update the larger embedding tables in the serving models, they used the sparsity and temporal locality in the embedding table updates. The Ekko system is orthogonal to QuickUpdate, and both can be implemented together. In the QuickUpdate, we optimized the design elements such as publishing interval, update granularity and parameter selection criteria to achieve the desired accuracy and minimize publishing the full model. Prioritized parameter selection is one of the techniques we used in this paper. In past studies (e.g., [1, 2, 12] Gradient based parameter selection has been explored for distributed training systems. Ekko [18] further expanded this criterion and additionally considered request frequency for each parameter and parameter freshness to the selection criteria. In QuickUpdate we decided to choose delta of gradient momentum which is a more stable measure than gradient itself, and additionally it publishes parameters such that it returns the highest accuracy compared with the baseline snapshot.

8 Conclusion

QuickUpdate is a system that enables online training to perform low-latency partial updates while providing a serving accuracy that is comparable to a fully fresh model. It offers a scalable solution for serving production-scale DLRM in real-time. This is particularly valuable because serving such models in real-time is challenging at scale due to limitations in network and storage bandwidth.

QuickUpdate achieves its scalability and accuracy goals by utilizing innovative techniques. One of these techniques involves selectively publishing the most important parts of each update, reducing the overall update size while maintaining accuracy. Additionally, QuickUpdate incorporates intermittent full model updates at a low frequency to ensure long-term accuracy. This combination of selective partial updates and intermittent full updates enables QuickUpdate to balance between low-latency serving and preserving accuracy over time.

We evaluated QuickUpdate using real world data for a large personalized ads model and showed that QuickUpdate is able to provide a serving accuracy that is comparable to a fully fresh model, while minimizing the required write bandwidth by over 13x.

References

[1] Dan Alistarh, Torsten Hoefler, Mikael Johansson, Nikola Konstantinov, Sarit Khirirat, and Cédric Renggli. The convergence of sparsified gradient methods. *Advances in Neural Information Processing Systems*, 31, 2018.

[2] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. Gradient compression supercharged high-performance data parallel dnn training. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 359–375, 2021.

[3] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10, 2016.

[4] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198, 2016.

[5] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-n-run: a checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 929–943, 2022.

[6] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 46(4):1–37, 2014.

[7] Carlos A Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)*, 6(4):1–19, 2015.

[8] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–501. IEEE, 2020.

[9] Bowei He, Xu He, Yingxue Zhang, Ruiming Tang, and Chen Ma. Dynamically expandable graph convolution for streaming recommendation. *arXiv preprint arXiv:2303.11700*, 2023.

[10] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, et al. Persia: An open, hybrid system scaling deep learning-based recommenders up to 100 trillion parameters. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3288–3298, 2022.

[11] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International conference on machine learning*, pages 2849–2858. PMLR, 2016.

[12] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J. Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018.

[13] Zhuoran Liu, Leqi Zou, Xuan Zou, Caihua Wang, Biao Zhang, Da Tang, Bolin Zhu, Yijie Zhu, Peng Wu, Ke Wang, and Youlong Cheng. Monolith: Real time recommendation system with collisionless embedding table. In *Proceedings of 5th Workshop on Online Recommender Systems and User Modeling, in conjunction with the 16th ACM Conference on Recommender Systems*, Seattle, WA, 2022.

[14] Ian MacKenzie, Chris Meyer, and Steve Noble. How retailers can keep up with consumers. *McKinsey & Company*, 18(1), 2013.

[15] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 993–1011, 2022.

[16] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[17] Pytorch. Crossentropy loss. In *https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html*.

[18] Chijun Sima, Yao Fu, Man-Kit Sit Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. Ekko: A large-scale deep learning recommender system with low-latency model update. In *16th USENIX Symposium on Operating Systems Design and Implementation.*, pages 821–839, Carlsbad, CA, 2022.

[19] Brent Smith and Greg Linden. Two decades of recommender systems at amazon. com. *Ieee internet computing*, 21(3):12–18, 2017.

[20] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 839–848, 2018.

[21] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. Kraken: Memory-efficient continual learning for large-scale real-time recommendations. 2020.

[22] Yishi Xu, Yingxue Zhang, Wei Guo, Huifeng Guo, Ruiming Tang, and Mark Coates. Graphsail: Graph structure aware incremental learning for recommender systems. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 2861–2868, 2020.

[23] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 974–983, 2018.

[24] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382, 2018.

[25] Peiyan Zhang and Sunghun Kim. A survey on incremental update for neural recommender systems. *arXiv preprint arXiv:2303.02851*, 2023.

[26] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. Aibox: Ctr prediction model training on a single node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 319–328, 2019.

[27] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs

Ziheng Jiang[1,*] Haibin Lin[1,*] Yinmin Zhong[2,*] Qi Huang[1] Yangrui Chen[1] Zhi Zhang[1]
Yanghua Peng[1] Xiang Li[1] Cong Xie[1] Shibiao Nong[1] Yulu Jia[1] Sun He[1] Hongmin Chen[1]
Zhihao Bai[1] Qi Hou[1] Shipeng Yan[1] Ding Zhou[1] Yiyao Sheng[1] Zhuo Jiang[1]
Haohan Xu[1] Haoran Wei[1] Zhang Zhang[1] Pengfei Nie[1] Leqi Zou[1] Sida Zhao[1]
Liang Xiang[1] Zherui Liu[1] Zhe Li[1] Xiaoying Jia[1] Jianxi Ye[1] Xin Jin[2,†] Xin Liu[1,†]

[1]*ByteDance* [2]*Peking University*

Abstract

We present the design, implementation and engineering experience in building and deploying MegaScale, a production system for training large language models (LLMs) at the scale of more than 10,000 GPUs. Training LLMs at this scale brings unprecedented challenges to training efficiency and stability. We take a full-stack approach that co-designs the algorithmic and system components across model block and optimizer design, computation and communication overlapping, operator optimization, data pipeline, and network performance tuning. Maintaining high efficiency throughout the training process (i.e., stability) is an important consideration in production given the long extent of LLM training jobs. Many hard stability issues only emerge at large scale, and in-depth observability is the key to address them. We develop a set of diagnosis tools to monitor system components and events deep in the stack, identify root causes, and derive effective techniques to achieve fault tolerance and mitigate stragglers. MegaScale achieves 55.2% Model FLOPs Utilization (MFU) when training a 175B LLM model on 12,288 GPUs, improving the MFU by 1.34× compared to Megatron-LM. We share our operational experience in identifying and fixing failures and stragglers. We hope by articulating the problems and sharing our experience from a systems perspective, this work can inspire future LLM systems research.

1 Introduction

Large language models (LLMs) [1] have emerged as a transformative technology in artificial intelligence (AI). Recent advancements in LLMs have significantly improved their capability. LLMs have demonstrated tremendous potential in a wide range of domains, such as machine translation, text summarization, and conversational agents [2]. As a company serving billions of users, we have been aggressively integrating AI into our products, and we are putting LLMs as a high priority to shape the future of our products.

Training LLMs is a daunting task that requires enormous computation resources. The scaling law [3] dictates that the model size and the training data size are critical factors that determine the model capability. To achieve state-of-the-art model capability, many efforts have been devoted to train large models with hundreds of billions or even trillions of parameters on hundreds of billions or even trillions of tokens. For example, GPT-3 [4] has 175 billion parameters and PaLM [5] has 540 billion parameters. Major players in this field build large-scale AI clusters with tens of thousands of GPUs to train LLMs.

Scaling LLM training to tens of thousands of GPUs brings unprecedented challenges. As AI has been at the core of many of our products, we have extensive experience in training deep neural networks (DNNs). Yet, training a model like ResNet [6] only takes tens or hundreds of GPUs. Compared to these models, the scale of training LLMs is unparallel. While we are not new to building and operating large-scale GPU clusters, these clusters are normally shared by many training jobs. Now, in the context of LLM training, a single job is occupying tens of thousands of GPUs and taking all the resources. The sheer scale of LLM training introduces two specific challenges from a systems perspective.

The first challenge is to achieve high training efficiency at scale. Model FLOPs utilization (MFU) is the ratio of the observed throughput to the theoretical maximum throughput assuming 100% of peak FLOPs [7]. It is a standard metric to evaluate training efficiency that directly translates to end-to-end training speed. LLM training is not embarrassingly parallel. To train an LLM, the model is split across GPUs and the GPUs heavily communicate with each other to make progress. Besides communication, other factors such as operator optimization, data preprocessing and GPU memory consumption also contribute significantly to MFU.

The second challenge is to achieve high training stability at scale, i.e., maintaining high training efficiency throughout

*Equal contribution.
†Correspondence to: Xin Liu <liuxin.ai@bytedance.com>, Xin Jin <xinjinpku@pku.edu.cn>.

the training process. Stability is particularly important from a production perspective, as LLMs take a long time to train. Training an LLM with one trillion tokens can take weeks. The scale and time are orders of magnitude larger than those of regular DNN training jobs. Failures and stragglers are the norm rather than the exception for LLM training. At such a scale, the consequences of failures and stragglers are devastating. Failures are very expensive, and it is critical to reduce the recovery time, given the large scale. A straggler not only affects its own work, but slows down the entire job involving tens of thousands of GPUs.

In this paper, we present the design, implementation and engineering experience of MegaScale, a production system for training LLMs at scale. MegaScale enables us to scale LLM training to more than 10,000 GPUs. We are able to harness the power of the massive number of GPUs to train LLMs with high training efficiency and stability. In building and operating MegaScale, we apply two systems principles: algorithm-system co-design and in-depth observability.

MegaScale is a specialized system tailored for LLM training. Algorithm-system co-design is a key principle to maximize performance for specialized systems, which has been applied widely in computer systems. We apply this principle to MegaScale in the context of LLM training with a full-stack approach that spans all important system components. We make several modifications and incorporate effective optimization techniques to the model architecture, including parallel transformer block [5], sliding window attention [8] and LAMB optimizer [9]. We leverage mixed parallelism strategies that combine data parallelism, pipeline parallelism, tensor parallelism, and sequence parallelism. Importantly, we design custom techniques based on the pattern of each parallelism strategy to maximize the overlapping between communication and computation. We apply prefetching and tree-based loading to optimize the data pipeline. We leverage non-blocking asynchronous operations and eliminate global barriers for large-scale collective communication group initialization. We design a custom network topology, reduce ECMP hash conflicts, customize congestion control, and tune retransmit timeout parameters for high network performance.

Stability problems including failures and stragglers in large-scale systems are notoriously hard to diagnose and fix. Many hard stability issues only emerge at large scale, which can stem from a wide range of software and hardware faults deep in the stack. Manually identifying and resolving every single issue is infeasible given the scale and complexity of the system. We apply the principle of in-depth observability to build a set of diagnosis tools. By 'in-depth observability', we mean a comprehensive monitoring and visualization strategy that penetrates beyond surface-level metrics to gather detailed, granular data across every component of the system stack, aiming to create a multidimensional view of system performance. The set of tools allows us to diagnose the system and identify root causes, by uncovering the intricate interactions and

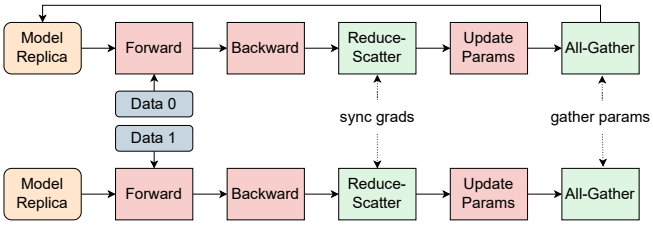


Figure 1: Data parallel training with Zero Redundancy Optimizer.

dependencies that contribute to stability issues. We develop a robust training framework to automate fault localization and recovery. We design heartbeat messages encapsulating various forms of information to facilitate real-time anomaly detection and provide early warnings. We implement a suite of diagnostic tests to identify nodes causing disruptions. We optimize the checkpointing and recovery procedure to reduce interruptions. To troubleshoot nuanced cases caused by stragglers, we develop a performance analysis tool to record fine-grained CUDA events and generate system-wide heat-map and timeline trace from a distributed view, and develop a 3D parallel training visualization tool to show data dependencies between ranks for diagnosis.

MegaScale is deployed in our datacenters to train LLMs for our products. Over the years, we have built several AI clusters with different size and hardware configurations. Our largest AI cluster has over 10,000 GPUs. In terms of training efficiency, MegaScale achieves 55.2% MFU when training a standard 175B transformer model on 12,288 GPUs, providing an improvement of 1.34× compared to the state-of-the-art open-source training framework Megatron-LM [10]. In terms of model converge and stability, we show a real production run of MegaScale that trains a proprietary model with hundreds of billions of parameters on multi-trillion tokens for several weeks. Over the weeks, the loss continues to converge, and MegaScale repairs and recovers the training process for over 100 times in presence of failures. We also share our experience in diagnosing and fixing some intriguing problems. We are working on open-sourcing components that can benefit the community on GitHub[3].

2 Background

The training of LLMs, characterized by their vast model architectures and massive datasets, is computationally intensive. Parallelism strategies distribute the training process across multiple devices.

Data parallelism. It replicates the model and optimizer states across multiple devices and the data is evenly divided among all devices. Each model replica executes the forward and backward propagation computation in parallel. Upon completion of each iteration, all model replicas synchronize to

[3]https://github.com/volcengine/veScale

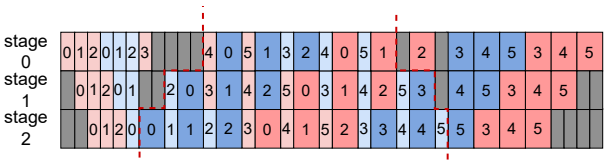


Figure 2: Interleaved 1F1B pipeline.

update the model. Instead of duplicating model states (like the optimizer states, gradients, and parameters), Zero Redundancy Optimizer (ZeRO) [11] shards these states across every data-parallel process. As a result, the traditional all-reduce operations that aggregate gradients are decomposed into separate reduce-scatter and all-gather operations. This is because every data-parallel process retains only a fraction of the total state. ZeRO is structured into three incremental stages of optimizations. Notably, the second stage is commonly adopted to shard both the optimizer states and gradients, while ensuring no additional communication overhead is introduced (Figure 1).

Pipeline parallelism. It distributes model layers among multiple devices and each device owns a portion of the model. Meanwhile, each training batch is subdivided into a number of micro-batches for pipelined execution. To reduce pipeline bubbles, various pipeline scheduling strategies are proposed, e.g., GPipe [12], PipeDream 1F1B [13], etc. Megatron-LM [7] employs the interleaved 1F1B scheduling. Each pipeline stage on every worker is subdivided into multiple virtual stages, which represents a subset of layers, referred to as a model chunk. Initially, workers enter a *warm-up* phase, executing the forward pass for a limited number of in-flight micro-batches. Following the warm-up, each worker progresses to the *steady* phase where workers perform one forward pass followed by one backward pass, often abbreviated as 1F1B. Upon concluding a batch, workers finalize the backward passes for any remaining in-flight micro-batches during this *cool-down* phase. Figure 2 shows an three-stage pipeline where each stage is further divided into two virtual stages.

Tensor parallelism. It distributes individual operators over multiple devices, with each device executing a portion of the computation in parallel. Depending on the specific partitioning strategy and its relationship to prior and subsequent operators in the model, partitioning can require communication among participating GPUs to split the input and then merge the output. For example, we can split GEMMs in the MLP and self-attention blocks among multiple GPUs to utilize more computational units. Some other operations like LayerNorm and Dropout are less computationally intensive but demand a considerable amount of activation memory. Another form of tensor parallelism called sequence parallelism is proposed to distribute these operators along the sequence dimension to effectively reduce the activation memory footprint.

Combination of parallelism strategies. These parallelism strategies can be combined into 3D parallelism to scale the training of LLMs across many GPUs [10]. Given the high communication overhead associated with tensor parallelism, it is preferable to confine such communication within a single cluster node. Conversely, data parallelism and pipeline parallelism are more amenable to inter-node communication. In this case, we choose to prioritize building the data parallelism groups over pipeline parallelism, which can mitigate cross-minipod communication for data parallelism.

3 Efficient Training at Scale

In the realm of LLMs, efficient training at scale becomes paramount. As we venture into deeper and more expansive models, the computational demands surge explosively. Handling such computation requirements without compromising on model accuracy necessitates the adoption of state-of-the-art algorithmic optimizations, communication strategies, data pipeline management, and network performance tuning techniques. This section delves deep into the methods employed to optimize the training of large models in order to achieve high training efficiency at scale.

3.1 Algorithmic Optimizations

We make a few modifications and incorporate recent optimizations at the algorithmic level to improve training efficiency, without compromising accuracy. We validate the impact of these techniques on model convergence in §6.2.

Parallel transformer block [14]. We adopt a parallel version of the transformer block in lieu of the standard serialized formulation. Specifically, the standard formula of the transformer block can be reformatted from

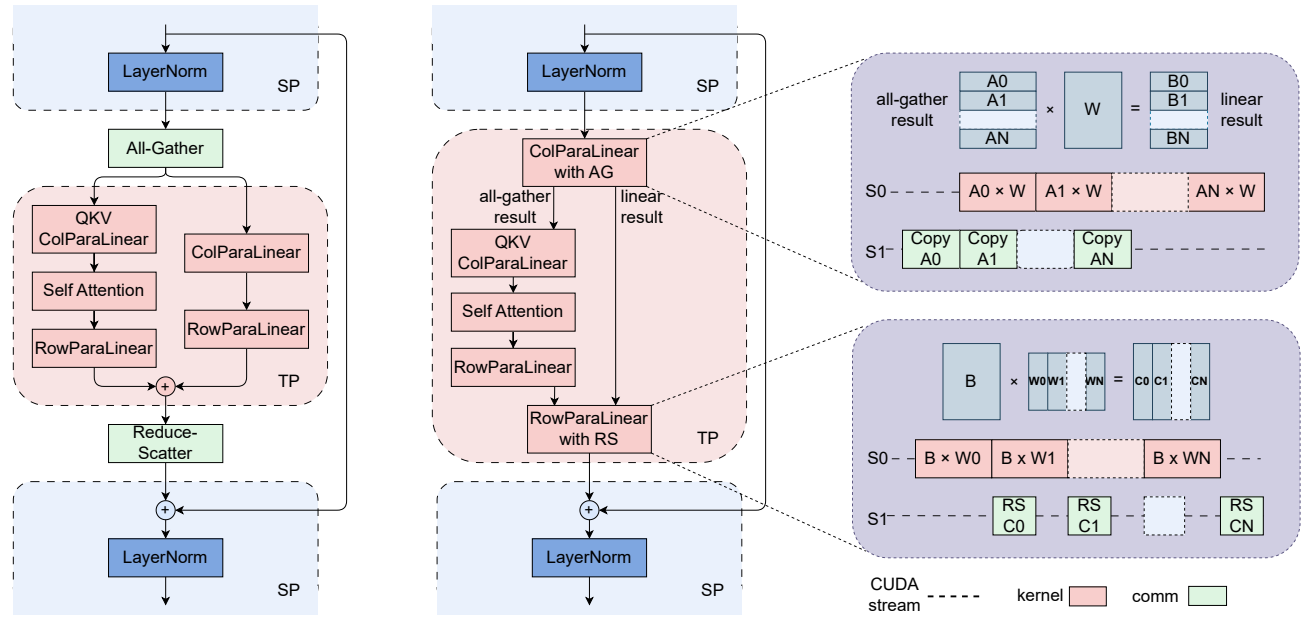
$$y = x + \text{MLP}(\text{LN}(x + \text{Attention}(\text{LN}(x)))) \quad (1)$$

into

$$y = x + \text{MLP}(\text{LN}(x)) + \text{Attention}(\text{LN}(x)) \quad (2)$$

With this approach, the computation of the attention block and the MLP block can be executed in parallel, thereby reducing the computation time. Prior work [5] shows that this modification does not degrade the quality of models with parameters in the hundreds of billions.

Sliding window attention (SWA). Sliding window attention [8] is a sparse attention mechanism that employs a fixed-size window surrounding each token in the input sequence. The computation complexity is $O(s \times w)$, where s is the input sequence length and w is the fixed window size. Sliding window attention is more efficient than the full self-attention, whose computation complexity is $O(s \times s)$, given that $w \ll s$. Past work [8] and our micro-benchmark (§6.2) have shown that the information across the entire input can be retained



(a) PTB with SP and TP (b) Fuse communication into Linears (c) Overlap communication with GEMM

Figure 3: Overlapping communication in tensor parallelism (TP) and sequence parallelism (SP) with parallel transformer block (PTB).

with a large receptive field created by stacking layers of such windowed attention. This enables faster training without compromising the accuracy.

LAMB optimizer. Efficient training at a large scale is often hindered by batch size constraints. Particularly, increasing the batch size may adversely affect model convergence. The LAMB optimizer [9] has been demonstrated to enable the scaling of BERT's training batch size to 64K without compromising accuracy. In the LLM setting, our experiments find that LAMB can scale the batch size to 4× without accuracy loss. With interleaved pipeline parallelism, the original schedule contains $\frac{4}{v}\frac{p-1}{m}$ pipeline bubbles when training four steps with 1× batch size [7], while the pipeline bubbles of training one step with 4× batch size are $\frac{1}{v}\frac{p-1}{4m}$. Hence, MegaScale reduces 87.5% of the pipeline bubbles via LAMB optimizer.

3.2 Communication Overlapping in 3D Parallelism

To reduce the iteration time, we systematically analyze the dependencies between computation and communication for all the operators in 3D parallelism, and design techniques to hide the overhead of all the off-the-critical-path operations.

Overlapping in data parallelism. As shown in Figure 1, for data parallelism, two main communication operations stand out. One is the *all-gather* operation, which fetches the most recent model parameters from workers in other data parallel ranks during the forward pass. The other is the *reduce-scatter* operation, which collect the gradients in the backward pass. In 3D parallelism, a single device may host multiple model chunks. Overlapping is implemented on a model chunk basis to maximize bandwidth utilization. The *all-gather* operation is triggered prior to the forward pass of a model chunk, and the *reduce-scatter* operation commences after its backward pass. This results in a challenge where the first *all-gather* operation and the last *reduce-scatter* operation cannot be hidden. Inspired by PyTorch FSDP [15], the initial *all-gather* operation is pre-fetched at the beginning of each iteration, allowing it to overlap with data loading operations, effectively reducing the communication time by a factor of $1/(2 * vpp_size)$. We also launch the high priority communication first to maximize overlapping. The priorities of communication operators are determined by the order of the corresponding computation operators that depend on the communication result.

Overlapping in pipeline parallelism. Pipeline parallelism features point-to-point *send/receive* communication. MegaScale uses the interleaved 1F1B scheduling method mentioned in 2. We note that in the warm-up phase, the forward pass only depends on its previous *receive*. We thus decouple the *send* and *receive*, which are often implemented together and can be blocked by the slower one. By breaking this dependency, we enable the send operation to overlap with the computation as shown in the left part of Figure 4. The cool-down phase can be viewed as the inverse of the warm-up phase, allowing for the inverse application of the same technique. As for the steady phase, both the forward and backward computation are independent of adjacent communication operations. Taking the backward as an example, as shown in the right part of Figure 4, its previous *receive* is for the next forward compu-

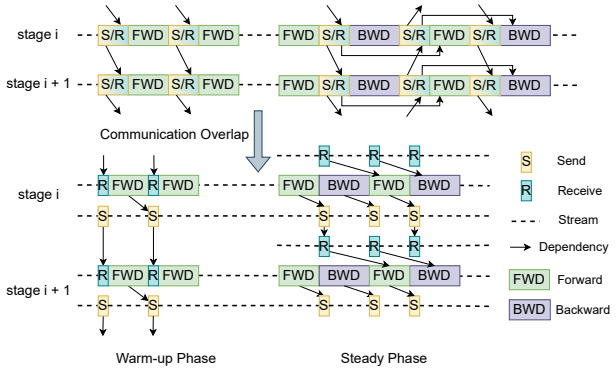


Figure 4: Overlapping communication in pipeline parallelism.

tation while the *send* is for the backward computation in the previous stage. So the *send* and *receive* operations can be launched asynchronously to overlap with the computation.

Overlapping in tensor/sequence parallelism. Tensor parallelism is commonly used to partition weights in computational-intensive operations, while operations like *LayerNorm* and *Dropout* are partitioned along the sequence dimension to save GPU memory. This necessitates *all-gather* and *reduce-scatter* operations for input collection and output redistribution across GPUs. Figure 3a shows this communication pattern in the parallel transformer block architecture. Here the two communication operators are in the critical path. To eliminate this overhead, we choose to fuse *all-gather* and *reduce-scatter* with the parallel *Linears* on the FFN path (Figure 3b). Since the GEMM kernels on the FFN path is larger, the communication can be hidden better. We break the GEMM kernel into small chunks, and pipeline the execution with the communication (Figure 3c). This strategy can be applied in the backward pass similarly.

3.3 Efficient Operators

Despite the optimization for GEMM operators in Megatron-LM, we identify opportunities for further enhancement in other operators. For the attention part, we adopt FlashAttention-2 [16], which improves work partitioning between different thread blocks and warps. For LayerNorm and GeLU, we observe that they are composed of fine-grained kernels in previous implementations. By fusing these kernels together, we reduce the overhead associated with launching multiple kernels and aid in optimizing memory access patterns, thereby achieving better performance.

3.4 Data Pipeline

Data preprocessing and loading are often overlooked. However, these operations create non-negligible GPU idle time at the beginning of each training step. Optimizing these operations are essential for efficiency of the training process.

Asynchronous data preprocessing. Data preprocessing is not on the critical path. As a result, while the GPU workers are synchronizing gradients at the end of each training step, the data preprocessing for the subsequent step can start, which hides the preprocessing overhead.

Redundant dataloader elimination. In a typical data loading phase of distributed training, each GPU worker is equipped with its own data loader, responsible for reading training data into the CPU memory before forwarding it to the GPU. This leads to competition among workers for disk read bandwidth, thereby creating a bottleneck. Notably, we observe that in the LLM training setting, GPU workers within the same machine are in the same tensor parallel group. Consequently, their inputs for each iteration are inherently identical. Based on this observation, we adopt a two-layer tree-based approach. We use a single, dedicated data loader on each machine to read the training data into a piece of shared memory. Subsequently, each GPU worker is responsible for copying the necessary data to its own GPU memory. This eliminates redundant reads and significantly enhances the efficiency of data transfer.

3.5 Collective Communication Group Initialization

In distributed training, the initialization phase involves the establishment of NVIDIA Collective Communications Library (NCCL) communication groups among GPU workers. Since this overhead is relatively negligible in small-scale scenarios, `torch.distributed` is used by default. As the number of GPUs scales to over ten thousand, the overhead introduced by naive implementations becomes intolerable. We conduct experiments on the same AI cluster in §6 and our empirical measurement indicates that the initialization time for Megatron-LM on 2,048 NVIDIA Ampere GPUs is approximately 1047 seconds. While this may appear relatively small compared to the training duration, it imposes a significant hurdle to routine testing and iterative development (e.g., minor code adjustments in hyperparameter tuning and debugging). It also hampers the implementation of fast restart-and-recovery mechanisms.

To address this issue, we perform a detailed profiling of `torch.distributed` [17] and identify two primary causes of excessive initialization time. The first issue resides in the synchronization step, where each process is involved in a barrier operation at the end of initialization a specific communication group. This barrier uses TCPStore, an inner distributed Key-Value Store implementation in Pytorch which operates in a single-threaded, blocking read-write manner. We replace TCPStore with Redis, which is non-blocking and asynchronous. This reduces the initialization time to 361 seconds on 2,048 GPUs. The second issue is related to the incautious usage of global barriers. Each process executes a global barrier after initializing its corresponding communication group. We care-

fully design the order in which communication groups are initialized to minimize the need for global barriers. This approach lowers the time complexity of the global barrier from $O(n^2)$ to $O(n)$. The initialization time is reduced to under 5 seconds on 2048 GPUs, and to under 30 seconds on more than 10,000 GPUs with those optimizations.

3.6 Network Performance Tuning

We analyze the traffic across machines in 3D parallelism and design techniques to improve network performance.

Network topology. Our datacenter network is built with high-performance switches based on Broadcom Tomahawk 4 chips. The total bandwidth of each Tomahawk chip is 25.6Tbps with 64×400Gbps ports. Three layers of switches are connected in a CLOS-like topology to connect more than 10,000 GPUs. For switches at each layer, the bandwidth percentage between downlink and uplink is 1:1. That is, 32 ports are used as downlink and 32 ports are used as uplink. The network provides high bandwidth with a small diameter. Every node can communicate with other nodes within a limited number of hops.

Reducing ECMP hashing conflicts. We carefully design the network topology and schedule network traffic to reduce ECMP hashing conflicts. First, at the top-of-rack (ToR) switch level, one 400G downlink port is split into two 200G downlink ports with specific AOC cables. The conflict probability is reduced as the bandwidth of each uplink is double of that of a downlink. Second, eight 200G NICs on the server is connected to eight different switches in a multi-rail way. The number of GPU servers connected by the same sets of ToR switches can reach 64. And we strategically schedule the data-intensive nodes from our training tasks to operate under the same Top of Rack (ToR) switch. This approach significantly reduces the number of switch hops required for communication and further reduce ECMP hashing conflicts probability.

Congestion control. In distributed training, all-to-all communication may lead to congestion and elevated levels of Priority Flow Control (PFC) [18] when employing the default DCQCN [19] protocol at scale. Excessive use of PFC can result in head-of-line (HoL) blocking [19], thereby diminishing network throughput. To mitigate these issues, we have developed an algorithm incorporating principles from both Swift [20] and DCQCN, which integrates the precise measurement of Round-Trip Time (RTT) with the rapid congestion response capabilities of Explicit Congestion Notification (ECN). This approach significantly enhances throughput and minimizes congestion related to PFC.

Retransmit timeout setting. Parameters in NCCL can be set to control retransmit timer and retry count. We tune these parameters for fast recovery under link flapping. To further reduce the recover time, we enable the adap_retrans feature on the NIC. This feature enables retransmission in a shorter interval and help recover the transmission more quickly when the link flapping period is short.

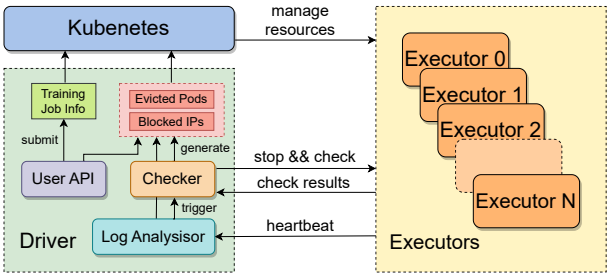


Figure 5: Robust training workflow.

4 Fault Tolerance

As the training cluster scales to over tens of thousands of GPUs, software and hardware faults become virtually inevitable. We introduce a robust training framework for LLM training that achieves automatic fault identification and fast recovery, enabling fault tolerance with minimal human intervention and negligible impact on ongoing training tasks.

4.1 Robust Training Workflow

As Figure 5 shows, upon receiving a submitted training task, the driver process interfaces with a custom Kubernetes to allocate computing resources and initiate the corresponding Pod for each executor. One executor manage one node. Once the executor has completed a series of initialization tasks, it creates the training process on each GPU and a robust training daemon which sends heartbeat to the driver periodically. These heartbeats encapsulate various forms of information to enable real-time anomaly detection and issue early warnings (§4.2). When the driver process detects an abnormal status in a particular training process, or fails to receive a heartbeat from an executor within a predefined time window, it triggers the fault recovery procedure. The driver will suspend the ongoing training task across all executors and command them to run a series of self-check diagnostics (§4.3). These diagnostic tests are carefully designed to be lightweight yet comprehensive, covering the majority of common hardware and software faults. Once the problematic nodes are identified, the driver submits the IP addresses of the nodes to be blocked, along with the information of the Pods running on them, to Kubernetes, which evicts the faulty nodes and replenishes the cluster with an equivalent amount of healthy ones which pass our diagnostic tests. Additionally, we provide a user interface that allows for manual eviction of nodes, particularly for those identified through manual analysis as in §5. After the recovery process is complete, the driver resumes training from the latest checkpoint. We optimize the checkpoint and resume process to minimize the loss of training progress (§4.4).

4.2 Data Collection and Analysis

The heartbeat messages includes the basic information of the executor, such as the IP address, the Pod name, and hardware information, etc. Additionally, the current status of the training processes is reported, enabling the driver to promptly detect any explicit anomalies. The *stdout/stderr* logs of training processes are also included. They will be aggregated, filtered and analyzed on the fly. If specific warning or error keywords are detected, the driver will report real-time diagnostic information. Moreover, RDMA traffic metrics are also included, serving as an indicator for network utilization and efficiency. Some anomalies in the training process may not manifest as explicit errors, giving the appearance that training is proceeding as expected. In such cases, RDMA traffic metrics serve as a critical indicator. Given the periodic nature of the training tasks, the network traffic characteristics for each step should exhibit similar patterns. Therefore, any significant decline or abnormal fluctuation in RDMA traffic is a signal of potential anomalies. Upon detecting such irregularities, the driver will issue alerts for manual investigation. If the traffic ceases entirely, the driver will automatically initiate the fault recovery procedure.

In order to enhance the monitoring of training stability and performance, we have developed a monitoring system with precision reaching the millisecond level. Different levels of monitoring are employed to track various indicators. Second-level monitoring is typically used to assess the overall health status and to rule out common configuration impacts on training. For instance, ECN/PFC/QoS configurations, link flapping, or any other issues of NICs. Millisecond-level monitoring, on the other hand, is used to determine if the network is congested and whether the data transfer speed of data parallelism and pipe parallelism has reached its physical limit.

4.3 Diagnostic Tests

There exists a trade-off between execution time and accuracy in self-check diagnostics. Extended diagnostic duration can adversely affect the effective training time, while high false positive rates can lead to unnecessary exclusion of machines that are actually functional. Through iterative experimentation and optimization, we have deployed a suite of lightweight diagnostic tests that effectively cover a broad spectrum of hardware and software faults encountered during actual training processes.

Intra-host network tests. To diagnose potential bottlenecks in intra-host network, we use our internally developed tool to test two things. The *Loopback* test measures the loopback bandwidth from all RDMA NICs (RNICs) to various intra-host endpoints, including memory nodes and GPUs. It conducts a full-mesh test within the host, covering all possible link combinations. This allows us to infer link-specific bandwidth degradation and irregularities in PCIe configurations based on end-to-end bandwidth results. The second *RNIC-to-RNIC* test examines the connectivity and bandwidth performance between different RNICs on the same host. These tests provide insights into whether the RNICs meet the hardware speed specifications and whether the underlying routing configurations are correctly configured.

NCCL tests. To identify potential faults in GPU communication, we run an *all-to-all* test among the GPUs within a single node to observe whether the bandwidth aligns with expected benchmarks. Once intra-host communication test is passed, each node also conducts an *all-reduce* test with neighboring machines under the same ToR switch to assess inter-node GPU communication.

4.4 Fast Checkpointing and Recovery

After identifying and evicting faulty machines, the driver needs to resume the training by loading model weights and optimizer states from the most recent checkpoint. It is critical to ensure that the latest checkpoint is as close as possible to the state of training progress when the faults happened, to minimize loss in computation and time. This require us to increase the frequency of checkpointing during training. However, we also want to reduce the latency introduced by the checkpointing process, especially the time on the critical path which blocks the training progress, thus impeding the overall system throughput.

To achieve fast checkpointing, we introduce an optimized, two-stage approach. In the first stage, each GPU worker writes its on-chip states to the host memory, and then continues the training process. After the optimization of Pytorch's serialization mechanism and the use of pinned memory, this process can be reduced to several seconds thanks to the high PCIe bandwidth, thereby minimally interrupting the ongoing training process. In the second stage, a background process takes over, asynchronously transferring the state from the host memory to a distributed file system (HDFS in our deployment) for centralized maintenance. This decoupling of operations into two stages allows the GPU workers to resume training almost immediately after dumping their state, while the more time-consuming process of writing to HDFS is offloaded to a separate, non-blocking process.

In the context of recovery from a checkpoint, it is on the critical path since training can not be started without the last checkpoint. The bottleneck is the bandwidth of HDFS, especially when each GPU worker needs to read its corresponding state partition. To alleviate this bottleneck, we propose an optimized data retrieval strategy. We recognize that multiple GPU workers often share the same state partition, e.g., the workers in the same data parallel group. Accordingly, we designate a single worker in the group to read the shared state partition from HDFS, thereby reducing the load linearly. This worker then broadcasts the state partition to all other GPU workers that share the same data. This approach effectively

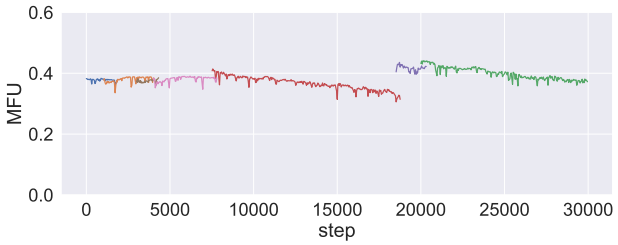


Figure 6: Inconsistent MFU observed in large-scale training. Different colors denote distinct executions of the same training job.

mitigates the bandwidth constraints of HDFS, leading to a substantial reduction in the recovery time.

5 Training Troubleshooting

Although our robust training framework automatically discovers, pinpoints, and resolves the majority of common faults, there remain certain hardware anomalies that manifest probabilistically and cannot be found by machine self-checks. Some anomalies may make the system appear to operate normally, yet significantly degrades the training efficiency. To address these nuanced cases, we have implemented several custom monitoring and analysis tools designed to support case-by-case anomaly detection.

5.1 Performance Diagnosis with CUDA Event Monitor

At the scale of tens of thousands of GPUs, we observe that, unlike in smaller-scale experiments, different runs exhibit varying computational efficiencies. Even with identical configurations , this inconsistency persist, as shown in Figure 6. We also observed that the performance of training tasks is not consistent at this scale. The MFU for various training tasks gradually declines over time. While this leads us to suspect variations between individual machines, no evident variations are detected under single GPU GEMM micro-benchmarks. To diagnose those performance issues, we develop a performance analysis tool that records the execution time of critical code segments on each machine rank during a run. In contrast to previous tools such as the torch profiler or the Megatron-LM timer, our tool times events based on the CUDA events method. This approach minimizes the need for CUDA synchronization, thus preventing performance degradation, allowing us to consistently run it in our production training jobs. This tool offers two visualization modes and can analyze the collected data from different perspectives.

The first mode uses a heat map to show time consumption differences between machines from various dimensions, depicted in Figure 7. We gather latency data of the computation phase (forward and backward) across devices and average the latency across steps. The aggregated data is visualized

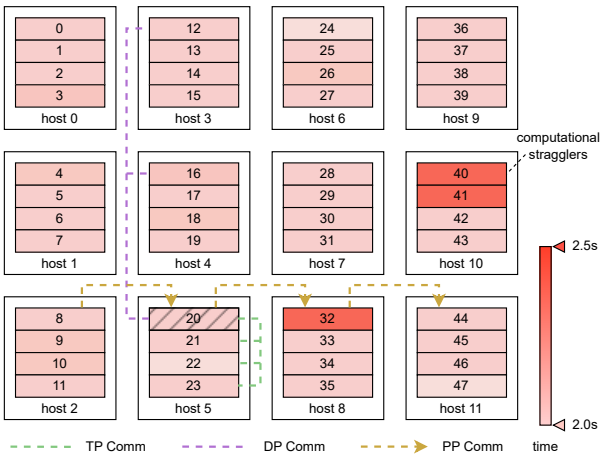


Figure 7: Performance heat-map. The color denotes the running time of the code segments on a rank. The figure also shows the 3D visualization feature, where rank 20 has been selected and the dependency across different parallelism dimensions become visible.

using a heat-map. The heat-map reveals that a minor fraction of machines (approximately 0.5%) exhibit substantially slower performance during training, thereby hindering overall training progress. The training efficiency is predominantly determined by the slowest machine's performance (i.e., stragglers), leading to inconsistencies in training efficiency across diverse runs, since machine scheduling within the cluster is stochastic. After excluding these outlier machines, the peak MFU across runs becomes consistent.

The other mode displays the event timeline on machines in a trace format from different distributed views (data parallelism, pipeline parallelism, tensor parallelism). Traditional profiler, such as PyTorch Profiler, is primarily designed for single-node activity analysis. This approach offers limited insight in distributed training scenarios where execution dependencies frequently span across multiple nodes. By aggregating the trace spans of various ranks onto a singular timeline, we gain a comprehensive perspective, revealing the overall execution order, pipeline bubbles, and synchronization characteristics among data parallel ranks. Figure 8 displays how our distributed tracer visualizes the actual execution of pipeline parallelism, detailing the data dependencies between different pipeline stages through the consolidation of event data across a pipeline parallelism group.

Every piece of data from the CUDA event timer is stored in a remote analytical database, allowing for easy retrieval of details from any step event. While the timer data is wrote to a local file in a line-by-line format, a separate streamer process then synchronizes this log file with a Kafka queue in real-time. The analytical database remains updated by consuming data from this Kafka queue, enabling on-the-fly analysis without interrupting the training job. All the monitoring features are turned on during real production training and the overhead is negligible compared to the training time.

Figure 8: The trace shows events collected in a pipeline group on a unified timeline. Dependencies become visible when an event is selected.

5.2 3D Parallel Training Visualization

With 3D parallelism and our optimization techniques (§3), the landscape of data flow and task sequencing is exceedingly intricate. Each GPU worker may be engaged in several synchronous or asynchronous operations at the given moment, leading to complex dependencies among them. This intricacy amplifies the challenges of fault diagnosis: when a single GPU worker experiences a fault, the entire cluster of nodes can stall in the NCCL communication operations, ultimately leading to a system-wide timeout. Externally, this situation manifests as a generic blockage, but the root cause of which is often buried under a deluge of timeout messages. To rapidly pinpoint the problematic nodes, we let each GPU worker log its own ongoing event upon communication timeout. These logs are then used to construct a visual representation of data dependencies based on the logical topology in the 3D parallel setting.

As Figure 7 shows, the cluster in 3D parallel training can logically be split into three dimensions: tensor parallelism, pipeline parallelism, and data parallelism. When we select a specific GPU worker, it displays its position within the logical topology, the direction of data flow and the different communication operations it involves. Importantly, in the event of an error, the tool provides direct access to the worker's error messages if any. This serves as a powerful tool for diagnosing training anomalies, enabling quicker identification and resolution of faults.

Consider the aforementioned case when defective GPUs probabilistically cause blocking when executing NCCL communication operations. Such blocking can hang the entire machine, leading to cascading timeouts across other dependent nodes and ultimately resulting in the paralysis of the entire training process. To swiftly identify these faulty nodes, we utilize the 3D parallel training visualization tool. Nodes that timeout due to waiting for the faulty ones will log their ongoing operations upon exiting. In contrast, the nodes with the faulty GPUs are hung and do not log any such information. Therefore, by examining the logs and the data flow within the visualization, these problematic nodes can be easily pinpointed. Once identified, these nodes can be manually isolated and flagged for maintenance through the robust training framework, as described in 4.1.

Model Size	Heads	Hidden Size	Layers	TP	PP
175B	128	12288	96	8	8
530B	160	20480	105	8	35

Table 1: Model configurations.

6 Experience

In this section, we describe our deployment and operational experience of MegaScale. We build dedicated AI clusters for LLM training. Over the years, we have iterated several versions of our specialized AI cluster architecture, and we are currently operating several AI clusters with varying size and hardware configurations. We use these AI clusters to train a wide range of models, from computer vision and recommendation models to LLMs. With the increasing importance of LLMs, we are building AI clusters with larger size to cater the need of LLM training. As of September 2023, the largest AI cluster in our production for LLM training contains more than 10,000 NVIDIA Ampere GPUs. We are also in the process of building large clusters based on the newest NVIDIA Hopper GPUs, as NVIDIA is ramping up production.

6.1 Training Performance

MegaScale is built on top of Megatron-LM [7], which is a state-of-the-art open-source LLM training framework that integrates 3D parallelism techniques and takes advantage of hardware resources. Our experiments use the Megatron-LM (commit hash: 285068c8) on Github [21], committed on January 11, 2023, chosen for its stability and feature set at the commencement of our experiments. We use the same batch size for Megatron-LM and MegaScale for fair comparison. We use two model sizes: 175B parameters and 530B parameters. We use interleaved pipeline-parallel schedule [22] with six and three interleaving stages for the 175B and 530B models, respectively. Sequence length is 2,048 and vocabulary size is 64,000 for all the cases. Table 1 shows the details of the model configuration.

Scalability. Figure 9 compares Megatron-LM and MegaScale when training the 530B model, where we set the batch size as the number of GPUs with adjusted learning rate to show the MFU results. We see that the MFU of MegaScale is higher than Megatron-LM by up to 6.1%. With increasing scales, the

Batch Size	Method	GPUs	Iteration Time (s)	Throughput (tokens/s)	Training Time (days)	MFU	Aggregate PFlops/s
768	Megatron-LM	256	40.0	39.3k	88.35	53.0%	43.3
		512	21.2	74.1k	46.86	49.9%	77.6
		768	15.2	103.8k	33.45	46.7%	111.9
		1024	11.9	132.7k	26.17	44.7%	131.9
	MegaScale	256	32.0	49.0k	70.86	65.3%(**1.23**×)	52.2
		512	16.5	95.1k	36.51	63.5%(**1.27**×)	101.4
		768	11.5	136.7k	25.40	61.3%(**1.31**×)	146.9
		1024	8.9	176.9k	19.62	59.0%(**1.32**×)	188.5
6144	Megatron-LM	3072	29.02	433.6k	8.01	48.7%	466.8
		6144	14.78	851.6k	4.08	47.8%	916.3
		8192	12.24	1027.9k	3.38	43.3%	1106.7
		12288	8.57	1466.8k	2.37	41.2%	1579.5
	MegaScale	3072	23.66	531.9k	6.53	59.1%(**1.21**×)	566.5
		6144	12.21	1030.9k	3.37	57.3%(**1.19**×)	1098.4
		8192	9.56	1315.6k	2.64	54.9%(**1.26**×)	1400.6
		12288	6.34	1984.0k	**1.75**	55.2%(**1.34**×)	2166.3

Table 2: Strong-scaling training performance for the 175B model. We set the batch size to 6144 when training with 3072 to 12288 GPUs. For 256 to 1024 GPUs, we decrease the batch size to 768 due to GPU memory limit. We report the training time required for training 300B tokens here. The number in parentheses in the MFU column represents the speedup of MegaScale compared to Megatron-LM.

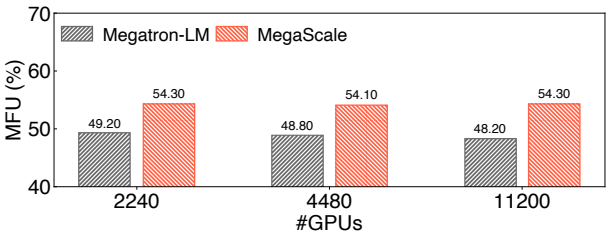


Figure 9: Weak-scaling training performance of Megatron-LM and MegaScale on the 530B model, where the batch size is scaled proportionally with the number of GPUs.

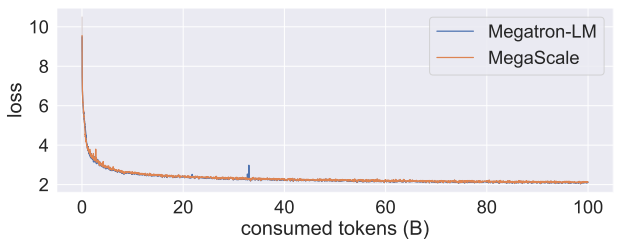
MFU of Megatron-LM decreases by 1.6% with more stragglers and communication, while MegaScale has near-linear scalability due to 3D-parallel communication overlapping.

In Table 2, we evaluate the strong-scaling training performance of Megatron-LM and MegaScale on the 175B model by increasing number of GPUs and maintaining a constant batch size. This experimental setting is more realistic, given that batch size is constrained by convergence effects and cannot be indefinitely scaled with the number of GPUs. MegaScale achieves up to 1.34× speedups over Megatron-LM across all settings. With increasing GPUs, we observe the MFU of MegaScale decreases from 59.1% to 55.2%. This is expected since the batch size is fixed and the computation-to-communication ratio decreases with more GPUs. Even in the largest scale with 12,288 GPUs, MegaScale still outperforms Megatron-LM by 14% MFU. For the smaller scale training, the speedup of MegaScale over the baseline ranges from 1.23× to 1.32×. Note that the difference in the maximum number of GPUs between this and the previous experiments (e.g., 12,288 vs. 11,200) is due to distinct 3D parallelism configurations for 175B and 530B models.

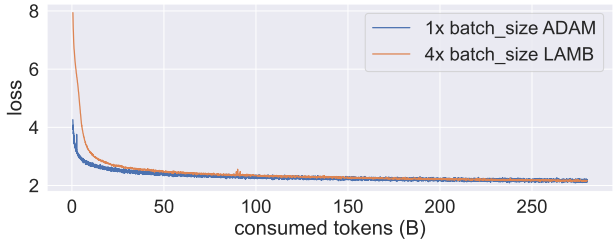
Ablation study. We evaluate the effectiveness of our optimization techniques of MegaScale. Table 3 shows the MFU improvement breakdown with different optimizations when training the 175B model on 256 GPUs. The baseline is the original Megatron-LM and has 47.7% MFU. It is worth noting that the networking optimizations are turned on for both Megatron-LM and MegaScale in this evaluation. We first apply two algorithmic techniques, parallel transformer block and sliding window attention, to Megatron-LM, achieving 5.6% MFU improvement. Communication is the major bottleneck of large-scale LLM training, and the 3D parallel communication overlapping of MegaScale hides the overhead and accelerates training by 6.2% MFU. We further adopt efficient operators and obtain 1.7% acceleration. Other optimizations such as data pipeline optimizations and the problematic code elimination mentioned in 6.3 further achieves 1.1% performance gain. Finally, we scale the batch size from 256 to 768 with LAMB optimizer, which significantly extends the steady phase in interleaved pipeline parallelism and achieves 3.0% MFU improvement. To sum up, MegaScale outperforms the baseline by 17.6% in the MFU number with all these optimizations.

6.2 Model Convergence and Stability

Model convergence microbenchmarks. We first conduct microbenchmark experiments to validate the algorithm tech-



(a) The training loss curve of MegaScale, which includes algorithm optimizations, in comparison with Megatron-LM.



(b) The training loss curve of ADAM optimizer and LAMB optimizer with four times of the batch size.

Figure 10: The training loss curves in microbenchmark experiments.

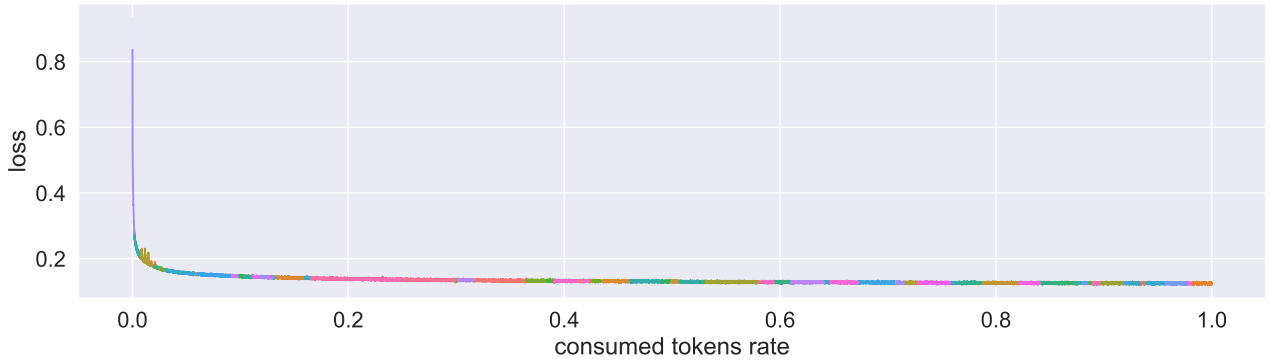


Figure 11: The normalized training loss curve of a real production run on more than 10,000 GPUs for several weeks. This run trains a model with hundreds of billions of parameters on multi-trillion tokens. Different colors indicate training restarts. MegaScale repairs and recovers the training process for over 100 times in presence of failures.

Idx	Method	MFU (Δ MFU)
1	baseline	47.7%
2	(1) with PTB	52.3% (4.6%)
3	(2) with SWA	53.3% (5.6%)
4	(3) with TP overlap	55.5% (7.8%)
5	(4) with PP overlap	58.0% (10.3%)
6	(5) with DP overlap	59.5% (11.8%)
7	(6) with efficient operators	61.2% (13.5%)
8	(7) with misc optimizations	62.3% (14.6%)
9	(8) with LAMB (BS×3)	65.3% (17.6%)

Table 3: MFU improvement breakdown when training the 175B model with 256 GPUs and batch size 256.

niques do not affect the model convergence. Due to the resource limit, the microbenchmarks are done on the 13B model. As shown in Figure 10a, while MegaScale adopts algorithm techniques, including parallel transformer block and sliding window attention, it achieves comparable loss results with the baseline when training with more than 100B tokens. We also evaluate the effect of LAMB optimizer as depicted in Figure 10b, which shows that LAMB optimizer with four times of batch size achieves the same loss as ADAM optimizer after around 250B tokens. Based on these observations, we turn on all the algorithmic optimizations in production training.

Model convergence and stability in real production LLM training. We show the model convergence and stability from a real production run. This run trains a proprietary model with hundreds of billions of parameters on multi-trillion tokens. This run uses more than 10,000 GPUs and lasts for several weeks. Figure 11 shows the loss continues to converge, with distinct colors indicating the training is restarted. Over the several weeks of this run, we experience training restarts over 100 times. With the robust training framework, over 90% of software and hardware faults are automatically identified and fixed by the techniques detailed in §4. The rest of the problems are handled with the help of the troubleshooting tools described in §5.

6.3 Problems Discovered and Fixed

We conduct an analysis of the fault records for the aforementioned production training job over several weeks. Our findings indicate that over 90% of the exceptions among them are automatically detected, located, and recovered using our robust training framework, such as CUDA error and segmentation fault. The average time required for detecting failure and

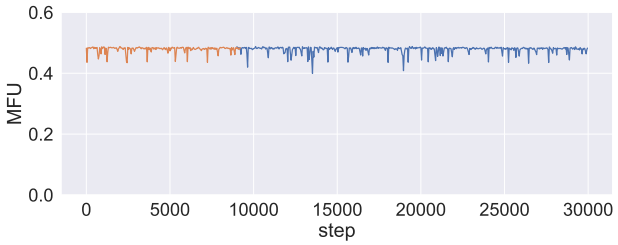


Figure 12: The MFU becomes stable after addressing the stragglers and problematic code segments. Different colors represent different training trials with the same setup.

executing diagnostic tests is less than 10 minutes. Moreover, the system can catch up to the training progress prior to the crash within 15 minutes from the latest checkpoints, maintaining over 90% effective training time rate, which is calculated as the number of iterations multiplied by the iteration training time, divided by the total training time. Below we show our experience in diagnosing and fixing some intriguing problems, which need to be analyzed using the troubleshooting tools in §5.

Computational stragglers. Building upon our utilization of CUDA event timers, we made another pertinent observation across multiple experimental setups. We noted that specific hosts took approximately 10% more time to execute the same forward computations compared to other ranks. This consistency across different experiments led us to conclude that the issue was not with the software but rather inherent to certain machines in the cluster. After isolating and removing these problematic hosts from the cluster, we observed an approximate 0.7% improvement in MFU.

MFU decreasing. In such large-scale training experiments, another phenomenon we observed is that training efficiency did not remain consistent over time. Instead, as the training progressed, the MFU of our training job gradually decreased. Through a step-by-step analysis based on CUDA event timer metrics, we noted several key findings. While the time consumed per training step was increasing, the time spent on forward, backward, and optimizer computations remained stable, irrespective of the increasing number of steps. This led us to infer that the time increase must be attributed to the collective communication overhead. Upon a reverse chronological examination, we identified the last collective communication step as the gradient reduce-scatter in data parallelism. If this step is delayed, the overall time per step elongates. Since we observed network bandwidth to be largely stable, we ruled out slowed communication speed as a factor for the increased time. According to the synchronization characteristics of collective communication, this leaves us with one conclusion: some ranks initiate the reduce-scatter operation later than others, forcing a wait for the slowest rank to catch up. In a scaled-down experiment involving only two ranks per data parallel group, we measured the launch times for reduce-scatter calls and found them to not be consistently staggered but rather fluctuating reciprocally. Furthermore, the size of this time stagger increased as more steps were executed. Specifically, Rank A may initially lag behind Rank B but might eventually surpass Rank B in speed and by a growing margin. Ultimately, all ranks waited for the slowest rank. To trace back the root cause of this time skew, we located the variance to occur during the forward computation stage. Digging deeper into the code, we attributed this irregularity to fluctuations caused by some code segments. For instance, irregular garbage collection can introduce disturbances into the training procedure, and certain PyTorch operations can lead to performance fluctuations. These operations are on the critical path but can be affected along the training procedure. After modifying or removing those problematic code segments, we no longer observed a significant decline in MFU, as shown in Figure 12.

Frequent network interface flapping problem. We occasionally encounter training stall or training speed drop problem due to frequent network interface flapping. When the network interface flapping phenomena happens, the network interface goes down at first then goes up again. The interval between down and up time usually lasts for several seconds. During the down process, all the packets in transmission will be dropped. The first lesson we learn is the timeout threshold should be set explicitly to a larger value , otherwise the default value will make NCCL timeout very quickly and return a completion error before the network card up again. The second lesson we learn is that the root cause of this problem is the bad link quality between network card, AOC cable and switch. The flapping frequency can be reduced to a satisfactory level by doing lower level quality control over network card signal strength, AOC cable quality and switch side signal strength.

7 Related Work

LLM training. A lot of efforts have been put to the training of pre-trained LLMs, including proprietary ones such as GPT-3 [1], GPT-4 [23], GShard [24], PaLM [5], and many others [25–29], as well as open-source alternatives like OPT [30], BLOOM [31], Llama [32], Llama-2 [33]. Existing technical reports in the field predominantly focus on model performance comparisons, leaving out the specific details of the system infrastructure that makes such training possible. This paper fills this gap by sharing our experience of end-to-end LLM pre-training at the scale of over 10,000 GPUs from a systems perspective.

After pre-training, pre-trained base models can be further fine-tuned to adapt to downstream tasks better. This has led to the emergence of a range of dialogue models [34–37] exemplified by ChatGPT. However, it is worth noting that the computational power and data requirements for fine-tuning are substantially lower than those for pre-training. With the application of optimization techniques such as quantization [38–41]

and low-rank adaptation [42], fine-tuning can be efficiently accomplished with limited resources.

LLM optimizations. In addition to the techniques mentioned previously in the paper, there exists a lot of other works targeted at improving the efficiency of LLMs. Sparse or linear attentions [43–45] are proposed to make the memory consumption scales approximately linearly. Several studies aim to design new architectures rather than conventional transformer architectures to address the efficiency issue, such as RWKV [46] and RetNet [47]. Many recent studies have been devoted to developing communication acceleration techniques for LLMs. Some works reduce communication traffic using gradient compression [48] or mixed-precision training [49], while others schedule communication to overlap it with computation. Many popular ML frameworks, such as TensorFlow [50] and PyTorch [51], enable overlapping communication with backward propagation by default. Recent works [52–55] further overlap gradient synchronization with forward computation via tensor partitioning, at the cost of extra overhead. Some works [56, 57] introduce fixed staleness to the training pipeline for full overlapping communication and communication. However, the staleness may degrade the model performance.

Diagnosis tools in datacenters. Many diagnosis tools have been developed to identify and pinpoint hardware and software problems in datacenters. Pingmesh [58] is an active probing system based on end hosts. Network wide RTT and packet loss and measured by sending probing ping packets and doing data analysis. Network-wide SLAs are provided and network problems including packet-blackhole and packet silent drop are detected. EverFlow [59], LossRadar [60], NetBouncer [61] exploits the capability of switches to diagnose detailed network problems like network path failures or specific network port failures. NetBouncer leverages IP-in-IP tunnel techniques to do path probing. EverFlow requires mirroring network packets to a centralized server to do debugging. Hostping [62] is a diagnosis system based on end hosts that focuses on intra-host bottlenecks. It actively senses complex GPU server PCIe/NVLINK interconnects and does loopback bandwidth and latency tests.

Fault tolerance in large-scale distributed systems. Fault tolerance has been a major concern in large-scale distributed systems, where a wide range of hardware and software failures can occur. Many fault tolerance techniques have been proposed in the past that cater the needs of different systems and deployment scenarios. Reactive fault tolerance techniques are used to reduce the impact of failures on a system when the failures occur. There are many techniques in this category such as Retry [63], Replication [63], Checkpointing [64] and Message Logging [65]. These techniques incur some system overhead to recover from failures. Proactive fault tolerance techniques keep healthy components in place as backups of the faulty components, obviating the need of recovery from faults and errors, e.g., preemptive migration [66–68] and load balancing [69]. However, these approaches often assume that failures are predictable, while it is challenging for real large-scale distributed systems to predict the failures due to the complexity of the systems.

8 Conclusion

In this paper, we offer an in-depth look at the design, implementation and deployment of MegaScale, a production-grade system built to train LLMs at the scale of over 10,000 GPUs. MegaScale exploits algorithm-system co-design to optimize training efficiency. MegaScale achieves 55.2% MFU when training a 175B LLM model on 12,288 GPUs, a 1.34× improvement over Megatron-LM. We emphasize the need for fault tolerance throughout the training process and implement a tailored robust training framework to locate and fix faults automatically. We provide a comprehensive set of monitoring tools for deep observability into system components and events, facilitating root cause identification for intricate anomalies. We believe that our work not only offers practical insights for those working on LLM training, but also paves the way for future research in this rapidly evolving field.

Acknowledgments

We thank our shepherd Deepak Narayanan and the anonymous NSDI reviewers for their insightful and constructive feedback. We thank Zhenyuan Yang and Wenjia Zhu for their thorough feedback on earlier drafts of this manuscript. We thank Zuquan Song, Gaohong Liu and the broader ByteDance Applied Machine Learning team for their support throughout this project.

References

[1] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[2] "Introducing chatgpt." https://openai.com/blog/chatgpt, 2022.

[3] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020.

[4] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, pp. 681–694, 2020.

[5] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[7] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. A. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, "Efficient large-scale language model training on gpu clusters using megatron-lm," 2021.

[8] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," 2020.

[9] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh, "Large batch optimization for deep learning: Training bert in 76 minutes," in *International Conference on Learning Representations*, 2020.

[10] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2020.

[11] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models." ArXiv, May 2020.

[12] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, *et al.*, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism ," in *NeurIPS*, 2019.

[13] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *ACM SOSP*, 2019.

[14] B. Wang and A. Komatsuzaki, "GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model." https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

[15] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer, A. Desmaison, C. Balioglu, P. Damania, B. Nguyen, G. Chauhan, Y. Hao, A. Mathews, and S. Li, "Pytorch fsdp: Experiences on scaling fully sharded data parallel," 2023.

[16] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," *arXiv preprint arXiv:2307.08691*, 2023.

[17] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "Pytorch distributed: Experiences on accelerating data parallel training," 2020.

[18] I. Group, "Ieee 802.1 qbb - priority-based flow control." https://1.ieee802.org/dcb/802-1qbb/, 2009.

[19] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion Control for Large-scale RDMA Deployments," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 523–536, 2015.

[20] G. Kumar, N. Dukkipati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, *et al.*, "Swift: Delay is Simple and Effective for Congestion Control in the Datacenter," in *SIGCOMM*, pp. 514–528, 2020.

[21] "Megatron-LM." https://github.com/NVIDIA/Megatron-LM/tree/main, 2021.

[22] V. A. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro, "Reducing activation recomputation in large transformer models," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[23] OpenAI, "Gpt-4 technical report," 2023.

[24] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, "Gshard: Scaling giant models with conditional computation and automatic sharding," 2020.

[25] A. Askell, Y. Bai, A. Chen, D. Drain, D. Ganguli, T. Henighan, A. Jones, N. Joseph, B. Mann, N. DasSarma, N. Elhage, Z. Hatfield-Dodds, D. Hernandez, J. Kernion, K. Ndousse, C. Olsson, D. Amodei, T. Brown, J. Clark, S. McCandlish, C. Olah, and J. Kaplan, "A general language assistant as a laboratory for alignment," 2021.

[26] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," 2022.

[27] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, E. Zhang, R. Child, R. Y. Aminabadi, J. Bernauer, X. Song, M. Shoeybi, Y. He, M. Houston, S. Tiwary, and B. Catanzaro, "Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model," 2022.

[28] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. de Las Casas, L. A. Hendricks, J. Welbl, A. Clark, T. Hennigan, E. Noland, K. Millican, G. van den Driessche, B. Damoc, A. Guy, S. Osindero, K. Simonyan, E. Elsen, J. W. Rae, O. Vinyals, and L. Sifre, "Training compute-optimal large language models," 2022.

[29] H. Su, X. Zhou, H. Yu, X. Shen, Y. Chen, Z. Zhu, Y. Yu, and J. Zhou, "Welm: A well-read pre-trained language model for chinese," 2023.

[30] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. ChenG, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "Opt: Open pre-trained transformer language models," 2022.

[31] T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé, *et al.*, "Bloom: A 176b-parameter open-access multilingual language model," *arXiv preprint arXiv:2211.05100*, 2022.

[32] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023.

[33] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023.

[34] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto, "Stanford alpaca: An instruction-following llama model." https://github.com/tatsu-lab/stanford_alpaca, 2023.

[35] W.-L. Chiang, Z. Li, Z. Lin, Y. Sheng, Z. Wu, H. Zhang, L. Zheng, S. Zhuang, Y. Zhuang, J. E. Gonzalez, I. Stoica, and E. P. Xing, "Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality," 2023.

[36] X. Geng, A. Gudibande, H. Liu, E. Wallace, P. Abbeel, S. Levine, and D. Song, "Koala: A dialogue model for academic research." Blog post, April 2023.

[37] Y. Ji, Y. Deng, Y. Gong, Y. Peng, Q. Niu, B. Ma, and X. Li, "Belle: Be everyone's large language model engine." https://github.com/LianjiaTech/BELLE, 2023.

[38] Z. Li, E. Wallace, S. Shen, K. Lin, K. Keutzer, D. Klein, and J. Gonzalez, "Train big, then compress: Rethinking model size for efficient training and inference of transformers," in *International Conference on Machine Learning (ICML)*, 2020.

[39] G. Xiao, J. Lin, M. Seznec, J. Demouth, and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," *arXiv*, 2022.

[40] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," *arXiv*, 2022.

[41] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "Llm. int8 (): 8-bit matrix multiplication for transformers at scale," *arXiv*, 2022.

[42] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021.

[43] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," 2019.

[44] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, "Transformers are rnns: Fast autoregressive transformers with linear attention," 2020.

[45] C. Zhu, W. Ping, C. Xiao, M. Shoeybi, T. Goldstein, A. Anandkumar, and B. Catanzaro, "Long-short transformer: Efficient transformers for language and vision," 2021.

[46] B. Peng, E. Alcaide, Q. Anthony, A. Albalak, S. Arcadinho, H. Cao, X. Cheng, M. Chung, M. Grella, K. K. GV, X. He, H. Hou, P. Kazienko, J. Kocon, J. Kong, B. Koptyra, H. Lau, K. S. I. Mantri, F. Mom, A. Saito, X. Tang, B. Wang, J. S. Wind, S. Wozniak, R. Zhang, Z. Zhang, Q. Zhao, P. Zhou, J. Zhu, and R.-J. Zhu, "Rwkv: Reinventing rnns for the transformer era," 2023.

[47] Y. Sun, L. Dong, S. Huang, S. Ma, Y. Xia, J. Xue, J. Wang, and F. Wei, "Retentive network: A successor to transformer for large language models," 2023.

[48] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding," in *NeurIPS*, 2017.

[49] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed Precision Training," in *ICLR*, 2018.

[50] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *OSDI*, 2016.

[51] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library ," in *NeurIPS*, 2019.

[52] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based Parameter Propagation for Distributed DNN Training ," in *MLSys*, 2019.

[53] S. H. Hashemi, S. Abdu Jyothi, and R. Campbell, "TicTac: Accelerating Distributed Deep Learning with Communication Scheduling," in *MLSys*, 2019.

[54] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed DNN training acceleration," in *SOSP*, 2019.

[55] Y. Bao, Y. Peng, Y. Chen, and C. Wu, "Preemptive All-reduce Scheduling for Expediting Distributed DNN Training," in *INFOCOM*, 2020.

[56] Y. Li, M. Yu, S. Li, S. Avestimehr, N. S. Kim, and A. Schwing, "Pipe-SGD: A Decentralized Pipelined SGD Framework for Distributed Deep Net Training," in *NeurIPS*, 2018.

[57] Y. Chen, C. Xie, M. Ma, J. Gu, Y. Peng, H. Lin, C. Wu, and Y. Zhu, "Sapipe: Staleness-aware pipeline for data parallel dnn training," *Advances in Neural Information Processing Systems*, vol. 35, pp. 17981–17993, 2022.

[58] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, "Pingmesh: A large-scale system for data center network latency measurement and analysis," *SIGCOMM Comput. Commun. Rev.*, vol. 45, p. 139–152, aug 2015.

[59] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, "Packet-level telemetry in large datacenter networks," *SIGCOMM Comput. Commun. Rev.*, vol. 45, p. 479–491, aug 2015.

[60] Y. Li, R. Miao, C. Kim, and M. Yu, "Lossradar: Fast detection of lost packets in data center networks," in *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, (New York, NY, USA), p. 481–495, Association for Computing Machinery, 2016.

[61] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang, "Netbouncer: Active device and link failure localization in data center networks," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, (USA), p. 599–613, USENIX Association, 2019.

[62] K. Liu, Z. Jiang, J. Zhang, H. Wei, X. Zhong, L. Tan, T. Pan, and T. Huang, "Hostping: Diagnosing intra-host network bottlenecks in RDMA servers," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, (Boston, MA), pp. 15–29, USENIX Association, April 2023.

[63] S. Haider, N. R. Ansari, M. Akbar, and M. R. Perwez, "Fault tolerance in distributed paradigms," 2011.

[64] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–14, 2018.

[65] A. S. Tanenbaum, *Distributed systems principles and paradigms*. 2007.

[66] S. Chakravorty, C. L. Mendes, and L. V. Kalé, "Proactive fault tolerance in mpi applications via task migration," in *International Conference on High-Performance Computing*, pp. 485–496, Springer, 2006.

[67] S. Chakravorty, C. Mendes, and L. V. Kale, "Proactive fault tolerance in large systems," in *HPCRI Workshop in conjunction with HPCA*, vol. 2005, pp. 1–7, Citeseer, 2005.

[68] Y. Chen, Y. Peng, Y. Bao, C. Wu, Y. Zhu, and C. Guo, "Elastic parameter server load distribution in deep learning clusters," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 507–521, 2020.

[69] I. Behera and C. R. Tripathy, "Performance modelling and analysis of mobile grid computing systems," *International Journal of Grid and Utility Computing*, vol. 5, no. 1, pp. 11–20, 2014.

Resiliency at Scale: Managing Google's TPUv4 Machine Learning Supercomputer

Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda,
Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, Steve Lacy, Hang Wang,
Aaron Wisner, Chris Lewis, Henri Bahini
Google
tpuv4-nsdi-paper@google.com

Abstract

TPUv4 (Tensor Processing Unit) is Google's 3rd generation accelerator for machine learning training, deployed as a 4096-node supercomputer with a custom 3D torus interconnect. In this paper, we describe our experience designing and operating the software infrastructure that allows TPUv4 supercomputers to operate at scale, including features for automatic fault resiliency and hardware recovery. We adopt a software-defined networking (SDN) approach to manage TPUv4's high-bandwidth inter-chip interconnect (ICI) fabric, using optical circuit switching to dynamically configure routes to work around machine, chip and link failures. Our infrastructure detects failures and automatically triggers reconfiguration to minimize disruption to running workloads, as well as initiating remediation and repair workflows for the affected components. Similar techniques interface with maintenance and upgrade workflows for both hardware and software. Our dynamic reconfiguration approach allows our TPUv4 supercomputers to achieve 99.98% system availability, gracefully handling hardware outages experienced by ~1% of the training jobs.

1 Introduction

Machine Learning (ML) models continue to grow in size and complexity [9, 24], enabled by the massive compute capability of heterogeneous supercomputers, where CPUs handle runtime task coordination and I/O, and accelerators such as TPUs [18–20] and GPUs deliver the computational performance needed for model training. Scaling up a supercomputer's node count enables more capable models because the training process can be effectively parallelized along the batch, tensor, and pipeline dimensions [17, 33].

The hardware/software ecosystem of ML supercomputers faces two challenges at scale: first, effectively parallelizing model training workloads, and secondly – and the focus of this paper – maintaining high availability of compute resources and consequently high goodput for ML training jobs. The latter has become increasingly difficult in recent years because:

- unlike loosely-coupled distributed applications such as Map-Reduce [12] that can effectively tolerate dynamically varying resource allocation, ML training jobs more often use static (compile-time) sharding strategies and gang scheduled execution, requiring all compute resources to be healthy simultaneously;
- modern ML models such as Large Language Models (LLMs) need an unprecedented amount of hardware (conventional compute, accelerators, networking and storage) [2], dropping the expected MTBF to hours or even minutes [15];
- in a cloud or shared cluster environment, many users contend for different subsets of supercomputer resources, making it essential to be able to reconfigure or rebalance resource allocations over time.

Google's TPUv4 machine learning supercomputing infrastructure is designed to meet these challenges. It comprises the following hardware and software components:

- A number of *cubes*: a cube is a hardware unit with 64 TPU chips arranged in a 4x4x4 3D mesh; each supercomputer or *pod* has 64 cubes, for a total of 4096 TPUs.
- A proprietary inter-chip interconnect (*ICI*): this is a high-speed network fabric that directly interconnects TPUs to allow direct device-to-device communication (i.e. RDMA) without involving the CPUs.
- Optical circuit switches (*OCSes*) [25]: these are used to dynamically cross-connect (xconnect) the ICI from different cubes to form the user-requested torus topology.
- Borg [31]: a cluster management service that admits, schedules and manages TPUv4 jobs (and others).
- Pod Manager: a cluster-level software service that manages multi-cube connectivity by actuating OCS xconnect setup in response to Borg scheduling decisions.

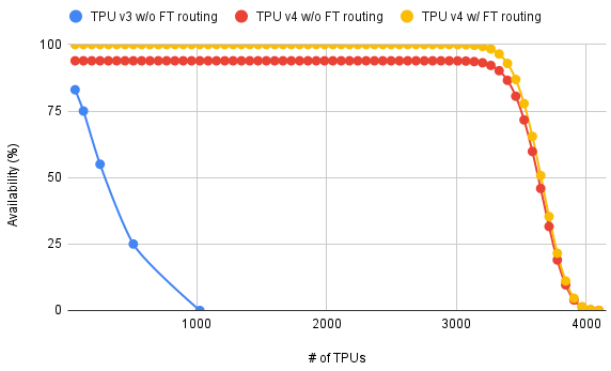


Figure 1: Availability of jobs size improves at scale with TPUv'4 cube configurability and fault-tolerant ICI routing.

- `libtpunet`: a software library that sets up the requested ICI network topology for each TPUv4 user job.
- `healthd`: a software daemon running on each host in a pod that continuously monitors machine hardware health and reports back to cluster-level software systems.

In TPUv4 pods, hardware and software are co-designed. Hardware provides a configurable compute substrate (the 4x4x4 cubes) with a programmable ICI protocol stack implemented by the OCS switches and on-chip ICI switches. Software dynamically manages the hardware by configuring the OCSes to combine multiple cubes into larger *slices* of the pod, and by programming the ICI routing policy via `libtpunet`. Connectivity is reconfigured according to the user-requested torus topology as Borg schedules jobs onto the various cubes. The Pod Manager mediates reconfiguration, and monitors ICI- and OCS-related health, excluding cubes from the resource pool as soon as faults are discovered.

In this paper, we describe how we make our TPUv4 supercomputers automatically resilient to faults at scale. More specifically, we:

– explain the TPUv4 supercomputer's configurable system architecture based on a programmable ICI protocol with OCS and ICI switches, optimizing for system availability and resiliency at scale;

– describe the software infrastructure that schedules, configures, and optimizes TPUv4 resources, focusing on our design principles of configurability and modularity;

– outline our optimized strategy for accelerator-side multi-hop RDMA routing for resilient collective operations over regular and twisted torus topologies; and

– report on our experiences to date in operating TPUv4 supercomputers in production.

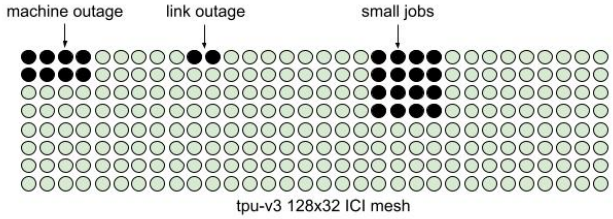


Figure 2: Static pods face resource fragmentation problems.

2 The Reconfigurable ML Supercomputer System Architecture

The TPUv4 reconfigurable supercomputer is designed for scalability, availability, resiliency, and cost [18]. At its core is a reconfigurable ICI fabric topology that connects different TPUv4 chips, backed by a set of programmable OCSes for each pod. Without TPUv4's OCS-based reconfigurability, job availability quickly drops as compute resource scales up. Figure 1 shows this effect with measured data from deployed TPUv3 *static pods* and TPUv4 *reconfigurable pods*.

For a conventional supercomputer like TPUv3 [19] where compute resources are statically interconnected, the overall availability of a job drops precipitously as the required amount of compute resources increases to 1024 chips. This is because in a static pod, all resources in a contiguous set of nodes must be simultaneously healthy to be assigned to a user, which becomes combinatorially less likely as the system scales. With TPUv4's cube-level configurability, availability stays high through about 94%, corresponding to ~50 cubes or 3200 TPUv4 chips.

The decreasing availability beyond this point is because of occasional machine and ICI link faults that can occur between different cubes. As we will show in Section 4, tolerating occasional OCS failures or maintenance events with fault-tolerant routing further increases availability to 99.98% because cubes are still accessible to users even in these rare events.

2.1 Lessons from Static Pod Architectures

Before the TPUv4 ML supercomputer, the state of the art was the TPUv2 and TPUv3 *static* pods [19] – 'static' because they feature a non-reconfigurable fixed ICI mesh. A TPUv2 pod has 256 TPUs connected in a 16x16 ICI torus, while TPVv3 had 1024 TPUs connected with a 32x32 torus. There is also a scale-up variant of TPUv3 that combines 4 pods into a 128x32 mesh with limited ICI routing capability. This so-called *multipod* version was co-designed with application collective patterns and used to explore scaling strategies for large models [21].

Figures 1 and 2 illustrate the availability challenges as the model size scales in a static pod. To train a model, all TPU processes must be simultaneously up to synchronously update their weights via ICI collectives. A single failed, or interrupted process will interrupt the whole training process.

Finding the appropriate compute resources for a user's job faces the following challenges:

1. Hardware outages: regular scheduled *maintenance* of hardware, firmware, and software at the ICI link, TPU chip and CPU host level can remove resources from the schedulable pool [22]. For a supercomputer with thousands of TPUs, an event affecting any one component occurs relatively frequently, making it difficult to find usable sets of resources. Furthermore, unexpected *faults* occur more frequently as systems and applications increase in both size and complexity. Without reconfigurability, obtaining decent availability for a job that requires 1024 hosts means that each individual host must sustain 99.9% availability; introducing reconfigurable OCS drops the host availability requirement to 99%.

2. Workload defragmentation: it is common for many jobs to contend for different subsets of a pod's schedulable resources. Since these jobs come and go at unpredictable times, sometimes Borg must move (preempt) smaller jobs to free up contiguous TPUs for pending larger training jobs. The scheduling complexity worsens with a mixture of user priorities. With OCS-based reconfigurability, Borg does not need to worry as much about the physical contiguity of TPU resources. Instead, any set of vacant cubes can be cross connected via the OCS for use by a user's job.

3. Deployment lead time: a static pod is not usable until *all* hardware is installed due to the tightly coupled nature of compute and network resources. With reconfigurable pods, once the OCS footprint is installed, cubes can be deployed and used as soon as they land.

The above challenges challenges motivated us to rethink things for the TPUv4 pod architecture.

2.2 TPUv4: OCS-based Reconfigurability

TPUv4 adopts a reconfigurable architecture which makes use of the Palomar Optical Circuit Switch (OCS) [25] to address the problems with static systems. By adopting this architecture, we have been able to effectively scale to 4096 TPU nodes, and to support a per-job choice of either 3D torus or 3D twisted-torus [7] topology.

The OCS is a dynamically configurable $N \times N$ switch based on an array of micro-electromechanical systems (MEMS) mirrors that can switch in milliseconds. Each OCS allows programmable cross-connect creation (xconnect) between any pair of ports on the (logical) north side of the switch to the (logical) south side. Once a connection between an N_i to S_j port is made, a dedicated ICI link connection is established such that optical signal from N_i can only be routed to S_j and vice versa, until these ports are reconfigured in some different permutation.

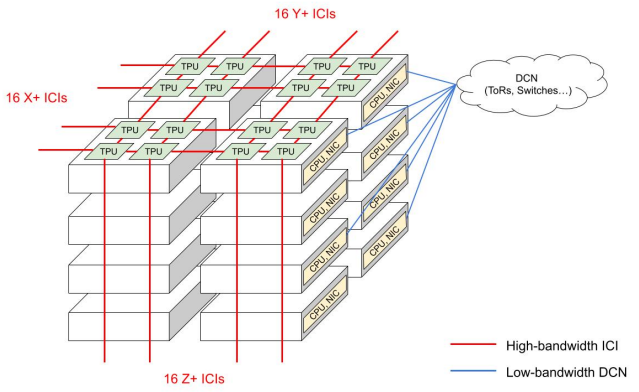


Figure 3: A 4x4x4 cube consists of 16 TPUv4 machines, each of which organizes 4 TPUs in a 2x2x1 mesh. The TPUs in a cube are interconnected over ICIs along *X*/*Y*/*Z* dimensions, with 16 optical links per cube face for OCS xconnect.

TPUv4 compute resources are organized at the granularity of multi-machine *cubes*. Each individual TPU machine has a CPU tray and a TPU tray, linked over PCIe. Each TPU tray has 4 TPUv4 chips arranged in a 2x2x1 ICI mesh; 16 TPU machines are grouped together as one datacenter rack; and the ICI links within the rack are interconnected over ICI to form a 4x4x4 mesh. This ensemble is a cube.

The optical switches interconnect multiple cubes to form larger ICI topology shapes with one or more cubes in each of the three dimensions. Each 3D cube exposes 16 ICIs on each face of the *X*/*Y*/*Z* dimension to the optical switches, totaling 96 ICIs per cube. A TPUv4 supercomputer consists of 64 cubes, with a total of 6144 optical ICI links connected to 48 distinct optical circuit switches. The lower-bandwidth CPU-side datacenter network is managed separately [25, 29].

TPUv4's OCS configurability greatly improves availability. Training jobs can use any cubes even if they are not physically contiguous, which mitigates resource fragmentation from competing jobs. Hardware failures remove the affected cube(s) from the resource pool but allow continued operation using healthy cubes. The 16-machine granularity for fault-tolerance was chosen to balance convenience (per rack deployment, power and networking) while retaining a relatively small blast radius in case of failure.

Reconfigurability is managed by the accompanying software infrastructure. Each job launch induces the software to establish a unique OCS xconnect depending on the required topology and cube selection. The large number of chips, links, and switches also requires automatic fault diagnosis, recovery, job rescheduling, and fault-tolerant ICI routing.

Using OCS scales TPUv4 pod with low cost: the OCS and optical fiber costs are $< 5\%$ of a TPUv4 pod's total capital cost, and their operating power is $< 3\%$ of a pod's total power. The capital and operating cost of TPUv4 OCS supercomputer is considerably lower than the alternative of scaling with packet switches such as Infiniband [18].

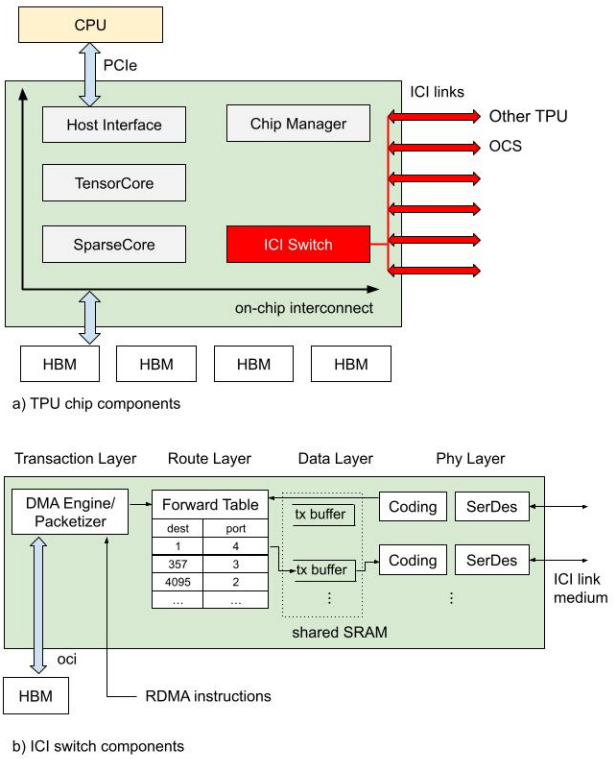


Figure 4: TPUv4's ICI switch implements layered, programmable ICI protocol.

2.3 Programmable ICI Protocol

TPUv4's ICI protocol is designed to be programmable so that software can tackle the operational complexity of reconfigurability and resilience. A TPUv4 pod is one ICI domain, where any pair of TPUs can RDMA to each other. Each ICI link can carry 50GBps uni-directional bandwidth. TPUv4 adopts a 3D ICI network topology for high bisection throughput, large system scale and low latency while maintaining low cost and supporting workload parallelization via collectives.

As shown in Figure 4, each TPUv4 chip has some compute, some high-bandwidth memory and an ICI switch that implements various ICI protocol layers. The ICI protocol facilitates per-job network partitioning, where connectivity, addressing, routing, and flow control are set up for each job, and where user sessions do not cross job boundaries. In this way, each job has exclusive ownership of all the links it uses, increasing security and removing additional system complexity for network sharing and congestion control. Table 1 shows the protocol layers and their corresponding software agents. From the bottom up, these are:

- **Physical Layer**: the SERDES, PCS, and link auto-establishment modules build a high-speed link, despite the inevitable presence of transmission errors. The Pod Manager controls xconnect of a physical channel by rotating OCS MEMS mirrors, and an on-chip manager automatically initializes and configures the physical links. The `healthd` daemon running in every TPU machine's Linux system container continuously reads link quality and connectivity signals to track hardware health.

Layer	Functionality	S/W Agent	ISA Visible?
Transaction	RDMA	XLA	**yes**
Routing	packet forwarding	`libtpunet`	*hint*
Data	link enable, flow control, retry, ordered delivery	`libtpunet` `healthd`	no
Physical	link xconnect port training	pod mgr, chip mgr, `healthd`	no

Table 1: ICI protocol layers.

- **Reliable Data Layer**: Packets are delivered in-order with automatic retransmit when data is lost at the physical layer, thus hiding the unreliable characteristics of the physical layer. Link-level, credit-based flow control is enforced. An *enabled* data layer signifies a ready-to-use ICI user session; prior to becoming ready, the system clears all data buffers to ensure we eliminate any architectural state pollution from prior ICI sessions. If one end of an enabled data layer is down, we automatically bring down the other end of the link to ensure a functional session. The `libtpunet` issues session start/stop commands, and adjusts optimal flow control buffer sizes. The privileged `healthd` machine daemon can explicitly disable a data layer link, forbidding its usage by any user session, in the case of an online link recovery (§3.6.3).

- **Routing Layer**: Packet forwarding tables are programmed by `libtpunet` with global load balancing. Each packet in a RDMA instruction goes from a source to destination TPU, indexing into the forwarding tables in each chip by the destination chip ID. The detailed routing policies are hidden from the ISA abstraction, although the `libtpunet` library can provide hints to the compiler to help guide program optimization.

- **Transaction Layer**: Compiler-generated RDMA instructions initiate hardware-mediated transfers which read data from memory and feed it to the ICI switch. A transaction spanning a group of individual RDMAs forms a collective communication operation.

Using a software-programmable ICI protocol stack allows us to flexibly cope with the complexity of a resilient 4096-node supercomputer, while allowing hardware to deal with real-time control of the links and offer high-bandwidth low-latency data transmission.

3 Automating Supercomputer Management

In the following we provide an overview of the end-to-end software infrastructure that we use to launch TPUv4 ML

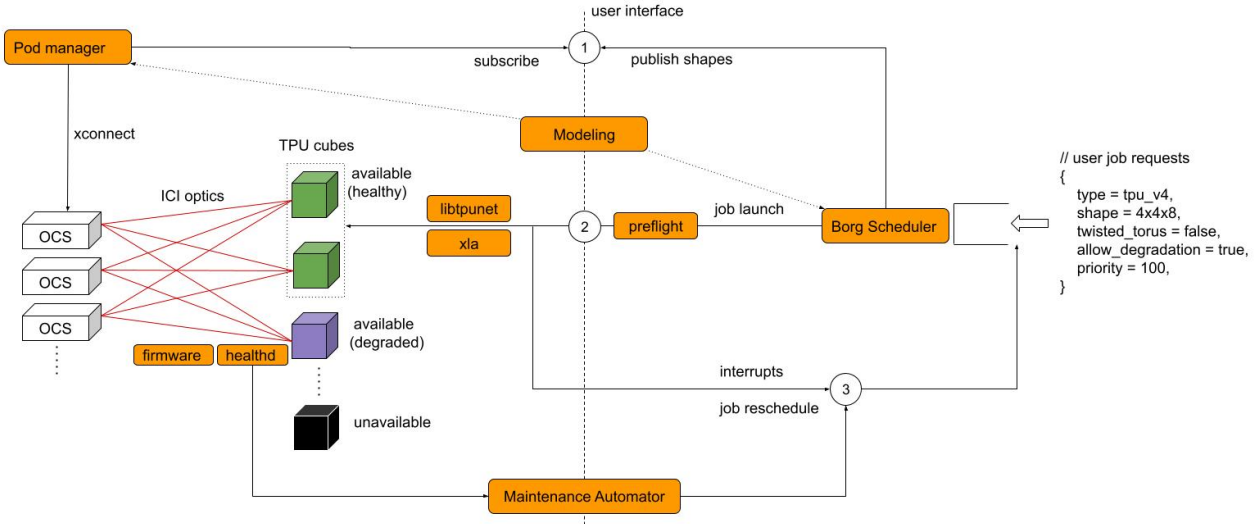


Figure 5: A TPUv4 job's life-cycle: the Pod Manager cooperates with the Borg scheduler to ask OCS to xconnect cubes, after which `healthd` preflight runs and `libtpunet` sets up the ICI network. XLA compiles programs with a distributed shared-memory system abstraction. In case a failure is detected, running jobs can be automatically interrupted and rescheduled.

training jobs, and to subsequently monitor and manage their life-cycle (see also Figure 5 for a summary).

3.1 Overview

When a user wishes to launch a large job[1] on a TPUv4 supercomputer, they specify their desired 3D *slice topology* in the form $(4x, 4y, 4z)$, along with other metadata. The Borg cluster scheduler [31] receives all such requests and queues them pending resource assignment. Once a job becomes eligible for scheduling, Borg will select a prospective set of cubes and then publish a xconnect request.

The Pod Manager periodically polls Borg to learn about any pending xconnect requests. For each one, it instructs the pertinent OCS switches to rotate their MEMS mirrors to establish the optical ICI physical channels. Assuming all OCS xconnects complete correctly, the Pod Manager sends a confirmation to Borg.

With Pod Manager's approval, Borg then dispatches the job binaries to the selected set of TPU machines. A preflight health check is first run to guarantee full hardware health for each TPU machine (any failures lead to Borg rescheduling onto different cubes). Following this, the ICI network is set up by `libtpunet` (i.e. validating the physical and link layers, and programming forwarding tables).

The XLA TPU compiler [3] takes the slice topology abstraction built by `libtpunet` and generates auto-parallelized TPU programs for distributed training. On each machine, the compiled TPU binary will be sent over PCIe to the TPU after which it can be executed. The above workflow is common to all ML frameworks, including TensorFlow [4], Jax [1] and Pathways [5].

During training, fleet maintenance services continuously monitor the hardware and software health of all the TPU machines. Any detected abnormality triggers a notification to Borg, which in turns notifies any affected running jobs so they can write an up-to-date model checkpoint (if possible). Once a job is rescheduled, it resumes from the latest model checkpoint. The faulty hardware is identified and sent to a repair workflow for diagnosis and repair.

The following sections describe this software infrastructure in some more detail.

3.2 Supercomputer Modeling

The foundation of the supercomputer software stack is a datacenter model [23] that reflects the TPUv4 machines and all related components. The model is stored in a dedicated database whose schema allows us to represent a graph of entities including racks, switches, RPC endpoints, and others. To support TPUv4 supercomputers some key entities are the TPUv4 cubes (i.e. the 16 machines with their trays and chips and static ICI inter-connection topology) as well as the ICI cabling from cubes to OCS along with additonal optics metadata. Once constructed, the model sets up the *intent* for cube deployment, job scheduling, OCS xconnect, network setup, and health checks. The model is consumed by both Borg and Pod Manager to serve as the source of truth for a each particular supercomputer configuration.

TPUv4 topologies are statically modelled up to cube size, as larger shapes require dynamic cube xconnect. For example, a single machine is a `2x2x1`, two adjacent machines can be combined to form a `2x2x2`, and so on up to `4x4x4`.

[1] We also support smaller (sub-cube) jobs. In these cases no OCS configuration is required, but the rest of the workflow is similar.

3.3 Cluster Scheduling

The Borg cluster scheduler [31] is responsible for assigning appropriate machines to each TPUv4 job. There are many Borg *cells* in Google's worldwide datacenters, and each cell may include several TPUv4 supercomputers. Each cell is managed by N replicated Borg service instances which, in combination, provide one logical Borg instance we call Borg Prime which includes a cluster scheduler.

The cluster scheduler combines the intended configuration (from the datacenter model) with its current view of the world to organize all of the TPUv4 resources it is responsible for into schedulable machine groups. Users generally select a cell in which launch their jobs, and indicate which 3D topology to use to train their model. Borg matches each user request to a set of *feasible* (usable) machines and creates a proposed assignment. In the case of multi-cube jobs, Borg publishes the proposed set of cubes to the Pod Manager and waits for it to signal xconnect success before proceeding.

Each TPU machine runs a `borglet` daemon that cooperates with Borg Prime to handle job life-cycle management. After Pod Manager approval, Borg Prime instructs each `borglet` in the assigned cubes to create a task container with the machine's TPUv4 devices[2] exposed in the task's container. The `borglet` then launches a sequence of binaries in the container, starting with the pre-flight check and finishing with the user binary.

Borg Prime and `borglet` combine to manage the response to events such as planned maintenance (e.g. firmware or software upgrades) or unexpected hardware faults. These events are aggregated from different sources, e.g. `borglet` is notified about critical local machine faults by the `healthd` daemon, and passes the details up to Borg Prime; the Pod Manager similarly forwards details about any critical OCS problems. Borg Prime also receives notification about less critical events from the Repair Automation System and the software Package Manager. In all cases, affected TPU machines are marked as unavailable, evicting any running jobs with notice, and excluding pending jobs from landing on them until things have been resolved.

Borg Prime implements priority scheduling (for higher and lower priority jobs). To help with fragmentation, Borg Prime can also choose to preempt a running workload (e.g. to relocate multiple sub-cube jobs to fit into a smaller number of cubes, or to move multi-cube workloads to a different pod so as to accommodate very large jobs). This happens in a controlled fashion, ensuring that jobs are minimally and fairly impacted.

[2] Very small jobs may use just a single TPU. In such cases, `borglet` will restrict container access to one device, and disable on-host ICI links.

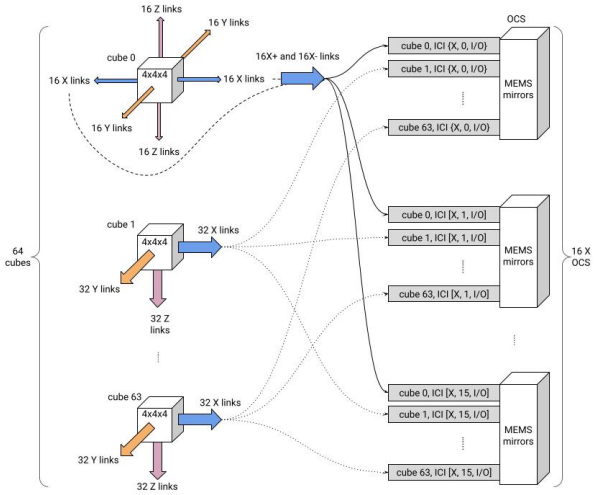


Figure 6: Each of the 64 cubes contributes two optical ICI links from two opposite sides of a ring to each of the 48 OCSes. 16 OCSes are needed for each dimension.

3.4 Pod Manager

Pod Manager is a highly available service critical to a TPUv4 system. It runs on dedicated network control servers that are independent of Borg, and interacts with clients such as Borg and OCS switches over the Google control plane network. The Pod Manager has two main functions: creating OCS xconnects to configure the user-requested TPU topology, and real-time monitoring of pod health .

The Pod Manager relies exclusively on model data (§3.2) to configure its services. It periodically polls the network model service for the latest information about the specific TPUv4s that it is serving, such as OCS endpoints and machines that are planned to be deployed. The OCS xconnect plan and continuous health check for every job is derived from the model. Using a model-driven Pod Manager design allows gradual deployment of a full TPUv4 supercomputer while having a subset of cubes available to customers early on.

The Pod Manager is replicated for high availability: a primary instance serves outside requests, while the remaining instances operate in hot stand-by mode so that one can quickly be elected primary if necessary. Our stand-by scheme relies on each follower continuously receiving copies of the checkpoints (also persisted externally), meaning fail-over is generally very fast. We rely on this for non-disruptive software upgrades and to tolerate hardware and software crashes.

The Pod Manager also serves as a central hub for a TPUv4 supercomputer's health monitoring. The service periodically checks the hardware health of all the optical switches by querying the OCS hardware over RPC. This telemetry is exported to Google's fleet-wide health management system (§3.6) and also used in real-time to guide fault-tolerant ICI routing optimizations (§4.2).

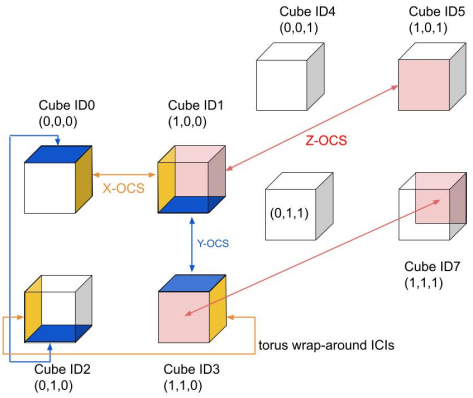


Figure 7: xconnect of an 8-cube 8x8x8 torus. Each OCS needs to pair 16 north/south OCS ports. Cube faces with the same color are interconnected to form multi-cube torus topologies.

3.4.1 **Torus** xconnect

Each 3D cube exposes 16 optical ICIs on each of the 6 faces of the X, Y, Z dimensions for a total of 96 ICI links. Pod Manager assigns each of these links a unique identifier $\{cube_id, dim, index, polarity\}$, where *cube_id* is a Google-wide cube UID, *index* ranges from 0–15 indicating the position within a cube face, and *polarity* can be *in* or *out*. 48 OCSes are use to xconnect these ICIs, 16 for each of the dimensions. Pod Manager gives each OCS a unique identifier $\{dim, index\}$ matching the ICI optical cables.

Fig 6 illustrates this cable connection scheme. Each OCS provides 128 ports for optical ICI connection from the cubes, allowing full connection of a single port for all 64 cubes. This scheme allows any $(4x, 4y, 4z)$ TPU topology shape to be formed, including a 4x4x4 single-cube full torus. Note that since the connected ICI and OCS have the same $\{dim, index\}$ parameters, if an OCS becomes unavailable, every cube observes one broken ICI link with the same $\{dim, index\}$ parameter.

To perform a job's cube xconnect, Pod Manager leverages its internal representation of the optical ICIs and OCSes. Fig 7 illustrates the process for a 8x8x8 torus:

- *Step 1:* Borg publishes the set of UIDs and the desired topology. Pod Manager assigns a 3D coordinate to each cube based on the topology; any cube can be chosen for any coordinate since the Pod Manager can instruct the switches to apply arbitrary port-port xconnect. For each (4x, 4y, 4z) shape, there are $x \cdot y \cdot z$ cube coordinates.

- *Step 2:* Pod Manager computes the inter-cube neighbor information based on the assigned coordinates; e.g. $(0,0,0)$ is adjacent to $(1,0,0)$ along `Xout` and `Xin`.

- *Step 3:* Pod Manager tells OCS to xconnect the ICIs $\{cubeA, dim, index, in\}$ and $\{cubeB, dim, index, out\}$ between every pair of adjacent cubes. For any topology, all 48 OCSes need to execute commands to xconnect ICI links, as all 16 ICI ports along all 3 dimensions and polarity must be connected to their remote neighbor. Each OCS can execute a variable number of commands, depending on the topology, although the OCSes for a single dimension always execute the same number. For 8x8x8 there are 8 cubes, and each OCS along the x, y, z dimensions must connect 8 pairs of ports to form the torus (one per cube).

- *Step 4:* The required connections are compared against the current configuration, cached inside Pod Manager, and we filter out any connections that will remain the same. RPCs are sent to xconnect the new connections.

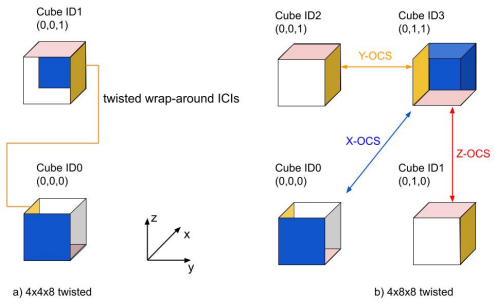


Figure 8: xconnect of (a) 2-cube 4x4x8 twisted-torus, and (b) 4-cube 4x8x8 twisted-torus. Cube faces with the same color are interconnected by OCS to form twisted-torus wrap-around ICIs. The twisted dimensions always have shorter ring sizes.

During any of the steps above, if Pod Manager determines that any port connections are infeasible, e.g., due to a hardware problem, Pod Manager will indicate this to Borg and reject the proposed set of cubes. Borg can then propose a new set of cubes set for the user's job.

3.4.2 **Twisted-torus** xconnect

In additional to a regular torus topology, we support the use of a *twisted torus* topology [7] if requested by the user. In TPUv4 twisted-torus [18], the wrap-around links are shifted with a vector offset, depending on the overall job shape. TPUv4 supports two families of twisted-torus topology: $(4k, 4k, 8k)$ and $(4k, 8k, 8k)$. Figure 8 illustrates how they are built.

For $(4k, 4k, 8k)$ shapes, the asymmetry grows along the Z dimension, with the X and Y dimensions being identical with the same size (i.e. half the Z dimension). The X and Y wrap-around links are shifted by a $(0, 0, 4k)$ vector offset.

For $(4k, 4k, 8k)$ shapes, the asymmetry grows along both the Y and Z dimensions, and the X dimension has the smaller size (i.e. half of the Y and Z dimensions). The X wrap-around links are shifted by a $(0, 4k, 4k)$ vector offset.

Cube coordinates are identical in both the regular and twisted torus case, but the latter changes which cube faces are deemed to be adjacent, and ultimately leads the Pod Manager to instruct each OCS to xconnect different north/south ports.

3.5 libtpunet

Once the ICI physical channels have stabilized after xconnect completion, Borg dispatches the job binaries to the host machines. The libtpunet library runs within a user's job to set up the ICI network (data and routing layer).

The first step is topology discovery. Discovery is a bottom-up process that scans each TPU's local neighbor ICI connectivity information, and runs breadth-first-search to ensure that the configured global topology matches the user request. In this process, each TPU in a job is assigned a unique chip id; this id is exposed as part of the ISA interface for RDMA instructions. The discovery process also identifies any faults that may need to be routed around, or exposed to users in the system abstraction. Topology discovery complements the intent-driven modeling of the network.

libtpunet then computes and programs the forwarding tables of each TPU based on the information curated during topology discovery. The forwarding tables are part of a job's globally optimized routing solution (more details in §4).

Along with ICI routing programming, libtpunet sets up the link-level flow-control buffer size, in proportion to the the link RTT. libtpunet also programs the configuration of consistent clocking on a job's distributed TPU set. The clock configuration is generated using a minimum spanning tree, factoring in the longest RTT of any ICI path. Using a consistent clock enables precise timestamps for performance tracing and debugging.

Finally, libtpunet starts an *ICI session*, allowing the use of various compiler-generated collective ops with RDMA. This is done by synchronously enabling the data layer of each ICI link across its two ends. An ICI handshake is performed in hardware to confirm the reliable data link enable request is initiated from both ends of the link.

libtpunet stays active through the job's lifetime to monitor the health of the ICI session. If any TPU observes an error, the link layer comes down, or the driver panics, a PCIe MSI-X interrupt is raised to libtpunet, which notifies Borg to initiate rescheduling.

3.6 Hardware Maintenance and Recovery

At global fleet scale, disruptive maintenance events (e.g. hardware repair or replacement, or critical software/firmware upgrades) occur relatively frequently. To maximize overall maintenance efficiency, Google operates a fleet automation system. Its remit covers hardware failure diagnosis, a hardware recovery workflow, and system software package installation (e.g. host kernel or device firmware).

Events generated by the fleet maintenance automation system send notifications to Borg to evict running jobs on impacted machines; any evicted jobs are queued for rescheduling. In case of suspected failures, the impacted hardware is sent to a repair workflow that marries automatic diagnosis with technician input if needed. Once the hardware is recovered it flows through an automated QA process before rejoining the resource pool. For our TPUv4 supercomputer we extended this system with continuous TPU hardware health telemetry, explicit preflight checks before job launch and a scheme for on-line ICI link repair.

3.6.1 healthd

We added a healthd daemon on every TPUv4 machine to perform real-time monitoring of hardware parts including the 24 unidirectional ICI links, the PCIe channels between the TPUs and CPUs, and the 4 TPU ASICs themselves. A set of hardware symptoms are defined for each of these components based on the telemetry data gathered by healthd. healthd consumes the same model as Pod Manager which provides the necessary details about monitoring endpoint, firmware, and ICI cable metadata.

For each ICI link, the cable connection and associated link quality are continuously checked against the modeled values and a set of predefined thresholds. Any detected *symptoms* are ranked by their criticality, with severe symptoms leading healthd to notify Borg to evict and reschedule affected jobs.

3.6.2 Preflight Check

A preflight check runs before every user job to ensure hardware is healthy. We currently include two different checkers: an *end-to-end* check validates the TPU hardware by running a mini sample workload, while an *intent-driven* checker validates physical-level hardware metrics against a set of golden "within spec" thresholds. The former provides broad coverage of both the hardware and software components including the TPU driver, firmware and libtpunet which all interact with the underlying chip and ICI; the latter allows detection of less obvious issues such as substandard link quality metrics. If the preflight checks fail, borglet will indicate to Borg Prime that the job should be rescheduled.

3.6.3 Online ICI Link Repair

For TPUv4, ICI link repair can be carried out online, automatically coordinated across the two ends of a link so that recovery can be reliably verified. The two endpoints can span two different machines, or a machine and an OCS switch. The Pod Manager coordinates all ICI network maintenance through *ICI link drains*. A drained ICI link is automatically excluded from user applications, although the TPU compute resources are not impacted (i.e. jobs can still land on the TPU machines providing they do not use the broken ICIs).

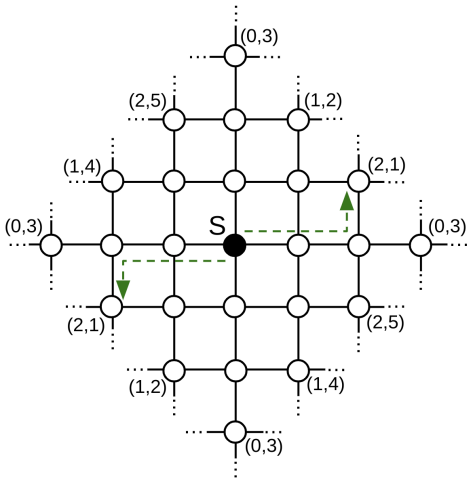


Figure 9: Shortest-path routes from the origin (S) in a 2D `3x6` twisted torus are confined to a diamond-shaped region [7]. Destinations along the boundary are labeled with their coordinates, with each destination appearing at least twice, meaning we need tiebreaking to pick among the shortest paths. For example, there are two possible paths to $(x = 2, y = 1)$, shown as dashed lines, and there are four possible paths to $(0, 3)$.

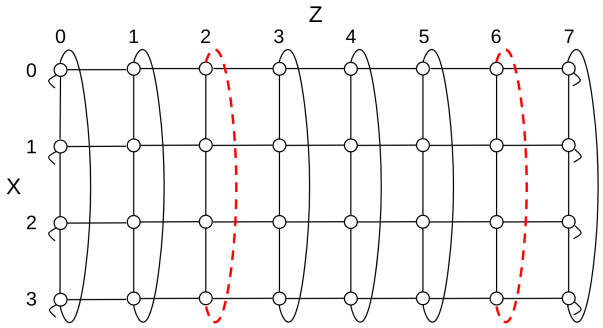


Figure 10: Example of ICI links impacted by an OCS being unavailable along the X dimension of a `4x4x8` torus. The unavailable OCS results in two unavailable X links along one XZ plane of the torus; the other XZ planes are unaffected. Unavailable links are emphasized with dashed red lines. The connectivity of the OCSs creates a periodic fault pattern, where unavailable links repeat every 4 hops along Z. This pattern is due to the OCS connectivity (Figure 6).

4 ICI Routing

We use multi-hop packet routing over high-bandwidth ICI links to provide fast TPU RDMA and collectives. ICI routing allows RDMA packets to be sent between arbitrary pairs of TPUs in the pod, and can work around certain ICI faults. The ICI forwarding tables are programmed once by `libtpunet` at job start-up, and remain fixed over the job's lifetime. Each source-destination pair sends packets along a single predetermined path through the ICI topology.

While simple, this approach is sufficient to achieve high performance on the typical collective communication patterns (e.g. all-gather, reduce-scatter, all-reduce, all-to-all) that arise during parallel decomposition of ML models [33].

There are two cases described in this section where `libtpunet` must carefully select a single path among multiple candidates to satisfy the ICI forwarding table constraints: *tiebreaking* and fault-tolerant *wild-first routing*. We perform path selection off-line using an integer linear programming approach and the results of this optimization are cached. This allows `libtpunet` to quickly load the precomputed solution during ICI network setup.

4.1 Fault-free Routing

When configured for regular torus topology, ICI uses dimension-order routing (DOR) [11]: all packets route one dimension at a time in a fixed order (e.g. X then Y then Z) following a shortest-path from source to destination in the torus. The dimension order is chosen so that longer dimensions of the torus are routed first as described in [8]. DOR is sufficient to balance load for the common traffic patterns of ML jobs. It can also be made deadlock-free with just two virtual channels [10], making it inexpensive to implement.

One complication occurs when a packet needs to travel exactly halfway around a dimension since tn this case, there are two shortest paths to choose from. For example, in an `8x8x8` torus, routing a packet from source $(x = 1, y = 0, z = 0)$ to destination $(x = 5, y = 0, z = 0)$ can travel 4 hops in either the X^+ or X^- direction.

We handle this *tiebreak* case algorithmically: a packet takes the positive direction when the relevant source node coordinate along that dimension is even, otherwise uses the negative direction. In our example, the source $(x = 1, y = 0, z = 0)$ has an odd X coordinate, so tiebreaking chooses X^-. This scheme balances load for common all-to-all traffic patterns.

Routing in the twisted torus also uses DOR and is deadlock free with two virtual channels. However tiebreaking in the twisted torus is more complicated because the dimensions are not separable as in the regular torus: Figure 9 illustrates this using two dimensions for simplicity. We decided to fold handling tiebreaking in the twisted torus case into a more general integer-linear programming framework that also handles fault-tolerant routing (§4.3). This obviates the need to develop an explicit tiebreaking algorithm in this case.

4.2 Fault-tolerant Routing

The reconfigurable ICI architecture is inherently resilient to machine outages due to the ability for dynamic cube selection. In `libtpunet`, further resilience against OCS unavailability events is added by supporting *fault-tolerant routing*. If an OCS is unavailable, a sparse set of links becomes unavailable. As shown in Figure 10, the patterns of unavailable links are

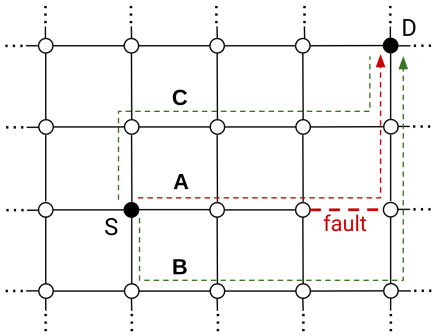


Figure 11: Example routes (dashed paths) between a source S and destination D. The XY dimension order route (A) crosses the unavailable link (red dashed line) so cannot be used. The two possible yXY wild-first routes (B and C) take a single hop in Y before continuing with XY DOR, avoiding the fault.

highly symmetric with respect to the cube granularity as a consequence of cube-OCS connectivity. With a small amount of routing flexibility, packets can avoid the broken links. We optimize routing algorithms for these scenarios off-line.

The path between a source-destination pair selected by dimension-order routing (DOR) becomes unusable if it crosses an unavailable link. In this case, alternative paths are created using *wild-first routing* (WFR). In WFR, a packet is allowed to take at most one wild hop along each dimension before reverting to use DOR to its destination.

Fig 11 shows an example of WFR routing in a two-dimensional torus. In this example, the dimension order is X then Y and only wild hops along Y (either Y^+ or Y^-) can help avoid unavailable links. We use the shorthand yXY to denote a wild-first routing algorithm that takes a wild hop along Y before continuing with XY dimension-order route. The yXY algorithm can avoid any single unavailable link in the X dimension.

There is a *sandwich rule* that captures the fault tolerance of WFR: to avoid a fault in one dimension, hops in another dimension must occur before and after it in the routing algorithm. For the yXY, X is "sandwiched" by hops in Y, so it avoids faults in X. Similarly, the xYX algorithm can avoid faults in Y. Extending to three dimensions, the $xyZYX$ algorithm can avoid a single fault in both the Y and Z dimensions.

The development of WFR was influenced by the microarchitecture of the ICI switches. While beyond the scope of this paper, WFR can be made deadlock-free with two virtual channels with one restriction: the wild hop order must be the reverse of the dimension order. For example, $xyZYX$ is deadlock-free with two virtual channels, but $yxZYX$ is not.

4.3 Offline Route Optimization

The previous sections described situations where multiple candidate paths can be produced, either due to tiebreaking or due to the wild-first routing algorithm. The ICI switch implementation, however, makes use of static forwarding tables, which are programmed with a single path for each source-destination pair. As mentioned previously, we formulate path-selection as an integer-linear program (ILP), calculating solutions offline and caching them for use at runtime.

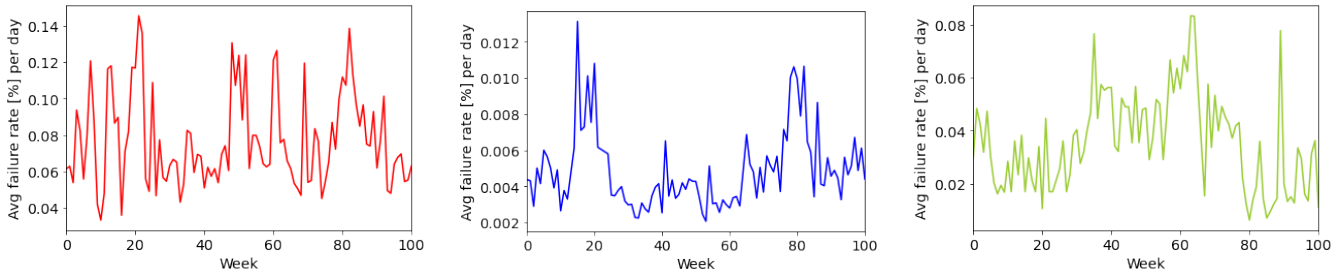
Slice	No faults (GB/s)	1 fault (GB/s)	1 fault (% of no faults)
4x4x4	75.9	70.0	92.2%
4x4x8 twisted	62.1	63.2	101.7%
4x8x8 twisted	54.3	53.7	98.8%

Table 2: Measured all-to-all throughput with (a) all OCSs healthly and (b) one OCS unavailable. The last column shows the throughput of the single-fault case versus the healthy case.

The goal of the ILP is to maximize the throughput of a predefined traffic pattern by solving a maximum concurrent flow problem [27]. All-to-all is typically chosen as the traffic pattern and supplemental constraints ensure other collectives (e.g. all-reduce) perform well. Per-path variables are constrained to Boolean values to adhere to the static routing constraints, with exactly one path per source-destination pair.

In both the fault-free and fault-tolerant cases, the ILP is designed to exploit translational symmetry to reduce the number of variables [30]. This makes finding optimal solutions for practical network sizes tractable. The torus and twisted torus are both vertex symmetric, so a single set of path variables can be used for a canonical source and then translated to all other sources. When an OCS is unavailable, the fault pattern is periodic with cube granularity, as was shown in Figure 10. While the resulting topology is less symmetric than the fault-free cases, the ILP can be still restricted to a set of canonical sources. This also enables a single canonical case to be solved offline and cached. The canonical fault pattern, trivially translated from the actual fault pattern, is initialized during network setup.

Table 2 compares measured all-to-all throughput in fault-free and single OCS failure scenarios when the forwarding tables are optimized using this methodology. For the regular torus networks, a single unavailable OCS reduces the ideal all-to-all performance to $15/16 \approx 93.4\%$ of the fault-free network. This corresponds to losing one of the 16 links along one face of the 4x4x4 cubes that connect to the OCSs. Interestingly the twisted torus shows better resilience, with the 4x4x8 performance *improving* slightly in the presence of an unavailable OCS . This is because of the flexibility offered by tiebreaking: by balancing different tiebreaking paths, traffic can be shifted from one dimension to another. This balancing is not possible in the regular torus. The slight improvement in the 4x4x8 case is a good illustration that the ILP formulation is only a proxy for real world performance.



(a) Machine daily failure rate. (b) ICI link daily failure rate. (c) OCS daily failure rate.

Figure 12: Weekly statistics of a supercomputer's hardware failure and recovery, including TPU machines, ICI cables, and OCS.

5 Fleet Statistics

In this section, we describe Google's fleet experience of operating TPUv4 supercomputers over the past two years. We focus on the software stack's automatic management of OCS xconnect, faults, and overall system availability.

5.1 Cube Reconfigurations

Thousands of training jobs are submitted to Google's TPUv4 supercomputers every day. Figure 13 shows a sample supercomputer's OCS xconnect actions over two months, correlated with the pod's number of admitted jobs. A higher number of jobs normally incurs more OCS xconnect changes because the Pod Manager updates ICI port xconnect for each job on arrival. We also see OCS xconnect changes when there are very large and/or long running training jobs which will experience reschedules to handle maintenance and failures. Overall, TPUv4 supercomputers function reliably with many tens of thousands of OCS xconnect changes per pod per day.

Training jobs can vary drastically in size and system topology, ranging from sub-cube mesh jobs for small scale experiments to large jobs that use almost the entire pod for LLM pretraining. We anecdotally observe that many embedding-heavy recommendation models adopt the twisted-torus topology, and some transformer-based models use model parallelism across more irregular torus topologies.

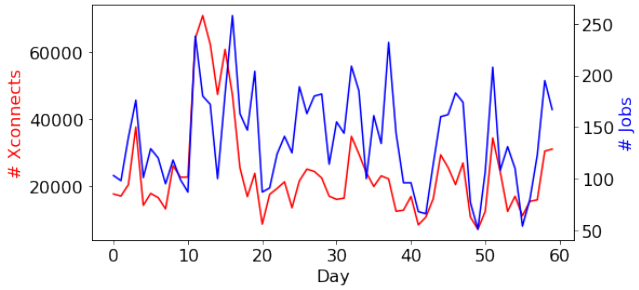


Figure 13: The OCS xconnect actions and jobs admitted by a TPUv4 supercomputer over two sample months.

5.2 Hardware Maintenance Automation

TPUv4's reconfigurability and fault-tolerant routing allow for resiliency against machine and OCS outages. Figure 12 shows the average failure rate of different hardware components in each supercomputer.

Faults are diagnosed at the TPU machine, ICI link, and OCS levels. The Pod Manager and `healthd` automate the repair and recovery process. In an average supercompuer, each day, 0.08% of the TPU machines, 0.005% of the ICI cables, and 0.04% of the OCS experience a failure. While these values are small, the number of jobs that are impacted by hardware outages is non-trivial because each supercomputer has a large number of machines, ICIs, and OCS. Machine and ICI outages are automatically tolerated by reconfiguring jobs to use spare healthy cubes. An OCS outage has larger blast radius as it can impact all cubes in a supercomputer. Fault-tolerant ICI routing lets us tolerate OCS outages with some performance impact; and we priority recovery time for OCS components compared with others to minimize this.

5.3 Fault-tolerant Jobs

In our experience to date, 95% of all TPUv4 training jobs opt in to fault-tolerant ICI routing so they can be resilient to OCS outages; the remaining jobs opt-out to rule out performance non-determinism caused by different routing strategies. Figure 14 shows the ratio of all fleet-wide jobs actively running with fault-tolerant routing across a 8-month sample period. In general at any time, fewer than 2% of the jobs are running with fault-tolerant routing. This quantity is highly correlated with OCS maintenance events and the per event recovery time. The spike around day 60 is due to a planned fleet-wide upgrade of OCS parts to improve reliability.

Fault-tolerant ICI routing comes with a performance penalty due to more congested traffic around faulty links. The load imbalance affects collective operations including all-to-all and all-reduce. We measured the performance impact across a range of key Google workloads spanning *Recommendation Models* (RM), *Large Language Models* (LLM), and BERT-based [13] models. Table 3 summarizes our results.

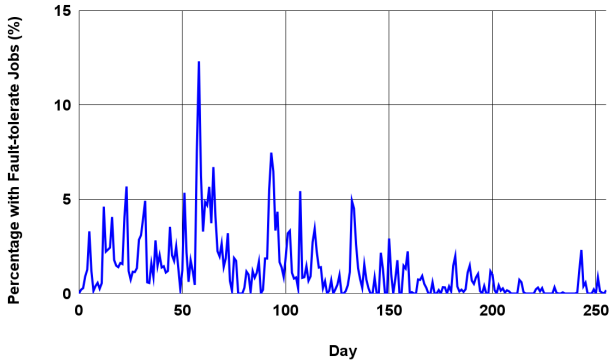


Figure 14: Percentage of jobs using fault-tolerant routing (OCS outage resiliency) over an 8-month window.

For all-to-all heavy workloads, the step time degradation is not significant because the offline routing optimizer has minimized, if not improved, all-to-all performance. This is particularly true for embedding-heavy twisted-torus shapes. The all-reduce workloads experience higher performance impact because the nearest-neighbor communication pattern receives 50% throughput hit. The impact on all-reduce operation can be improved by smarter overlapping between compute and communication. Overall, all workloads experience a small slowdown in step time.

6 Related Work

The architecture decision to use OCS for TPUv4 is discussed in [18], while [20] and [19] evaluate the design of prior generation static TPU pods. This paper describes the software ecosystem for TPUv4 and how it achieves resilience at scale. The usage of OCS in production-scale data center networks was described in [25], discussing considerations for scalability, cost, and topology engineering. This paper focuses on OCS use TPUv4 supercomputers.

Previous work has covered circuit switching for supercomputers [28] and proposed topology engineering for ML training [32]. Nvidia uses a 2-tier NVswitch-based fat tree network over NVlink for inter-GPU collectives. These represent a different design point compared with ours: OCS simplifies the ICI network design compared to introducing packet switches because it establishes dedicated physical channels without the need to control shared traffic, while the lower purchasing price and stand-by power also reduces operating cost [20].

Twisted tori are due to [6, 26]; the specific $(4k, 4k, 8k)$ and $(4k, 8k, 8k)$ twisted torus shapes supported TPUv4 conform to those of [7]. Finally software-defined datacenter networks have been described extensively in the literature (e.g. [14, 16, 25, 29]). To our knowledge, we are the first to describe this approach for an exascale supercomputer.

Workload	Topology	Step Time Slowdown
RM-1	4x4x8 twisted	0.5%
RM-2	4x4x8 twisted	3%
RM-3	4x4x8 twisted	3.9%
RM-4	4x4x8 twisted	8.6%
RM-5	4x4x8 twisted	8.3%
RM-6	4x4x8 twisted	4.7%
LLM-1	4x4x8	2.6%
BERT-1	4x4x4	1.2%
BERT-2	4x8x8	3.2%

Table 3: Performance impact from fault-tolerant ICI routing.

7 Future Work

Our main short term priorities are improving the performance and recovery overhead of TPU pods: ML supercomputing hardware is in high demand, and every little helps. In future, as well as supporting increased line rate for ICI links, we plan to introduce a randomized routing capability to ICI switches to enables better load balancing for both torus and twisted torus topologies in the presence of faults, particularly for nearest-neighbor communication patterns.

We also plan tighter integration between OCS-based configurability and workload reconfiguration by allowing jobs to continue mostly unaffected by failures. Our approach here is to provision a hot-standby cube in response to an outage event, and directly migrate accelerator state to the new TPUs without ever writing a persistent checkpoint. This work involves changes to the Borg scheduler (to provision on demand), `libtpunet` (to dynamically adjust a built ICI session) and the Pathways ML runtime [5] (to manage state transfer).

8 Conclusion

The TPUv4 supercomputer is an exascale 4096-chip computing system that addresses the availability and scalability challenges of fast-paced ML model evolution. TPUv4 offers approximately 2.1× performance compared to the previous generation, but also features cube-level *reconfigurability* based on optical circuit switching and uses fault-tolerant ICI routing to allow operation if switches fail.

This paper has described the end-to-end software infrastructure for TPUv4 that provides flexibility for topology, routing, scheduling, interrupting, monitoring, and hardware health management. TPUv4's software-defined ICI networking approach enables strong fault resiliency to machine and switch outages at scale. The software has been operating in production since 2020, running TPUv4 supercomputers for both Google Cloud clusters and internal users, and sustaining 99.98% system availability,

9 Acknowledgements

We wish to thank our collaborators: Xiangyu Dong, Sabastian Mugazambi, Sameer Kumar, Andy Koch, Jianqiao Liu for the project inception, and Wenbo Zhao, Hongji Li, Rajendra Gottipati, Yuechao Pan, Justin Duan, Peter Gavin for their work on continuous measurement and monitoring. We also thank Kais Belgaied and Varinder Singh for their support during the design and development of our software infrastructure. Finally, special thanks go to Robert Hundt, Norm Jouppi, Aamer Mahmood, and the anonymous NSDI reviewers for their feedback and suggestions on the paper.

References

[1] JAX: Autograd and XLA. https://github.com/google/jax [Accessed: 2023-04-26].

[2] The TPUv4 Exaflop ML Supercomputer. https://cloud.google.com/blog/topics/systems/tpu-v4-enables-performance-energy-and-co2e-efficiency-gains [Accessed: 2023-04-26].

[3] The XLA TPU Compiler. https://github.com/openxla/xla [Accessed: 2023-04-26].

[4] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[5] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent El Shafey, Chandramohan A. Thekkath, and Yonghui Wu. Pathways: Asynchronous Distributed Dataflow for ML, 2022.

[6] Wendell J Bouknight, Stewart A Denenberg, David E McIntyre, JM Randall, Amed H Sameh, and Daniel L Slotnick. The Illiac IV system. *Proceedings of the IEEE*, 60(4):369–388, 1972.

[7] Jose M Camara, Miquel Moreto, Enrique Vallejo, Ramon Beivide, Jose Miguel-Alonso, Carmen Martínez, and Javier Navaridas. Twisted torus topologies for enhanced interconnection networks. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1765–1778, 2010.

[8] Dong Chen, Noel A. Eisley, Philip Heidelberger, Robert M. Senger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David L. Satterfield, Burkhard Steinmacher-Burow, and Jeffrey J. Parker. The IBM Blue Gene/Q interconnection network and message unit. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2011.

[9] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[10] W. J. Dally and C. L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. Comput.*, 36(5):547–553, may 1987.

[11] William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[12] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2019.

[14] Andrew D Ferguson, Steve D Gribble, Chi-Yao Hong, Charles Edwin Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, et al. Orion: Google's Software-Defined Networking Control Plane. In *NSDI*, pages 83–98, 2021.

[15] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. Failures in large scale systems: long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017.

[16] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, et al. B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined WAN. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 74–87, 2018.

[17] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan

Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[18] Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th annual international symposium on computer architecture*, 2023.

[19] Norman P Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A domain-specific supercomputer for training deep neural networks. *Communications of the ACM*, 63(7):67–78, 2020.

[20] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, 2017.

[21] Sameer Kumar, James Bradbury, Cliff Young, Yu Emma Wang, Anselm Levskaya, Blake Hechtman, Dehao Chen, HyoukJoong Lee, Mehmet Deveci, Naveen Kumar, Pankaj Kanwar, Shibo Wang, Skye Wanderman-Milne, Steve Lacy, Tao Wang, Tayo Oguntebi, Yazhou Zu, Yuanzhong Xu, and Andy Swing. Exploring the limits of Concurrency in ML Training on Google TPUs, 2021.

[22] Sameer Kumar and Norm Jouppi. Highly Available Data Parallel ML training on Mesh Networks, 2020.

[23] Jeffrey C. Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with Modeling Network Topologies at Multiple Levels of Abstraction. In *17th Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[24] OpenAI. GPT-4 Technical Report, 2023.

[25] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter evolving: Transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of ACM SIGCOMM 2022*, 2022.

[26] Carlo H Sequin. Doubly twisted torus networks for VLSI processor arrays. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 471–480, 1981.

[27] Farhad Shahrokhi and D. W. Matula. The Maximum Concurrent Flow Problem. *J. ACM*, 37(2):318–334, apr 1990.

[28] John Shalf, Shoaib Kamil, Leonid Oliker, and David Skinner. Analyzing ultra-scale application communication requirements for a reconfigurable hybrid interconnect. In *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 17–17. IEEE, 2005.

[29] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network. *ACM SIGCOMM computer communication review*, 45(4):183–197, 2015.

[30] Brian Towles, William J. Dally, and Stephen Boyd. Throughput-Centric Routing Algorithm Design. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '03, page 200–209, New York, NY, USA, 2003. Association for Computing Machinery.

[31] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[32] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. TopoOpt: Co-optimizing Network Topology and Parallelization Strategy for Distributed Training Jobs, 2022.

[33] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. GSPMD: general and scalable parallelization for ML computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

NN-Defined Modulator: Reconfigurable and Portable Software Modulator on IoT Gateways

Jiazhao Wang[1], Wenchao Jiang[1], Ruofeng Liu[2], Bin Hu[3], Demin Gao[4], and Shuai Wang[5]

[1]*Singapore University of Technology and Design,* [2]*University of Minnesota,* [3]*University of Southern California,* [4]*Nanjing Forestry University,* [5]*Southeast University*

Abstract

A physical-layer modulator is a vital component for an IoT gateway to map the symbols to signals. However, due to the soldered hardware chipsets on the gateway's motherboards or the diverse toolkits on different platforms for the software radio, the existing solutions either have limited extensibility or are platform-specific. Such limitation is hard to ignore when modulation schemes and hardware platforms have become extremely diverse. This paper presents a new paradigm of using neural networks as an abstraction layer for physical layer modulators in IoT gateway devices, referred to as NN-defined modulators. Our approach addresses the challenges of extensibility and portability for multiple technologies on various hardware platforms. The proposed NN-defined modulator uses a model-driven methodology rooted in solid mathematical foundations while having native support for hardware acceleration and portability to heterogeneous platforms. We conduct the evaluation of NN-defined modulators on different platforms, including Nvidia Jetson Nano and Raspberry Pi. Evaluations demonstrate that our NN-defined modulator effectively operates as conventional modulators and provides significant efficiency gains (up to 4.7× on Nvidia Jetson Nano and 1.1× on Raspberry Pi), indicating high portability. Furthermore, we show the real-world applications using our NN-defined modulators to generate ZigBee and WiFi packets, which are compliant with commodity TI CC2650 (ZigBee) and Intel AX201 (WiFi NIC), respectively.

1 Introduction

In recent years, we have observed the swift progression of the Internet of Things (IoT), transitioning from theoretical concepts to tangible reality. IoT's objective is to connect many devices, such as sensors and actuators, globally via various Physical (PHY) layer technologies. These technologies are tailored to suit IoT connections based on factors like throughput, power consumption, and coverage area. For instance, IEEE 802.15.4 [19] is specifically designed for short-range, low-rate IoT connections, while NB-IoT [7] is intended for broader, low-power IoT connections. The IoT gateway functions as a central hub, establishing wireless communication links with IoT devices and bridging them to the rest of the Internet. Within PHY of IoT communication (Figure 1a), the modulator plays a vital role in generating signals for data transfer to transmit over the air.

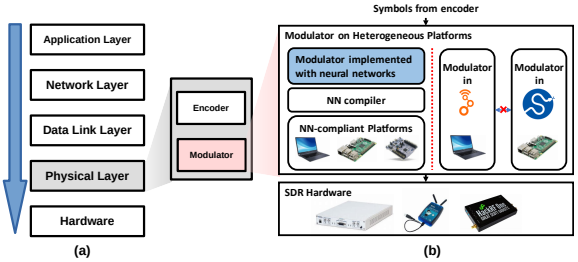


Figure 1: (a) Simplified layered model for IoT protocol stack. Our design is focused on the Physical layer. (b) Left: deployment of the modulator based on Neural Network. Right: deployment of the modulator based on SDR toolkits.

Therefore, it is essential for the gateway to be flexible, allowing it to support a variety of wireless technologies used by IoT devices and even accommodate emerging technologies for future readiness. However, numerous existing gateway designs [5, 21, 27, 43] employ hardware-based solutions, where wireless technologies are integrated into dedicated chipsets, which are either soldered onto the gateway's motherboard or connected through extension ports. Such hardware-based solutions offer limited adaptability, as their functions are fixed upon manufacturing. Software Defined Radio (SDR) is introduced as a flexible alternative for IoT gateways to address these limitations. Users can implement both current and emerging wireless transceivers as software, surpassing the extensibility of hardware-based solutions.

Despite the advantage of flexibility, SDR-based gateway design comes with several drawbacks. SDR-based designs consist of the radio frequency (RF) front-end and the computing device serving as the host device for the software radio application. The software radio application requires using signal processing toolkits or libraries, like GNURadio [4]

and SciPy [15]. Owing to the variety of development tools and host platforms, transferring the same functionality to a new platform demands a considerable learning curve and extensive effort in software development. Meanwhile, software radio comes at the cost of efficiency loss as we shift the signal processing from specialized hardware to general software systems. Many researchers and developers intend to optimize the software radio with the capability of hardware acceleration [26, 37, 38]. However, these works are targeted at specific platforms and require a considerable learning curve and extensive effort during development. Given the diversity of platforms and toolkits, deploying a highly efficient software modulator on multiple platforms becomes challenging.

To address these issues, we propose to develop a novel framework that facilitates the design of software transmitters on a variety of IoT gateways using neural networks. This approach would maintain the flexibility needed to accommodate a wide range of transmitters while enhancing portability and efficiency on different platforms. Our work is motivated by several interesting observations: **i)** the neural network module is widely supported across diverse hardware platforms due to the flourishing AI technologies, and **ii)** our research demonstrates that signal processing blocks within a modulator can be equivalently implemented using neural network models. These insights have led to the development of our innovative neural network-defined modulator [1], which offers a flexible and portable design for IoT gateways. The complete architecture of the design is depicted in Figure 1b. The proposed neural network-defined modulator functions are implemented by a unified neural network framework that can take advantage of accelerators across various platforms. In essence, the unified neural network framework operates as an abstraction layer for modulation tasks across heterogeneous platforms.

In summary, the original contributions of this paper are listed as follows:

- Conceptually, we propose an NN-defined physical layer modulator, which achieves high flexibility and extensibility to support multiple modulation schemes, and portability and efficiency on heterogeneous platforms.
- Technically, we adopt a model-driven approach to build the NN-defined modulators. The structure and parameters in the NN-defined modulators are rooted in a solid mathematics foundation from the modulation model.
- Experimentally, we deploy the NN-defined modulators on multiple hardware platforms (e.g., Nvidia Jetson Nano, Raspberry Pi) with extensive evaluations. We also employ our NN-defined modulators into the workflow of the IoT gateway to generate protocol-compliant signals, including ZigBee and WiFi.

[1]The code for reproduction is available at the anonymous repository: https://github.com/Repo4Sub/NSDI2024

2 Motivation

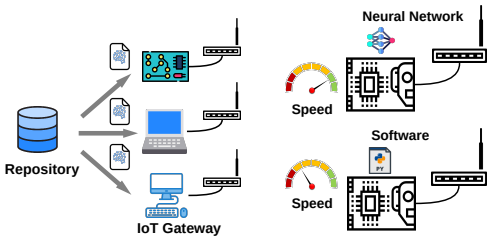
2.1 Problem Statement

Modern IoT gateways strive to offer adaptable transmitters to address the ever-evolving landscape of IoT connectivity technologies. Present IoT gateway solutions can be classified into hardware-based gateways and those based on software-defined radio (SDR). Hardware-based solutions, as the term implies, combine numerous chips/modules tailored for various connectivity technologies on a single board [5, 21, 27, 43]. While these hardware chips/modules exhibit merits such as cost-effectiveness and efficiency, they are limited by their lack of adaptability. This limitation stems from the fixed nature of technologies within the chips/modules and the restricted capacity for users to alter connectivity technologies. Gateway platforms can take the form of diverse devices, including personal computers, edge servers, and, increasingly, embedded computers, which can function as host devices for SDR. Consequently, it is possible to develop software radio for distinct IoT technologies, surpassing hardware-based solutions in terms of flexibility. Nevertheless, the multitude of development tools and deployment platforms for software radio can impede portability. For instance, GNU Radio establishes the signal processing blocks required for software radio construction, yet porting GNU Radio to embedded computers for IoT gateways is challenging, necessitating recompilation for target devices [28]. Moreover, platform/toolkit-specific implementations often depend on optimized designs that exploit acceleration capabilities, potentially resulting in efficiency loss when transferring SDR-based solutions to new platforms. For example, cuSignal [20] is a GPU-accelerated signal processing library, exclusively designed for devices equipped with NVIDIA GPUs, thus not providing a universal solution.

2.2 Opportunities

Our design is inspired by the extensive integration of AI frameworks and hardware across diverse computing platforms, which can serve as IoT gateways. Hardware manufacturers continuously enhance their devices to facilitate neural network deployments, incorporating specialized instruction sets [18] and distinct hardware accelerators [9], along with programming libraries that capitalize on these features. Concurrently, nearly all mainstream machine learning frameworks, such as Tensorflow [16] and PyTorch [13], endeavor to function across various operating systems and hardware architectures. Additionally, these frameworks encapsulate low-level acceleration libraries, promoting developers to speed up the execution of neural network models.

By constructing transmitters as neural network modules with widespread support, we can attain not only extensibility for an array of technologies but also portability and efficiency on platforms compatible with neural networks. For one thing, a gateway device can always update its supported modulation schemes by retrieving the corresponding neural network implementation from the repository server (Figure 2a). Si-



(a) (b)

Figure 2: (a) Different devices can retrieve various neural network models to update modulation schemes. (b) Neural network modulators can be accelerated to achieve better efficiency compared with software modulators.

multaneously, the neural-network-defined modulators are expected to achieve superior efficiency compared to traditional software modulators (Figure 2b), which are blessed by the advantages of the hardware accelerators.

2.3 Challenges

The primary technical challenge involves integrating signal processing blocks into neural network models. One direct method is to utilize general-purpose neural network models, such as fully-connected (FC) layers, as in the literature [46, 58]. Nevertheless, we contend that this approach has two principal disadvantages compared to the traditional digital modulation model. The operational mechanism of a general-purpose machine learning model is often perceived as a black-box approach [31], which raises concerns about its reliability.

To illustrate this, we present a straightforward example of modulators based on general-purpose neural networks. We develop an FC-based neural network model to modulate OFDM symbols and train it using the dataset gathered from the standard 64-S.C. (subcarrier) OFDM modulator. The FC-based OFDM modulator converges to a Mean Squared Error (MSE) loss of 1.5×10^{-6} for the training set, signifying that the generated signals from training symbols closely resemble the corresponding training signals. However, it fails to modulate new OFDM symbols from the test set. The produced signal samples are depicted in Figure 3. The output from the FC-based modulator substantially deviates from the standard signals. Although this is a simple case study, we can deduce from these results that the neural network ought to be meticulously designed and executed to achieve modulation tasks.

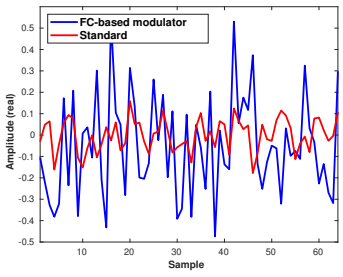


Figure 3: Waveform (real part) comparison of FC-based modulator and standard 64-S.C. OFDM modulator.

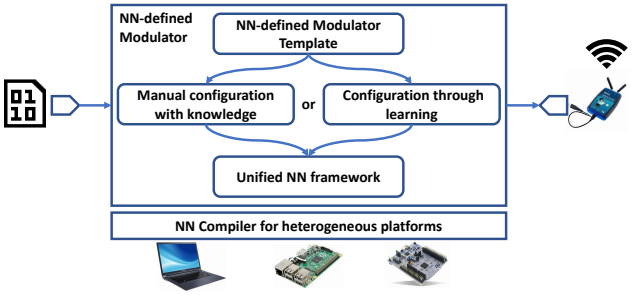


Figure 4: Architecture of NN-defined modulator.

Our research advocates for a model-driven strategy that integrates domain-specific knowledge of modulation techniques into the neural network design process. Instead of employing general-purpose neural network models or creating tailored neural network layers, our objective is to interpret fundamental neural network layers using domain-specific expertise regarding modulation schemes. By assembling the modulator with these neural network layers, which are comprehensively supported and efficiently implemented across diverse frameworks and platforms, we accomplish an interpretable, lightweight, and efficient neural network-based implementation for software modulators.

The architecture of the proposed Neural-Network-Defined (NN-defined) modulator is depicted in Figure 4. As in the figure, a modulator template (Section 3) rooted in solid mathematical foundations can be configured to implement specific modulation schemes either manually as in Section 4, or in a learning manner as in Section 5. Next, the NN-defined modulator will be transformed into a unified NN framework capable of executing across heterogeneous platforms (Section 6). The unified NN framework can be deployed onto various platforms and incorporated into the transmission pipeline (Section 7).

3 Template of NN-defined Modulator

In this section, we discuss how to use a model-driven methodology to construct neural networks for modulation tasks based on the underlying digital modulation models.

3.1 Mathematical Foundation of Digital Modulation

In wireless communication, a transmitter uses a modulator to convert symbols to signals before transmitting them to the air. The modulation process is usually analyzed through the Signal Space Analysis [29, 48], which is widely adopted in modeling amplitude/phase modulation techniques or named as linear modulation [29], including pulse amplitude modulation (PAM), phase-shift keying (PSK), and quadrature amplitude modulation (QAM). And the concepts are also applicable in modeling multicarrier modulation schemes, like OFDM.

Based on this method, a signal $S_i(t)$ modulated from a symbol s_i is considered a linear combination of the set of

basis functions. The synthesis process is given as

$$S_i(t) = \sum_{j=1}^{N} s_{ij}\phi_j(t) \tag{1}$$

where $\phi_j(t) \in \{\phi(t)\}^N$ is the j-th function in the set of N basis functions and s_{ij} is the j-th elements of the N-dimensional vector representation of the input symbol s_i. The basis functions and format of the symbol can be diverse and determine the different modulation schemes.

3.2 Modulator Template via Neural Network

After modeling the modulation process, we start to fit such a mathematical model into the neural network design to construct the proposed NN-defined modulator template.

3.2.1 Discrete-time Modulation Model

To accommodate the model into the neural networks, we first derive the discrete-time representation of the general model. The N-dimensional symbol vector s_i will be modulated to a series of signal samples as:

$$S_i[n] = \sum_{j=1}^{N} s_{ij}\phi_j[n] \tag{2}$$

where $\phi_j[n]$ is the discrete-time form of the basis functions. The symbols are processed sequentially, and the signal samples are concatenated into the final modulated signals for the whole symbol sequence, given as:

$$S[n] = \sum_i S_i[n - iL] \tag{3}$$

where L is the number of *samples per symbol*, meaning that L samples represent one symbol in the final modulated signals.

To extend Equation (2) to a complex I/Q signal, we have:

$$\begin{aligned} &S_I[n] + jS_Q[n] = \mathrm{Re}\{S_i[n]\} + j\mathrm{Im}\{S_i[n]\} \\ &= \sum_{j=1}^{N} [\mathrm{Re}\{s_{ij}\} + j\mathrm{Im}\{s_{ij}\}][\mathrm{Re}\{\phi_j[n]\} + j\mathrm{Im}\{\phi_j[n]\}] \\ &= \sum_{j=1}^{N} \mathrm{Re}\{s_{ij}\}\mathrm{Re}\{\phi_j[n]\} - \sum_{j=1}^{N} \mathrm{Im}\{s_{ij}\}\mathrm{Im}\{\phi_j[n]\} \\ &+ j(\sum_{j=1}^{N} \mathrm{Re}\{s_{ij}\}\mathrm{Im}\{\phi_j[n]\} + \sum_{j=1}^{N} \mathrm{Im}\{s_{ij}\}\mathrm{Re}\{\phi_j[n]\}) \end{aligned} \tag{4}$$

where S_I is the In-Phase signal and S_Q is the Quadrature. We can observe Equation (4) is composed of multiple $\sum_{j=1}^{N} s_{ij}\phi_j[n]$ patterns.

3.2.2 Basics of Transposed Convolutional Layer

Then we convert the $\sum_{j=1}^{N} s_{ij}\phi_j[n]$ pattern to a neural network. We find the *transposed convolutional layer* is a mathematically equivalent implementation. We first introduce the basic computation of a transposed convolutional layer in Figure 5.

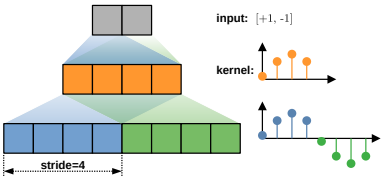


Figure 5: Diagram of the basic operation of the transposed convolutional layer.

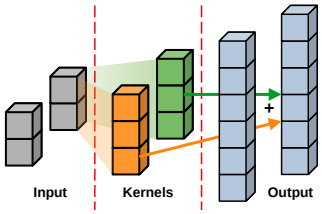


Figure 6: Diagram of the operation of multi-channel transposed convolutional layer.

The 1-D transposed convolutional layer has only 1 input channel and 1 output channel. The elements in input sequence $[+1,-1]$ are multiplied by a *kernel*. The multiplication results are mapped to the output. The step between each multiplication result is determined by the *stride* parameter.

The transposed convolutional layer supports multiple input channels and multiple output channels [13, 16]. In Figure 6, we visualize the operation of the transposed convolutional layer with multiple input and output channels (both are 2 in this figure). As illustrated here, each input channel will *convolve* with a set of 2 kernels, and the results are combined to generate one output channel. The calculation process of the transposed convolutional layer is the same as in one channel of Equation 4, *if the kernel is set to the same as the real/imaginary parts of the basis functions, i.e.,* $\mathrm{Re}\{\phi_j[n]\}$ *and* $\mathrm{Im}\{\phi_j[n]\}$*, and the stride is set to the samples per symbol, i.e., L as in Equation 3.*

3.2.3 NN-defined Modulator Template

With the transposed convolutional layer, we can express the whole modulation process in Equation (4) as an NN-defined template modulator in Figure 7. The input channel comprises real and imaginary parts of the symbol vectors, which form two *groups* of the transposed convolutional layer. The kernels of the transposed convolutional layer are determined by the basis functions. After that, a linear (fully-connected) layer is added to merge the four-channel outputs to generate the real and imaginary parts of the modulated signals. Its weight are set as $[+1,0,0,-1]$ and $[0,+1,+1,0]$ according to the coefficients in Equation (4) as shown in Figure 7.

Thus, we begin with the mathematical foundation of the modulation process and derive a generalized modulation model. Subsequently, we show how to adapt the general model within our template for the NN-defined modulator. The universal template comprises a transposed convolutional layer followed by a fully-connected layer. By meticulously con-

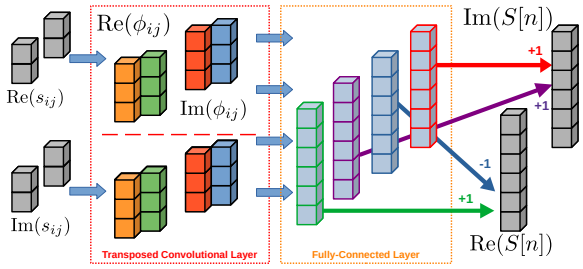


Figure 7: Diagram of the template of NN-defined modulator. 0-weight connections are omitted in the fully-connected layer.

figuring the template for the NN-defined modulator, we can accomplish a range of modulation schemes.

4 Instances of NN-defined Modulator Template

With the NN-defined modulator template, we further study how to generate specific modulation schemes.

4.1 Common NN-defined Modulators

4.1.1 Single Carrier Amplitude/Phase Modulation

For amplitude/phase modulation on the single carrier, the information is carried by the modulated signal's amplitude and/or phase. The most general case is quadrature amplitude modulation (QAM). For example, ZigBee [24] adopts Offset-Phase-Shift Keying (O-QPSK) as its modulation scheme, which is a variant of QPSK or 4-QAM scheme.

The QAM symbols are represented in a complex scalar as $s_k = \mathrm{Re}\{s_k\} + j\mathrm{Im}\{s_k\}$. The symbols pass a real-valued pulse-shaping filter to generate signals. Similar to equation (4), we represent the I/Q signals as in

$$\begin{aligned} S_I[n] &= \mathrm{Re}\{S[n]\} = \sum_k \mathrm{Re}\{s_k\} p[n-kL] \\ S_Q[n] &= \mathrm{Im}\{S[n]\} = \sum_k \mathrm{Im}\{s_k\} p[n-kL] \end{aligned} \tag{5}$$

where $p[n]$ represents the pulse-shaping filter, and L is the number of samples per symbol.

Based on Equation (4) and (5), when applying the NN-defined modulator template, we can configure the kernels of the transposed convolutional layer to be the values of shaping filter $p[n]$. This also implies the potential simplification of the template. If the shaping filter is real-valued, we can omit two channels of the transposed convolutional layer that correspond to the imaginary parts. We can also discard the fully-connected layer in the template because the output from the remaining 2 output channels from the transposed convolutional layer directly forms the desired modulated signals. For better illustration, an NN-defined QPSK modulator with a half-sine wave shaping filter is depicted in Figure 8. The output from the transposed convolutional layer is I/Q signals.

4.1.2 Multicarrier Modulation

We also extend our design for multicarrier modulation schemes, more specifically, the widely used OFDM scheme.

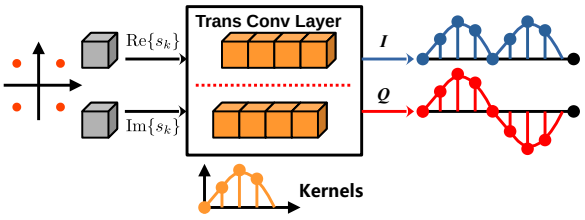


Figure 8: Diagram of a simplified NN-defined QPSK modulator with half-sine wave shaping filter.

We consider a N-S.C. OFDM modulator as an example, of which the input symbol vector consists of N elements, $s_0, s_1, \cdots, s_{N-1}$, that correspond to the components in the frequency domain. Thus, to get the signal samples $S[n]$, they are transformed to the time domain by performing an inverse Discrete Fourier Transform (IDFT) on the input N elements, given as

$$S[n] = \sum_{i=0}^{N-1} s_i e^{j2\pi ni/N},\ 0 \le n \le N-1. \tag{6}$$

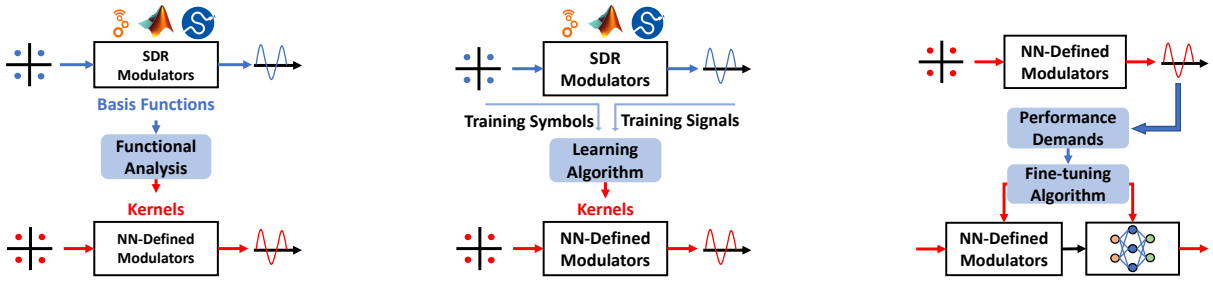
The transformation can be interpreted as mapping the complex symbol vector $\mathbf{s} = [s_0, s_1, \cdots, s_{N-1}]$ of N dimensions to signal $S[0], \cdots, S[N-1]$ with the basis functions set $\phi_i[n]$, which consists of N functions in total, like $\phi_i[n] = e^{j(2\pi ni/N)}$.

The OFDM scheme is consistent with the general case as in Equation (4). As discussed in Section 3, the input to the NN-defined modulator consists of real and imaginary elements from the complex symbol vectors. They are divided into 2 groups at the transposed convolutional layer. For each group, the kernels are set based on the real and imaginary parts of $e^{j2\pi ni/N}$. Then, the four-channel output from the transposed convolutional layer is fed into the fully-connected layer to generate the final In-phase and Quadrature signals.

4.2 Protocol-specified NN-defined Modulators

IoT protocol modulators may incorporate additional operations to enhance system reliability. For instance, ZigBee adopts an offset operation to the QPSK modulator by shifting the quadrature signals by half a symbol duration. OFDM systems used in WiFi adopt cyclic-prefix to improve robustness against multipath effects. Concurrently, some IoT protocols introduce intricate frame structures containing various fields for signaling. For example, WiFi frames typically encompass different signal fields. Although all these fields utilize the OFDM modulator, they may require different operations.

To address these additional operations, we draw inspiration from the inheritance feature in computer programming. The NN-defined modulators serve as the foundational component, and we attach operations to the temporal output from the base NN-defined modulator to generate the ultimate output signals. The attached processes are also achieved through operators supported by neural networks, allowing us to derive specialized NN-defined modulators for diverse protocols. We will discuss the protocol-specific NN-defined modulator in greater detail in Section 7.



(a) (b) (c)

Figure 9: Different approaches to configure kernels. (a) Manual setting with expert knowledge, (b) Learning from existing datasets, (c) Fine-tuning with other ML modules.

5 Modulator Kernel Configuration

From the previous sections, we designed a template for the general modulation model, where the kernels of the template can be derived for a specific modulation scheme. In this section, we will discuss how to use the NN-defined modulator template to learn from signals whose analytical expression is unknown or fine-tune the NN-defined modulator to compensate for hardware distortion in practical systems.

5.1 Manual Setting with Expert Knowledge

As shown in Figure 9a, for a modulation scheme with a known analytical expression, communication experts can take a direct way to derive the kernels of the transposed convolutional layer as discussed in Section 4. It is an efficient and accurate way to construct signals in the NN-defined modulator, similar to the conventional Software modulator in the SDR development.

5.2 Learning from Dataset

As shown in Figure 9b, for a signal with an unknown analytical expression or a non-expert developer, the kernels of the template can be derived by training the NN-defined modulator. For example, a non-expert developer who intends to shift an existing software radio to another platform can utilize the learning ability of the NN-defined modulator from the existing system to reconstruct the modulator, which will significantly ease the development complexity. One can treat it as a standard machine learning task to minimize the mean squared error. Thanks to the model-driven approach, the trained kernels imply a potential signal processing pipeline to mimic the target signal.

More specifically, the training input has the dimension of $[Batch_size, 2 \times Symbol_dimension, \ Sequence_length]$, where $2 \times Symbol_dimension$ indicates the input is represented using the real and imaginary parts. And the training output has the size of $[Batch_size, Signal_length, 2]$, where the 2 on the last dimension also indicates the real and imaginary parts of complex signals. There are $2 \times Symbol_dimension$ kernels to train in total.

For demonstration, we apply the NN-defined modulator template to learn the 64-S.C. OFDM scheme. The NN-defined OFDM modulator is trained with the same training settings as the example FC-based modulator in Section 2 with the same dataset and training epochs. The training set contains 256 different OFDM symbol sequences, each of which represents 128 input complex symbols. The FC-based modulator is implemented with two fully-connected layers, with almost ~ 60000 trainable parameters in total. We calculate the mean squared error between the modulated signals from two kinds of modulators and the standard signals on the training and test sets, respectively. Both the FC-based modulator and our NN-defined modulator have tiny errors on the training set. Our NN-defined modulator outperforms the FC-based modulator significantly on the test set. We plot signals generated from our NN-defined and the FC-based modulator in Figure 10. As in the figure, our NN-defined modulator can modulate the symbols correctly, while the FC-based modulator fails. The NN-defined modulator has much fewer parameters to train compared with the FC-based modulator, and the parameters are physically meaningful, which ensures that our NN-defined modulator is more reliable.

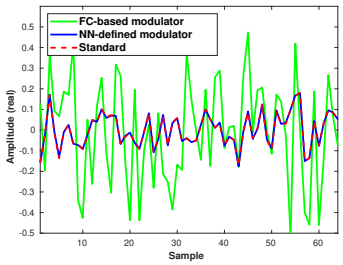


Figure 10: Waveform (In-phase) comparison of FC-based modulator, NN-defined modulator, and standard modulator for 64-S.C. OFDM scheme.

5.3 Fine-tuning for Better Performance

As shown in Figure 9c, the NN-defined modulator can be combined with extra AI/ML models to fine-tune to meet specific performance demands. During the fine-tuning procedure, the kernels of the NN-defined modulators and parameters within the appended AI/ML module are adjusted to fulfill the goals. The fine-tuning process is an open design because the performance demands and the AI/ML extra modules are diverse. For better illustration, we discuss combining the proposed NN-defined modulator with additional AI/ML modules to handle the hardware distortion in the transmitter systems.

The modulated signals are processed at the RF front-end in the transmitter systems to send over the air. Due to the characteristics of the circuits, there is some non-linearity in the RF front-end hardware, which will introduce distortion to the output signal compared with the ideal output. One efficient approach to reduce the distortion effect is to apply the predistortion process to the modulated signals before feeding into the RF front-end [52]. Here, we propose to use a neural network-based predistortion (NN-PD) module. Without loss of generality, we focus on the non-linearity introduced by the power amplifier.

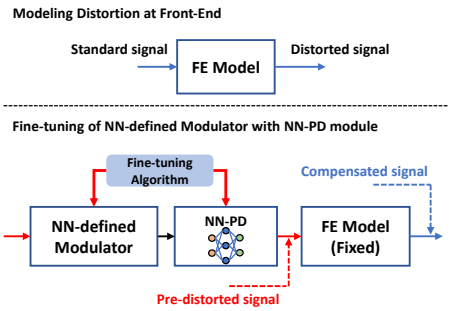


Figure 11: Diagram of Front-End model and NN-defined modulator with NN-PD module.

As illustrated in Figure 11, we first use a neural network, the front-end (FE) Model, to model the nonlinear behavior of the RF front-end. The FE model serves as the simulator of the RF front-end for the fine-tuning procedure. Next, we construct the NN-PD and insert it between the NN-defined modulator and the FE model. The predistorted signals from the NN-PD will pass the FE model, and the compensated signal is generated. The compensated signal is supposed to be as similar as possible to the ideal output signal. So, we set the training goal of our fine-tuning algorithm and tune the kernels in the NN-defined modulator and the parameters in NN-PD module while the parameters in the FE model are fixed. Once the fine-tuning procedure is finished, the NN-defined modulator and NN-PD module can generate predistorted signals, which can compensate for the non-linearity of the RF front-end.

As for verification, we compare the Bit Error Rate (BER) performance of QAM-modulated signals with predistortion and those without predistortion. The simulation is conducted in additive white Gaussian noise (AWGN) channel. We plotted the BER curves in Figure 12. The BER curve of the ideal signals is also visualized as the baseline. Furthermore, we conduct Error Vector Magnitude (EVM) test on the modulated signals. EVM can be evaluated by a percentage scale that reflects the deviation of the modulated and standard constellations. We measure the root mean squared EVM of the signals at different SNR levels. The results are illustrated in Table 1. When the SNR is low (SNR$<$ 0dB), the noise is dominant in such conditions, so all three signals suffer from noisy environments, resulting in high error rates and high EVM. However, when SNR is relatively high (SNR$>$ 0dB), the distortion effect of the RF front-end is more significant than the noise. Hence, the signals with predistortion perform much better than those without predistortion because the hardware distortion is reduced. However, the compensation is imperfect, so the error rates and EVMs of the predistorted signals are still slightly larger than the ideal signals. The above results indicate the great potential that the proposed NN-defined modulator can be integrated with other AI/ML modules and deliver better performance.

	SNR=-10dB	0dB	10dB
EVM of ideal signals	65.9%	31.2%	15.4%
w/ predistortion	66.6%	32.1%	15.7%
w/o predistortion	79.5%	33.4%	21.7%

Table 1: Root mean squared EVM of ideal modulated signals, signals with predistortion, and signals without predistortion.

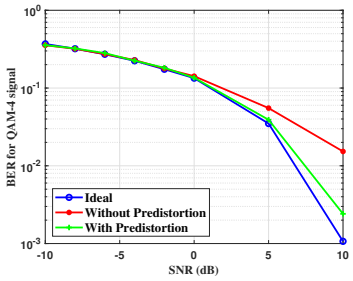


Figure 12: BER of NN-defined Modulator with NN-PD.

6 Modulator with Portability

We first highlight better portability of NN-defined modulators compared with conventional software radio systems by demonstrating the pipeline of the software modulator to generate signal samples. The software radio relies on some libraries which contain signal processing operations. We choose SciPy [15], a scientific computing library in Python, and GNURadio [4], the recognized software radio library, as case studies. Moreover, without loss of generality, we consider the QAM with Root Raised Cosine (RRC) pulse shaping filter as an example. It requires two major steps for modulation: upsampling and pulse-shaping filtering. We list the corresponding implementation for the GNURadio-based and the SciPy-based modulators in Table 2. As shown in the table, although the two kinds of implementations share the same pipeline, the functions used are quite different, which requires the developer to master new development tools for smooth conversion. Besides, we also notice that GNURadio provides several predefined shaping filter blocks, like `Root Raised Cosine Filter(rrc_fir)` for quick usage. In contrast, SciPy does not provide such predefined functions, so we need to configure the filter manually, which also increases the difficulty of porting.

Operations	GNURadio	SciPy
Upsampling	`interp_fir`	`scipy.interpolate`
Filtering	`rrc_fir`	`scipy.convovle`

Table 2: Operations for QAM modulator in different toolkits.

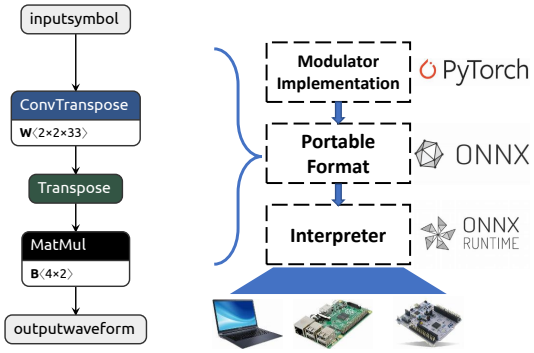
6.1 Framework-Independent NN-defined Modulators

Framework-dependent design implies that the NN-based modulators depend on unique functions or models provided by specific machine learning frameworks. Although machine learning frameworks offer some mathematical functions that can be employed to develop customized neural network layers for modulation tasks, as seen in NVIDIA Sionna [33] based on TensorFlow, the customized neural network modulator remains reliant on the development framework. To exemplify, we demonstrate the implementation details of the Sionna-based QAM modulator. Sionna employs the built-in operations and encapsulates them into the customized neural network layers, `Upsampling` and `Filter`, to emulate the functions as in the conventional pipeline. `Upsampling` layer applies `tf.pad` and some dimensional operations like `tf.expand_dims` to insert zeros between symbols, and `Filter` layer applies `tf.math.convolve` which takes the upsampled symbol sequences and filter taps as input to generate the modulated signals. We compare the availability of mathematical functions used in the Sionna modulator across other mainstream ML frameworks. The results are presented in Table 3. Although there are similar functions such as `pad` and `convolve`, the direct transition among different frameworks is still hard. Consequently, the framework-dependent modulator can be ported to platforms running the same framework, but it is challenging to deploy it on platforms operating with different frameworks.

	Tensorflow	PyTorch
NN-defined	`Conv1DTranspose`	`ConvTranspose1d`
	`Linear`	`Linear`
Sionna	`pad`	`pad+concatenate`
	`expand_dims`	`unsqueeze`
	`convolve`	`convolve`

Table 3: Original operations and converted ONNX operators in our NN-defined and Sionna QAM modulator.

Framework-independent design means our NN-based modulators are implemented by the share functions or models by various machine learning frameworks. Unlike NVIDIA Sionna, which constructs customized layers fro modulators, our NN-defined modulators utilize the fundamental neural network layers that are considered basic components of existing machine learning frameworks. More specifically, the transposed convolutional layer and the fully connected layer are generally supported by various frameworks [13, 16, 23]. Although the layer names vary (Table 3), they share the same functionalities, which ensures that the proposed NN-defined modulator can be a framework-independent design.



(a) (b)

Figure 13: (a) Example converted ONNX format of a QAM NN-defined modulator, (b) Diagram of development and deployment of NN-defined modulators.

We utilize the ONNX [10] as an intermediate framework to ensure the interoperability. ONNX is an open ecosystem for technology companies and research organizations to store and import neural network models onto different frameworks. ONNX defines a common set of operators that contains the fundamental layers of neural network models, including the transposed convolutional layer and the fully-connected layer used in our design. As a validation, we visualize the graph of the ONNX model of our NN-defined modulator template in Figure 13a. As depicted in the figure, the transposed convolutional layer operator is *ConvTranspose*, and the fully-connected layer is represented by the *MatMul* operator. Almost all mainstream machine learning frameworks support conversions between their native models and ONNX ones. It is also worth noting that porting customized neural network layers to ONNX models demands significant effort. Consequently, the custom layers in NVIDIA Sionna are challenging to convert to ONNX models, while our NN-defined modulator built upon fundamental layers exhibits better interoperability across different frameworks.

6.2 Seamless Acceleration

As previously illustrated, IoT gateway hardware platforms provide acceleration capabilities to expedite the execution of neural network models. The proposed NN-defined modulator is constructed based on fundamental neural network layers that are generally supported and well-optimized for execution on various hardware platforms. Therefore, the NN-based modulator can leverage these capabilities to enhance efficiency, speeding up the modulation process.

A typical development and deployment workflow for the proposed NN-defined modulators is depicted in Figure 13b. The prototype of the NN-defined modulators can be developed in mainstream machine learning frameworks, such as PyTorch. Then, NN-defined modulators are converted to a portable ONNX format for improved interoperability across different platforms. Deploying ONNX models requires a compiler, such as ONNX runtime [11] or Apache TVM [2]. Using

Figure 14: Prototype of NN-defined modulator. Left box: Nvidia Jetson Nano as the host device for the NN-defined modulator. Right box: ADI Pluto SDR as SDR hardware.

ONNX runtime as an example, it can utilize different accelerator backends. Numerous accelerator backends have been developed by the community. For instance, it can employ Nvidia GPU [8] on GPU-equipped systems, Arm ACL [3] for Arm SoC platforms, and OpenVINO [12] for Intel x86 platforms. Therefore, the NN-defined modulator can be seamlessly accelerated on various platforms.

7 Evaluation

7.1 Implementation

7.1.1 Framework and hardware

We design the NN-defined modulator in PyTorch [13] with `ConvTranspose1d` and `Linear` layers. Once the NN-defined modulators are ready for port, we convert the modulators into ONNX format. We port the ONNX NN-defined modulator to Nvidia Jetson Nano [9] and Raspberry Pi [14] for verification. Both devices support the ONNX runtime, and Jetson Nano is equipped with a GPU, which can be used to accelerate the execution of the ONNX NN-defined modulators.

Besides, we also implement an NN-defined modulator prototype. We connect the host (Nvidia Jetson Nano) running the NN-defined modulator with the SDR hardware (ADI Pluto SDR [1]) as shown in Figure 14. We use this prototype to transmit the modulated signals over the air.

7.1.2 Modulation schemes

Without loss of generality, we choose several typical schemes for 1) PAM-2 with the rectangular filter, 2) QPSK with the half-sine wave filter, 3) 16-QAM with RRC filter for amplitude/phase modulation, and 4) 64-S.C. OFDM scheme for multicarrier modulation. We use MATLAB Signal Processing Toolbox [6] to generate the symbols and the signals as for training sets. When evaluating the efficiency and portability, we select 16-QAM modulator with RRC filter as the example. The conventional SDR modulators for 16-QAM with RRC filter are implemented with signal processing libraries, GNU-Radio [4] on x86 laptop, and SciPy [15] on Nvidia Jetson Nano, as the baselines. For comparison, we also implement a 16-QAM modulator with RRC filter using Nvidia Sionna [33] on the x86 laptop for comparison.

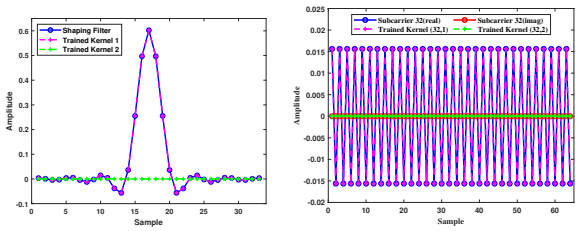
7.2 Signal Quality of NN-defined Modulator

7.2.1 Trained kernels in NN-defined modulators

As discussed in Section 5, the kernels within the NN-defined modulator can be trained with training sets. Here, we use 16-QAM with RRC filter and 64-S.C. OFDM scheme as examples to analyse the trained kernels.

For the 16-QAM scheme with RRC shaping filter, the input symbol is 1-dimensional, so there are 2 kernels trained. According to the analysis in Section 4, the trained kernels are supposed to be the real and imaginary parts of the shaping filter. We visualize trained kernels and the original RRC shaping filter in Figure 15a. One of the trained kernels is nearly identical to the original shaping filter. The other one is almost zero-valued, which is consistent with the zero-valued imaginary parts of the shaping filter.

For the 64-S.C. OFDM scheme, there are 2×64 kernels trained. According to the analysis in Section 4, the kernels are supposed to be the real and imaginary parts of the subcarrier functions, i.e., $e^{j\frac{2\pi in}{64}}$. We also visualized a pair of the trained kernels in Figure 15b. And these two kernels are the same as real and imaginary parts of the standard subcarrier $e^{j\frac{2\pi\times 32n}{64}}$. The NN-defined OFDM modulators share the same conclusion that the trained kernels perfectly match the signal processing pipeline in conventional modulators.



(a) (b)

Figure 15: Trained kernels from NN-defined modulators for (a) QAM with RRC filter, (b) 64-S.C. OFDM.

7.2.2 Transmission performance in AWGN channel

We apply the trained NN-defined modulators to generate signals and pass the signals in the additive white Gaussian noise (AWGN) channel to verify the transmission performance. And we plot the Bit Error Rate (BER) curves in Figure 16. Meanwhile, The BER curves of the signals from standard modulators in MATLAB are also plotted as the baseline. As illustrated in the figure, the NN-defined modulators for the selected modulation schemes can modulate the symbols correctly so that the modulated signals can achieve the same error performance as standard modulators in AWGN channels.

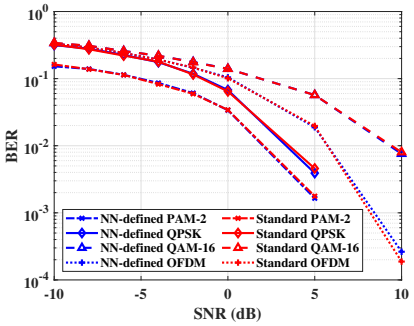


Figure 16: The BER performance of NN-defined modulator compared with standard modulators.

7.3 Efficiency and Portability

7.3.1 Efficiency improvement

To verify the efficiency improvement of the NN-defined modulator, we measure the running time of the conventional SDR QAM modulator, the NN-defined QAM modulator, and the Nvidia Sionna QAM modulator on an x86 laptop and compare these time recordings in Figure 17. All the QAM modulators modulate a batch consisting of 32 symbol sequences with 256 symbols.

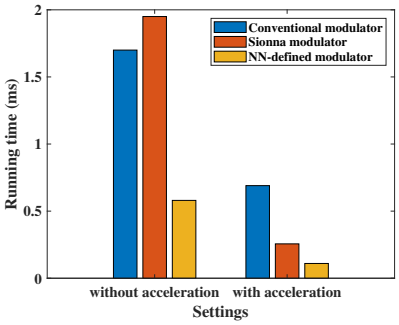


Figure 17: Running time of different implementations.

When all three modulators run without acceleration, it takes 0.58ms for our design to finish, which is much faster than the conventional (1.7ms) and Sionna modulator (1.9ms). Our NN-defined modulator applies the fundamental neural network layers that are well-optimized so that it performs better than the customized neural network layers implemented in Nvidia Sionna as well as the conventional SDR modulator.

The NN-defined and Sionna modulators support hardware acceleration thanks to the neural network implementation. We measure their running time with acceleration enabled. We also implement an accelerated QAM modulator with cuSignal [20]. The NN-defined modulator and Sionna modulator execute much faster than without acceleration. The running time of the NN-defined modulator is reduced to 0.059ms from 0.58ms, which is 28 times faster than the conventional SDR modulators without acceleration, even 10 times faster than the implementation using cuSignal. For the Sionna modulator, the running time is also reduced to 0.25ms. Both our NN-defined modulator and Sionna modulator run faster than the conventional modulator. These results prove that the NN-defined modulator can significantly improve efficiency compared with conventional SDR modulators.

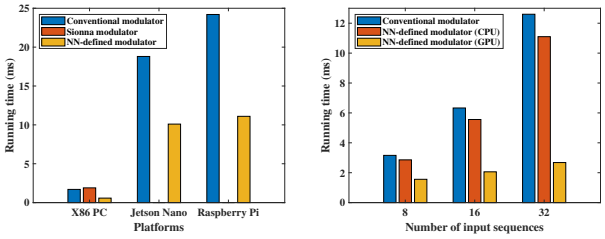
7.3.2 Portability

Porting among platforms. As aforementioned in Section 6, porting the conventional SDR implementations and Sionna-based one from one platform to another requires considerable effort. Here, we focus on the portability of our NN-defined modulator. Following the development diagram of the NN-defined modulators, we first implement the NN-defined QAM modulator in PyTorch and convert it into the ONNX model. We list the converted operations in the ONNX framework in Table 4. The transposed convolutional layer (`torch.ConvTranspose1d`) and the linear layer (`torch.Linear`) are widely supported so that they can be converted to the portable format.

Implementations	PyTorch layer	ONNX operator
NN-defined	`ConvTranspose1d`	`ConvTranspose`
	`Linear`	`MatMul`

Table 4: Original operations and converted ONNX operators in the NN-defined modulator.

Performance on different platforms. We now deploy the ONNX NN-defined QAM modulator on embedded computers such as Nvidia Jetson Nano and Raspberry Pi. Figure 18a illustrates the running time on different platforms. Sionna modulator fails to be ported because the customized layers are hard to be transformed into ONNX models. Although the running time of the NN-defined modulator on embedded systems is longer than that on the x86 laptop, we successfully port our NN-defined modulators to different platforms.



(a) (b)

Figure 18: (a) Running time on different platforms of x86 PC, Nvidia Jetson Nano, Raspberry Pi. (b) Acceleration evaluation on the target platform of Nvidia Jetson Nano.

The following evaluation demonstrates the acceleration capability of the target platform. We configure the ONNX NN-defined QAM modulator on Nvidia Jetson Nano to run with GPU acceleration as discussed in Section 6. We compare the running time of the conventional modulator and our NN-defined modulator modulate symbol batches of different sizes. The evaluation results are visualized in Figure 18. We can observe a considerable efficiency improvement compared with the conventional modulator as well as the CPU-only

NN-defined modulator. Moreover, the efficiency of our NN-defined modulator is still much better than the conventional modulator implemented with an accelerated signal processing library. More specifically, when the number of input symbol sequences is 32, the accelerated NN-defined modulator is 4.7 times faster than the conventional modulator and even 2.5 times faster than the accelerated modulator. These results showcase that we can easily run the NN-defined modulators on target platforms with acceleration capability.

7.4 Application in IoT Technologies

The proposed NN-defined modulators are employed to generate protocol-compliant signals, showcasing representative use cases in IoT gateways.

7.4.1 ZigBee-compliant Signals

ZigBee [24], developed based on IEEE 802.15.4 [19], utilizes Offset-QPSK, a variant of amplitude/phase modulation, as its modulation scheme. The diagram of the O-QPSK modulator is illustrated in Figure 19. The modulator input comes from a 4-QAM constellation, where the symbols are $\{\pm 1 \pm 1j\}$. The real and imaginary parts of the input symbols are processed separately to generate I/Q signals. The quadrature branch of signal samples is shifted by a delay to introduce the offset. As evident in the output waveform, the quadrature branch exhibits a slight lag.

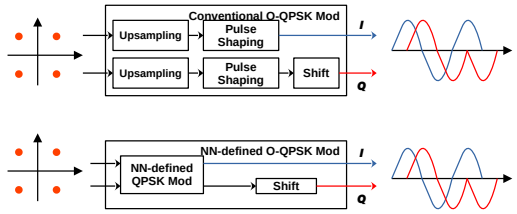
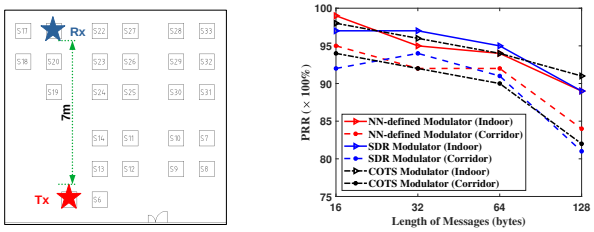


Figure 19: Diagram of conventional O-QPSK modulator (top) and the NN-defined O-QPSK modulator (bottom).

To construct an O-QPSK modulator for ZigBee protocol, we combine the NN-defined QPSK modulator with a shifting process to form the NN-defined O-QPSK modulator, as depicted in Figure 19. We generate symbols from messages following the specification and feed them into our NN-defined O-QPSK modulator. The modulated signals are sent over the air utilizing the prototype in Figure 14. We employ the TI CC2650 Kit [17] as the ZigBee receiver, which can parse the captured signals into messages.

We generate ZigBee packets with varying message lengths and transmit 100 packets. At the receiver side, the received ZigBee packets without errors are recorded, and we calculate the packet reception ratio (PRR) in different settings, repeating the evaluation 5 times. We compare the performance with the SDR implementation using signal processing libraries. And we conduct the same experiments on commercial off-the-shelf (COTS) TI devices as a baseline. The evaluation is conducted indoors and outdoors. The settings of indoor environments are demonstrated in Figure 20a. As depicted in Figure 20b, the ZigBee signals generated by the NN-defined modulator can be successfully received by the commercial device, achieving performance comparable to the existing SDR implementation and commercial devices.



(a) (b)

Figure 20: (a) Evaluation settings in indoor environment. (b) Packet Reception Ratio of ZigBee packets modulated by NN-defined modulator and SDR modulator.

7.4.2 WiFi-compliant Signals

WiFi, which is also extensively utilized for IoT communication, typically employs the OFDM scheme. When implementing the NN-defined modulator for WiFi communication, the process becomes slightly more complex, as WiFi utilizes the CP-OFDM [48] modulator, and WiFi frames generally consist of signals generated from various fields.

Taking IEEE 802.11a/g as an example in Figure 21, WiFi frames comprise four fields: Short Training Field (STF), Long Training Field (LTF), Signaling Field (SIG), and Data Field (DATA). The STF and LTF primarily serve detection, synchronization, and channel estimation purposes at the receiver. The SIG contains information about the current frame, such as frame length and modulation and coding scheme information, while the DATA field carries the data.

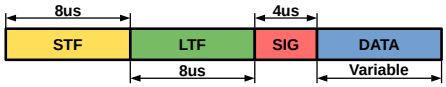


Figure 21: Fields in IEEE 802.11a/g frame.

Different fields need specific operations. The STF and LTF involve repeating the signals from the OFDM modulator, while the SIG and DATA require adding a cyclic prefix to the modulated signals by copying the ending parts of the OFDM signals to the front. Following the discussion in Section 4, we combine an NN-defined OFDM modulator with additional operations to add the cyclic prefix.

Four NN-defined modulators corresponding to the four fields in IEEE 802.11a/g WiFi frames are implemented. These modulators are then combined to create a single NN-defined WiFi modulator. The overall structure is illustrated in Figure 22. The NN-defined modulators for `STF`, `LTF`, `SIG`, and `DATA` fields collectively form the NN-defined WiFi modulator, allowing for a comprehensive modulation process that addresses the unique requirements of each field.

We generate beacon packet signals using the NN-defined WiFi modulator and transmit them over the air. A laptop is

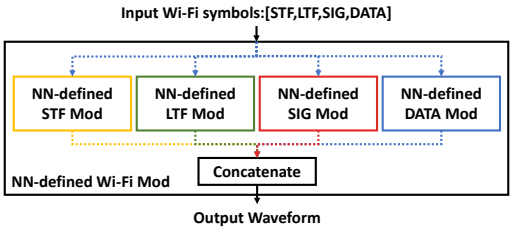


Figure 22: NN-defined Wi-Fi modulator.

used to sniff the beacon packets. We test beacon reception in an indoor environment at the 5GHz band, transmitting 100 beacon packets for 5 times. Figure 23 demonstrates that the laptop can successfully receive the beacon with an SSID of "NN-definedModulator", achieving a PRR at 96%.

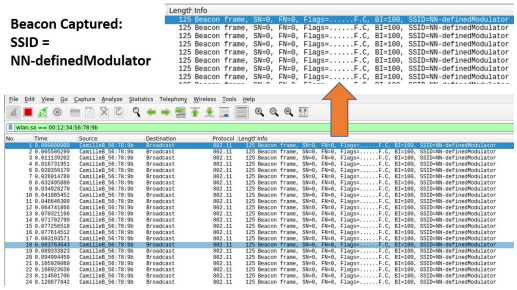


Figure 23: Reception of beacon signals generated by NN-defined Wi-Fi modulator.

Next, we extend our design to transmit data by generating data packets and passing the signals through simulated AWGN channels. We follow the standard process to detect and synchronize WiFi frames using STF signals, conduct channel estimation and equalization using LTF signals, and then demodulate and decode the SIG and DATA signals. In our evaluation, the symbols for a grayscale image data are generated using 16-QAM and 64-QAM. The results are listed in Figure 24. As shown in the figure, we can successfully reconstruct the transmitted images under different settings, further demonstrating the effectiveness and versatility of our NN-defined modulator design in practical applications.

(a) (b) (c)

Figure 24: (a) Original image of 256×256 pixels; Received images using (b) 16-QAM at SNR=10dB and (c) using 64-QAM at SNR=20dB.

8 Related Works

SDR solutions for IoT gateway: SDRs are proposed as universal gateways operating across technologies. Past work [22, 51, 59] has developed smart home gateways using the USRP radio with GNUradio support. [45] revisited the SDR-based IoT gateway for decoding collapsed packets. Other solutions employed the cross-technology communication technique as an alternate for IoT gateways [34–36, 39–42, 53–56].

Machine learning for communication system: Neural networks or machine learning has been extensively used in physical layer designs [25, 31, 44, 47, 49, 57, 61]. [46] introduced a method to learn an end-to-end communication system by interpreting it as an autoencoder [30]. In [60], a DNN model replaces all blocks in the conventional OFDM receiver. In [50], the researchers propose to replace processing blocks in the OFDM receiver with neural network models and deploy them on IoT devices.

Our work has innovations in two tiers, distinctive objectives, and different methodologies. Objective-wise, most literature views the neural network as an optimizer and seeks performance gains under complex conditions [61]. In contrast, we use the neural network as an abstraction layer for the portability of IoT gateway functionalities. Methodology-wise, the literature commonly adopts data-driven approaches that employ general-purpose neural networks [25, 49, 57]. We adopt a model-driven approach that designs the neural network-based modulators with reference to the mathematical models.

9 Discussion

It's worth pointing out that we only discuss the linear amplitude/phase modulation schemes in this paper. Other modulation schemes require further study, such as frequency modulation, also known as non-linear modulation. Following the similar idea, We can model the frequency modulation based on the phase changes and construct another NN-defined modulator template that can be used for the Gaussian frequency shift keying (GFSK) modulators used in Bluetooth [32]. Moreover, we intend to extend the application of the learning ability.We can further apply the neural network to learn to reduce the adjacent channel leakage ratio (ACLR) for single carrier scheme or to reduce the peak-average power ratio (PAPR) for OFDM scheme. We can also apply the NN-defined modulator to learn from noisy signal samples to reconstruct noiseless modulators. The model-driven approach can also be applied to the receiver design, including demodulation and decoding, which is an emerging topic in wireless communication.

10 Conclusion

In this paper, we present an NN-defined modulator template for various modulation schemes that can be converted to a unified NN framework for portable deployment for IoT gateway design. The proposed NN-defined modulator has the extensibility to achieve various modulation schemes for IoT connections, and the evaluation results show that they can perform well. The NN-defined modulator outperforms the existing SDR solutions in terms of portability and efficiency thanks to the wide support of the NN on heterogeneous computing platforms. Meanwhile, the NN-based implementation also enables our design with the learning ability, featuring the potential for intelligent communication systems.

Acknowledgments

We extend our heartfelt thanks to the reviewers for their rigorous and constructive feedback and the shepherding committee for their invaluable guidance. Special appreciation goes to our collaborator, Shuai Wang from George Mason University, whose diverse skills also significantly enriched this work. This research is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (MOE-T2EP20221-0017), National Natural Science Foundation of China under Grant No.62272098 and The Future Network Scientific Research Fund Project (Grant No. FNSRFP-2021-YB-17). This research is also supported by the National Research Foundation, Singapore, and Infocomm Media Development Authority under its Future Communications Research & Development Programme.

References

[1] Adi adalm-pluto. https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/adalm-pluto.html.

[2] Apache tvm. https://tvm.apache.org/.

[3] Arm compute library. https://www.arm.com/technologies/compute-library.

[4] Gnu radio. https://www.gnuradio.org/.

[5] Hardwario iot platforms. https://www.hardwario.com/.

[6] Matlab signal processing toolbox. https://www.mathworks.com/products/signal.html.

[7] Narrowband–internet of things (nb-iot). https://www.gsma.com/iot/narrow-band-internet-of-things-nb-iot/.

[8] Nvidia cuda toolkit.

[9] Nvidia jetson nano developer kit. https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

[10] Onnx: Open neural network exchange. https://onnx.ai.

[11] Onnx runtime. https://onnxruntime.ai/.

[12] Openvino. https://docs.openvino.ai/latest/home.html.

[13] Pytorch. https://pytorch.org/.

[14] Raspberry pi. https://www.raspberrypi.com/.

[15] Scipy. https://www.scipy.org/.

[16] Tensorflow. http://tensorflow.org/.

[17] Texas instruments launchxl-cc2650. https://www.ti.com/tool/LAUNCHXL-CC2650.

[18] Up edge computing kit. https://www.aaeon.com/en/c/up-edge-computing-kit.

[19] Ieee standard for low-rate wireless networks. *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, pages 1–709, 2016.

[20] Cusignal. https://developer.nvidia.com/blog/accelerated-signal-processing-with-cusignal, 2023.

[21] Amiruddin Amiruddin, Anak Agung Putri Ratna, Ruki Harwahyu, and Riri Fitri Sari. Secure multi-protocol gateway for internet of things. In *2018 Wireless Telecommunications Symposium (WTS)*, pages 1–8, 2018.

[22] Carlos J Bernardos, Antonio De La Oliva, Pablo Serrano, Albert Banchs, Luis M Contreras, Hao Jin, and Juan Carlos Zúñiga. An architecture for software defined wireless networking. *IEEE wireless communications*, 21(3):52–61, 2014.

[23] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[24] Sinem Coleri Ergen. Zigbee/ieee 802.15. 4 summary. *UC Berkeley, September*, 10(17):11, 2004.

[25] Xuanxuan Gao, Shi Jin, Chao-Kai Wen, and Geoffrey Ye Li. Comnet: Combination of deep learning and expert knowledge in ofdm receivers. *IEEE Communications Letters*, 22(12):2627–2630, 2018.

[26] Georgios Georgis, Alexios Thanos, Marcin Filo, and Konstantinos Nikitopoulos. A dsp acceleration framework for software-defined radios on x86 64. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1648–1652. IEEE, 2020.

[27] Egidio Gioia, Pierluigi Passaro, and Matteo Petracca. Amber: An advanced gateway solution to support heterogeneous iot technologies. In *2016 24th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–5. IEEE, 2016.

[28] Gwenhael Goavec-Merou. Gnuradio running on embedded boards: porting to buildroot. *Proceedings of the GNU Radio Conference*, 2021.

[29] Andrea Goldsmith. *Wireless communications*. Cambridge university press, 2005.

[30] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[31] Hengtao He, Shi Jin, Chao-Kai Wen, Feifei Gao, Geoffrey Ye Li, and Zongben Xu. Model-driven deep learning for physical layer communications. *IEEE Wireless Communications*, 26(5):77–83, 2019.

[32] Robin Heydon and Nick Hunn. Bluetooth low energy. *CSR Presentation, Bluetooth SIG https://www. bluetooth. org/DocMan/handlers/DownloadDoc. ashx*, 2012.

[33] Jakob Hoydis, Sebastian Cammerer, Fayçal Ait Aoudia, Avinash Vem, Nikolaus Binder, Guillermo Marcus, and Alexander Keller. Sionna: An open-source library for next-generation physical layer research. *arXiv preprint arXiv:2203.11854*, 2022.

[34] Wenchao Jiang, Song Min Kim, Zhijun Li, and Tian He. Achieving receiver-side cross-technology communication with cross-decoding. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 639–652, 2018.

[35] Wenchao Jiang, Zhimeng Yin, Song Mim Kim, and Tian He. Transparent cross-technology communication over data traffic. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[36] Wenchao Jiang, Zhimeng Yin, Ruofeng Liu, Zhijun Li, Song Min Kim, and Tian He. Bluebee: a 10,000 x faster cross-technology communication via phy emulation. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, pages 1–13, 2017.

[37] Tarik Kazaz, Christophe Van Praet, Merima Kulin, Pieter Willemen, and Ingrid Moerman. Hardware accelerated sdr platform for adaptive air interfaces. *arXiv preprint arXiv:1705.00115*, 2017.

[38] Kaipeng Li, Bei Yin, Michael Wu, Joseph R Cavallaro, and Christoph Studer. Accelerating massive mimo uplink detection on gpu for sdr systems. In *2015 IEEE dallas circuits and systems conference (DCAS)*, pages 1–4. IEEE, 2015.

[39] Zhijun Li and Tian He. Webee: Physical-layer cross-technology communication via emulation. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, pages 2–14, 2017.

[40] Ruofeng Liu, Zhimeng Yin, Wenchao Jiang, and Tian He. Lte2b: Time-domain cross-technology emulation under lte constraints. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, pages 179–191, 2019.

[41] Ruofeng Liu, Zhimeng Yin, Wenchao Jiang, and Tian He. Xfi: Cross-technology iot data collection via commodity wifi. In *2020 IEEE 28th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2020.

[42] Ruofeng Liu, Zhimeng Yin, Wenchao Jiang, and Tian He. Wibeacon: Expanding ble location-based services via wifi. In *Proceedings of the 27th annual international conference on mobile computing and networking*, pages 83–96, 2021.

[43] Roberto Morabito, Riccardo Petrolo, Valeria Loscrì, and Nathalie Mitton. Legiot: A lightweight edge gateway for the internet of things. *Future Generation Computer Systems*, 81:1–15, 2018.

[44] Tianjie Mu, Xiaohui Chen, Li Chen, Huarui Yin, and Weidong Wang. An end-to-end block autoencoder for physical layer based on neural networks. *arXiv preprint arXiv:1906.06563*, 2019.

[45] Revathy Narayanan and Swarun Kumar. Revisiting software defined radios in the iot era. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 43–49, 2018.

[46] Timothy O'shea and Jakob Hoydis. An introduction to deep learning for the physical layer. *IEEE Transactions on Cognitive Communications and Networking*, 3(4):563–575, 2017.

[47] Timothy O'shea and Jakob Hoydis. An introduction to deep learning for the physical layer. *IEEE Transactions on Cognitive Communications and Networking*, 3(4):563–575, 2017.

[48] John G. Proakis. *Digital Communications*. McGraw-Hill New York, 2007.

[49] Morteza Soltani, Wael Fatnassi, Ahmed Aboutaleb, Zouheir Rezki, Arup Bhuyan, and Paul Titus. Autoencoder-based optical wireless communications systems. In *2018 IEEE Globecom Workshops (GC Wkshps)*, pages 1–6. IEEE, 2018.

[50] Nasim Soltani, Hai Cheng, Mauro Belgiovine, Yanyu Li, Haoqing Li, Bahar Azari, Salvatore D'Oro, Tales Imbiriba, Tommaso Melodia, Pau Closas, et al. Neural network-based ofdm receiver for resource constrained iot devices. *arXiv preprint arXiv:2205.06159*, 2022.

[51] Manolis Surligas, Antonis Makrogiannakis, and Stefanos Papadakis. Empowering the iot heterogeneous wireless networking with software defined radio. In *2015 IEEE 81st Vehicular Technology Conference (VTC Spring)*, pages 1–5. IEEE, 2015.

[52] Chance Tarver, Liwen Jiang, Aryan Sefidi, and Joseph R Cavallaro. Neural network dpd via backpropagation through a neural network model of the pa. In *2019 53rd Asilomar Conference on Signals, Systems, and Computers*, pages 358–362. IEEE, 2019.

[53] Shuai Wang, Jianlin Guo, Pu Wang, Kieran Parsons, Philip Orlik, Yukimasa Nagai, Takenori Sumi, and Parth Pathak. X-disco: Cross-technology neighbor discovery. In *2022 19th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, pages 163–171. IEEE, 2022.

[54] Shuai Wang, Woojae Jeong, Jinhwan Jung, and Song Min Kim. X-mimo: Cross-technology multi-user mimo. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, pages 218–231, 2020.

[55] Shuai Wang, Song Min Kim, and Tian He. Symbol-level cross-technology communication via payload encoding. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 500–510. IEEE, 2018.

[56] Shuai Wang, Zhimeng Yin, Zhijun Li, Yongrui Chen, Song Min Kim, and Tian He. Networking support for bidirectional cross-technology communication. *IEEE Transactions on Mobile Computing*, 20(1):204–216, 2019.

[57] Chao-Kai Wen, Wan-Ting Shih, and Shi Jin. Deep learning for massive mimo csi feedback. *IEEE Wireless Communications Letters*, 7(5):748–751, 2018.

[58] Hao Ye, Geoffrey Ye Li, and Biing-Hwang Juang. Power of deep learning for channel estimation and signal detection in ofdm systems. *IEEE Wireless Communications Letters*, 7(1):114–117, 2017.

[59] Chaorui Zhang, Peng Xie, Deyuan Li, Jiekai Zhang, and Rong Yu. Wireless home gateway: Software-defined radio architecture and applications. In *IET International Conference on Communication Technology and Application (ICCTA 2011)*. IET, 2011.

[60] Zhongyuan Zhao, Mehmet Can Vuran, Fujuan Guo, and Stephen D Scott. Deep-waveform: A learned ofdm receiver based on deep complex-valued convolutional networks. *IEEE Journal on Selected Areas in Communications*, 39(8):2407–2420, 2021.

[61] Banghua Zhu, Jintao Wang, Longzhuang He, and Jian Song. Joint transceiver optimization for wireless communication phy using neural network. *IEEE Journal on Selected Areas in Communications*, 37(6):1364–1373, 2019.

Democratizing Direct-to-Cell Low Earth Orbit Satellite Networks

Lixin Liu[1], Yuanjie Li[1,2], Hewu Li[1,2], Jiabo Yang[1], Wei Liu[1], Jingyi Lan[1], Yufeng Wang[1], Jiarui Li[1], Jianping Wu[1,2], Qian Wu[1,2], Jun Liu[1,2], Zeqi Lai[1,2]
[1]*Tsinghua University,* [2]*Zhongguancun Laboratory*

Abstract

Multi-tenant Low Earth Orbit (LEO) satellites emerge as a cost-effective win-win solution for direct 4G/5G access to our regular phones/IoTs anywhere on Earth. However, the current hop-by-hop stateful cellular session impedes this effort due to its need for tight functional coupling and stable service relationships among satellite operators, mobile operators, and users. Our empirical study with real satellite data shows that, it restricts LEO satellites' serviceable areas, limits the use of available (possibly competitive) satellites, and suffers from signaling storms and dynamic many-to-many relationships in extreme LEO mobility. We thus devise MOSAIC to strive for *self-serve multi-tenant* LEO satellites. MOSAIC defines policy-embedded one-time tokens for *pay-as-you-go* local satellite access. These tokens allow satellites to self-serve users anywhere without relying on remote mobile operators, alleviate inter-satellite coordinations to enjoy competitive satellites, and simplify many-to-many service relationships for on-demand multi-tenancy. MOSAIC is attack-resilient and incrementally deployable using our SIM-based solution. Our evaluations with the real satellite data and commodity 3GPP NTN protocol stack validate MOSAIC's viability.

1 Introduction

Space is the new business growth point for cellular networks. The emergent direct-to-cell LEO satellites, such as SpaceX's Starlink [1, 2], Iridium [3, 4], Globalstar [5], AST [6, 7], and Lynk [8] complement terrestrial networks to eliminate their coverage holes for 2.7 billion "unconnected" global users [9] and offer their regular phones/IoTs direct satellite access via 4G, 5G, and beyond. They can significantly save operators' infrastructure costs in under-served areas and expand their service to anywhere on Earth for new subscribers and revenues. So, mobile network operators (MNOs) and satellite network operators (SNOs) have actively partnered to deploy [10–15] and standardize [16–27] direct-to-cell LEO satellite services.

Rather than owning dedicated satellites by every MNO, building *multi-tenant* direct-to-cell satellites for sharing is a more practical and favorable win-win solution for MNOs *and* SNOs [1] (Figure 1). On the one hand, satellites are a scarce and competitive resource for MNOs. The highly congested near-Earth space leaves insufficient orbital slots for all MNOs' satellites [28–30]. LEO satellites' capital expenses are also prohibitive for MNOs [31, 32]. Instead, renting satellites is more affordable and lowers the barriers to entry for MNOs. On the other hand, SNOs also have incentives to partner with MNOs due to their lack of licensed 4G/5G spectrums to serve

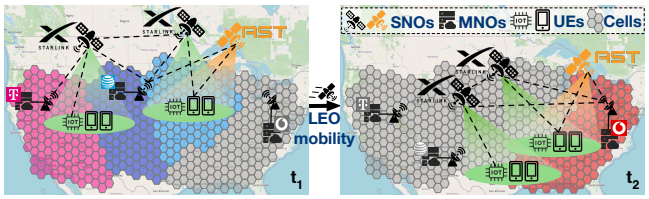


Figure 1: Dynamic multi-tenant direct-to-cell LEO satellites.

regular phones/IoTs independently. Akin to cloud computing, leasing satellites to more MNOs increases SNOs' revenues and return on investment through economies of scale.

Sharing satellite access has been a decades-old practice. Traditional geostationary satellites are single-hop physical pipes that are easily sharable using the infrastructure-as-a-service model. While ideal for multi-tenancy, this transparent pipe model suffers from low service coverage, missed radio processing deadlines, and unaffordable bandwidth demands in 4G/5G due to its heavy reliance on remote ground stations (§3.1). To this end, modern LEO satellites like Starlink, AST, and Lynk have adopted onboard cellular network functions for scalable and performant direct-to-cell services [8, 33–36].

However, cellular network functions in LEO satellites are not easily sharable due to their requirement for tight functional coupling and stable service relationships among SNOs, MNOs, and user equipment (UEs). This requirement is rooted in the cellular architecture's stateful hop-by-hop session that assumes fixed, always-on, and trusted infrastructure. It is hard to meet in multi-tenant LEO satellites due to their fast mobility, intermittent accessibility to MNOs for remote control, and 3rd-party nature as intermediate session nodes. Our empirical study with real satellite data shows that (§3), this defect is detrimental to everyone: It impedes UEs' flexible use of *any* available satellites, restricts MNOs' serviceable areas, complicates MNOs' use of diverse (potentially competitive) SNOs' satellites, and exhausts SNOs' satellites with signaling storms and dynamic many-to-many relationship management.

We explore an alternative cellular scheme for *self-serve multi-tenant* direct-to-cell satellites. Our solution, MOSAIC (**M**ulti-**O**perator **S**atellite **A**ccess via **I**n-band **C**ontrol), adopts the *pay-as-you-go* paradigm that is more suitable than hop-by-hop stateful sessions for sharing the mobile infrastructure like LEO satellites (§5): Akin to mobile bike sharing [37], each MNO supplies its UEs with self-certified one-time tokens (a satellite version of restrictive blind signatures from the offline cash system [38]) as "coins" to pay for local satellite access on demand. These tokens embed UE-specific roaming, billing, and QoS policies to let *any* authentic satellite self-serve UEs without contacting remote MNOs. This UE-initiated in-band

state provision enables near-stateless/sessionless satellites for transparent pipe-like multi-tenancy. It also simplifies inter-satellite coordination and dynamic many-to-many service relationships to avoid signaling storms and encourage the use of competitive SNOs' satellites. MOSAIC's tokens retain the same security level as the legacy 4G/5G and are incrementally deployable using our SIM card-based deployment (§6).

We prototype MOSAIC and evaluate it using real satellite data and Amarisoft's commodity 3GPP non-terrestrial network (NTN) protocol stack. Compared to the NTN [16–27] and Starlink [34–36], MOSAIC scales to a large number of satellites, COTS UEs, and MNOs with signaling storm freedom, 116% serviceable area expansion to the LEO constellation's entire coverage, and 4.71–14.25× service resumption latency reduction in LEO mobility at negligible costs.

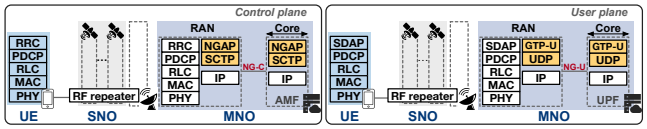
2 Why Multi-Tenant LEO Mobile Satellites?

In this section, we motivate the need for direct-to-cell LEO satellites (§2.1) and the incentives for sharing them among MNOs from the MNO, SNO, and UE perspectives (§2.2).

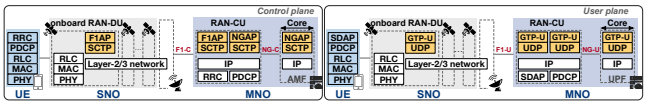
2.1 The Need for Direct-to-Cell LEO Satellites

Direct-to-cell satellites originate as a complementary method to connect phones/IoTs where the terrestrial infrastructure cannot reach. To offer ubiquitous access, terrestrial cellular networks should deploy radio access networks (RANs) with numerous radio base stations to cover broad geographic areas and bridge them to the Internet via core networks (Figure 2a). While profitable in urban areas with sufficient subscribers, such capital-intensive infrastructure loses revenues when covering rural areas with few subscribers [39, 40] and is even undeployable in oceans and airplanes, thus leaving 2.7 billion global users unconnected [9, 41]. Instead, direct-to-cell satellites can complement terrestrial networks with their broad coverage, offer direct cellular access to phones/IoTs, and save MNOs' capital and operation costs in under-served areas.

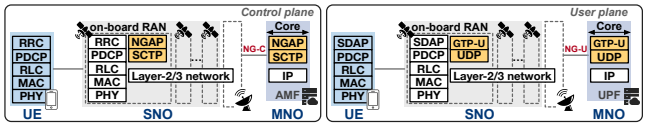
Traditional direct-to-cell satellites, such as Inmarsat [42], Thuraya [43], and Tiantong [44], operate in the geostationary orbit (GEO) at an altitude of 35,786 km. While excellent for broad coverage, GEO satellites are unfriendly to commodity phones/IoTs since their distant transmission is power-hungry, slow, and noisy. Dedicated satphones with high-gain antennas can alleviate this issue, but they are not widely available or affordable to most consumers. Instead, modern satellites like Starlink [1, 2], Iridium [3, 4], Globalstar [5], AST [6, 7], and Lynk [8] operate in LEOs at the altitude of 340–2,000 km to be closer to phones/IoTs for faster network speed, lower energy costs, and more affordable hardware. Due to each LEO satellite's smaller coverage, a *constellation* with 10s–1,000s satellites is typically adopted for global coverage.



(a) Transparent satellite pipe [20, 44] (option 8 in Fig.5a)



(b) Onboard RAN-DU [20] (option 2 in Fig.5a, also used in Starlink [33–36])



(c) Onboard RAN [20] (option 1 in Fig.5a, full-fledged satellite base stations)

Figure 2: State-of-the-art cellular network function splits in 3GPP NTN and Starlink for direct-to-cell satellites.

2.2 Incentives for LEO Satellite Multi-Tenancy

The traditional satellite network market is monopolistic: Each SNO independently launches its own satellites (most of which reside in the GEO) to operate its own network services using dedicated satellite-specific radio spectrums. While feasible for standalone GEO satellites for customized terminals, this dedicated mode is not technically feasible or commercially profitable for both MNOs and SNOs in direct-to-cell LEO constellations due to three fundamental constraints:

1. MNOs: Scarce orbital slots. LEOs are highly crowded, with about 8,300 satellites [29] and 27,000 space junks [28], leading to 3,500–60,000 conjunction events per month [45, 46] and 24,410 collision avoidance maneuvers per year [47]. The recent mega-constellation deployment further congests LEOs and raises collision risks [30] and RF interferences [48]. This situation leads to more stringent and time-consuming orbit allocations (typically years) by ITU [49]. There are insufficient orbital slots to accommodate all MNOs' satellites.

2. SNOs: Shortage of licensed 4G/5G spectrums. To be compatible with commodity phones/IoTs, direct-to-cell satellites should use the legacy 4G/5G spectrums, most of which have been allocated to terrestrial MNOs by ITU/FCC. SNOs alone do not have sufficient licensed cellular spectrums to offer direct-to-cell services.

3. MNOs & SNOs: Prohibitive capital costs. Despite advances in satellite miniaturization and rocket reusability, deploying a LEO constellation is still capital-intensive for both MNOs and SNOs [31, 32]. Building dedicated direct-to-cell LEO constellations raises the barrier to entry for MNOs and lowers the commercial return on investment for SNOs.

To this end, SNOs like Starlink [10–12, 33, 50], AST [6, 7, 51], and Lynk [8] have recently partnered with global MNOs (e.g., T-Mobile, AT&T, Vodafone, and KDDI) to advocate *multi-tenant* direct-to-cell satellite services. FCC has recently proposed a regulatory framework to foster collaborations between SNOs and MNOs, allowing SNOs to utilize spectrum previously allocated only to MNOs through lease

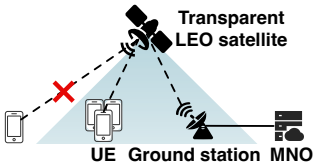
(a) Direct-to-cell UEs

(b) Open probes [54]

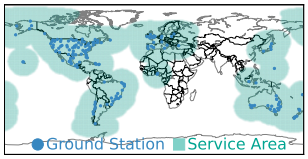
Dataset	Scale	Description
TLEs [55]	50,922,755 Two-Line Elements	Real LEO satellites' ephemeris
Iridium RF	943,003 RF link measurements	From our Iridium RF receivers
RIPE open probes [54]	7,800,177 ping and 382,464 traceroute logs	From 50 open Starlink dishes as network probes
GEO direct-to-cell	3,903,065 PHY, MAC, RLC, RRC, & NAS logs	From Thuraya X5-Touch and Tiantong T900 satphones

(c) Real satellite dataset

Figure 3: Equipment and satellite data in our empirical study.



(a) Reliance on ground stations (GS)



(b) Servicable areas limited by GS

Figure 4: Limited coverage for transparent LEO satellites.

agreements or partnerships [52]. Each MNO rents SNOs' satellites and grants its licensed 4G/5G spectrums to them for direct phone/IoT connection. The SNO's satellites can host multiple MNOs' direct-to-cell services, while each MNO can rent diverse SNOs' satellites for coverage and cost optimizations. Akin to cloud computing, LEO multi-tenancy yields a win-win situation: it lowers the barrier to entry for MNOs, complements MNOs' terrestrial coverage in under-served areas at low costs, and increases SNOs' revenues and return on investment through economies of scale. UEs also benefit from it through a more open, competitive market that offers cheaper satellite data plans and complementary coverage[1].

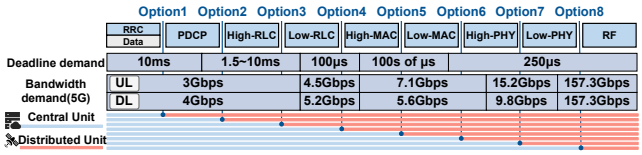
3 Challenges for Multi-Tenant LEO Satellites

While appealing to SNOs, MNOs, and UEs, multi-tenant direct-to-cell LEO satellites are still in their infancy. The ongoing direct-to-cell satellite solutions under development, such as Starlink [34–36] and 3GPP NTN [16–27], have not started to support multi-tenancy. Our empirical study driven by real satellite data in Figure 3 shows that traditional 2G/3G GEO transparent pipe sharing model is not feasible for LEOs due to its incomplete coverage, 4G/5G deadline violations, and unaffordable bandwidth demands (§3.1). While moving 4G/5G functions to LEO satellites can resolve this issue, its multi-tenancy can be impeded by the cellular hop-by-hop stateful session's need for tight functional coupling and stable service relationships among SNOs, MNOs, and UEs (§3.2).

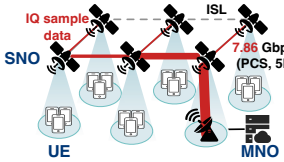
3.1 Transparent Satellite Pipe as a Service?

Sharing satellite access is a decades-old practice. Most GEO satellites today are transparent physical pipes that can multiplex radio signals from diverse MNOs. Similar to fiber rentals, it is easy for MNOs to rent these satellites to relay raw RF signals (IQ samples) between UEs and terrestrial infrastructure, as shown in Figure 4a. This physical network slicing has

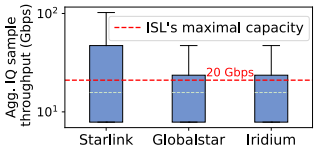
[1]The latest Huawei Mate 60 Pro+ smartphone has supported both Tiantong and Beidou satellites for messaging services [53].



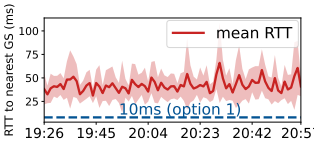
(a) Standard functional split options in 4G/5G radio functions [56–58]



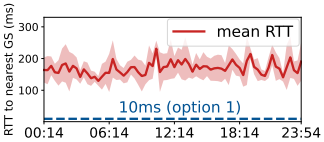
(b) IQ sample data aggregation at ISLs



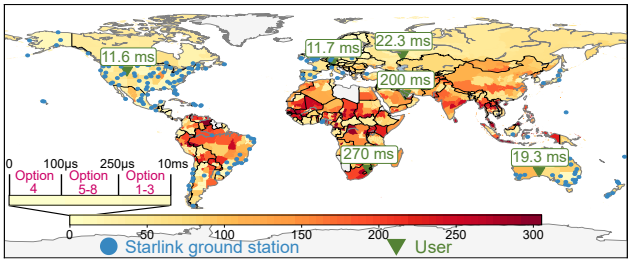
(c) Aggregated IQ sample throughput



(d) Starlink terminal's RTT w/o ISLs



(e) Starlink terminal's RTT w/ ISLs



(f) RTT between the user and its nearest ground station (w/ ISLs)

Figure 5: Transparent satellite pipes cannot meet 4G/5G's deadline and throughput requirement for basic functionality.

been successful in direct-to-cell GEO satellites for low-speed 2G/3G. It is a preferable multi-tenancy model for both MNOs and SNOs due to its *simplicity* and *transparency*: MNOs can rent satellites on demand as plug-and-play pipes without exposing their internal network functions to SNOs, thus retaining complete control of their mobile services. SNOs can transparently accommodate and isolate diverse (potentially competitive) MNOs without deploying or maintaining complex, per-MNO cellular functions in satellites.

Unfortunately, while excellent for low-speed 2G/3G GEO satellites, this transparent pipe sharing becomes technically infeasible for faster 4G/5G LEO satellites for three reasons:

(1) Incomplete coverage: For a standalone satellite, UEs and ground stations must concurrently reside in its coverage for functional direct-to-cell services. While acceptable for GEO satellites with broad coverage, this issue limits low-coverage LEO satellites' globally serviceable areas (Figure 4b). Most "unconnected" UEs today [9] reside in remote areas with sparse or no ground stations, thus leaving them still disconnected by bent-pipe satellites. This prevents MNOs from expanding their services to these UEs for more revenue.

(2) Missed 4G/5G radio deadlines: To expand coverage, LEO satellites can be networked via inter-satellite links (ISLs) to reach remote ground stations. However, such a networked physical pipe still cannot meet 4G/5G's basic functional requirements. From the functional split view, networked trans-

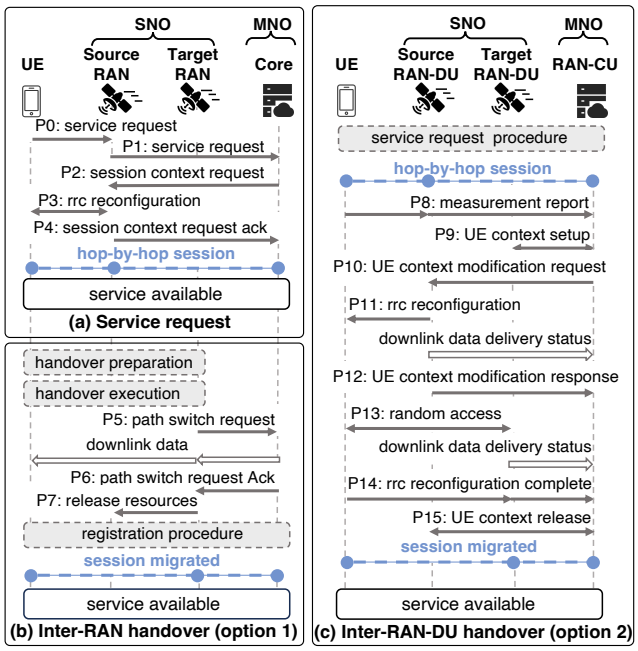


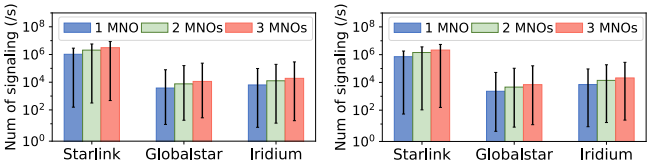
Figure 6: Signaling procedures in 4G/5G satellites [61].

parent LEO satellites essentially follow the standard 3GPP option-8 split [56–58] in Figure 5a: Each satellite relays all serving UEs' RF IQ samples to remote ground stations for processing. For functional correctness, each 4G/5G IQ sample should be delivered to the ground station within 250μs, which accounts for $\leq$80 km distance between the satellite and ground station [56]. This stringent deadline is unsatisfiable for LEO satellites, whose distance to ground stations is at least 340 km. Figure 5d–5e quantify real Starlink dish terminals' ping RTTs to ground stations with/without multi-hop ISL traversals based on the RIPE's global open Starlink dish probes [54]. Both RTTs exceed the 250-μs deadline by 1–2 orders of magnitude, thus crashing the basic 4G/5G radio functions if the transparent satellite pipe model was used.

(3) Unaffordable bandwidth demands: Each transparent LEO satellite relays all serving UEs' IQ samples to ground stations for processing, which requires 7.86 Gbps for a typical 5 MHz radio channel [58]. As shown in Figure 5b–5c, all these satellites' IQ samples to remote ground stations will accumulate and congest ISLs (typically with $\leq$20 Gbps capacity [59, 60]), thus further impeding the 4G/5G functionality.

3.2 In-Orbit Cellular Function as a Service?

To avoid the transparent LEO satellite pipe's constraints for functional 4G/5G in §3.1, SNOs have started to offload terrestrial cellular network functions to LEO satellites. Starlink Gen 2 satellites have equipped medium access control (MAC) and radio link control (RLC) protocols [33, 34], resulting in an option-2 split in Figure 5a. 3GPP also extends its 4G/5G standards to place partial (option-2) or complete (option-1) radio functions to satellites [20] (Figure 2b–2c). Due to the stringent 4G/5G radio deadlines in Figure 5a, only option-1 and 2 split are feasible for LEO satellites. Both localize the radio processing to avoid missing deadlines and wasting ISL bandwidth, thus enabling functional direct-to-cell 4G/5G LEO satellite services for commodity phones/IoTs.



(a) RAN (Option 1 in Fig.6b) (b) RAN-DU (Option 2 in Fig.6c)

Figure 7: Signaling storms in satellites under LEO mobility.

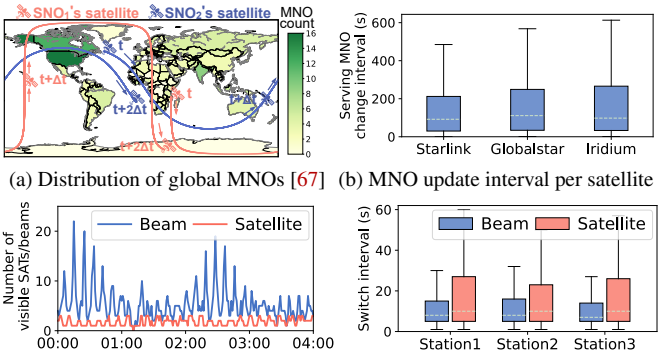
However, in-orbit cellular functions are not easily sharable from the multi-tenancy perspective. Localizing radio processing in satellites in Figure 2 requires exposing MNOs' internal cellular functions to SNOs' fast-moving, intermittently accessible, and potentially untrusted LEO satellites. This functional split among SNOs, MNOs, and UEs becomes burdensome in LEO satellites due to two fundamental issues:

Tight functional coupling: Today's cellular network adopts a stateful session-based architecture for carrier-grade services. To activate services for each UE, it should set up a hop-by-hop session across the UE, base station, and core network that binds this UE's ID, QoS, billing, and security states. As the UE moves, it should migrate this hop-by-hop session to the new infrastructure node to retain services. This stateful session implies a tight coupling and trust among the UE, base station (SNO), and core network (MNO) to coordinate for successful network services, which is not flexible or efficient for LEO multi-tenancy from all participants' perspectives:

○ *SNOs: Signaling storms.* The exposure of MNOs' stateful cellular functions can exhaust the SNO's satellites. Each LEO satellite has a short-lived coverage for each area due to its fast mobility (e.g., ≈3 minutes for a Starlink satellite at 7.6km/s). When entering a new area, each LEO satellite should take over all active UEs' sessions from the previous LEO satellite for continuous services. The session migration procedures differ between functional split options (F1AP [62] for option-2 between local and remote radio functions in Figure 6c, and NGAP [63] for option-1 between RAN and core network in Figure 6b). We evaluate the number of signaling processed by each satellite per second in Figure 7. Since each satellite can cover multiple MNOs (each having 1,000s of UEs), both incur signaling storms for LEO satellites.These signaling storms can consume up to 15.75% of total ISL bandwidth.

○ *MNOs: Restricted serviceable areas.* Due to the tight functional coupling, the radio functions in SNOs' satellites in Figure 2b–2c still rely on the remote ground station to contact the terrestrial MNO and fetch each UE's session states for functioning. Note that most LEO satellites today do not have active or reliable ISLs[2]. Similar to transparent pipes in §3.1, they cannot serve UEs in remote areas without ground stations even if physically covering them (Figure 4). To independently

[2]For example, Starlink's ISLs have just started to operate in limited areas [64, 65], while AST, Globalstar, and Lynk's satellites have no ISLs.



(a) Distribution of global MNOs [67] (b) MNO update interval per satellite

(c) Visible satellites/beams (Iridium RF) (d) Switch interval (Iridium RF)

Figure 8: Dynamic SNO-MNO-UE service relationship.

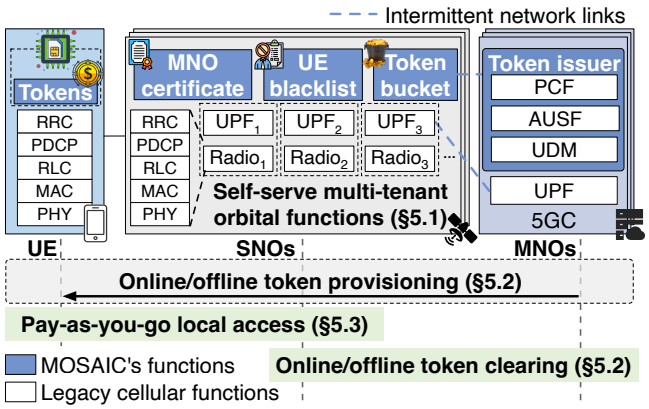


Figure 9: Overview of MOSAIC.

function without MNOs, SNOs' satellites should equip the MNO's full-fledged radio and core network functions, which becomes out of the MNO's control due to satellites' intermittent accessibility and is vulnerable to attacks in space [66].

○ *MNOs & UEs: Inflexible use of competitive SNOs.* As explained in §2.2, both MNOs and UEs have incentives to employ diverse LEO satellites from multiple SNOs. However, the hop-by-hop session in Figure 2 limits this flexibility due to its tight coupling among the UE, SNO, and MNO. As the LEO satellite moves, the UE and its MNO should re-establish or migrate the session among *competitive* SNOs' satellites that are unlikely to coordinate directly. Re-establishing the session requires the new satellite to fetch UE states from the ground station, thus limiting its serviceable area without ISLs (Figure 4). Migrating the session from the old SNO's satellite to the new SNO's satellite is prohibitive due to its competitive nature. While the MNO's terrestrial core can bridge these satellites for indirect migration, its reliance on remote ground stations again limits LEO satellites' serviceable area.

Dynamic SNO-MNO-UE service relationship: Due to the extreme LEO mobility, the many-to-many relationship among SNOs, MNOs, and UEs fluctuates and challenges everyone:

○ *SNOs: Exhaustive MNO reconfigurations.* Each SNO aims to host as many MNOs as possible for higher revenue. As its LEO satellite moves, it should repeatedly reconfigure its serving MNOs in the current coverage (Figure 1 and 8a–8b). With its cellular functions coupled to MNOs, this reconfiguration is more exhaustive than transparent pipes in §3.1 since it should manage MNOs' per-UE stateful sessions (Figure 7).

○ *MNOs & UEs: Dynamic trust establishment.* As shown in Figure 8c–8d, the LEO satellites that each MNO and UE can employ change over time due to their transient coverage, some of which can be untrusted. They must frequently re-establish trust with the new incoming satellite to maintain the hop-by-hop stateful session (thus signaling storms), which is hardly scalable to LEO constellations from diverse SNOs.

4 Overview

We propose an alternative cellular network scheme for multi-tenant direct-to-cell LEO satellites to address issues in §3. Our solution, MOSAIC, adopts in-orbit cellular functions for functional direct-to-cell 4G, 5G, and beyond (§3.1). As shown in §3.2, these functions are not easily sharable due to their tight functional coupling and demands for stable service relationships among SNOs, MNOs, and UEs. Both are hard to meet in SNOs' LEO satellites due to their fast mobility, intermittent accessibility to remote MNOs when ISLs are not reliable or active, and 3rd-party nature for MNOs and UEs.

To this end, MOSAIC shifts from the hop-by-hop session-based cellular service to **pay-as-you-go satellite self-service**. We note that, the need for tight cellular functional coupling and stable service relationships stems from the hop-by-hop stateful session for carrier-grade services. These sessions are inherently vulnerable to extreme LEO mobility, intermittent satellite connectivity, and untrusted satellites. Instead, the pay-as-you-go paradigm is more suitable for sharing mobile infrastructure like LEO satellites[3]: Analogous to bike-sharing systems [37], each UE pays for its runtime satellite access on-demand using one-time digital tokens ("coins") provisioned by its MNO. *Any* SNO's authentic local satellite can accept these tokens to serve this UE without pre-establishing sessions or contacting MNOs. When this satellite can reach the remote MNO (e.g., using ISLs immediately if available or geographically covering the MNO's ground station afterward), it uses these tokens as a proof of service to charge the MNO for on-demand satellite leasing. This scheme departs from the session-based cellular architecture today in three aspects:

○ *Loose coupling:* Each satellite locally self-service UEs without contacting remote ground stations or other satellites. This alleviates the coordination among satellites and ground stations, prevents signaling storms for SNOs under fast LEO mobility, lets the MNO employ remote satellites for more coverage and revenue, and facilitates the UE to take advantage of diverse (potentially competitive) SNOs' satellites.

○ *Simple SNO-MNO-UE relationship:* The pay-as-you-go tokens eliminate stateful sessions and decouple the SNO, MNO, and UE to benefit everyone. SNOs' in-orbit cellular

[3] Although the pay-as-you-go paradigm has also existed in terrestrial mobile networks (e.g., prepaid SIM cards [68]), it is still based on hop-by-hop stateful session. It is infeasible for multi-tenant LEO satellites (§3.2).

functions become near-stateless, thus more transparent and friendly for multi-tenancy (similar to pipes in §3.1). MNOs and UEs can rent SNOs' satellites on demand, which is more cost-effective and flexible to use competitive satellites.

○ *Retain carrier-grade services:* Despite near-stateless, this scheme can still retain carrier-grade services. Its pay-as-you-go paradigm naturally enforces the billing and even avoids traditional 4G/5G's exhaustive charging data signalings [69]. The MNO can embed the roaming, QoS, and other policies into its digitally signed tokens. When accepting these tokens from the UE, the SNO's satellite locally learns these policies to enforce them without directly contacting remote MNOs.

Figure 9 overviews MOSAIC. It comprises multi-tenant satellites from SNOs, the remote terrestrial home network from MNOs, and UEs. Satellites equip full-fledged radio and data-plane functions to meet the 4G/5G deadline and bandwidth requirements in §3.1. These functions' states are decoupled from satellites and provisioned by UEs' tokens, resulting in near-stateless satellites for transparent multi-tenancy. To foster self-service, MOSAIC does *not* mandate ISLs in these satellites for immediate access to ground stations. Each MNO's terrestrial home maintains its subscriber database and provisions its UEs with self-certified, policy-embedded tokens online or offline. For compatibility and integration with terrestrial 4G/5G, the MNO can reuse its core network to manage these tokens. The UEs store these tokens in their SIM cards and feed them to the satellites via runtime in-band signaling for pay-as-you-go services.

5 Design of MOSAIC

This section addresses three key issues to realize MOSAIC:

1. How can SNOs arrange cellular functions in satellites to enable self-serve multi-tenancy for MNOs (§5.1)?
2. How can MNOs generate pay-as-you-go tokens for UEs to leverage these 3rd-party satellites, while still retaining full (remote) control of carrier-grade services (§5.2)?
3. How can UEs consume these tokens to access any local authentic satellites without mutual trust (§5.3)?

5.1 Self-Serve Multi-Tenant Orbital Functions

The first step for MOSAIC is to enable multi-tenant direct-to-cell LEO satellites for SNOs that are (I) as easily sharable by MNOs as the classic transparent pipe in §3.1 but (II) free of their incomplete global coverage, 4G/5G deadline violations, and bandwidth exhaustion in §3.1. The challenge arises from the tension between both goals: (II) requires pushing 4G/5G functions to LEO satellites, which, however, suffers from tight coupling with MNOs and complex SNO-MNO-UE service relationship dynamics to meet (I). As explained in §3.2, this tension is rooted in the hop-by-hop stateful cellular session.

To this end, MOSAIC rearranges orbital cellular functions to be self-contained for (II) and decouples them from stateful sessions for (I). Each satellite deploys self-contained 4G/5G radio and user-plane functions, so that it can independently serve local UEs in case its ISLs are unavailable/disrupted and remote ground stations are unreachable. These orbital cellular functions are near-stateless and decoupled from MNOs: At runtime, they are locally driven by UE-paid tokens in §5.2 provisioned by MNOs that have embedded session states, thus free of signaling coordination (storms) among satellites and ground stations. This design allows the satellite to serve MNOs and UEs on demand and approximates the transparent pipes in §3.1. We next elaborate on the design details.

Step 1: Self-contained cellular functions in satellites. As shown in Figure 9, each satellite in MOSAIC runs full-fledged 4G/5G radio functions as standalone base stations, thus free of deadline violations and bandwidth exhaustion in §3.1. It also integrates the core network's user-plane functions to enforce self-serve traffic forwarding, QoS, and billing. To decouple these functions from stateful sessions, MOSAIC adopts a session state proxy in each satellite. This local proxy emulates control-plane core network functions for radio and user-plane functions through the standard interface (e.g., NGAP [70] and GTP-U [71] in 5G). It provisions session states to these functions using the UE-paid tokens rather than contacting remote MNOs. Even without ISLs or access to ground stations in the worst case, each satellite can still independently offer the minimum 4G/5G access and local communications (e.g., voice calls, short messaging, and emergency SoS) for UEs inside its coverage[4]. In this case, each satellite can still serve a considerable number of users due to its wide coverage area (covering 10–100s km^2). When the ground stations or ISLs are available, it can also roll back to the legacy 4G/5G to fetch states from MNOs (though not necessary in most cases).

Step 2: Transparent MNO multi-tenancy. To serve a MNO in a given geographic area, each LEO satellite should be (1) granted the use of this MNO's licensed 4G/5G spectrum in this area for the radio access; (2) paid by UE-side tokens for runtime on-demand access; and (3) paid by MNOs offline by showing these tokens as a proof of service. To achieve this, the MNO signs a digital certificate for each authorized satellite that binds its spectrum grant with this satellite's identity and geographic areas to use these spectrums. The satellite stores this certificate as a proof for UEs and global spectrum regulators to offer services when covering the target areas. At runtime, it accepts UE-side tokens with session states to drive its stateless 4G/5G functions, and saves these tokens into its bucket for offline financial clearing with the MNO (detailed in §5.2). Its stateless functions can easily accommodate multiple MNOs with their certificates for multi-tenancy.

Step 3: Dynamic multi-tenancy under LEO mobility. As the LEO satellite rapidly moves over time, its serving MNOs also update frequently in response to its terrestrial coverage change. MOSAIC's stateless nature simplifies this process: Except for 4G/5G spectrums and cell broadcast information,

[4]Services that mandate ground stations (e.g., emergency calls) may be unavailable but are also impossible for *any* solutions without the connectivity.

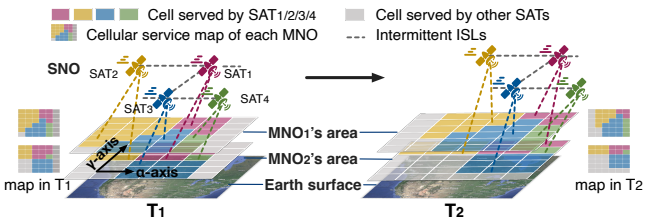


Figure 10: MOSAIC's dynamic multi-tenancy.

no other stateless cellular functions need to be reconfigured since they are decoupled from MNOs. To further simplify spectrum and cell information updates, MOSAIC inherits the idea of Earth-fixed geographic cells from SpaceCore [66] that are stable despite satellite mobility and coupled with each MNO's location-dependent roaming, QoS, billing, and spectrum policies. As shown in Figure 10, for each MNO, MOSAIC defines its 4G/5G cells by their geography rather than their fast-changing service satellites. MOSAIC uses SpaceCore's affine spherical coordinate to align these cells with satellites' orbital parameters, thus linearizing each satellite's runtime mapping of cells it covers. Each satellite tracks its runtime coverage to these cells, determines the MNOs to be served using its local certificates, and reconfigures its radio with these MNOs' 4G/5G spectrums and geographic cell identity. If a LEO constellation is used, the successive satellite takes over a cell's service when the previous satellite leaves, while retaining the same geographic cell and service area IDs for UEs to prevent unnecessary handover signalings.

5.2 Pay-as-you-go Satellite Access Tokens

The key enabler of MOSAIC's pay-as-you-go satellite access is its self-certified, policy-embedded one-time tokens. These tokens are generated by MNOs, provisioned to each UE online or offline based on its data plan, consumed by UEs for runtime local satellite access, and deposited by SNOs back to MNOs online or offline to close the financial transaction. The issue is that, UEs in remote areas may have to pay these tokens for access to SNOs' satellites out of the MNO's reachability. This threatens the MNO's controllability of its carrier-grade services and the security of the token-based satellite access.

To this end, MOSAIC adopts the idea of cryptographic "restrictive blind signatures" from the offline cash system [38] as provably secure one-time tokens, embeds MNO-controlled carrier-grade service policies into these tokens for remote enforcement, locally restrains the UE-side token misuses using MNO-issued tamper-resistant SIM cards, and lets the MNO detect and penalize token misuses as the ultimate defense.

Semantics of policy-embedded tokens: Each one-time token represents a unit of satellite access defined by each MNO based on its billing granularity (e.g., by time, data volume, message count, satellite access count, or even flat rate). The MNO embeds each UE's location-dependent roaming, QoS, billing, access control, and other policy states into its token to enforce its carrier-grade services. The UE should consume the corresponding token to enjoy the carrier-grade satellite services in the target location. Some MNOs may want to offer multi-stage data plans (e.g., "$10/GB for the first 5GB data, then free data throttled at 128Kbps afterward"). MNOs can enforce them by generating and provisioning a corresponding quota of differentiated tokens for each stage for the UE.

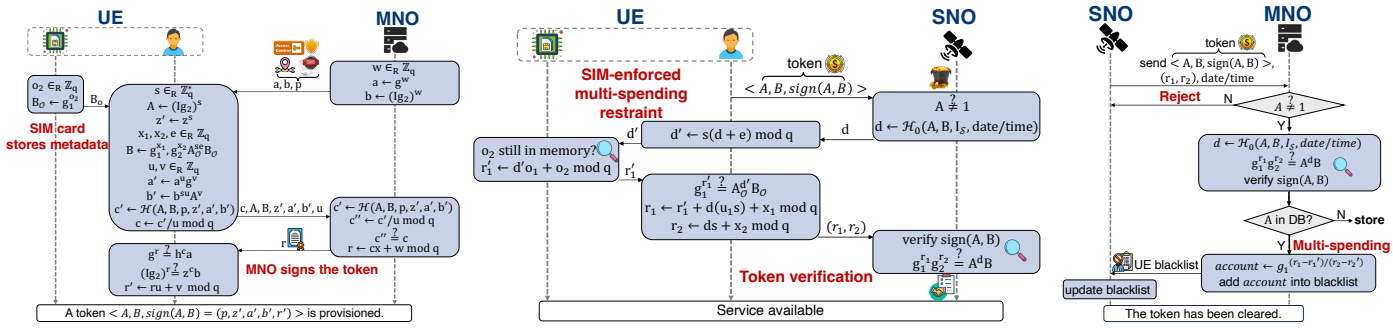
Token generation/provision: Figure 11a illustrates how each MNO generates and provisions tokens for each UE. It can occur *offline* (e.g., the user purchases the prepaid SIM card with pre-generated tokens) or *online* (e.g., the MNO offers tokens to the UE after the initial registration over a secure channel). It involves the MNO, the UE, and the MNO-issued tamper-resistant SIM card to the UE. There are two steps:

(1) Preparation: The MNO maintains two databases for the UE subscription and the consumed token, both of which can reuse the legacy cellular functions (detailed in §6). Following the one-time restrictive blind signature [38], the MNO generates tokens using a function pair $\langle x, (G_q, g, g_1, g_2, \mathcal{H}, \mathcal{H}_0)\rangle$, where x is its private key, (G_q, g, g_1, g_2) is its public key for UEs/SNOs based on the classic discrete logarithm challenge [72], $\mathcal{H}$ is a public hash function for token generation and verification, and $\mathcal{H}_0$ is a public hash function for SNOs for runtime challenge-response token verification.

(2) Token generation: To create a token, the MNO generates a random number pair (a, b) to the UE. The UE with the SIM card generates a third random number o_2, and computes a number pair (A, B) based on (a, b, o_2) and MNO's public key in Figure 11a. To restrain the UE-side token manipulation, the SIM card stores o_2 for later token consumptions below. To retain MNOs' control of carrier-grade services, MOSAIC extends [38] to bind (A, B) with each UE's session states p (including but not limited to location-dependent roaming, QoS, billing, and access control profile) using the hash function $\mathcal{H}$ and digitally signs this binding as $sign(A, B)$ using the private key x. The UE stores the tuple $\langle A, B, sign(A, B)\rangle$ as a token.

SIM-enforced one-time token consumption: At runtime, the UE should consume a token to "pay" for the satellite access (whose detailed workflow will be introduced in §5.3). For correctness, each token should be used at most once. This can be threatened by selfish or malicious UEs that attempt to gain free or unauthorized satellite access by spending a used token multiple times. The issue is that such token misuse may occur for SNOs' satellites in remote areas without ISLs or ground stations to reach the MNO for detection. MOSAIC must mitigate it locally *without* the remote MNO's assistance.

MOSAIC mitigates this threat using SIM card-assisted token consumption. Recall that SIM cards are MNO-issued, trusted, and tamper-resistant hardware that are resilient to UE manipulations. They are mandatory for UEs to gain network services. Although the MNO may not be able to remotely detect the UE's token multi-spending, it can delegate this task to SIM cards for local enforcement. A side benefit of this approach is that its local nature also saves time and signaling overhead among satellites and remote ground stations.



(a) Token generation/provision (MNO–UE) (b) One-time token consumption (UE–SNO) (c) Token clearing (SNO–MNO)

Figure 11: MOSAIC's self-certified, policy-embedded one-time tokens for pay-as-you-go local satellite access.

Figure 11b shows how the SIM card enforces the local one-time token consumption. To use a token $\langle A, B, sign(A,B)\rangle$, the UE should correctly respond to the serving satellite's runtime challenge. The computation of this response must rely on this token's random number o_2, which has been stored in the SIM card and never exposed to the UE. After computing the response and receiving the token acknowledgment from the satellite, the SIM card deletes this token's o_2. In case the UE double-spends the same token, its random number o_2 has been lost in the SIM card, thus failing to pass the satellite's challenge and access network services. If the token packet from the UE to the satellite is lost over the air, the UE will not receive the token acknowledgment. After the timeout, the UE will retransmit the token for loss recovery.

Token clearing and misuse penalty: Upon the successful token validations, the SNO's satellite obtains the token as proof of its service to remote UEs. It can use this token to charge the corresponding MNO by usage, as shown in Figure 11c. This token-clearing process can happen either *online* (if the satellite has ISLs or direct ground station coverage to contact the MNO) or *offline* otherwise. For the latter case, the LEO satellite should maintain a token bucket to hold and clear these tokens when it eventually moves to cover the MNO's accessible ground station. On receiving the satellite's tokens, the MNO records them in its consumed token database. In the extreme case when the UE manages to crack the SIM card to bypass its local one-time consumption enforcement (though we are unaware of such attacks for commercial SIM cards in reality), the MNO can still eventually detect it by comparing the multi-spent token with those in this database. In this case, the MNO notifies this UE to the satellite. The satellite blacklists this UE and later denies its service as a penalty. In practice, this process is responsive due to the considerable ground station availability, as we will evaluate in §7.

5.3 Local Self-Service via In-Band Control

We last present how the UE can use tokens in §5.2 to enjoy self-serve satellites in §5.1. Its policy-embedded tokens can locally drive the stateless satellite to establish and enforce carrier-grade services via *in-band control*, thus alleviating the need to redirect to MNOs via ISLs (which may not always exist in LEO satellites today) or ground stations (unavailable in remote areas in Figure 4b) and minimizing the amount of signaling. To maximize the number of phones/IoTs (thus higher revenue) to enjoy MOSAIC, this in-band control requires no changes for the UE hardware or the standard 4G/5G interfaces between UE and infrastructure.

Preparation: Each UE should prepare tokens in §5.2 in advance for later satellite access. There are various ways to achieve this. For example, the UE can purchase prepaid SIM cards with tokens, retrieve tokens from the MNO offline through a secure out-of-band channel (e.g., WiFi or terrestrial 4G/5G when available), or fetch tokens from the MNO online through an end-to-end secure channel if the serving satellite has active ISLs or direct coverage to the ground station.

Service establishment: There are two scenarios:

• *Uplink service:* When the UE needs to send data but has no active connectivity, it should establish the service with a satellite. As shown in Figure 12, the UE first selects *any* authentic satellite based on its broadcasted identity and sets up an insecure radio connectivity. It then performs identity-based cryptography [73, 74] to authenticate this satellite using its certificate in §5.1 and negotiates local security keys for this connection, thus preventing the invocation of signaling procedures with the remote MNO. After the successful satellite authentication, it follows the procedure in Figure 11b to consume token. For backward compatibility, the UE and satellite piggyback the necessary information using standard transparent ULInformationTransfer and DLInformationTransfer message containers in 4G/5G [26, 27]. After the token consumption, the satellite extracts the UE's session states from the token to enforce carrier-grade services. If any of these procedures fails, the satellite will terminate the service or roll back to the legacy procedure in Figure 6 if it can reach the MNO.

• *Downlink service:* When a satellite receives data destined for the UE, it should forward the data to satellites covering this UE or establish downlink services if covering the UE by itself. The challenge is that MOSAIC's stateless satellite has no prior knowledge of the UE's location. 4G/5G resolves this issue by paging the UE through all base stations in the service area through the downlink broadcast channel, which, however, is expensive for satellites due to their broad global coverage and many more UEs to serve in these areas.

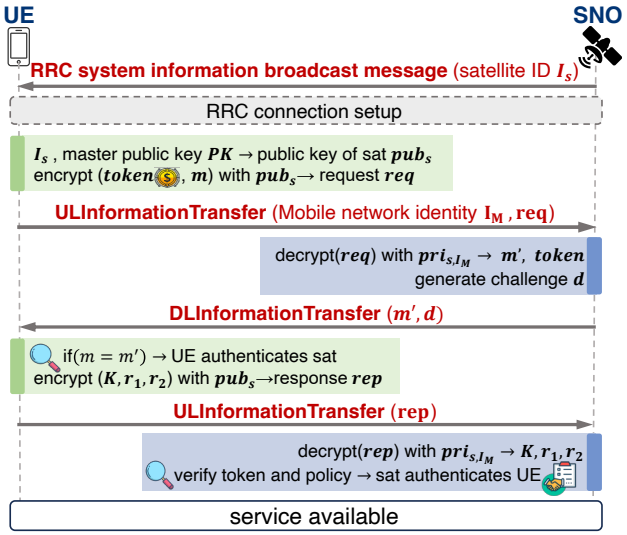


Figure 12: MOSAIC's in-band control for local self-service.

Instead, MOSAIC leverages the Earth-fixed geographic cells in §5.1 and Figure 10 as the UE's location reference for downlink services. MOSAIC assigns each UE's IP address as PLMN.cell-ID.UE-identity, where PLMN is the UE's serving MNO's 4G/5G Public Land Mobile Network identity, cell-ID is this UE's residing geographic cell's identity, and UE-identity is this UE's globally unique identity (e.g., IMSI in 4G and SUCI in 5G). This IP address is globally unique and locates this UE at the cell granularity. For each geographic cell, MOSAIC assigns one satellite per MNO that fully covers this cell to serve it. This is achieved in the LEO constellation through an initial bipartite matching between satellites and geographic cells per MNO and linear cell mapping update over time in Figure 10. For multiple MNOs, the SNO runs this matching separately to distribute MNOs among multiple satellites. Given the UE's packet with its geographic IP address, the satellite can easily detect if it is responsible for this cell for paging or locates the satellite serving this cell for geographic satellite routing. Upon receiving the paging message, the UE follows Figure 12 to establish the connectivity to this satellite and receive the downlink data.

Carrier-grade service enforcement: With the MNO-signed token from the UE, the serving satellite can extract the session states p from each token to locally enforce various carrier-grade services per UE, including but not limited to:

- *Billing:* The pay-as-you-go nature readily enforces it;
- *QoS:* The token embeds the 4G/5G QoS indicator for the satellite to run MAC-layer QoS-aware radio scheduling;
- *Location-based service:* The MNO can embed the applicable location in each token to let the satellite enforce location-specific roaming and access control; and
- *Customized service:* Some services (e.g., emergency calls) mandate redirecting data to specific infrastructure. The MNO can embed these rules into the corresponding token for the satellite to guide its traffic forwarding.

UE-driven mobility support: MOSAIC's pay-as-you-go paradigm implicitly moves the mobility support out of the infrastructure. The extreme LEO satellite mobility does not trigger any signaling procedures for a static UE, thus avoiding signaling storms in §3.2. The UE's in-band state provision also prevents inter-satellite coordination for handovers, letting the UE access any available and even competitive satellites. When the UE roams to a new geographic cell, it does not have to update its location to the remote core network like the legacy 4G/5G or SpaceCore [66]. Instead, it can immediately gain services using the tokens tailored to this new cell.

Security analysis: MOSAIC's pay-as-you-go paradigm departs from today's stateful session-based cellular architecture. It does not require the strong trust among the SNOs, MNOs, and UEs that is a must-have for hop-by-hop stateful sessions (§3.2), thus more dependable for multi-tenancy in untrusted satellites. Appendix A details how MOSAIC is resilient to threats from UEs, SNOs, MNOs, and external attackers to retain at least the same security as the legacy cellular network.

6 Practical Deployment

We focus on three issues for MOSAIC's practical large-scale deployment: (1) How to minimize SNOs' satellite changes for MOSAIC? (2) How to integrate MOSAIC satellites with MNOs' terrestrial 4G/5G for coexistence? (3) How to enable MOSAIC for commercial off-the-shelf (COTS) phones/IoTs?

Our deployment resolves these issues with three principles:

(I) *MOSAIC as a plug-and-play proxy:* For the minimal changes for SNO's satellites, we realize MOSAIC as an external proxy for the unmodified cellular functions.

(II) *Seamless integration with terrestrial 4G/5G:* MOSAIC reuses and incrementally upgrades terrestrial cellular core networks to integrate pay-as-you-go satellites.

(III) *SIM-based deployment for UEs:* Rather than changing the COTS UE hardware, we realize MOSAIC's UE-side logic as SIM applet and mobile app. UEs can readily enjoy MOSAIC with a new SIM card.

Figure 9 shows MOSAIC's practical deployment scheme based on these principles. It spans three participants:

• **SNO-side satellites:** Each satellite installs the legacy cellular RAN and user-plane function (UPF). It further installs a proxy that emulates the cellular core network's control-plane functions. This proxy packages MOSAIC's self-serve multi-tenant functions in §5.1. It maintains this satellite's certificates from MNOs, accepts tokens from each UE through ULInformationTransfer messages, extracts the UE's session states from tokens, and feeds these states to RAN and UPF to enforce carrier-grade services. As the LEO satellite moves, this proxy also updates its mapping to cells and MNOs (§5.1 and §5.3). To facilitate this proxy's deployment with minimal satellite changes, we can reuse operational satellites' built-in software-defined telemetry, tracking, and control (TT&C) features to remotely upgrade onboard satellite software/firmware[5].

[5]For instance, Starlink has demonstrated its quick satellite software up-

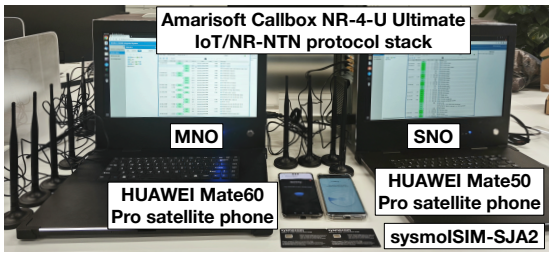


Figure 13: Our MOSAIC prototype and testbed setup.

• **MNO-side terrestrial infrastructure:** For tokens in §5.2, MOSAIC reuses the standard subscription server (HSS in 4G [77] and UDM in 5G [78]) as the user profile database and charging function (OCS/OFCS in 4G [77] and CHF in 5G [78]) as the consumed token database. It offers tokens to UEs and their SIM cards through the standard APDU interface for remote over-the-air update [79]. This unifies the management between satellite and terrestrial cellular networks. It also uses this core network to bridge the handover between satellites and terrestrial base stations using procedures in Figure 6. This integration requires no changes for terrestrial base stations.

• **UE-side SIM card:** MOSAIC requires no hardware changes for COTS phones/IoTs. Instead, each UE just needs to (1) get a SIM card from the MNO with MOSAIC's token capabilities in §5.2 as an applet and (2) install a mobile app to store tokens in §5.2 and invoke the in-band control in §5.3. This app coordinates with the SIM/eSIM via TelephonyManager APIs [80] to jointly perform procedures in §5.2–5.3. MNOs can remotely upgrade SIMs/eSIMs and provision tokens using the standard over-the-air update interfaces [81, 82].

Proof-of-concept prototype (Figure 13): We follow the above methodology to prototype MOSAIC with Amarisoft Callbox NR-4-U Ultimate [83], one of the first available 3GPP NTN software protocol stacks on the market for direct-to-cell satellite communication testing. This suite realizes full-stack 3GPP-R17 IoT/NR-NTN protocols and standard-compliant LEO satellite RF channel emulators [19]. We use one Amarisoft node to emulate the SNO's LEO satellite, one Amarisoft node to emulate the MNO's terrestrial 4G/5G core, and COTS phones/IoTs with programmable sysmoISIM-SJA2 SIM cards [84] and an Android app to realize MOSAIC's pay-as-you-go paradigms. We test various COTS UEs, including the Huawei Mate 60 Pro with direct-to-cell communication with the Tiantong GEO satellite via 2G GMR, and Huawei Mate 50 with messaging services via Beidou GEO satellites.

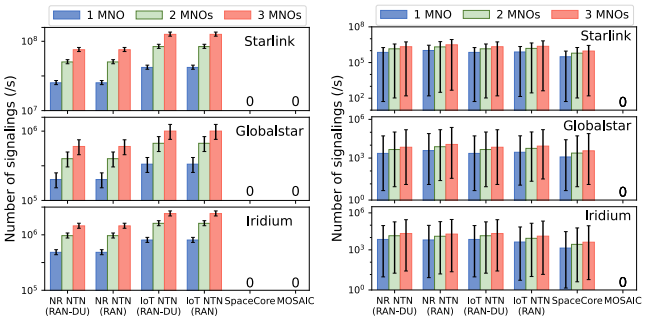
7 Evaluation

We evaluate MOSAIC using a combination of qualitative analysis, quantitative micro-benchmark test with our prototype in Figure 13, and large-scale what-if emulations driven by satellite data in Figure 3. We compare MOSAIC with state-of-the-art in Figure 2, assess its modules in §5.1–5.3, and show how it addresses issues in §3.2 for SNOs, MNOs, and UEs.

grade capability [75] to defend against jamming attacks. Spire Global deployed software-defined satellites to facilitate remote firmware upgrade [76].

	COTS UEs?	4G/5G?	Self-service?	Multi-tenancy?	Competitive satellites?
Transparent pipe	√	×	×	√	√
Starlink [33–36]	√	√	×	Partial	Partial
NR NTN (5G) [20]	√	√	×	×	×
IoT NTN (4G) [23]	√	√	×	×	×
SpaceCore [66]	×	√	√	×	×
MOSAIC	√	√	√	√	√

Table 1: Comparison of direct-to-cell satellite solutions.



(a) Service establishment (b) Satellite handover in LEO mobility

Figure 14: MOSAIC reduces signalings for satellites.

7.1 Qualitative Advantages over SOTAs

Table 1 compares MOSAIC with existing work in Figure 2 regarding their support for multi-tenant LEO satellites to directly connect COTS phones/IoTs. By offloading cellular functions to satellites, MOSAIC avoids the transparent satellite pipes' shortage of enabling satellite 4G/5G for UEs (§3.1). By replacing the stateful hop-by-hop cellular sessions with the pay-as-you-go paradigm, MOSAIC also overcomes the limitations in §3.2 to enable self-serve satellites for SNOs for transparent multi-tenancy. Its near-stateless nature lets UEs and MNOs flexibly use competitive SNOs' satellites for complementary coverage and lower costs. While SpaceCore [66] achieves similar stateless 5G core functions in LEO satellites with UE-driven state management, it cannot readily support multi-tenancy or competitive satellites because it assumes all satellites belong to a single MNO (i.e., no trust concerns). Moreover, unlike MOSAIC, SpaceCore requires modifying 4G/5G radio resource control (RRC) messages to piggyback UE states, thus not applicable to COTS phones/IoTs.

7.2 Self-Serve Multi-Tenant Orbital Functions

We assess how MOSAIC's self-service paradigm in §5.1 frees SNOs' satellites from signaling storms, expands their global serviceable areas for more revenues, and simplifies their dynamic many-to-many mapping to MNOs and UEs.

Self-contained orbital cellular functions: MOSAIC's function split in §5.1 lets each satellite independently serve UEs without interacting with other satellites or ground stations, thus free of signaling storms in §3.2. Figure 14a counts the signalings for each fast-moving LEO satellite to establish services for incoming UEs using the procedure in Figure 6a. It confirms that MOSAIC avoids the signaling storms with its

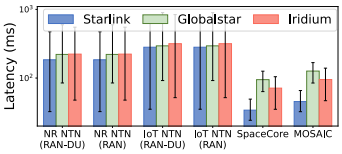


Figure 15: Service setup latency.

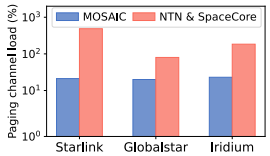


Figure 16: DL paging cost.

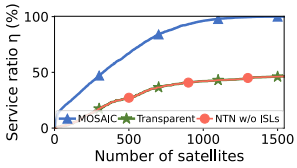


Figure 17: Serviceable areas.

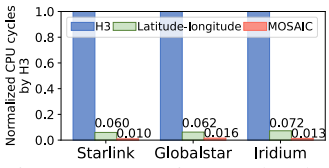


Figure 18: Mapping overhead.

pay-as-you-go tokens. Note that SpaceCore also saves signalings with its UE-driven state fetching but requires modifying 4G/5G standards, thus not applicable to COTS UEs.

A side benefit of MOSAIC's self-contained orbital cellular functions is that it saves UE-perceived service latencies. Figure 15 shows that, compared to NTN, MOSAIC's localized service setup reduces signaling latency by up to 5.19×, 1.33×, and 2.33× in Starlink, Globalstar, and Iridium, respectively. Its service setup latency is slightly higher than SpaceCore by 0.33× as a tradeoff for compatibility with COTS UEs with additional standard-compliant signalings (Figure 12).

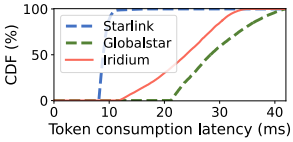
Expansion of serviceable areas: MOSAIC's self-service nature eliminates satellites' reliance on ground stations for broader serviceable areas. We define each LEO constellation's cellular service ratio as $\eta = \frac{\text{Areas with functional satellite 4G/5G}}{\text{Total areas covered by all satellites}}$. Figure 17 showcases Starlink Phase II satellites' service ratio as a function of activated satellites. When ISLs are not reliable, NTN's serviceable areas are constrained by ground stations similar to transparent pipes in §3.1. More satellites do not suffice to fulfill serviceable areas due to the non-uniform distribution of ground stations in Figure 4b. Instead, MOSAIC is free of this deficiency. It expands these satellites' serviceable areas with up to 116% increment and achieves 100% service ratio within this LEO constellation's terrestrial coverage.

Simplified dynamic many-to-many mapping: MOSAIC adopts the orbit-aligned geographic cell division in §5.1 to simplify the dynamic SNO-MNO-UE service relationships. Figure 18 compares this method with H3 hexagonal cells [85] (likely used by Starlink [86]) and the latitude-longitude rectangular cells in terms of their computation cost of dynamic re-mapping. MOSAIC reduces CPU cycles by 100×, 63×, and 73× in Starlink, Globalstar, and Iridium, respectively.

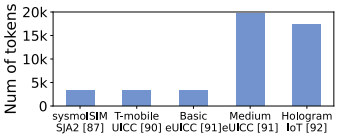
7.3 Pay-as-you-go Satellite Access Tokens

We next evaluate the efficiency, scalability, and resiliency of MOSAIC's pay-as-you-go tokens in §5.2 for MNOs and UEs.

Efficiency of token consumption: MOSAIC's localized token consumption in Figure 11b slightly incurs extra delays between the UE and satellite. As shown in Figure 19a, each token's consumption latency is $\leq$ 16.4 ms, 41.9 ms, and 34.8 ms in Starlink, Globalstar, and Iridium, respectively. They are marginal compared to MOSAIC's savings in Figure 15.



(a) Token consumption latency



(b) Num. tokens each (e)SIM can support

Figure 19: MOSAIC token's system overhead.

Scalability of SIM-enforced tokens: MOSAIC's SIM-enforced one-time token requires storing per-token metadata o_2 in the SIM card. Figure 19b quantifies the number of available 1024-bit tokens that each commercial standard-compliant SIM card (with 64KB–384KB RAM [84, 87–89]) can support. Each SIM card can store 3,279-19,661 tokens' 160-bit metadata o_2. If the MNO lets each token represent 1 minute, 1 text, or 1 MB of data quota, then the quota of MOSAIC based on today's SIM card is comparable to prepaid SIM cards on the market today (e.g., 250 units/month SIM [68])[6].

Scalability of token generation and verification: Both are scalable to many UEs. Our test with a workstation using a single core of 2.30GHz Intel Xeon Gold 5218 shows that, each MNO in this setup can generate 1,175 tokens/s (Figure 11a) and verify 1,401 tokens/s (Figure 11c) on average.

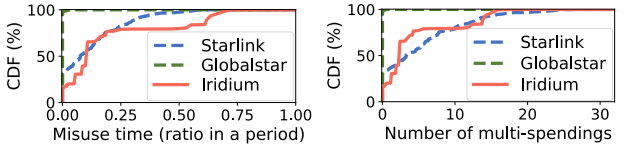
Resiliency to token manipulations: MOSAIC's SIM-enforced token consumption locally restrains most token misuse since the SIM is tamper-resistant. In the extreme case where the MNO-controlled SIM is cracked *and* remote LEO satellites have no ISLs to reach remote MNOs, the token misuse is still detected when the LEO satellites eventually move to the ground station to interact with MNOs in Figure 11c. Figure 20 quantifies this worst-case token multi-spending by a misbehaved UE before it is detected by blacklisted by SNOs and MNOs. Due to LEO satellites' fast mobility and considerable ground station availability (exemplified in Figure 4b), even this worst-case misuse time is still bounded by 0.2–0.88 satellite orbital periods (22.8–87.7 minutes). It can be further shortened if ISLs exist for timely token clearing in Figure 11c.

7.4 Local Self-Service via In-Band Control

We last assess how MOSAIC's UE-driven procedures in §5.3 localize satellite access, simplify signalings in LEO mobility, and facilitate the use of competitive SNOs' satellites.

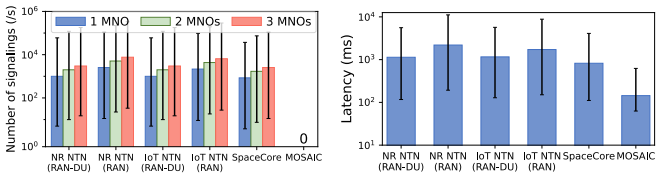
Service establishment: MOSAIC's uplink service establishment has been studied in §7.2. Its downlink service is similar to the uplink one except for the additional paging procedure, which could overload the 4G/5G broadcast channel given numerous UEs in satellites' broad coverage [91]. Figure 16 compares the paging load assuming 5MHz 4G/5G radio bands for each satellite and 400 UEs/km^2 by following [91]. Compared to NTN and SpaceCore, MOSAIC reduces paging channel load by 23×, 4×, 8× in Starlink, Globalstar, and Iridium, respectively. Note that NTN and SpaceCore's paging load can even exceed the 4G/5G channel capacity, thus unable to support numerous UEs. Instead, by mapping each

[6] Besides, existing SIM cards allow for 100,000–500,000 read/write operations during their lifetime [90], so the number of writes won't be a bottleneck.



(a) Misuse time per token (b) Misuse count per token

Figure 20: Worst-case token misuse when SIMs are cracked.



(a) Handover signaling message costs (b) Service resumption latency

Figure 21: Inter-SNO satellite handover under LEO mobility.

geographic cell to a corresponding satellite in §5.3, MOSAIC reduces the areas to page for functional 4G/5G services.

UE-driven mobility support: As explained in §5.3, MOSAIC's UE-driven primitive avoids procedures triggered by extreme LEO satellite mobility. As shown in Figure 14b, this helps MOSAIC eliminate the signaling storms in §3.2: Rather than coordination between satellites, each new satellite directly fetches each UE's states from its one-time tokens.

Use of competitive SNOs' satellites: By minimizing the coordination among satellites and ground stations, MOSAIC's in-band control lets UE freely choose *any* authentic satellite for use. Instead, existing solutions require the MNO's involvement to indirectly coordinate competitive SNOs' satellites, which require ISLs to remote ground stations and incur signaling storms. Figure 21 quantifies these costs assuming two SNOs (Starlink and Iridium) with ISLs. Compared to the state-of-the-art, MOSAIC saves 850–7,640× signaling costs and 4.71–14.25× latencies due to its local in-band control.

8 Limitations

MOSAIC is our first step to enable self-serve multi-tenant direct-to-cell satellites for our regular phones/IoTs. While encouraging, we believe that it could be further improved in at least three aspects in the future work: *(1) For SNOs*, while MOSAIC's pay-as-you-go token grants service access, it does not guarantee *verifiable* carrier-grade service. Selfish SNOs may not offer carrier-grade services after gaining tokens, thus causing overbilling. This issue could be resolved using recent two-sided measurement and negotiation mechanisms [69, 92]. *(2) For MNOs*, while MOSAIC's policy-embedded tokens let MNOs retain cellular policy control, the offloaded cellular functions to satellites still cannot be directly managed by MNOs. How to enhance MNOs' configurability and manageability of on-board satellite cellular functions deserves further research. *(3) For UEs*, MOSAIC's tokens suppress signaling overhead at the cost of some UE-to-satellite bandwidths due to its in-band control. It is worth exploring how to compress tokens for more bandwidth-efficient pay-as-you-go services.

9 Related Work

We are witnessing a boom in LEO networks. Extensive efforts have been made on LEO physical topology [30, 93], link-layer scheduling [94, 95] and handovers [96, 97], network-layer routing [98–100], edge computing applications [101, 102], and real-world measurements [103–105]. Instead, enabling direct-to-cell satellites for commodity phones/IoTs has gained less attention until recently. These LEO satellites differ from other non-terrestrial 4G/5G via balloons [106] or UAVs [107–109] due to their extreme mobility and scale in foreign outer space. Recent studies [20, 66, 110] focus on coping with these challenges for functional satellite 4G/5G. Our work complements them by exploring the sharing of direct-to-cell satellites among SNOs and MNOs as a cost-effective win-win solution.

Cellular network multi-tenancy has been a long-standing desire and is becoming a reality in terrestrial networks with the maturity of standards [111–113], cloudified 5G [114, 115], and decentralized architecture [92, 116, 117]. But these terrestrial efforts cannot support sharing the extremely mobile LEO satellite infrastructure (§3), which motivates our design. MOSAIC borrows the idea of UE-driven state management from SpaceCore [66] but extends it as pay-as-you-go tokens for multi-tenancy. MOSAIC's SIM-based solution also avoids SpaceCore's incompatibility with regular phones/IoTs.

10 Conclusion

We present MOSAIC, a pay-as-you-go solution to enable self-serve multi-tenant LEO direct-to-cell satellites for our regular phones/IoTs in under-served areas via 4G, 5G, and beyond. MOSAIC departs from today's hop-by-hop stateful cellular session that impedes multi-tenancy due to its need for tight functional coupling and stable relationships among SNOs, MNOs, and UEs. Instead, it adopts policy-embedded one-time tokens for on-demand satellite services. This paradigm realizes near-stateless satellites for transparent multi-tenancy, alleviates their reliance on remote ground stations for self-service, and alleviates inter-satellite coordinations to encourage the use of diverse satellites from competitive SNOs.

In a broader context, MOSAIC is an attempt to democratize cellular networks from space. While cellular democratization has been nice-to-have in terrestrial networks to foster market competition, it has become a must-have in space due to the mutual need for scarce orbital slots from satellite operators, licensed spectrums from mobile operators, and lowered capital costs of LEO constellations for both participants. We hope our lessons can spur this effort toward a win-win sharing of space networks *by* the community and *for* the community.

Acknowledgment: We thank our shepherd, Dr. Deepak Vasisht, and all reviewers for their valuable feedback. This work is supported by the National Key Research and Development Plan of China (2022YFB3105201) and the National Natural Science Foundation of China (62202261). Hewu Li and Yuanjie Li are the corresponding authors.

References

[1] SpaceX. SpaceX Invites World's Carriers to Collaborate: No More Cell Phone Dead Zones. https://www.spacex.com/updates/#direct2cell, 2022.

[2] FCC. Application for Mobile Satellite Service by SpaceX. https://fcc.report/IBFS/SAT-PPL-20221206-00170, Jun. 2022.

[3] Qualcomm. Qualcomm Introduces Snapdragon Satellite, The World's First Satellite-Based Solution Capable of Supporting Two-Way Messaging for Premium Smartphones and Beyond. https://tinyurl.com/yjy325am, 2023.

[4] Iridium. Iridium certus as a 5g companion for land mobile communications. https://tinyurl.com/ype5s2zt, 2019.

[5] SpaceNews. Apple lends globalstar $252 million for satellite-enabled iphones. https://tinyurl.com/4bxapj4x, 2023.

[6] AST SpaceMobile, Mobile Network Operators. https://ast-science.com/company/mobile-network-operators/, 2023.

[7] AT&T leases spectrum to AST SpaceMobile. https://tinyurl.com/mr356tha, 2023.

[8] SatelliteToday. Lynk Co-Founder Says Satellite-to-Cell Tech Will Be 'Bigger than 5G'. https://tinyurl.com/2p9a3an8, 2021.

[9] Measuring digital development: Facts and Figures 2022. https://www.itu.int/itu-d/reports/statistics/facts-figures-2022/, 2022.

[10] SpaceX and T-Mobile partner for direct-to-cellphone satellite service. https://spacenews.com/spacex-and-t-mobile-partner-for-direct-to-cellphone-satellite-service/, 2022.

[11] KDDI Signs Agreement with SpaceX to Bring Satellite-to-Cellular service to Japan. https://tinyurl.com/yax9seaz, 2023.

[12] SpaceX's Starlink signs direct-to-cell deal with Swiss telco Salt. https://tinyurl.com/2beh6rkp, 2023.

[13] Emergency SOS via satellite on iPhone 14 and iPhone 14 Pro lineups made possible by $450 million Apple investment in US infrastructure. https://www.apple.com/newsroom/2022/11/emergency-sos-via-satellite-made-possible-by-450m-apple-investment/, 2022.

[14] Samsung targets satellite-enabled smartphone chips after surprise iPhone 14 feature. https://www.cnbc.com/2023/02/23/samsung-turns-to-space-with-satellite-enabled-smartphone-chip.html, 2023.

[15] Qualcomm and Leading Global Smartphone Manufacturers Collaborate to Bring Snapdragon Satellite to Smartphones. https://tinyurl.com/yyrn7edz, 2023.

[16] 3GPP. Technical specification group services and system aspects; study on using satellite access in 5g; stage 1 (release 16). *https://www.3gpp.org/DynaReport/22822.htm*.

[17] 3GPP. TR22.926: Guidelines for Extraterritorial 5G Systems, Dec. 2021.

[18] 3GPP. TR23.737: Study on Architecture Aspects for Using Satellite Access in 5G, Jun. 2018.

[19] 3GPP. TR38.811: Study on New Radio (NR) to support non-terrestrial networks, 2020.

[20] 3GPP. TR38.821: Solutions for NR to support non-terrestrial networks (NTN), 2020.

[21] 3GPP. TR24.281: Study on PLMN Selection for Satellite Access, Sep. 2021.

[22] 3GPP. TR28.808: Study on Management and Orchestration Aspects of Integrated Satellite Components in a 5G Network, Dec. 2021.

[23] 3GPP. TS36.331: E-UTRA; Radio Resource Control (RRC), Jun. 2023.

[24] 3GPP. TS38.331: 5G NR: Radio Resource Control (RRC), Jun. 2023.

[25] 3GPP. TS24.008: Core network protocols; Stage 3, Mar. 2023.

[26] 3GPP. TS24.301: Non-Access-Stratum (NAS) for EPS, Apr. 2023.

[27] 3GPP. TS24.501: Non-Access-Stratum (NAS) for 5G, Apr. 2023.

[28] NASA. Space Debris and Human Spacecraft. https://www.nasa.gov/mission_pages/station/news/orbital_debris.html, 2021.

[29] ESA. Space debris by the numbers. https://www.esa.int/Safety_Security/Space_Debris/Space_debris_by_the_numbers, April 2022.

[30] Yuanjie Li, Hewu Li, Wei Liu, Lixin Liu, Wei Zhao, Yimei Chen, Jianping Wu, Qian Wu, Jun Liu, Zeqi Lai, and Han Qiu. A Networking Perspective on Starlink's Self-Driving LEO Mega-Constellation. In *The 29th International Conference on Mobile Computing and Networking (MobiCom)*. ACM, 2023.

[31] Starlink Hits 250,000 Customers, Elon Musk Hints: SpaceX Booking Over $300 Million/Year. https://tinyurl.com/ycxscdae, 2022.

[32] SMALLSAT RIDESHARE PROGRAM. https://www.spacex.com/rideshare/, 2023.

[33] SPACEX GEN2 DIRECT-TO-CELLULAR SYSTEM, ATTACHMENT A. https://fcc.report/IBFS/SAT-MOD-20230207-00021, 2023.

[34] Jayasuryan V Iyer, Khasim Shaheed Shaik Mahammad, Yashodhan Dandekar, Ramakrishna Akella, Chen Chen, Phillip E Barber, and Peter J Worters. System and Method of Providing a Medium Access Control Scheduler, 2022. US Patent 11,540,301.

[35] Chen Chen, Pavel Chikulaev, Sergii Ziuzin, David Sacks, Peter Worters, Darshan Purohit, Yashodhan Dandekar, Vladimir Skuratovich, Andrei Pushkin, and Phillip Barber. Low Latency Schedule-driven Handovers, 2023. US Patent 11,729,684.

[36] Jared Michael Greene, Mohammed Faraz Admani, Jacob Nelson Glueck, Sergii Ziuzin, Francesco De Paolis, Dhruv Dawar, and Christopher Yu. System and Method of Providing Access to Compute Resources Distributed Across a Group of Satellites, 2023. US Patent App. 17/955,401.

[37] Bicycle-sharing System. https://en.wikipedia.org/wiki/Bicycle-sharing_system, 2023.

[38] Stefan Brands. Untraceable Off-line Cash in Wallets with Observers. In *13th Annual International Cryptology Conference (CRYPTO)*. Springer, 1994.

[39] China makes big investments in 5G. http://english.www.gov.cn/news/topnews/202107/25/content_WS60fca12bc6d0df57f98dd886.html, Jul 2021.

[40] A Huge Investment On 5G Base Station For A Faster Network. https://howmuchhub.com/5g-base-station-cost/, 2022.

[41] Mobile network coverage. https://www.itu.int/itu-d/reports/statistics/2022/11/24/ff22-mobile-network-coverage/, 2022.

[42] Inmarsat satellite communications. https://www.inmarsat.com/.

[43] Thuraya Telecom. https://thuraya.com/.

[44] Andrew Jones. China kicks off a busy 2021 in space with communications satellite launch. https://www.space.com/china-launches-tiantong-1-03-communications-satellite, Jan 2021.

[45] Tereza Pultarova. SpaceX Starlink Satellites Responsible for Over Half of Close Encounters in Orbit. https://www.space.com/spacex-starlink-satellite-collision-alerts-on-the-rise, 2022.

[46] Average monthly conjunction rates surge from 2017 to 2020. https://spacenews.com/space-traffic-management-idling-in-first-gear/, 2020.

[47] SpaceX Constellation Status Report: June 1, 2023–November 30, 2023. https://licensing.fcc.gov/myibfs/download.do?attachment_key=25957549, Dec. 2023.

[48] ITU. ITU Radio Regulatory Framework for Space Services. https://www.itu.int/en/ITU-R/space/snl/Documents/ITU-Space_reg.pdf, 2022.

[49] ITU. Orbit/Spectrum International Regulatory Framework: Challenges in the 21st Century. https://tinyurl.com/2ar7kt2a, 2016.

[50] Starlink direct to cell. https://www.starlink.com/business/direct-to-cell, 2023.

[51] MobileEurope. AT&T, Vodafone and AST SpaceMobile Hit New Satellite-to-Mobile 5G Milestone. https://www.mobileeurope.co.uk/att-vodafone-and-ast-spacemobile-hit-new-satellite-to-mobile-5g-milestone/, 2023.

[52] FCC. Single Network Future: Supplemental Coverage from Space. https://docs.fcc.gov/public/attachments/DOC-400678A1.pdf, Feb. 2024.

[53] GSMArena. Huawei Mate 60 Pro+ debuts: a souped-up Mate 60 with satellite voice call support. https://www.gsmarena.com/huawei_mate_60_pro_unveiled_an_soupedup_mate_60_with_satellite_voice_call_support-news-59832.php, Sep. 2023.

[54] RIPE Atlas. List of Starlink Terminals as Open Network Probes. https://atlas.ripe.net/frames/probes/?search=Starlink&status=&af=&country=, 2023.

[55] Space Track. https://www.space-track.org, 2023.

[56] Open RAN functional splits, explained. https://tinyurl.com/2h8zxbbc, 2021.

[57] 3GPP. TR38.821: Study on new radio access technology: Radio access architecture and interfaces, 2020.

[58] China Mobile. Transport requirement for CU&DU functional splits options. https://www.3gpp.org/ftp/tsg_ran/WG3_Iu/TSGR3_93/Docs/R3-161813.zip, 2016.

[59] Inigo Del Portillo, Bruce G Cameron, and Edward F Crawley. A Technical Comparison of Three Low Earth Orbit Satellite Constellation Systems to Provide Global Broadband. *Acta Astronautica*, pages 123–135, 2019.

[60] Nils Pachler, Inigo del Portillo, Edward F Crawley, and Bruce G Cameron. An updated comparison of four low earth orbit satellite constellation systems to provide global broadband. In *2021 IEEE international conference on communications workshops (ICC workshops)*, pages 1–7. IEEE, 2021.

[61] 3GPP. TS38.401: NG-RAN; Architecture description. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3219, 2023.

[62] 3GPP. TS38.473: NG-RAN; F1 Application Protocol (F1AP). https://tinyurl.com/yp4mev39, 2023.

[63] 3GPP. TS38.413: NG-RAN; NG Application Protocol (NGAP). https://tinyurl.com/yc55kmaf, 2023.

[64] Reddit. Starlink's laser links are active. https://www.reddit.com/r/Starlink/comments/xsupcn/laser_links_are_active/, Sep. 2022.

[65] Ramish Zafar. Starlink Turns On Laser Satellites For Region With Four Months Long Night. https://tinyurl.com/yws6um6y, Nov. 2022.

[66] Yuanjie Li, Hewu Li, Wei Liu, Lixin Liu, Yimei Chen, Jianping Wu, Qian Wu, Jun Liu, and Zeqi Lai. A Case for Stateless Mobile Core Network Functions in Space. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2022.

[67] List of Global Mobile Network Operators. https://www.frequencycheck.com/countries/, 2023.

[68] SpeedTalk Mobile. Flexible "Pay as You Go" Phone Plans. https://speedtalkmobile.com/pay-as-you-go-phone-plans/, 2023.

[69] Yuanjie Li, Kyu-Han Kim, Christina Vlachou, and Junqing Xie. Bridging the Data Charging Gap in the Cellular Edge. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 15–28. ACM, 2019.

[70] 3GPP. TS38.413: NG Application Protocol, Oct. 2021.

[71] 3GPP. Ts29.281: 3gpp evolved packet system (eps); evolved general packet radio service tunneling protocol for user plane (gtpv1-u), Sep. 2021.

[72] Discrete Logarithm— Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Discrete_logarithm.

[73] Identity-based cryptography— Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Identity-based_cryptography, 2023.

[74] Identity-based encryption— Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Identity-based_encryption, 2023.

[75] Pentagon Impressed by Starlink's Fast Signal-Jamming Workaround in Ukraine. https://www.pcmag.com/news/pentagon-impressed-by-starlinks-fast-signal-jamming-workaround-in-ukraine, 2022.

[76] How Software-Defined Satellites Will Shape Communications. https://interactive.satellitetoday.com/via/april-2021/how-software-defined-satellites-will-shape-communications/, 2021.

[77] 3GPP. TS23.401: General Packet Radio Service enhancements for Evolved Universal Terrestrial Radio Access Network (E-UTRAN) access, Dec. 2015.

[78] 3GPP. TS23.501: System architecture for the 5G System (5GS), Apr. 2023.

[79] ETSI. Smart Cards; Card Application Toolkit (CAT) (Release 17), 2022.

[80] Android TelephonyManager API. https://tinyurl.com/5aswc3b7.

[81] GSMA. SGP.02 v4.2. https://www.gsma.com/esim/resources/sgp-02-v4-2/, 2020.

[82] GSMA. SGP.22 v3.0. https://www.gsma.com/esim/resources/sgp-22-v3-0/, 2022.

[83] Amarisoft Callbox Ultimate Series. https://www.amarisoft.com/ultimate/, 2023.

[84] sysmoISIM-SJA2. https://osmocom.org/projects/cellular-infrastructure/wiki/SysmoISIM-SJA2.

[85] H3: Uber's Hexagonal Hierarchical Spatial Index. https://s2geometry.io/, 2021.

[86] Does Starlink use H3? https://github.com/uber/h3/issues/717, 2022.

[87] T-Mobile's Triple Cut Prepaid Sim Card (64K). https://www.walmart.com/ip/T-Mobile-Triple-Cut-Sim-Card/193539534.

[88] GSMA. SGP.22: eSIM Technical Specification, Oct. 2022.

[89] Hologram Global IoT SIM cards. https://www.hologram.io/products/global-iot-sim-card/.

[90] Do I need an industrial sim card? https://support.open-m2m.com/en/knowledgebase/article/do-i-need-a-standard-or-industrial-sim-card, 2023.

[91] 3GPP. TP for TR 36.763 capturing RAN2 #114e agreements. https://tinyurl.com/46ebmru9, 2021.

[92] SVR Anand, Serhat Arslan, Rajat Chopra, Sachin Katti, Milind Kumar Vaddiraju, Ranvir Rana, Peiyao Sheng, Himanshu Tyagi, and Pramod Viswanath. Trust-free Service Measurement and Payments for Decentralized Cellular Networks. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks (HotNets)*, 2022.

[93] Debopam Bhattacherjee and Ankit Singla. Network Topology Design at 27,000 km/hour. In *ACM CoNEXT*, 2019.

[94] Deepak Vasisht, Jayanth Shenoy, and Ranveer Chandra. L2D2: Low Latency Distributed Downlink for LEO Satellites. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 151–164, 2021.

[95] Bill Tao, Maleeha Masood, Indranil Gupta, and Deepak Vasisht. Transmitting, Fast and Slow: Scheduling Satellite Traffic through Space and Time. In *The 29th International Conference on Mobile Computing and Networking (MobiCom'23)*. ACM, 2023.

[96] Jian Li, Kaiping Xue, Jianqing Liu, and Yongdong Zhang. A user-centric handover scheme for ultra-dense leo satellite networks. *IEEE Wireless Communications Letters*, 9(11):1904–1908, 2020.

[97] Senbai Zhang, Aijun Liu, Chen Han, Xiang Ding, and Xiaohu Liang. A network-flows-based satellite handover strategy for leo satellite networks. *IEEE Wireless Communications Letters*, 10(12):2669–2673, 2021.

[98] Yuanjie Li and Hewu Li and Lixin Liu and Wei Liu and Jiayi Liu and Jianping Wu and Qian Wu and Jun Liu and Zeqi Lai. "Internet in Space" for Terrestrial Users via Cyber-Physical Convergence. In *Twentieth Workshop on Hot Topics in Networks (HotNets)*. ACM, 2021.

[99] Debopam Bhattacherjee, Waqar Aqeel, Ilker Nadi Bozkurt, Anthony Aguirre, Balakrishnan Chandrasekaran, P Brighten Godfrey, Gregory Laughlin, Bruce Maggs, and Ankit Singla. Gearing up for the 21st century space race. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNets)*, 2018.

[100] Handley, Mark. Delay is Not an Option: Low Latency Routing in Space. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNet)*, pages 85–91. ACM, 2018.

[101] Bhattacherjee, Debopam and Kassing, Simon and Licciardello, Melissa and Singla, Ankit. In-orbit Computing: An Outlandish thought Experiment? In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets)*, 2020.

[102] Bradley Denby and Brandon Lucia. Orbital Edge Computing: Nanosatellite Constellations as a New Class of Computer System. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 939–954, 2020.

[103] François Michel, Martino Trevisan, Danilo Giordano, and Olivier Bonaventure. A First Look at Starlink Performance. In *Proceedings of the 22nd ACM Internet Measurement Conference (IMC)*, pages 130–136, 2022.

[104] Sami Ma, Yi Ching Chou, Haoyuan Zhao, Long Chen, Xiaoqiang Ma, and Jiangchuan Liu. Network characteristics of leo satellite constellations: A starlink-based measurement from end users. In *IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2023.

[105] Melisa López, Sebastian Bro Damsgaard, Ignacio Rodríguez, and Preben Mogensen. An empirical analysis of multi-connectivity between 5g terrestrial and leo satellite networks. In *2022 IEEE Globecom Workshops (GC Wkshps)*, pages 1115–1120. IEEE, 2022.

[106] Frank Uyeda, Marc Alvidrez, Erik Kline, Bryce Petrini, Brian Barritt, David Mandle, and Chandy Aswin Alexander. SDN in the Stratosphere: Loon's Aerospace Mesh Network. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2022.

[107] Moradi, Mehrdad and Sundaresan, Karthikeyan and Chai, Eugene and Rangarajan, Sampath and Mao, Z Morley. SkyCore: Moving Core to the Edge for Untethered and Reliable UAV-based LTE Networks. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 35–49. ACM, 2018.

[108] Ayon Chakraborty, Eugene Chai, Karthikeyan Sundaresan, Amir Khojastepour, and Sampath Rangarajan. SkyRAN: A Self-organizing LTE RAN in the Sky. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 280–292, 2018.

[109] Ramanujan K Sheshadri, Eugene Chai, Karthikeyan Sundaresan, and Sampath Rangarajan. SkyHAUL: A Self-Organizing Gigabit Network In The Sky. In *Proceedings of the Twenty-second International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing (MobiHoc)*, pages 101–110, 2021.

[110] Ruolin Xing, Xiao Ma, Ao Zhou, Schahram Dustdar, and Shangguang Wang. From Earth to Space: A First Deployment of 5G Core Network on Satellite. *China Communications*, 20(4):315–325, 2023.

[111] 3GPP. TS 23.251: Network Sharing; Architecture and functional description, Mar. 2022.

[112] 3GPP. TR28.835: Study on Management Aspects of 5G Multiple Operator Core Network Sharing Phase 2. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=4035, 2023.

[113] Multi-Tenant and Sharing (MORAN and MOCN). https://www.parallelwireless.com/products/multi-tenant-and-sharing/, 2021.

[114] Xenofon Foukas and Bozidar Radunovic. Concordia: Teaching the 5G vRAN to Share Compute. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 580–596, 2021.

[115] Binh Nguyen, Tian Zhang, Bozidar Radunovic, Ryan Stutsman, Thomas Karagiannis, Jakub Kocur, and Jacobus Van der Merwe. ECHO: A Reliable Distributed Cellular Core Network for Hyper-Scale Public Clouds. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2018.

[116] Zhihong Luo, Silvery Fu, Natacha Crooks, Shaddi Hasan, Christian Maciocco, Sylvia Ratnasamy, and Scott Shenker. LOCA: A Location-Oblivious Cellular Architecture. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[117] Zhihong Luo, Silvery Fu, Mark Theis, Shaddi Hasan, Sylvia Ratnasamy, and Scott Shenker. Democratizing cellular access with cellbricks. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021.

A Security Analysis

MOSAIC's pay-as-you-go paradigm departs from traditional stateful session-based cellular architecture. It does not require the strong trust among the SNOs, MNOs, and UEs that is a must-have for conventional hop-by-hop stateful sessions (§3.2), thus more feasible for multi-tenancy over untrusted satellites. We next show how MOSAIC is resilient to various threats from UEs, SNOs, MNOs, and external attackers to retain at least the same security as the legacy cellular network.

Mitigating threats from UEs: A selfish UE is incentivized to forge, manipulate, or multi-spend a token for unauthorized satellite access. MOSAIC avoids so with its cryptographically restrictive blind signature-based tokens (whose security has been formally proved in [38]) and SIM-assisted protection. Without the MNO's secret key x in §5.2, the UE cannot forge or modify a valid token. The MNO-issued tamper-resistant SIM locally denies the UE's token multi-spending. In the worst rare case that the SIM is cracked, the MNO can still detect the multi-spending and permanently blacklists this UE for later services in the clearing phase in Figure 11c.

Mitigating threats from MNOs: A selfish MNO may want to save its rental bill to SNOs by underclaiming its usage of satellites. MOSAIC's pay-as-you-go tokens offer undeniable proof of service for SNOs to mitigate this threat: Each satellite can only gain the token if serving the UE. By showing the ownership of these tokens to trusted third parties (e.g., court), the SNO can prove the MNO's satellite usage for charging.

Mitigating threats from SNOs: An unauthorized satellite without the certificate from the MNO cannot legally use this MNO's licensed 4G/5G spectrums; otherwise, it fails to pass the authentication in Figure 12 and can be detected by spectrum regulators (e.g., ITU/FCC). Besides, a selfish SNO may refuse to offer carrier-grade services after gaining the UE's token, thus overbilling the UE and MNO. This issue has existed in terrestrial cellular networks for decades [69, 92, 117] due to the lack of verifiable accounting and QoS. To avoid this attack, MOSAIC can leverage the two-sided measurement and negotiation in [69,92] between the UE and satellite to form an undeniable, publicly verifiable proof of service enforcement. The SNO can only convincingly charge the MNO when both this proof and token are available.

Mitigating threats from external attackers: MOSAIC retains at least the same security as the legacy 4G/5G. Like 4G/5G, it adopts authentication and key agreement in Figure 12 to protect its over-the-air signaling and data. Its tokens and certificates are exchanged afterward over the encrypted channel, thus resilient to eavesdropping or manipulation by man-in-the-middle attackers. MOSAIC's in-band control also localizes the UE's state retrieval to mitigate potential leaks during the state migrations between satellites and ground stations in the outer space filled by untrusted 3rd-party satellites.

B Acronyms in This Work

AMF	Access and Mobility Management Function
AUSF	AUthentication Server Function
COTS	Commercial Off-The-Shelf
CU	Central Unit (in radio access networks)
DU	Distributed Unit (in radio access networks)
F1AP	F1 Application Protocol [62]
GEO	Geostationary Earth Orbit
GTP-U	GPRS Tunneling Protocol–User plane [71]
HSS	Home Subscriber Server
IMSI	International Mobile Subscriber Identity
ISL	Inter-Satellite Link
LEO	Low Earth Orbit
MNO	Mobile Network Operator
MM	Mobility Management protocol
NAS	Non-Access Stratum
NGAP	Next Generation Application Protocol [20, 70]
NTN	Non-Terrestrial Network
OCS	Online Charging System
OFCS	OFfline Charging System
PCF	Policy and Charging Function
PDU	Protocol Data Unit
PDCP	Packet Data Convergence Protocol
PLMN	Public Land Mobile Network
RAN	Radio Access Network
RLC	Radio RLC Control protocol
RRC	Radio Resource Control protocol
SCTP	Stream Control Transport Protocol
SM	Session Management protocol
SMF	Session Management Function
SNO	Satellite Network Operator
SUCI	Subscription Concealed Identifier
SUPI	Subscription Permanent Identifier
UDM	Unified Data Management
UE	User equipment
UP	User Plane
UPF	User Plane Function

Known Knowns and Unknowns: Near-realtime Earth Observation Via Query Bifurcation in `Serval`

Bill Tao, Om Chabra, Ishani Janveja, Indranil Gupta, Deepak Vasisht
University of Illinois Urbana-Champaign

Abstract

Earth observation satellites, in low Earth orbits, are increasingly approaching near-continuous imaging of the Earth. Today, these satellites capture an image of every part of Earth every few hours. However, the networking capabilities haven't caught up, and can introduce delays of few hours to days in getting these images to Earth. While this delay is acceptable for delay-tolerant applications like land cover maps, crop type identification, etc., it is unacceptable for latency-sensitive applications like forest fire detection or disaster monitoring. We design `Serval` to enable near-realtime insights from Earth imagery for latency-sensitive applications despite the networking bottlenecks by leveraging the emerging computational capabilities on the satellites and ground stations. The key challenge for our work stems from the limited computational capabilities and power resources available on a satellite. We solve this challenge by leveraging predictability in satellite orbits to bifurcate computation across satellites and ground stations. We evaluate `Serval` using trace-driven simulations and hardware emulations on a dataset comprising ten million images captured using the Planet Dove constellation comprising nearly 200 satellites. `Serval` reduces end-to-end latency for high priority queries from 71.71 hours (incurred by state of the art) to 2 minutes, and 90-th percentile from 149 hours to 47 minutes.

1 Introduction

Low Earth Orbit (LEO) satellites promise to deliver continuous, high-resolution imagery of the Earth through large constellations of low cost cubesats. These constellations, e.g., Planet Dove [50], deploy imaging sensors on cubesats in low orbits nearly 500 Kilometers above the Earth's surface. Due to their low orbits and large constellation size, they can capture an image of every location on Earth multiple times per day. The imagery from these satellites is useful for many applications such as disaster monitoring [11, 18], precision agriculture [9, 40], disease modeling [14], climate monitoring [64], and financial analytics [62].

However, Earth observation constellations today cannot support many latency sensitive applications because they suffer from large latency of few hours to days between an image capture and its availability to the end user [59]. For example, a fire department needs the images of a wildfire within a few minutes so as to limit risks to human lives, forest ecosystems,

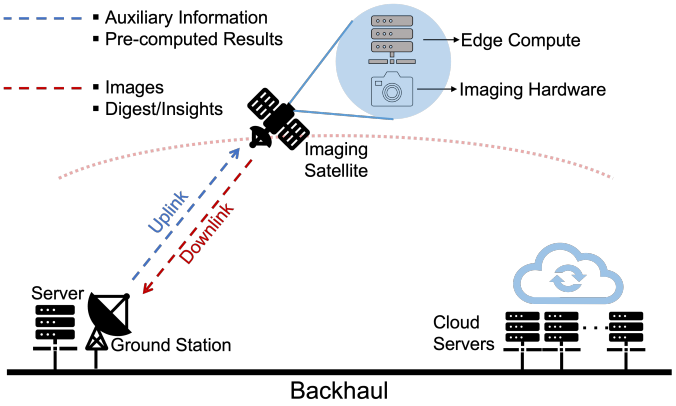


Figure 1: `Serval` **distributes computation across satellites and ground stations to prioritize latency-sensitive imagery and insights.**

and property. Such delays largely arise from a satellite's data transfer process (see Fig. 1). Satellite imagery must be transported to ground stations on Earth, and from there to the cloud, where processing, storage, and insight generation can occur. This pipeline can incur a delay of hours and sometimes days. This is due to several factors: (a) orbital dynamics, as satellites have intermittent access to ground stations on Earth—about ten minutes per contact, with 4-5 good quality contacts per day; and (b) the bandwidth from satellite to ground station is limited due to the large distance from satellite to Earth, making it nearly impossible to downlink all the images from a satellite during every contact.

This paper's goal is to enable *near-real-time insights* from satellite imagery by reducing the time-to-insight for satellite imagery to O(minutes). We do so by by leveraging emerging compute capabilities on satellites and ground stations. This is feasible today due to the ongoing push to equip satellites with small amounts of compute resources such as a Raspberry Pi or a NVIDIA Jetson, e.g., in 2020, the European Space Agency (ESA) deployed a neural-network based cloud detector on their Φ-Sat-1 mission [28]. Similarly, many recent proposals from industry [7, 46] and academia [60, 61] argue for co-locating ground stations and data centers to reduce terrestrial networking delays and enable compute on ground stations. Our core idea is to leverage the emerging general-purpose computational capabilities available on satellites and ground stations to *prioritize latency-sensitive images* such as those containing forest fires, while deprioritizing other images that are relatively less latency-sensitive.

To achieve this goal, we build Serval [1], a novel edge computing framework designed to derive near-real-time insights from LEO satellites. Serval allows multiple long-running queries to execute simultaneously on incoming satellite imagery. Given a query set, Serval intelligently distributes compute across satellite, ground station, and the datacenter. Similar to comodity products for satellite imagery analysis such as Planet Analytics [1], Serval represents each query as a logical intersection of a sequence of *filters*, e.g. "forest fires in California" is denoted as three sequential filters *{California, forest, fire}*, each of which involves geographical or statistical computation (e.g., a neural network). Different queries may have different latency-sensitivity and compute requirements. Unlike past work [23], Serval does not discard any images because new applications can emerge post-collection (e.g., historical data analysis or disaster management). For example, recently Planet [51], a leading Earth observation company, used their satellite imagery to retroactively track the origin and flight of a balloon that entered the United States airspace [5], which would be impossible if images had been discarded. Instead, Serval focuses on dynamically *reordering* image delivery to reduce end-to-end delays for latency-sensitive content.

The key challenges in Serval stem from the scale of satellite imagery and the limited compute capabilities available inside a LEO satellite. First, each satellite generates nearly a Terabyte of data per day. Second, a satellite needs to perform the query compute on this data using its limited compute capacity. LEO satellites generally have small solar panels that generates limited power. For instance, the model used in [23] has a 7W solar panel, a large fraction of which is utilized for critical satellite function. However, a Jetson TX2 itself consumes 11.3W. Moreover, solar panel power supply is further diminished because it generates no power when the satellite is on the dark side of the Earth. This means that the computer onboard cannot be always on. Together, this means it is infeasible for all images being collected by the satellite to be processed on-board.

Serval's key insight is based on our observation that a query is typically composed of *two kinds* of filters, determined by the rate of change of the data the filter pertains to. The data beneath some filters may change quite quickly—we call these as *dynamic filters*. Examples include (the outline of) fires, (position of) boats, etc. However, the data beneath the second class of filters changes much more slowly—we call these as *glacial filters*. Examples include forest identification, and ocean and land boundaries—these boundaries do not often change within a day (or even weeks).

Serval *bifurcates* a query—it assigns the temporally static (glacial) parts of the query to spatially static entities (ground stations, the cloud) while assigning temporally dynamic parts of the query to spatially dynamic entities (satellites). Consider a query such as "forest fires in California". Serval can decompose this query into identifying "California", identifying "forests", and identifying "fire". In this set, "California" and "forests" are both glacial filters, while "fire" is a dynamic filter. Our key insight is that *glacial filters can be pre-computed on the ground stations using stale imagery (e.g., a day-old image)*. Such glacial filter computation can be done even before an image is captured at the satellite and the results can be conveyed to the satellite. Serval's bifurcated approach has two advantages. First, the pre-computed glacial filter results on the ground means that the LEO satellite only needs to compute the dynamic filters of a query. Second, the same glacial filter inferences can be reused by multiple satellites (single compute, multiple use). The glacial filter offload to ground stations is enabled by the predictability of satellite orbits and as a result, predictability of the geographical location and time of each image.

For some filters that have to run in real-time, such as cloud detection, we can infer high-quality priors by additionally incorporating *auxiliary information* available at the ground station, e.g., weather forecast information. Cloud detection is an important component of RGB image analysis because clouds occlude useful information and must be rejected prior to processing. For instance, if the forecasted cloud cover is either very low or very high, Serval can skip the cloud detection step altogether aboard the satellite.

We evaluate Serval using a combination of trace-driven simulations and hardware emulation on two applications: 'forest fires in California', and 'vessel counting at ports'. These applications represent opposite ends of a spectrum: the former outputs a set of images, while the latter outputs counts. We designed a LEO satellite simulator and evaluated Serval using real image traces collected from PlanetScope, a constellation of over 200 CubeSats launched by Planet Lab. Our paper is, to the best of our knowledge, the largest evaluation performed using data collected by a real operational satellite constellation. Specifically, our traces contain ten million images collected using 151 satellites across 20 days. We evaluate Serval using two different satellite configurations and two different ground station configurations.

Contributions: We summarize our contributions below:

- We present the first system that distributes compute across satellites and ground stations to deliver near-real-time insights from Earth observation satellites.
- We propose a new bifurcated query execution approach that offloads glacial (slowly-changing) filter computation to the ground in order to reduce computational load on satellites, and end-to-end latency.
- To the best of our knowledge, we are the first to evaluate our system on a real-world raw and continuous trace collected by the world's largest LEO satellite constellation for Earth observation.

[1] Smart Edge-based Realtime Visual Analytics for LeoSats

- Our evaluation shows that the `Serval` scheduler improves end-to-end median percentile latency on high-priority images from over 70 hours to 2 minutes (90-th percentile from 149 hours to 47 minutes), while also improving detection accuracy and reducing satellite compute load by over 80%.

2 Background

In the 1990's, Iridium, Globestar, and Teledesic [27, 30, 42] planned constellations of tens of satellites to provide direct connectivity to handheld terminals. Similarly, early imaging constellations, such as NOAA's series for weather sensing, comprised of a couple of satellites. These early constellations triggered important research in satellite networking [19, 21, 34, 38, 49, 52].

More recently, within the last 5-10 years, the emergence of large LEO constellations, comprising of hundreds of satellites, has been driven by lower launch and manufacturing costs of small satellites. For instance Planet's Dove constellation for Earth imagery is composed of nearly 200 low-cost cubesats ('shoebox-sized' satellites) with off-the-shelf components. Our work focuses on these modern constellations for Earth observation. These modern constellations differ from traditional satellite constellations in three ways:

- **Constellation size:** Modern satellite constellations (e.g., Planet Inc. [26], Spire Inc. [3], etc.), consist of hundreds of satellites as opposed to few satellites in traditional constellations. This allows modern constellations to get more frequent imagery of any part of Earth with revisit frequency of few hours as opposed to a delay of several days from traditional constellations.
- **Data volumes:** The low orbit of LEO satellites and improved imaging hardware enables high resolution imagery (e.g. $1m^2$ per pixel). They capture images of Earth in different parts of the frequency spectrum, e.g., RGB, Radio Waves, Infrared, etc. The multi-spectral imagery, increased satellite number, and high resolution lead to increased data volumes—from few GBs of data per day to TBs of data per day. For instance, Planet's Dove satellites generates approximately one Terabyte of data per satellite per day.
- **Applications:** Traditional Earth observation satellites could only support delay-tolerant applications like crop yield estimates, land cover use, etc. Modern constellations gather images more frequently and offer the promise of real-time applications like disaster monitoring, traffic analysis, maritime monitoring, etc.
- **Processing pipelines:** Modern data processing pipelines increasingly rely on modern Machine Learning (ML) methods, in contrast to (merely) traditional signal processing based approaches. For example, European Space Agency's Φ-Sat-1 [28] recently demonstrated the ability to perform neural network-based cloud detection on board a satellite.

Network pipeline: Finally, we provide a brief description of satellite network pipeline as context for the rest of the paper. LEO satellites for Earth observation operate in polar orbits and go around the Earth once every 1.5 hours appoximately. During each orbit, they pass over a different part of the Earth due to Earth's rotation. The data from these satellites is usually downloaded using few dedicated ground stations with Gbps link capacities [25]. Due to a satellite's orbital motion, it can contact each ground station four to six times a day, with each contact lasting up to ten minutes. To improve download latency and increase the number of contacts, recent work has proposed distributed ground station designs with multiple general-purpose ground stations [60,61]. In the industry, Amazon and Microsoft have launched ground-station-as-a-service platforms [7, 46], wherein satellite operators can rent time on existing ground stations to download data from satellites.

3 `Serval`'s Design

In this section, we present the problem setup, `Serval`'s approach to distributing compute across Earth and space, and `Serval`'s incorporation of auxiliary information. Finally, we describe `Serval`'s execution engine.

3.1 Problem Setup

Fig. 1 shows the three layers in a satellite networking system: (1) the LEO *satellite constellation* containing hundreds of orbiting satellites, (2) few *ground stations* across the world, each of which can communicate with satellites, and (3) *the cloud* consisting of one or more data centers. The satellites continuously image the Earth's surface and send the captured images down whenever they make the next contact with a ground station. The ground stations eventually upload images to the cloud.

The satellites are energy-constrained, i.e., they have access to limited amounts of power, and only a fraction of the power is available for compute. The primary usage of the power is to maintain temperature, power attitude determination, and control systems, communicate with ground stations, and other critical satellite functionality. We assume each satellite has limited computational capability such as a Jetson TX2 or Jetson AGX Orin. This assumption is validated by recent proposals of incorporating compute in satellites in academia [15, 22, 23, 44, 45], and by recent launches that incorporate computational capabilities on satellites [4, 28].

Ground stations have no power limits and are equipped with more (though not infinite) computational resources than satellites. This assumption is reasonable in both traditional ground station designs which were capital-intensive, as well

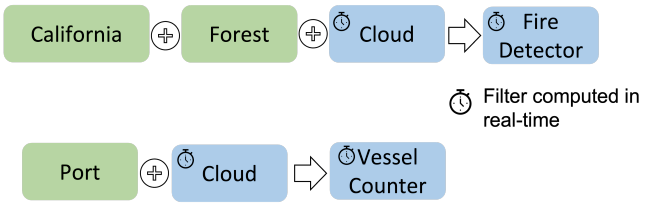


Figure 2: `Serval` **represents queries as a set of sequential filters. The picture above represents two queries: 'images containing forest fire in California', and 'Number of vessels at ports'.**

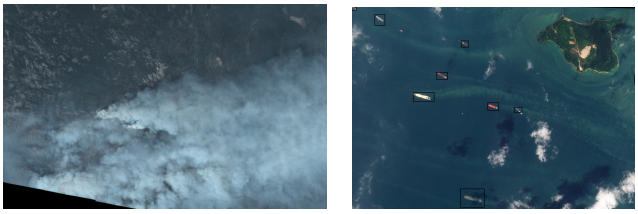


Figure 3: **Example output images for queries 'California Forest Fire' (left) and 'Vessels Counting in Port' (right).**

as in modern ground station designs wherein cloud providers co-locate compute resources at the ground stations.

Graph representation: Application developers who are customers of the satellite constellation operator company submit queries that will run on the Earth observation imagery. For example, Planet Inc. has a product named "Analytics" which allows users to run user-defined analytic applications. The user can choose a specific object detection model to run, and limit the set of images to run the selected model on by defining additional filters based on image metadata, including date, time, geographic location, cloud cover ratio, etc. [1]. However, today such queries are executed offline on the cloud once all the images have been delivered to the cloud — we seek to execute them in real-time.

While some queries may not be sensitive to taking days to generate a response, several queries are latency-sensitive and require responses in minutes. The satellite operator company may charge a premium for such latency-sensitive queries. Examples include: (a) a forest department may design a query like 'images of forest fires in California'; (b) a trading firm may be interested in a query like 'number of ships at major ports across the world'; (c) a disaster response team might be interested in 'images of, or number of, flood-damaged buildings in Florida'.

`Serval` supports two types of queries: (I) *image outputs:* queries that require images to be outputted matching the query, e.g., forest fire images from California may be needed for a detailed inspection of the damage and to create a plan for action, (II) *statistic outputs:* queries that only require the inference to be delivered, e.g., a hedge fund may need the count of cars in different parking lots across Beijing, rather than the actual images, or, financial traders may need to know the count of ships at a port.

We represent each query as a sequence of filters, which is consistent with the current commodity products such as Planet Analytics [1]. For example, 'forest fires in California' can be represented as *California→forest→cloud→fires*. Similarly, 'ships around ports' is represented as *Port→Cloud→Ship count*. Note that each query contains a cloud filter to remove images that are occluded by clouds. The filter representation is depicted visually in Fig. 2.

Each filter is a computational block that takes an image as an input and outputs either a boolean value or a number. For example, the *California* filter performs a geographical check and returns *True* or *False*, i.e., do the geographical coordinates of the image overlap with the geographical boundaries of *California*. Similarly, the filters *Vessel counter* may be a neural network (a stock network, or one supplied by the application developer) that detects and counts the number of vessels in an image. *Cloud*, *Forest*, and *Fire* are other neural network-based filters in examples above.

`Serval`'s goal is to prioritize images that pass all the filters corresponding to at least one latency-sensitive query. We also note that the filter representation of our queries lends itself to cross-query optimization. For example, if two different queries rely on the same underlying filters, `Serval` does not need to perform the computation twice, e.g., if two queries both rely on forests in California, then we do not need to perform forest detection twice. Similarly, *Cloud* filter needs to be computed only once for an image even though the image may be relevant to multiple queries.

Note on inter-satellite links: While Inter-satellite Links (ISLs) have generated a lot of interest, none of the Earth observation constellations today are equipped with ISLs. Hence we do not consider ISLs. Starlink demonstrated feasibility of laser-based ISLs [41, 57], and Planet and Telesat announced their plans to explore radio-frequency ISLs across orbits (e.g., from Low Earth Orbit to Middle Earth Orbit or Geostationary Orbits) [36]. Given the uncertainty over feasibility and type of ISLs, we choose to exclude them in our analysis. If ISLs mature in the future, `Serval`'s design can be generalized to accommodate and exploit ISLs.

3.2 Distributing Compute Across Earth and Space

Today's delay between image capture at satellite and image delivery to the cloud/end-user ranges from several hours to many days. To enable near-realtime insights for the end user(s), `Serval`'s primary goal is to reduce this delay to minutes for latency-sensitive queries, while accommodating the constraints imposed by the satellite's limited power and compute capabilities. Towards this goal, we first consider the placement of computation at the different compute units: satellite, ground station, and the cloud servers. There are two existing

approaches to perform inference on Earth imagery:

(i) In-order Delivery and Computation: In the traditional approach, images are delivered to the cloud in the same timestamp order as they were captured by the satellite, and then inference is run on these images in the cloud. This approach suffers from large delays because of networking bottlenecks. Specifically, the LEO satellites exhibit fast motion with respect to ground stations on the Earth [23, 61]. Therefore, any ground station-satellite contact is fleeting – less than ten minutes per contact, few contacts per satellite per day. This intermittent connectivity, in conjunction with the small number of ground stations, leads to large networking delays. Since images cannot be processed before they are received at the data center, the time-to-insight for this approach is very large.

(ii) In-orbit Computation: Recently, there is a growing push to place computation on the satellites. The first wave of this push demonstrated the use of satellite computation to reject cloud-occluded images [28]. Orbital edge computing [23] broadens the scope of satellite computing to reject images that do not meet the application goals, e.g., if the goal is to select images containing buildings, the satellite runs a building detector and rejects images that contain no buildings. This approach places a high computational load on satellites.

Specifically, such in-orbit computation techniques suffer from two problems: (a) as the number of applications increases or becomes more complex (such as the compositional filters discussed in Sec. 3.1), satellites cannot accommodate this compute with their limited resources, and (b) new applications emerge for historical data. Such applications may not be known *a priori*. If a satellite discards data that does not match current applications, new applications that emerge later cannot be served.

`Serval` takes the practical approach of distributing compute between *satellites and ground stations*. Our approach is centered around three properties of satellite imagery:

- **Image locations are predictable:** Satellites follow predictable orbital paths that can be estimated using their orbit descriptors, e.g., using Two Line Element (TLE) orbit descriptors published by observatories as well as many satellite operators. Therefore, even before an image is taken, we can predict what the geographical content of the image is.
- **Content of most images is glacial:** For most images taken at a given location, most of their content is glacial, i.e., stationary—it does not change rapidly within a few days. An image that contains buildings yesterday, will likely contain buildings today. Similarly, forests, desserts, farms, and other land types rarely change over the time span of a few days.
- **Ground station compute capabilities are rising:** Increasingly, ground stations are designed to include computational resources. These computers are more powerful and better resourced in terms of power and networking as compared to computational devices on the satellite.

Based on these observations, `Serval` divides the filters for all the queries into two types: (a) fast-changing dynamic filters that need to be executed on the satellite at run-time, and (b) glacial slow-changing filters that can be executed using stale imagery on the ground stations. `Serval` executes the glacial filters at the ground station or the cloud *before* the image is captured and uses the ground stations to communicate the results to the satellite in advance. This allows the satellite to process a very small fraction of images on the satellite in the bottleneck execution path.

Pre-computing such glacial information can significantly reduce the compute requirements on the satellite. As satellites orbit rapidly around the earth, their footprint looks like thin belts that extend almost vertically from the south pole to the north pole. Therefore, ordinary user requests, such as forests in California or farm land in the U.S., generally only makes up a tiny portion of a satellite's imagery.

`Serval` attaches a *dynamic* or *glacial* attribute to each filter to aid the execution engine. This can either be supplied by the developer, or estimated from data. Consider the query 'forest fires in California'. We find that among the millions of images captured by a constellation, only 0.4% contain any land area in California. Two-thirds of California images contain forests, i.e., only 0.25% of all images captured by the constellation contain 'forests in California'. Since we can estimate the exact set of images that will contain 'forests in California' using historical data, the satellite needs to run a fire detector on only this small set of images. In this case, *California* and *Forest* are glacial filters, and *Fire* is a dynamic filter. Only dynamic filters need to run on satellites using fresh data.

3.3 Incorporating Auxiliary Information Sources

Ground stations, unlike satellites, have continuous access to auxiliary information sources like weather forecasts. We ask if such information can be used by `Serval` to either improve inference quality or to skip computation on the satellite.

We first make the observation that clouds occlude many of the images taken by a satellite, and a cloud-occluded image is not useful for any query. In fact, there has been a lot of work in cloud detection for satellite imagery before [28, 35, 47, 53], both using statistical methods [35, 53] and neural networks [28, 47].

Given this observation, `Serval`'s key idea is to leverage weather forecasts in order to skip on-board processing of images over areas that are *forecast* to have high cloud cover. Concretely, if the probability of cloud cover is high, then `Serval` assigns a 'cloudy' tag to the image and exclude it from any other computation and de-prioritize its transfer to the ground station. Conversely if the probability of cloud cover is very low, then `Serval` assigns a 'cloud-free' tag to the image. In this case, the processing pipeline on the satellite can skip

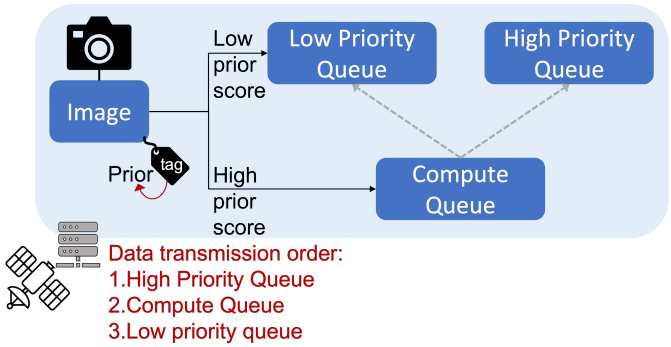


Figure 4: **Serval's satellite execution engine re-orders the images to prioritize latency-sensitive data download.**

the *Cloud* filter and proceed to the next steps of computation. Finally, all other images that are not tagged in either of the above ways are processed through the *Cloud* filter on-board the satellite.

In practice, we use cloud probability thresholds of $\{0.2, 0.8\}$ for the above processing. If the cloud cover probability for an image is less than 0.2, `Serval` assumes the image to be cloud-free. If the probability is greater than 0.8, `Serval` de-prioritizes the image. If the probability is between these two values, `Serval` processes the image through the *Cloud* filter, which is a neural network model.

While the current version of `Serval` does not incorporate weather forecasts in other inferences, there are many other potential optimizations for future work. For applications like forest fire, weather forecasts also contain information regarding forest fire risks, which can be used to select high-likelihood images to run filters on. Also, cloud detectors are known to confuse smoke and cloud which can be improved by combining cloud detection results and weather forecasts. Finally, auxiliary information may contain other types of information such as information extracted from previous satellite images, e.g., if a different satellite observed forest fire in the same region a few hours ago.

3.4 `Serval`'s Execution Engine

We describe how `Serval` captures the insights discussed in Sections 3.1-3.3. We discuss how `Serval`'s execution engine works on a satellite, at a ground station, and in the cloud.

3.4.1 The Satellites

Fig. 4 shows that each satellite maintains three queues for images: (i) a low priority queue, (ii) a high priority queue, and (iii) a compute queue. The low priority queue contains images that do not match any latency-sensitive query. The high priority queue contains images that definitely match latency-sensitive queries. The compute queue contains images that need more computation to be performed on the satellite in order to ascertain their status.

Network delivery order: When the satellite comes into contact with the ground station, it will first downlink images in the high priority queue. If the high priority queue is emptied and ground station contact remains, it will downlink the images in compute queue. This is because even though computation hasn't run on these images on the satellite, these images have a higher likelihood of being latency-sensitive than images in the low-priority queue. Finally, if the compute queue is emptied and ground station contact remains, it will downlink images from the low priority queue. This multi-queue architecture allows `Serval` to benefit from limited compute available on satellites, even when it may not be able to compute on all images.

Image placement in queues: When a satellite comes in contact with a ground station, the ground station sends the following information to the satellite: (a) pre-computed values for *glacial* or slow-moving filters for each image that the satellite is expected to capture in its upcoming orbital path leveraging the observation from Sec. 3.2 that satellite orbits are predictable, and (b) weather predictions for the geographical location corresponding to each image the satellite is expected to capture. For each image captured by a satellite, `Serval`'s first goal is to identify whether the image meets the requirements for *any* (at least one) of the on-board queries. `Serval` does so by immediately applying all pre-computed filters. For a given query, if any of its pre-computed filters rejects the image, the image is considered rejected by that query. If an image is rejected by all queries, then the image is placed in the low priority queue. Note that in practice this entire filtering process is fast as it does not involve significant compute but is a quick classifier based on pre-computed inferences received from the ground station.

To be placed in the compute queue, an image must meet two criteria: (a) all the pre-computed glacial filters in at least one query must *select* that image, and (b) there must be at least one dynamic filter in that query that needs computation. For example, for 'forest fires in California', if an image is selected by the *California* and *Forest* filters, it is placed in the compute queue to run the *Cloud* and *Fire* filters.

Any image that is selected by *all* the filters in at least one query is placed by `Serval` into the high priority queue. If a query has some dynamic filters, then an image moves to the high-priority queue from the compute queue. For example, in the example above, if the image in the compute queue is selected by both *Cloud* and *Fire* filters, it will move to the high priority queue. Note that, if all the filters in a query are glacial filter (precomputed on the ground station), then, some images can skip the compute queue and directly move to the high priority queue as well.

Finally, if the output of a query is an inference and not an image (e.g., *Vessel Count*), then `Serval` places the image in the low priority queue and creates an empty image with just

the metadata information (e.g., number of cars in parking lots or number of vessels at a port). Serval places this (much) smaller metadata value in the high priority queue.

In situations where the satellite has sufficient surplus energy to compute on an image, Serval pulls an image from the compute queue and runs the dynamic filters corresponding to the query on the image. At any time, if the satellite determines that an image is not possibly high priority in the multi-stage filtering process, it will immediately put it into the low priority queue and start doing computation on the next image. On the other hand, if the satellite determines that an image is high-priority, it will put the image into the high priority queue without running additional filters on that image. In each queue, images are ordered by their capture timestamp.

3.4.2 The Ground Stations

Recall that ground stations have much higher computation capability than the satellites, while having much fewer power constraints. In Serval, the ground station maintains two queues: a high priority queue and a low priority queue. When a ground station receives an image from the satellite, it first runs the uncompleted filters of the query and the places the image in the high priority if the image is selected by all the filters in at least one query, otherwise it places it in the low priority queue. Whenever the ground station has free computational resources, Serval opportunistically uses the downtime to compute the results of 'glacial' filters for upcoming satellite contacts. A ground station has a steady backhaul connection to the cloud, and uses this backhaul to constantly stream images to the cloud starting from the high-priority queue.

3.4.3 The Cloud

The cloud acts as a frontend to all users. The images and insights will be uploaded to the cloud and made available for users to download. The cloud also acts as the central coordinator of the entire Serval system and computes the future orbits, positions of satellites and other preemptively computed values for the images taken in the future. The cloud will also schedules satellite-ground station contacts and sends the pre-computed values to the appropriate ground station to be relayed to the satellites in the control plane. Fig. 5 demonstrates Serval's placement of different filters for the example queries in Fig. 2.

4 Experimental Setup

We evaluate Serval using a combination of trace-driven simulations and emulations. Our code is open source and available at https://github.com/ConnectedSystemsLab/Serval.

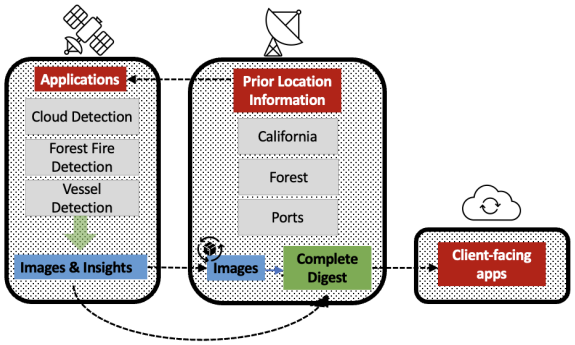


Figure 5: **Serval Example Execution: The placement of different compute units for queries defined in Fig. 2.**

4.1 Applications

We picked two sample user applications to evaluate Serval's performance: (a) identification of all *Forest Fire images in California*, and (b) *Counting Marine Vessels in Ports* in the world's busiest ports, including Shanghai, Singapore, Hong Kong, Hamad, and Jebel Ali. These applications cover vast geographical areas, and yet they are different in output (images vs. count) and in their internal query filters, ML models, and resultant computational needs.

Fig. 2 showed that the California Forest Fire application is a sequence of four filter stages:
1. *Is the Image from California?* A geographic filter (Glacial filter);
2. *Is the Image in a woody zone?* Computed from stale data (Glacial filter);
3. *Is the Image cloud-free?* (Dynamic filter); and
4. *Does the Image contain smoke or haze from a forest fire?* (Dynamic filter).

Using Planet images, we trained three deep-learning models for the identification of the forest, fire, and cloud. The forest and fire identification is based on a ResNet architecture [31] while the cloud identification is based on a MobileNet architecture [33]. The label for woody/non-woody is obtained from the USDA Forest Service; the labels for cloud and non-cloud are obtained from Planet UDM2; the fire labels are manually created. We divided our data into a training set and a testing set.

Our second application, Marine Vessel in Ports Counting, is a sequence of three filter stages:
1. *Is the Image near one of the focus port areas?* A geographic filter (Glacial filter);
2. *Is the Image cloud-free?* (Dynamic filter); and
3. *Counting the number of vessels in the image.* (Dynamic filter).

We trained a ResNet-based [31] vessel classifier using a Kaggle planet imagery dataset[2]. This dataset did not contain bounding boxes or vessel counts as training labels; it just contained vessel classification labels. Therefore, we used

[2]https://www.kaggle.com/datasets/rhammell/ships-in-satellite-imagery

the trained classifier and a weakly supervised object detection [58] (WSOD) to draw object bounding boxes on sample images taken by Planet. We then trained a yolov5 [37] model using our automatically labeled training images. We divided our data into a training set and a testing set.

Obtaining historical data for forests: To simulate extracting forest data from stale images, we initially run the forest model over images collected from the first 10 days. We, then, evaluated `Serval` over the last 10 days of the trace we obtained. To label each image in the later 10 days, we find all images in the first 10 days that intersect with the image, and if any such image is labeled as forest, we determine that the target image is forest.

Weather data: We obtained weather forecast data from openmeteo for all the images [2]. For each image that requires cloud detection, if the cloud coverage is more than 80%, we determine that it is cloudy without running the cloud detection. Similarly, if cloud coverage is less than 20% we determine that it is not cloudy without running the model. If the cloud coverage is between 20% and 80%, we run the cloud detection model.

4.2 Real-world Dataset

We obtained a large image dataset from Planet's Dove satellite constellation [50], comprising nearly 200 satellites. This dataset contains the metadata (size, location, etc.) for all images captured by all 200+ satellites between July 1st and July 20th, 2021. We use this time interval because of the prevalence of forest fires in California during the summer of 2021. This dataset contains metadata for ten million images. Due to the limited quota of images we can download, we analyzed the metadata to find images in California and multiple port areas. Based on our analysis of the metadata, we requested around 40k complete images from Planet's Planetscope API [51]. Each image contains 4 channels (red, green, blue, NIR). The collective size of our downloaded images is 13 TB.

Planet has 3 different types of sensors to collect images: PS2, which covers an area of 24km by 8km; PS2.SD, which covers an area of 24km by 16km; and PSB.SD, which covers an area of 32.5km by 19.6km. The runtime of a neural network largely depends on the image size and not the content of the image. Therefore, for the images that we only downloaded the metadata, we used the average running time of the models as an assumed running time for the corresponding model on those images, when necessary.

4.3 Hardware-benchmarking and Simulator Design

First, we use real hardware to benchmark the performance of different filters, such as the Machine Learning models for various filters. Table 1 shows that we evaluated `Serval` using **two different modes of on-board computers for satellites**: Jetson ORIN 30W and 15W modes. The microcomputer is equipped with a GPU that is able to run neural network based models in the images.

Mode	15W	30W
CPU Cores	4x1.1MHz	8x1.7MHz
GPU Speed	420 MHz	624 MHz
RAM	32 GB	32 GB
Storage	64 GB	64 GB

Table 1: **Jetson AGX Orin Hardware Configurations**

We tested the performance of `Serval` at scale in simulation. Our simulator computes the orbits of satellites using Two-Line-Element [32] orbit descriptors for each Planet satellite. The simulator tracks satellite-ground station contacts and the link quality of each of these contacts by following the specifications of Planet's ground stations in [25]. The simulator also keeps track of the energy generation and consumption on each satellite, as well as the compute time on it. We ran the simulation with a time granularity of $\Delta t = 1min$. For each satellite, we assume that it is equipped with the Jetson AGX Orin and operates in one of its two power modes.

Power management: Our simulator models the satellite power profile in Table 2. The satellite generates power using a solar panel. The satellite's ADACS (Advanced Data Acquisition and Control System) and other systems always need to be on and consume continuous power. We select these numbers to match previously reported numbers [23]. When the satellite takes photos or transmit power, it will also cost energy. We schedule the computation in a greedy approach: whenever there is spare power available and images requiring computation, the satellite will run the next-queued computation (if any).

Resource limit: We run two applications that target a small fraction of the Earth's surface area. In practice, many more applications may run on satellite imagery and cover a larger geographical area. Therefore, for fairness, we limit network and compute usage for `Serval` to 1% of the total network and compute capacity available on the satellite (roughly proportional to the surface area of the geography we cater to as compared to all the area imaged by Planet's satellites). Note that high-priority images identified by `Serval` are bottlenecked by this network restriction, other images (e.g., from other locations) can still use the full network.

Ground stations: We evaluated `Serval`'s performance **under two different ground station configurations**: (a) a state-of-the-art distributed ground station architecture [61] where we placed 200 ground stations across the globe; and (b) a traditional monolithic ground station configuration from Planet's current ground station system [20,24]. We model approximate

Component	Power
ADACS	1.13W
Computation	Depends on hardware
Camera	6J per photo
Transmission	50 W

Table 2: **Power consumption of the satellites**

Application 1		Application 2	
Filter	Amount	Filter	Amount
Total	**10097097**	**Total**	**10097097**
California	37037	Port	2642
Forest	26153	Cloud-less	1769
Cloud-less	24578	Vessel	1769
Fire	243		

Table 3: **Number of images in each stage of the pipeline.**

locations of Planet's ground stations from publicly accessible information. The ground station system in our simulation consists of 12 different ground stations distributed across the globe, carrying 48 antennas in total, where each antenna can talk to a single satellite at a time. We assume that the ground stations have a steady backhaul Internet connection that is sufficient to transmit everything it receives to the cloud.

We used the link quality model described by Planet in [25] for traditional ground station setup. For distributed ground station network, we scaled down the downlink data rate for each ground station to 25%—this makes the total downlink bandwidth constant across the two scenarios (as done in [61]).

5 Microbenchmarks

5.1 Number of Final Images $\ll$ Number of Collected Images

Table 3 quantifies the proportion of latency-sensitive images across different days. For the California forest fire application, just 0.36% of the images pass the geographic filter (*California*) and just 0.26% of the images pass the *Forest* filter. The number of actual fire images is just 243. Therefore, in `Serval`, the satellite needs to analyze at most 26k images and select 243 latency-sensitive images to download. For the vessel in ports application, we find 1769 images containing vessels at ports. These small final image counts indicate the potential latency benefits from transferring only a small (but just right!) set of images from the satellite down to the cloud.

5.2 Preemptive Compute at Ground Station

We estimate the burden on the ground station to run the required preemptive computation task by profiling it on a large cluster of computation nodes. The results show that on one

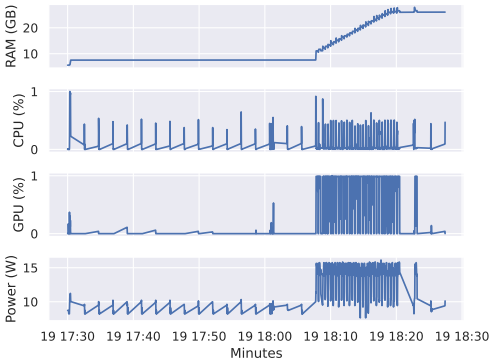


Figure 6: **Resource utilization over time for satellite "103b" passing over California**

data-center-grade GPU such as NVIDIA Tesla K80, it takes an average of 4.39s to run the forest detector on one image taken by the satellite. Running a forest detector on all California images across ten days takes no more than 24 GPU hours. This cost is distributed across multiple ground stations (at least 12 stations in our setup) and due to the station's higher power, this is preferable to running the same compute on the satellites. Further, because such data is glacial, this station runtime (O(hours)) suffices to update the satellite's glacial filters and auxiliary information once a day.

5.3 Hardware Emulation

We tested the system load of a typical satellite under `Serval` via hardware emulation. We emulate the satellite with a Raspberry Pi serving as an onboard control system, connected to a Jetson ORIN serving as the computation system. The Raspberry Pi "captures" images and sends them to the Jetson for running on-board computation. After receiving results from the Raspberry Pi, the Jetson will execute `Serval`'s execution engine (Section 3). The Raspberry Pi will send images over a TCP connection for emulating the satellite-ground station link, to a dedicated server. We emulated the time period from 17:30 on July 19, 2021, to 18:30 on July 19, 2021: when satellite "103b" flies over California and does on-board computation on the interesting images.

Figure 6 shows the variation of system load over time. We observe that before 18:05, the Jetson periodically wakes up to do heartbeat communication with the controller (Raspberry Pi) and consumes low energy. When the satellite starts to pass over California, we see a significant increase in all types of resource usage on the Jetson machine, especially GPU usage, because the applications require running neural networks on the GPU. After the satellite passes California, the CPU, GPU utilization, and power consumption drop to normal levels. We use the numbers obtained from these emulations to benchmark different filters in our simulated at-scale evaluation below.

We trained different deep learning models (as described above) to perform tasks of cloud detection, forest detection,

vessel detection, and forest fire detection. Table 4 presents the average run time of each filter on the Jetson device. Specifically, we report the time taken to classify one image using the filter.

Model	15W	30W
Cloud	1.5	3
Forest	17	8.5
Fire	0.246	0.245
Vessel	3.85	1.846

Table 4: **Average runtime of filters on Jetson machine (seconds)**

6 End-to-End Results

6.1 End-to-end Performance

We evaluate the end-to-end latency for all the latency-sensitve imagery that matches our two applications. We define the latency as the delay between when the corresponding image is generated and when the information is delivered to the cloud and available for the user to download. For the vessel counting application, just the vessel count needs to be delivered to the user. Fig. 7 shows the distribution of the end-to-end latency achieved by `Serval` compared to in-order delivery. We plot this result for two hardware configurations of the AGX ORIN – 15W power consumption and 30W power consumption, to benchmark `Serval`'s benefits across different types of computational platforms. AGX ORIN can operate in these power modes and offers less computational capability in the lower power mode.

Using the 15W power mode, and a monolithic ground station architecture, `Serval` reduces the median latency by 70× from 78.2 hours to 1.1 hours, and 90-th percentile from 145.55 hours to 2.71 hours. For `Serval`, a large part of this latency stems from the fact that even after high-priority images have been identified, they must wait for the first ground station contact. Distributed ground stations (DGS) [7, 46, 61] have recently been designed to provide more opportunities for data download. In the DGS case, `Serval` can download images with a median latency of 0.03 hours, compared to 71.71 hours median latency for in-order delivery. Even the 90th percentile for `Serval` is 0.78 hours, compared to 149.05 hours for in-order delivery. This result shows that even with simple compute capabilities on the satellite, `Serval` can achieve near-realtime delivery of latency-sensitive insights.

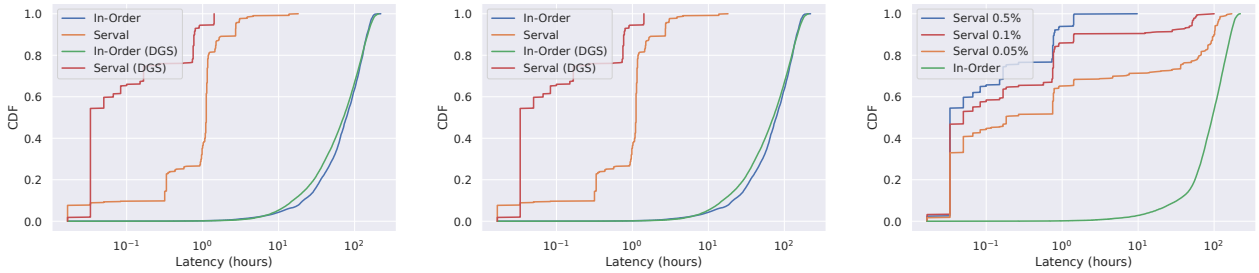
Next, we compare results for the high-power 30W mode on Jetson AGX ORIN deployed on the satellite. In this case, the delays for the baseline do not change because it includes no compute. For `Serval`, the median delay is 1.1 hours (90-th percentile – 2.7 hours) and 0.03 hours (90-th percentile – 0.78 hours) for monolithic ground stations and DGS respectively. Note that the median delay for monolithic stations does not change. This validates our hypothesis that the delay stems from the sparsity of ground station locations. When combined with DGS, `Serval` can achieve a median latency of few minutes. Unless noted otherwise, the evaluation below is performed using the 30W, DGS, 1% resource-constrained setting.

Impact of scaling up applications: Recall that, for the evaluation below, we limit the `Serval`'s resource usage for high-priority images to 1%. We would like to see how having more or less applications would impact the performance of `Serval`, which is equivalent to changing the resource usage threshold. We tested resource limit values of 0.05%, 0.1% and 0.5%. Figure 7c shows that `Serval`'s performance remains steady even when the resource limit is reduced to 0.5% from 1%. Its performance starts to decrease when the resource limit is reduced to 0.1%, and the computation starts to emerge as a bottleneck. As discussed in our future work (Sec. 8), such bottlenecks can be overcome by architectural optimizations.

Effect of running computation on satellites: To evaluate the effect of running compute on satellite, we compared the performance of `Serval` against the case when satellites will do no computation tasks on board but only prioritize all images based on glacial filters alone. The results are illustrated in Figure 8a. By running compute on the satellite, `Serval` was able to reduce the median latency by 23×.

One might wonder why filtering based on glacial filters is insufficient since the number of California forest images is small. This is because the high-priority traffic arrives in a bursty manner: when a satellite is over an area of interest, it continuously captures images that need to run the computation. When a satellite is not over an area of interest, it does not require computation. This is true for all applications whose images are not evenly distributed across the globe. Therefore, when a satellite is in contact with a ground station, the proportion of images in transmission queues with high "pre-computed" scores could be much larger than the average during a satellite-ground station contact. On the other hand, we know that for instance, only 1% of images are forest fire in all California forest images. Therefore, running computation on the satellite can help filter a significant chunk of images that pass the glacial filter pre-computed on the ground.

Benefit of using auxiliary information: To validate our decision to use side-channel information for cloud detection, we first checked the accuracy of weather information. The results are shown in Table 5. In this experiment, we only run the cloud detector on images that have a cloud coverage value between 20% and 80%. From the table, we can see that this method saves 83.6% computation on cloud detection while yielding an accuracy of 96.3% and recall of 99.3%. Indeed, we can precisely estimate the cloudiness of a majority of images without sacrificing accuracy, saving a large amount of compute.

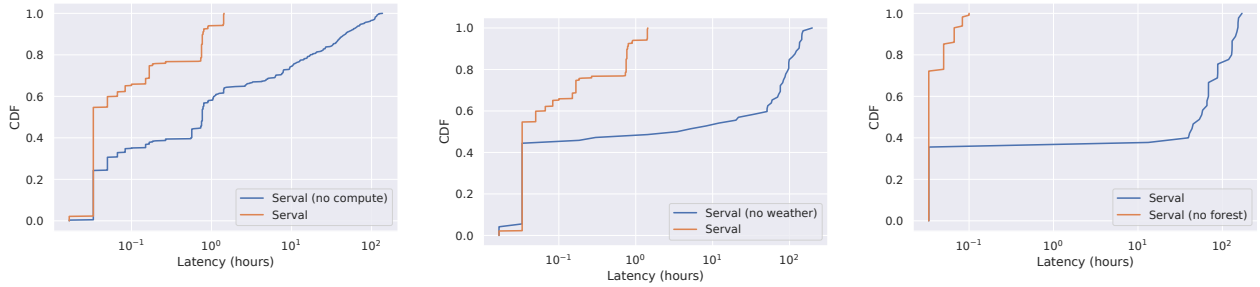


(a) Latency with Jetson ORIN in 15W mode

(b) Latency with Jetson ORIN in 30W mode

(c) `Serval`'s performance with varying resource constraints

Figure 7: **End-to-end latency of `Serval` compared to in-order baseline.** `Serval` can achieve a median end-to-end latency of 0.1 hours when using distributed ground stations. `Serval`'s performance is stable across different hardware constraints.



(a) Effect of running computation on satellites.

(b) Effect of weather information.

(c) Effect of using historical data on forest (California forest fire).

Figure 8: **Effect of different components in `Serval`. (Ground station setup is DGS)**

Prediction	Cloudy (>0.5)	Not Cloudy (<0.5)
High (>0.8)	251	226
Medium (0.2-0.8)	1668	4873
Low (<0.2)	1245	31750

Table 5: **Accuracy of weather information. Each row indicates the weather forecast, and each column indicates the output of the *Cloud* filter.**

We evaluated the weather information's contribution to the end-to-end performance by comparing `Serval` against when the side channel information is disabled. The results are shown in Figure 8b. We can see that by employing weather information, `Serval` improves the median latency by 8.8×.

Benefit of using historical data: Does pre-computation of glacial filters on the ground station have an advantage? To test this hypothesis, we move the *Forest* filter to the satellite and test if this move hurts the latency. For this experiment, we consider a single application: "California forest fire". The comparison is shown in Figure 8c. We can see that the median latency increased by 1670× when we don't use historical data. We observe that there are two reasons for this large performance drop: (a) running forest model on satellites consumes a great amount of computation time (the number of images is larger, and each image requires more computational resources) and (b) since the model is being run in real-time, some of the fire images do not look like forest for the neural network, because of the presence of the smoke. Therefore, these images get misclassified as 'not forest' and placed in the low priority queue. However, `Serval` considers stale data for such analysis when it places the glacial filter execution on ground stations. Such images are not occluded by forest. The second benefit is an unintended consequence of running glacial filters on ground stations.

Comparison to early discard : As an extension to the above experiment, we compared `Serval`'s performance against an early discard scheme inspired by OEC [23]. OEC relies solely on computation on satellites and discards all images deemed low priority. Similar to the above experiment, since OEC does not use historical data or ground station computation, it incorrectly discards some images that contain forest fire. In fact, OEC only successfully identified and downlinked 14.7% of all images with forest fire. Since it discards all "low priority" images on the satellite, the "false negative" images do not even have a chance to reach the user. The "false negative" images appear in the "long tail" distribution in Figure 8c. In contrast, `Serval`'s classifier makes fewer mistakes because

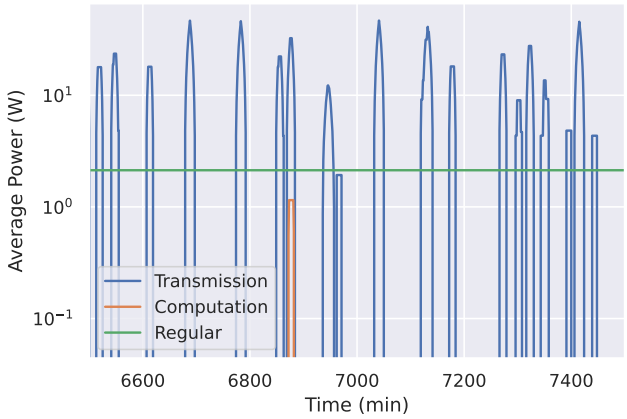


Figure 9: **Average Power over time for different applications**

of its reliance on historical data. Even when Serval makes a mistake, it just adds additional latency to that image rather than discarding that image. For the vessel counting application, both OEC and Serval are able to downlink all images containing vessels primarily because historical data doesn't help with the classification strategy.

6.2 Satellite Power Usage

We monitored the power usage for different functions during the simulation period. The energy cost consists of 3 main parts: regular power (ADACS and other essential functions to keep the satellite alive), transmission power and compute power. Figure 9 illustrates a sample satellite's ("Dove 103b") power consumption profile during a period in which it flys over California. We observed that while the satellite is constantly consuming power for regular functions and from time to time for transmission, the compute function is only activated when the satellite receives some potential high-priority image. We can see that the compute power consumption is much more sparse than either transmission or regular maintenance. In our simulation, we saw 68.1% energy being consumed by transmission, 28.9% consumed by regular maintenance, and 2.9% consumed by computation. Because of our limit on resource utilization for the high-priority applications, we only used 1% of the 2.9% total compute energy, and the rest of the energy was reserved for other computation tasks (e.g., for other tenant applications, satellite maneuvering, etc.).

7 Related Work

Our work builds on results in terrestrial edge computing, video processing systems, and orbital edge computing.

Terrestrial edge computing pipelines: Edge computing has been a widely studied topic in terrestrial networks [8,13,17,29, 54, 55] for diverse applications such as traffic camera analytics [8, 12], augmented reality systems [56], and robotics [63]. These systems tend to push computation as close to the video sensor as possible while being cognizant of the resource constraints of the edge devices. Serval naturally builds on this line of research. Satellites are similar to resource-constrained mobile devices with relatively weak connectivity to the cloud. The LEO setting presents the additional challenge that the sensor devices (satellites) themselves are moving. Serval addresses this by leveraging the predictable orbital paths of satellites, query filters and bifurcation, and using auxiliary information.

Video processing systems: Much recent work [6,8,13,39,48, 65] has focused on improving the execution of video analytics pipelines on edge devices. These systems consider different aspects of optimizing video analytics such as efficient model retraining [13], model merging for efficient execution on edge GPUs [48], etc. This line of work makes varying assumptions about the availability of compute resources such as powerful GPUs and continuous connectivity with the cloud—these luxuries are not available on satellites. Nevertheless, ideas from video processing systems are complementary to Serval, and our work opens an avenue to explore such future directions. For instance, model merging, an idea from video processing systems, can be useful if the models (inside the filters of different Serval queries) share a lot of common layers.

Satellite edge computing: Traditional satellites packaged radiation-hardened specialized hardware [10]. Due to the lower orbits of LEOSats which suffer much less radiation exposure, and for economies of scale in manufacturing, there has been a move towards general-purpose hardware for LEOSats [28]. This has blossomed research in satellite edge computing systems [15, 16, 22, 23, 43–45]. Some papers propose new edge-enabled functionalities for communication megaconstellations [15, 16], e.g., deploying content delivery networks in space to improve network performance. This work is independent of Serval due to its focus on a different class of satellites. [45] and [44] evaluate the performance of compression techniques and Machine learning models on satellite-compatible hardware such as NVIDIA Jetson Nano and/or NVIDIA Jetson AGX. We believe such optimizations demonstrate the feasibility of deploying Machine Learning workloads on satellites, and such architectural optimizations (including from other domains like approximate computing) can be applied orthogonally to Serval to improve performance.

Our work is closest to Orbital Edge Computing (OEC [23]) and Kodan [22]. Both OEC and Kodan aim to reduce the amount of satellite imagery transferred to Earth by enabling early rejection of imagery that is not considered useful. While OEC reimagines different satellites in a constellation as a computational nanosatellite pipelines that seamlessly distribute

computational tasks, Kodan focuses more on the computation at each satellite. In contrast to both these works, Serval does not discard any images on the satellite, and focuses on reducing the delay of latency-sensitive images via prioritization and reordering. This has the advantage that *post facto* queries on historical can be performed, e.g., in a recent high-profile incident, Planet used historical satellite imagery to trace the historical motion of a Chinese balloon entering into the US airspace from its origin to destination [5]. Kodan's contribution is orthorgonal to ours, since Kodan aims to train the optimal neural network model for specific user applications, while we focus on how to optimally schedule compute on satellites and ground stations given a fixed neural network. Hence Kodan can be used alongside Serval.

8 Concluding Discussion

We build Serval, a distributed computation framework for near-realtime insights from Earth imagery satellites. Serval can deliver latency sensitive imagery such as forest fire imagery in minutes as opposed to hours or days of delay for traditional in-order delivery systems. We conclude by listing some possible extensions of Serval in future work:

- **Multi-modal imagery:** Serval currently focuses on RGB imagery captured by satellites. Increasingly, satellites capture other forms of imagery such as radar, hyperspectral, and multi-spectral. We believe Serval can naturally extend to support these emerging image types.
- **Merging filters across queries:** As the number of applications scales, there are more opportunities to reduce redundant compute within and across queries. Multiple queries may share filters to reduce compute on satellites. Moreover, multiple neural network models may share weights for a subset of the layers and present opportunities for model merging techniques like [48] to optimize compute on the satellite.
- **Architectural optimizations:** We did not consider architectural optimizations such as model pruning or precision drop to pack more compute on the limited satellite resources. Such techniques can further improve Serval's performance.

Acknowledgments

We are grateful to anonymous reviewers and our shepherd Lixia Zhang for feedback on this work. This work was partially sponsored by NSF Award CNS-2237474, by NSF Award CNS-1908888, and by generous contributions from Cisco and Microsoft. We are grateful to Kiruthika Devaraj from Planet Inc. for discussions and feedback on early ideas. Access to Planetscope imagery was provided by the UIUC library's contract with Planet Inc.

References

[1] Analytics Overview. https://developers.planet.com/docs/analytics/.

[2] Open-meteo: Open-source weather api. https://open-meteo.com/.

[3] Spire Global Inc. https://spire.com/.

[4] Living on the edge: Satellites adopt powerful computers. https://spacenews.com/living-on-the-edge-satellites-adopt-powerful-computers/, 2022.

[5] One more way ai can help us harness one of the most underutilized datasets in the world. https://www.planet.com/pulse/one-more-way-ai-can-help-us-harness-one-of-the-most-underutilized-datasets-in-the-world, 2023.

[6] Neil Agarwal and Ravi Netravali. Boggart: Towards General-Purpose acceleration of retrospective video analytics. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 933–951, Boston, MA, April 2023. USENIX Association.

[7] Amazon Inc. AWS Ground Station . https://aws.amazon.com/ground-station/.

[8] Ganesh Ananthanarayanan, Victor Bahl, Landon Cox, Alex Crown, Shadi Nogbahi, and Yuanchao Shu. Video analytics-killer app for edge computing. In *Proceedings of the 17th annual international conference on mobile systems, applications, and services*, pages 695–696, 2019.

[9] Bruno Aragon, Rasmus Houborg, Kevin Tu, Joshua B. Fisher, and Matthew McCabe. CubeSats Enable High Spatiotemporal Retrievals of Crop-Water Use for Precision Agriculture. *Remote Sensing*, 2018.

[10] William Bamford, Luke Winternitz, and Curtis Hay. Gps world, innovation: Autonomous navigation at high earth orbits. 2005.

[11] Panagiotis Barmpoutis, Periklis Papaioannou, Kosmas Dimitropoulos, and Nikos Grammalidis. A review on early forest fire detection systems using optical remote sensing. *IEEE Sensors*, 2020.

[12] Johan Barthélemy, Nicolas Verstaevel, Hugh Forehead, and Pascal Perez. Edge-computing video analytics for real-time traffic monitoring in a smart city. *Sensors*, 19(9):2048, 2019.

[13] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 119–135, Renton, WA, April 2022. USENIX Association.

[14] A. Bhattachan, N. Skaff, S. Vimal, J. Remais, and D. P. Lettenmaier. Using geospatial datasets to characterize mosquito larval habitats in the Los Angeles Basin. In *AGU Fall Meeting Abstracts*, 2019.

[15] Debopam Bhattacherjee, Simon Kassing, Melissa Licciardello, and Ankit Singla. In-orbit computing: An outlandish thought experiment? In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, 2020.

[16] Vaibhav Bhosale, Ketan Bhardwaj, and Ada Gavrilovska. Toward loosely coupled orchestration for the LEO satellite edge. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, June 2020.

[17] Christopher Canel, Thomas Kim, Giulio Zhou, Conglong Li, Hyeontaek Lim, David G Andersen, Michael Kaminsky, and Subramanya Dulloor. Scaling video analytics on constrained edge nodes. *Proceedings of Machine Learning and Systems*, 1:406–417, 2019.

[18] Kejie Chen, Jean-Philippe Avouac, Saif Aati, Chris Milliner, Fu Zheng, and Chuang Shi. Cascading and pulse-like ruptures during the 2019 ridgecrest earthquakes in the eastern california shear zone. *Nature Communications*, 2020.

[19] M. Chu, D. Drynan, and L. R. Benning. Integrating satellite links into a land-based packet network. *ACM SIGCOMM Comput. Commun. Rev.*, 1985.

[20] Kyle Colton, Joseph Breu, Bryan Klofas, Sydney Marler, Chad Norgan, and Matthew Waldram. Merging Diverse Architectures for Multi-Mission Support. In *Small Satellite Conference*, 2020.

[21] Olivier L. de Weck, Richard de Neufville, and Mathieu Chaize. Staged deployment of communications satellite constellations in low earth orbit. *Journal of Aerospace Computing, Information, and Communication*, 2004.

[22] Bradley Denby, Krishna Chintalapudi, Ranveer Chandra, Brandon Lucia, and Shadi Noghabi. Kodan: Addressing the computational bottleneck in space. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, New York, NY, USA, 2023. Association for Computing Machinery.

[23] Bradley Denby and Brandon Lucia. Orbital edge computing: Nanosatellite constellations as a new class of computer system. In *ACM ASPLOS*, 2020.

[24] Kiruthika Devaraj, Ryan Kingsbury, Matt Ligon, Joseph Breu, Vivek Vittaldev, Bryan Klofas, Patrick Yeon, and Kyle Colton. Dove High Speed Downlink System. In *Small Satellite Conference*, 2017.

[25] Kiruthika Devaraj, Matt Ligon, Eric Blossom, Joseph Breu, Bryan Klofas, Kyle Colton, and Ryan Kingsbury. Planet High Speed Radio: Crossing Gbps from a 3U Cubesat. In *Small Satellite Conference*, 2019.

[26] Anna Escher. Inside Planet Labs' new satellite manufacturing site. TechCrunch. https://techcrunch.com/2018/09/14/inside-planet-labs-new-satellite-manufacturing-site/, 2018.

[27] C.E. Fossa, R.A. Raines, G.H. Gunsch, and M.A. Temple. An overview of the iridium (r) low earth orbit (leo) satellite system. In *IEEE National Aerospace and Electronics Conference*, 1998.

[28] Gianluca Giuffrida, Luca Fanucci, Gabriele Meoni, Matej Batič, Léonie Buckley, Aubrey Dunne, Chris van Dijk, Marco Esposito, John Hefele, Nathan Vercruyssen, Gianluca Furano, Massimiliano Pastena, and Josef Aschbacher. The -sat-1 mission: The first on-board deep neural network demonstrator for satellite earth observation. *IEEE Transactions on Geoscience and Remote Sensing*, 2022.

[29] Peizhen Guo, Bo Hu, and Wenjun Hu. Mistify: Automating DNN model porting for On-Device inference at the edge. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 705–719. USENIX Association, April 2021.

[30] JOHN HANSON, MARIA EVANS, and RONALD TURNER. *Designing good partial coverage satellite constellations*. Astrodynamics Conference. 1990.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[32] Felix R. Hoots and Ronald L. Roehrich. Models for Propagation of NORAD Element Sets. Technical report, Aerospace Defense Command, United States Airforce, 1980.

[33] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[34] Y.C. Hubbel. A comparison of the iridium and amps systems. *IEEE Network*, 1997.

[35] Gary J. Jedlovec, Stephanie L. Haines, and Frank J. LaFontaine. Spatial and temporal varying thresholds for cloud detection in goes imagery. *IEEE Transactions on Geoscience and Remote Sensing*, 2008.

[36] Rachel Jewett. Planet to support nasa relay networks for telesat, ses. Satellite Today. https://www.satellitetoday.com/government-military/2022/08/23/planet-to-support-nasa-relay-networks-for-telesat-ses/.

[37] Glenn Jocher, Alex Stoken, Jirka Borovec, Ayush Chaurasia, Liu Changyu, Adam Hogan, Jan Hajek, Laurentiu Diaconu, Yonghye Kwon, Yann Defretin, et al. ultralytics/yolov5: v5. 0-yolov5-p6 1280 models, aws, supervise. ly and youtube integrations. *Zenodo*, 2021.

[38] H. Keller, H. Salzwedel, G. Schorcht, and V. Zerbe. Comparison of the probability of visibility of the most important currently projected mobile satellite systems. In *IEEE Vehicular Technology Conference. Technology in Motion*, 1997.

[39] Mehrdad Khani, Ganesh Ananthanarayanan, Kevin Hsieh, Junchen Jiang, Ravi Netravali, Yuanchao Shu, Mohammad Alizadeh, and Victor Bahl. RECL: Responsive Resource-Efficient continuous learning for video analytics. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 917–932, Boston, MA, April 2023. USENIX Association.

[40] J. Kong, Y. Ryu, R. Houborg, and M. Kang. Monitoring canopy photosynthesis in high spatial and temporal resolution using CubeSat imagery. In *AGU Fall Meeting Abstracts*, 2019.

[41] Kuiper Systems LLC. Application of kuiper systems llc for authority to launch and operate a non-geostationary satellite orbit system in ka-band frequencies. FCC, https://licensing.fcc.gov/myibfs/download.do?attachment_key=1773885.

[42] R.J. Leopold. The iridium communications systems. In *Singapore ICCS/ISITA '92*, 1992.

[43] Israel Leyva-Mayorga, Marc M. Gost, Marco Moretti, Ana Pérez-Neira, Miguel Ángel Vázquez, Petar Popovski, and Beatriz Soret. Satellite edge computing for real-time and very-high resolution earth observation. arXiv 2212.12912, 2022.

[44] Martina Lofqvist and José Cano. Optimizing data processing in space for object detection in satellite imagery. *CoRR*, abs/2107.03774, 2021.

[45] Martina Lofqvist and José Cano. Accelerating deep learning applications in space. arXiv 2007.11089, 2020.

[46] Microsoft. Azure Orbital. https://azure.microsoft.com/en-us/services/orbital/.

[47] Sorour Mohajerani and Parvaneh Saeedi. Cloud-net: An end-to-end cloud detection algorithm for landsat 8 imagery. In *IGARSS 2019 - 2019 IEEE International Geoscience and Remote Sensing Symposium*, 2019.

[48] Arthi Padmanabhan, Neil Agarwal, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Guoqing Harry Xu, and Ravi Netravali. Gemel: Model merging for Memory-Efficient, Real-Time video analytics at the edge. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 973–994, Boston, MA, April 2023. USENIX Association.

[49] C. Partridge and T.J. Shepard. Tcp/ip performance over satellite links. *IEEE Network*, 1997.

[50] Planet Inc. Dove Satellite Constellation. https://www.planet.com/our-constellations/.

[51] Planet Inc. Planetscope. https://developers.planet.com/docs/data/planetscope/.

[52] Stephen R. Pratt, Richard A. Raines, Carl E. Fossa, and Michael A. Temple. An operational and performance overview of the iridium low earth orbit satellite system. *IEEE Communications Surveys*, 1999.

[53] William B Rossow and Leonid C Garder. Cloud detection using satellite measurements of infrared and visible radiances for isccp. *Journal of climate*, 1993.

[54] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[55] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[56] Yushan Siriwardhana, Pawani Porambage, Madhusanka Liyanage, and Mika Ylianttila. A survey on mobile augmented reality with 5g mobile edge computing: architectures, applications, and technical aspects. *IEEE Communications Surveys & Tutorials*, 23(2):1160–1192, 2021.

[57] SpaceX. Spacex non-geostationary satellite system. FCC, https://fcc.report/IBFS/SAT-LOA-20161115-00118/1158350.pdf.

[58] Peng Tang, Xinggang Wang, Angtian Wang, Yongluan Yan, Wenyu Liu, Junzhou Huang, and Alan Yuille. Weakly supervised region proposal network and object detection. In *Proceedings of the European conference on computer vision (ECCV)*, pages 352–368, 2018.

[59] Bill Tao, Maleeha Masood, Indranil Gupta, and Deepak Vasisht. Transmitting, fast and slow: Scheduling satellite traffic through space and time. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, ACM MobiCom '23, New York, NY, USA, 2023. Association for Computing Machinery.

[60] Deepak Vasisht and Ranveer Chandra. A distributed and hybrid ground station network for low earth orbit satellites. In *ACM HotNets*, 2020.

[61] Deepak Vasisht, Jayanth Shenoy, and Ranveer Chandra. L2d2: Low latency distributed downlink for low earth orbit satellites. In *ACM SIGCOMM*, 2021.

[62] Maria Fernandez Vidal and Peter Bull. Using satellite data in financial inclusion. 2019.

[63] Yiding Wang, Weiyan Wang, Duowen Liu, Xin Jin, Junchen Jiang, and Kai Chen. Enabling edge-cloud video analytics for robotics applications. *IEEE Transactions on Cloud Computing*, pages 1–1, 2022.

[64] M. Wiseman and A. Bradley. Impact of the length of the sea ice-free summer season on Alaskan Arctic coastal erosion rates. In *AGU Fall Meeting Abstracts*, 2019.

[65] Zhujun Xiao, Zhengxu Xia, Haitao Zheng, Ben Y. Zhao, and Junchen Jiang. Towards performance clarity of edge video analytics. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 148–164, 2021.

Spectrumize: Spectrum-efficient Satellite Networks for the Internet of Things

Vaibhav Singh, Tusher Chakraborty, Suraj Jog, Om Chabra[†], Deepak Vasisht[†], and Ranveer Chandra
Microsoft Research, [†]UIUC

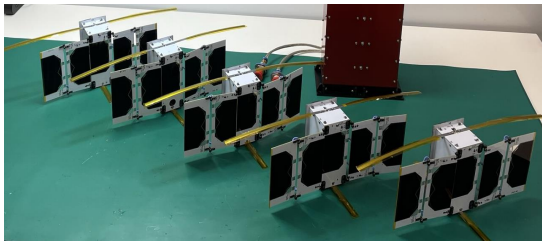
Abstract

Low Earth Orbit satellite constellations are gaining traction for providing connectivity to low-power outdoor Internet of Things (IoT) devices. This is made possible by the development of low-cost, low-complexity pico-satellites that can be easily launched, offering global connectivity without the need for Earth-based gateways. In this paper, we report the space-to-Earth communication bottlenecks derived from our experience of deploying an IoT satellite. Specifically, we characterize the challenges posed by the low link budgets, satellite motion, and packet collisions. To address these challenges, we design a new class of techniques that use the Doppler shift caused by the satellite's motion as a unique signature for packet detection and decoding, even at low signal-to-noise ratios and in the presence of collisions. We integrate these techniques into our system, called Spectrumize, and evaluate its performance through both simulations and real-world deployments. Our evaluation shows that Spectrumize performs 3× better compared to the classic approach in detecting packets with over 80% average accuracy in decoding.

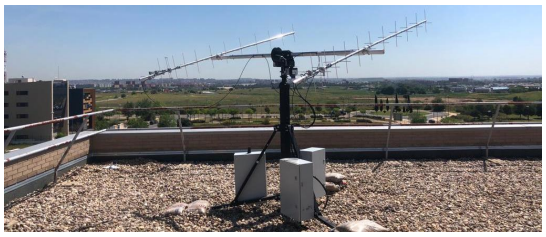
1 Introduction

The emergence of the "new space" era has led to the growing popularity of small satellites aimed at connecting Internet of Things (IoT) devices. These satellites offer connectivity to devices where terrestrial networks do not exist, for example, in farms, forests, oceans, and others. With their small size and low-complexity hardware, these satellites are easy to build and launch into orbit, resulting in the rapid growth of the IoT satellite industry. Over a dozen companies [3–6, 33, 35] are investing heavily in deploying constellations of hundreds of these small IoT satellites to provide connectivity using the Direct-to-Satellite (DtS) model. This allows IoT devices on the ground to uplink data directly to orbiting satellites, without requiring deployment of gateway devices. The satellites collect and downlink data to ground stations which have terrestrial backhaul for cloud-based data aggregation.

The main focus of IoT satellites is to achieve a low price point, setting them apart from traditional earth observation and broadband connectivity satellites. These satellites are smaller in size and optimized for IoT applications, which typically require low data rates, usually in the range of hundreds of bits per second, and operate at a bandwidth of around a hundred kilohertz. Therefore, IoT satellites are equipped with simple connectivity hardware, such as half-wave dipole cable antennas with no beamforming capability or inter-satellite link (ISL) [43]. Similarly, ground stations (GS) and IoT devices on the ground are designed to be simple and can be deployed anywhere using small antennas that are either omnidirectional or have limited directionality. In conclusion, the connectivity design and hardware of IoT satellites closely resemble that of state-of-the-art terrestrial IoT networks.



(a) *Our satellite and 4 others from the same constellation*



(b) *Our industry collaborator's ground station*

Figure 1: *Our real-world testbed: an IoT satellite launched in collaboration with FOSSA [7], and multiple ground stations.*

We collaborated with FOSSA Systems [7], an IoT satellite company, to launch an IoT pico-satellite and investigate the characteristics of this emerging network class. We utilized their ground stations in Spain to facilitate data downloads from the satellite. To our surprise, despite using standard hardware, we encountered significant packet loss during the downlink from the satellite to the ground station. Specifically, we found that 65.28% of the packets transmitted by the satellite were lost and remained undecoded by the ground station. This paper stems from the challenges of enabling a robust downlink from low-cost, low-power IoT satellites. In particular, we observe that the following three challenges limit our ability to decode downlink transfer from IoT satellites.

Challenge 1: Low link budget – The link budget for satellite to ground communication is low because of the large distance (around 500 km), limited directionality of antennas on satellites, ground stations without beamforming capabilities, and atmospheric attenuation. We observe an average link budget of only −133.78 dBm during a satellite pass over a ground

station. Increasing transmission power is not feasible due to power constraints on the satellite and regulatory restrictions. Though the use of a high-end setup like a phased array antenna on the ground can increase the link budget, it requires a large antenna setup and is expensive, especially for IoT satellites that operate on lower frequency bands like VHF and UHF. Lower link budget leads to frequent packet drops and a low data rate. It is particularly problematic for the downlink transmissions because ground stations also experience increased noise floor due to terrestrial transmissions.

Challenge 2: Packet collisions and spectral inefficiency – LEO IoT satellites have limited antenna directionality and can transmit signals across vast areas on Earth, referred to as their footprint. These footprints can span up to a million square kilometers of surface area. Since multiple satellites operate within the same frequency band, their transmissions collide at ground stations located within the overlapping footprints (Fig. 2). For example, consider the Swarm constellation consisting of 170+ satellites [9]. A ground station situated in a polar region, ideal due to the satellites' orbital dynamics, falls within the view of an average of 7 satellites simultaneously. Consequently, satellites may encounter a high collision rate when downloading data to the ground station. As a preemptive measure, ground stations are scheduled in advance to communicate with only one satellite, even when in the line of sight of multiple satellites. This scheduling strategy, while necessary, has a significant impact on spectral efficiency.

Challenge 3: Satellite motion – The satellite-ground link exhibits fluctuations in link quality due to two distinct types of motion: (a) the orbital motion of the satellite introduces variations in the distance between the ground device and the satellite, and (b) localized satellite motion, often characterized by tumbling. This phenomenon is particularly pronounced in small-sized IoT satellites, as they lack an altitude control system to stabilize their motion. While large broadband satellites tackle this issue through real-time bit-rate adaptation, it presents a more formidable challenge with IoT satellites due to their lower data rates and limited bandwidth. This challenge is especially evident in receive-only distributed ground stations employed for downloading data from the satellite [2, 48]. As a result, the satellite chooses conservative bit rates for transmission that can work even for the lowest signal-to-noise ratio (SNR). This leads to sub-optimal spectrum efficiency.

This paper presents Spectrumize, a novel approach designed to enhance the spectral efficiency of IoT satellite downlinks, thereby enhancing reliability, robustness, and scalability in satellite-based IoT networks. Spectrumize stands out by not treating satellite motion as a problem but as a unique enabler for satellite-based IoT networks. The core concept behind Spectrumize hinges on the predictability of satellite motion, which results in predictable Doppler shifts in the signal unique to each satellite. Consequently, we leverage the temporal variation of these Doppler shifts as signatures, akin to codes in CDMA. Much like CDMA, Doppler signatures empower us to elevate SNR and demultiplex simultaneous transmissions, resulting in higher spectral efficiency.

SNR boosting: As mentioned previously, distinguishing the signal received from a satellite from noise at a ground station can be challenging. Traditional methods of packet detection using preamble-based correlation are often ineffective because of the signal's frequency shifts due to Doppler effects. We have made two key observations: first, Doppler shifts between satellites and ground stations are predictable due to stable orbital paths, and second, satellites typically transmit a periodic train of packets. To address this, Spectrumize employs a series of preambles, each adjusted with the appropriate Doppler correction, for correlation with the received signal to facilitate packet detection. In our empirical studies, we confirm that both Doppler shift correction and signal repetition are necessary for successful packet detection, even in the presence of hardware errors like carrier frequency offsets.

Collision resolution: Spectrumize introduces a new approach to address packet collision challenges in satellite IoT networks, particularly those caused by overlapping satellite footprints. This approach relies on three distinguishing factors: the unique Doppler shift of each satellite-ground link, the symbol reception time, and the received signal intensity. By leveraging the distinct Doppler shift patterns experienced by colliding packets from different satellites, we use the Doppler signature as a self-contained code to demultiplex signals from multiple transmissions. As a result, a ground station can simultaneously download data from multiple satellites, leading to a significant enhancement in spectral efficiency.

We conduct evaluations of Spectrumize through simulation, emulation, and real-world testbed, which encompasses our satellite in orbit. This testbed also involves other satellites from the same constellation and multiple ground stations provided by our industry collaborator, FOSSA. Utilizing this testbed, we characterize the nature of satellite-ground transmissions for IoT satellites and validate our design. The results demonstrate that Spectrumize outperforms state-of-the-art methods, achieving a 3× enhancement in packet detection with an average decoding accuracy exceeding 80%.

Summary of Contributions:

- We characterize the communication challenges posed by the constraints in a deployed IoT satellite network.
- We propose a new 'Doppler signature as a code' approach to boost satellite decoding and demultiplexing performance.
- We evaluate Spectrumize in a real-world testbed including satellites in orbit.

2 Motivation from Real-world Experience

IoT satellites constitute a rapidly growing category of low-cost, low-complexity mini-satellites. As of 2022, approxi-

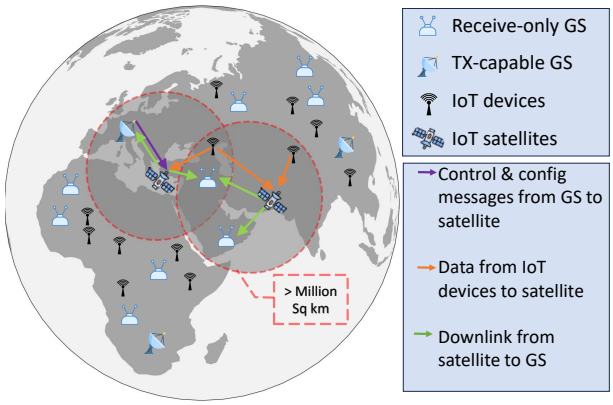


Figure 2: *Overview of end-to-end satellite IoT communication using Direct-to-Satellite technology.*

mately two thousand such satellites have been launched into orbit [8]. Swarm, FOSSA, Astrocast, EchoStar, Myriota, and more than ten other companies are leading this growing satellite IoT industry [3–6, 35]. To gain a better understanding of the industry practices, challenges, and the need for innovation in this burgeoning industry, we collaborated with an industry leader, FOSSA Systems [7], to launch an IoT satellite shown in Fig. 1a. Our collaborator designed and launched the satellite according to industry standards, without any design input from us. We utilized the satellite as a real-world testbed for our research along with the collaborator's ground station to communicate with the satellite, creating a testbed that accurately represents the ecosystem of today's satellite IoT industry. In this section, we delve deeper into the practices of the satellite IoT network industry and share our firsthand experiences with the challenges it presents.

2.1 Satellite as the Global IoT Gateway

In traditional terrestrial wide-area IoT networks such as LoRaWAN, IoT devices follow a star-of-stars topology with a gateway aggregating data from the IoT devices. In contrast, in satellite IoT networks, the IoT gateway is carried by the satellites themselves. This is achieved through direct-to-satellite (DtS) communication, where IoT devices on the earth send data directly to the overhead satellites. As the satellite orbits around the Earth, it can collect data from different locations and work as the global IoT gateway (Fig. 2). Similarly, our satellite carries a SX1302 LoRa gateway produced by Semtech [10]. The satellite has dimensions of around $10cm \times 5cm \times 5cm$ – this class of satellites is smaller than cube-satellites and is referred to as pico-satellite. Due to its small size, the satellite's power is restricted, with a maximum average power of 1.6 watts produced by its solar panel and stored in rechargeable batteries. To maintain low hardware complexity and cost-effectiveness, the satellite does not have any active altitude control system. It orbits Earth in a low earth orbit (LEO), typically between 450 to 550 kilometers above Earth's surface, allowing it to observe different parts of the planet over time. Because of its orbital motion, the satellite can only communicate with ground devices for brief periods, typically lasting 5-9 minutes for a location on Earth. There are usually only 2-3 such communication windows per day for a location. As a result, a constellation of 150+ satellites is required to ensure 24/7 global coverage.

2.2 Ground Stations

To communicate with our satellite, we rely on two ground stations provided by our collaborator (Fig. 1b), which we use for downloading data and configuring the satellite through Telemetry, Tracking, and Command (TT&C) communication. To keep costs low, these ground stations use low-end connectivity hardware, including omnidirectional or directional Yagi antennas, rather than phased array antennas (Fig. 1b). The directional Yagi antennas have a beamwidth of 30° and a gain of 12 dBic, and they are coupled with rotators that can move them to point towards the satellite when it is in view. While we primarily use Yagi antennas, several other companies opt for omnidirectional antennas, which can further simplify their designs [2, 25]. The antenna is linked to a software-defined radio (SDR) to process the communication signal. The idea of using SDR-based ground stations is also popular in the ground station as a service industry [14, 29]. Furthermore, due to the short contact time and low data rate of the satellite-ground communication link, there is currently a significant effort to develop receive-only distributed ground stations that can reduce data latency in satellite IoT networks [1, 2, 48]. For example, SatNOGS is an open-source global network of ground stations equipped with low-cost SDR and Raspberry Pi [1]. Our satellite also broadcasts health beacons targeting these distributed ground stations.

2.3 Communication

Our IoT satellite communicates with ground devices in three ways: 1) the satellite collects data from IoT devices, 2) the satellite communicates with ground stations for TT&C and downloading data to ground stations, and 3) the satellite sends data/beacons to distributed ground stations. In this section, we focus on communication between the satellite and ground stations. The satellite is equipped with a half-wave dipole cable antenna and does not have beamforming capability. It employs a Semtech SX1262 radio to communicate with the ground stations, which supports both LoRa and 2-FSK modulation [21]. We use the 401 - 402 MHz band assigned by the local regulators for Earth exploration satellites, meteorological aids, and meteorological satellites. The uplink communication (ground station to satellite) takes place on 401.3 MHz, while the downlink communication (satellite to ground station) occurs on 401.9 MHz. The satellite also sends a health beacon every 30-120 seconds on 401.7 MHz. The

channel bandwidth is max 125 kHz with LoRa modulation and 39 kHz with 2-FSK.

Before a satellite-ground station communication takes place, the contact is scheduled in advance using the satellite's two-line element (TLE) and the ground station's location [47]. During the contact, the communication is initiated by sending specific commands from the ground station to the satellite. Once the satellite receives a command, it relays it back to the ground station. The ground station confirms the command if it matches the one it sent, and then the command is executed. Depending on the command, the satellite either updates its onboard configuration or sends data to the ground station. When the satellite sends data, it does so in a stream of N data packets, where N is defined in the command. If the satellite does not have N packets to send, it sends all the packets that it has. Since the channel is not expected to be used for multiple satellite-ground communications at the same time, the packets are transmitted at a plain interval without any random back-off. A constant bit rate is maintained during communication, given the low bandwidth and short contact time.

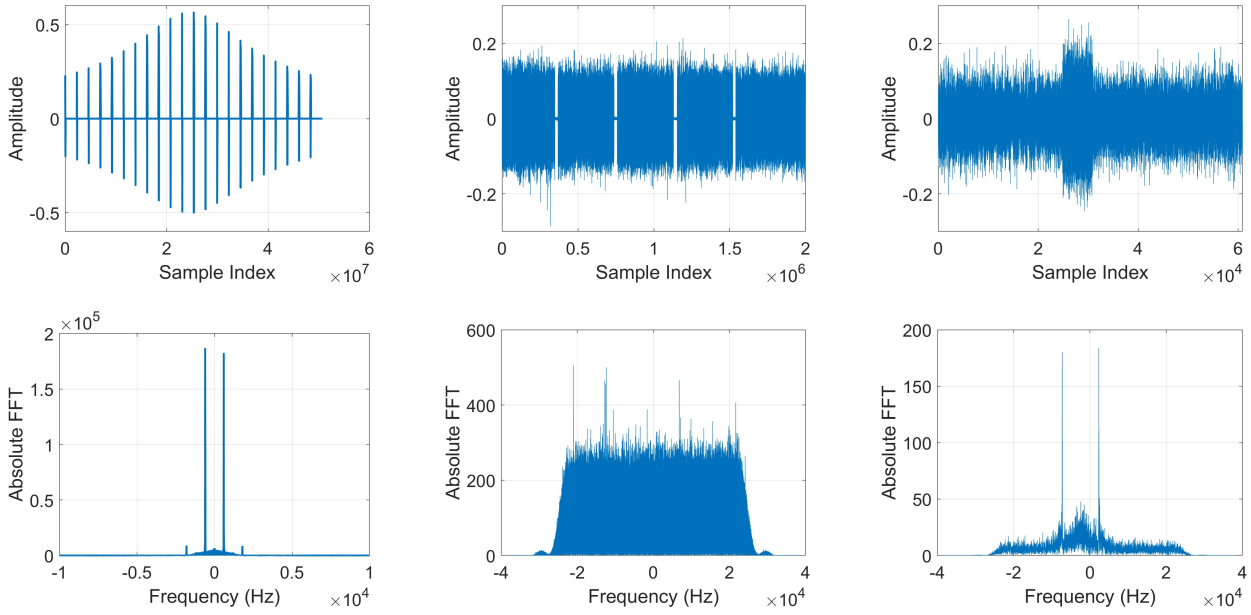
2.4 Our Experience and Challenges

Following the successful launch of our satellite, we commenced the commissioning phase which involves several stages such as initial orbit determination and tracking, first contact with the satellite, sequential activation and checks of spacecraft subsystems, and calibration of payloads, sensors, and control systems. Ideally, this phase should take about two weeks to complete. Unfortunately, we encountered some issues from the first contact with the satellite which extended the duration of the commissioning phase by several weeks. It was persistent across other satellites launched together. The issue primarily appeared to be related to the downlink communication, prompting us to conduct further investigations to identify the root cause.

The expected signal from our satellite using 2-FSK modulation can be seen in Fig. 3a(top) in the time domain, which was emulated in the lab using our satellite's RF setup. When we transfer the signal in the frequency domain using Fast Fourier Transformation (FFT) as shown in Fig. 3a(bottom), we see two distinct peaks with the perfect frequency deviation configured in 2-FSK modulation. In Fig. 3b, we can see the signal recorded by the SDR of our ground station during a satellite pass of approximately 10 minutes. According to the config of our satellite's regular beacon transmission, we expect to receive 10-14 packets. However, in the received signal shown in Fig. 3b, we do not see any clear signs of the packets as we saw in the emulated signal (Fig. 3a). When we transfer the received signal to the frequency domain using FFT, as shown in Fig. 3b(bottom), we do not see any clear peaks corresponding to 2-FSK modulated packets. It is apparent that the packets are buried under the heavy noise and interference on the ground. Nevertheless, we attempted to run a classic preamble-based packet detection by running correlation between the received signal and the known packet preamble. Ideally, we should have seen a prominent correlation peak corresponding to the beginning of each packet in the signal, translating to 10-14 peaks in our case. However, as shown in Fig. 4, we hardly see a prominent correlation peak. We then conducted a manual inspection of the received signal in search of the packets and found 12 packets in the signal, as shown in Fig. 3c after zooming in on the signal. We ran FFT on this part of the signal and found clear 2-FSK peaks in the frequency domain, confirming that the packet was transmitted from the satellite. In aggregate, the SNR of the received signal was so low that the classic preamble correlation-based technique failed in detecting packets. This experience led us to dig deeper into the reasoning behind our experience.

To begin with, the link between the satellite and ground station has a low link budget due to several factors, such as the omnidirectional radiation pattern of the satellite's antenna, the long distance between the two (hundreds of kilometers), the satellite's power limitation, and regulatory restrictions on the power flux density of satellite transmission on the ground. For instance, in the case discussed, the link budget is -112 dBm with the satellite's transmission EIRP of 22 dBm, which is comparable to other satellite service providers like Swarm reporting -108.24 dBm [43]. However, these values are calculated by assuming an elevation angle of 50°. The elevation angle has a direct impact on path loss and consequently, the link budget. In other words, a lower elevation angle means the satellite is closer to the horizon, resulting in a lower link budget. Using data from the TinyGS, an opensource distributed ground station network for IoT satellites, over a six-week period, we found that the average elevation angle across the ground stations is 26.68° indicating that the link budget is lower in the real world than the value calculated for nominal elevation angle of 50°. Furthermore, the link budget in reality is even lower due to environmental factors such as weather and surrounding infrastructure that cause signal attenuation. The average RSSI across the TinyGS ground stations was found to be -133.78 dBm, which is -128.12 dBm for our ground stations. Note that this data had packet survival bias since we have the RSSI values for only the detected packets, whereas, the packets having a RSSI below the minimum detectable intensity (MDI) of ground station radio or very poor SNR values, are not detected and hence not recorded. Additionally, we noticed that the rotation of the satellite around its own axis, known as tumbling, leads to the antenna depointing, resulting in decreased signal strength.

The ability to detect and decode packets despite having an RSSI above the MDI depends on the level of noise and interference at the ground station, which can originate from both terrestrial and non-terrestrial sources. Interference from other satellite constellations is common since IoT satellites often use a band designated for Earth exploration, meteorological, and mobile satellites. In addition, interference from



(a) *Emulated signal containing 2-FSK packets transmitted by satellite. Time domain (top) and frequency domain (bottom).*

(b) *Signal received at the ground station in real-world. Time domain (top) and frequency domain (bottom).*

(c) *Packet found after manual inspection by zooming in the signal. Time domain (top) and frequency domain (bottom).*

Figure 3: *Our experience in processing signal received from our satellite at the collaborator's ground station.*

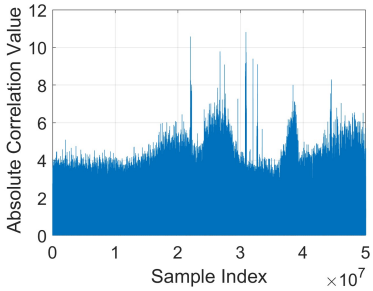


Figure 4: *Packet detection from a real-world signal using single preamble correlation. Results: no clear correlation peak at the beginnings of packets.*

terrestrial communication can significantly affect the satellite signal due to the low link budget, and operation in the ISM band can make them susceptible to direct terrestrial interference [15, 28, 31, 59]. Besides, the use of low-cost and low-complexity hardware in ground stations also contributes to the noise. As a combined effect of these factors, the SNR value gets extremely poor and becomes negative [19]. To highlight the extent of the poor SNR values, the average SNR at our ground station locations was found to be as low as -18 dB, while it was -17.11 dB across TinyGS ground stations.

In addition to external interference, IoT satellites also face interference from other satellites within their own constellation. This is due to the omnidirectional radiation pattern of a satellite's antenna, which results in a footprint that can cover millions of square kilometers. The footprints of adjacent satellites from the same constellation overlap with each other as shown in Fig. 2. As mentioned earlier, ground stations are strategically placed in locations where they can maximize the number of satellite passes, such as the polar regions. With over 170 IoT satellites in a constellation like Swarm, a ground station in such an optimal location spends 83% of its time in the overlapping footprint of two or more satellites. At any given time, the ground station can see up to 23 satellites with a median of 4 (see Appendix A). Now, the satellites having overlapped footprints create interference with each other as they transmit to a ground station in an overlapping region. As a solution, when downloading data from a satellite at a ground station, only one satellite is scheduled for communication at a time, despite multiple satellites being within view. The ground station sends a command to the scheduled satellite to initiate data transfer while keeping the others silent. This approach, while addressing interference issues, results in significantly reduced spectral efficiency. This impact is even more pronounced for ground stations located in polar regions, which are almost all the time in overlapping footprints of multiple satellites.

3 Spectrumize's Algorithm

In this section, we present our technique for detecting and decoding the packets received from IoT satellites that are severely buried under the noise floor (Fig. 3b (top)). We found that conventional technique of correlating with a single preamble exhibits poor performance (Fig. 4), where the peaks corresponding to packet start time are obscured by the

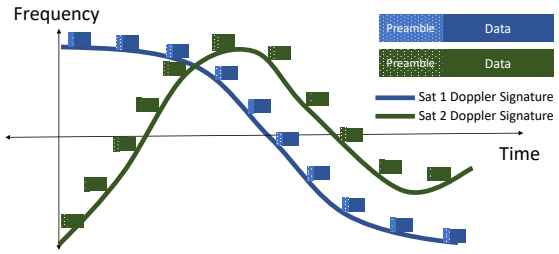


Figure 5: *Unique Doppler Signatures allow us to disentangle the collisions from two different satellites at a ground station.*

overwhelming noise.

To enhance the prominence of the peaks associated with packet start times, it is crucial to increase the correlation score of the genuine packet within the noisy received signal. To do so, we leverage two unique observations specific to the IoT satellite domain.

1. Downlink from satellite to ground stations are contention-free transmissions: Traditional wireless networks, such as WiFi and cellular, operate on a contention-based system where transmitters employ carrier sense before transmitting packets. Consequently, packet transmissions in traditional wireless networks do not typically adhere to a periodic or predetermined pattern. However, the dynamics in IoT satellite networks differ. Techniques like carrier sense are ineffective due to the vast footprint of satellites on the Earth's surface, making it possible for distant satellites to inadvertently collide at a ground station. As a result, the current industry standard for deploying downlink communication protocols from IoT satellites to ground stations adheres to a contention-free schedule.

More specifically, the IoT satellite network employs a centralized downlink scheduler that assigns ground stations to receive packets from specific satellites during each satellite's contact period with the ground station. The IoT satellite receives this transmission schedule in advance, along with commands that specify the number of packets, denoted as N, that it can transmit. During the satellite's contact period with its designated ground stations, it transmits a train of N data packets, each periodically spaced. It's essential to note that because the satellite does not perform carrier sensing before transmitting data packets, its packet transmissions follow a deterministic periodic pattern, as illustrated in Fig. 5. The key insight of Spectrumize is that this known periodic sequence of N packets can be harnessed to construct a longer virtual *preamble train*. This virtual preamble train serves to enhance the correlation score of the target signal, elevating it above the noise floor and facilitating the detection of packets.

2. Orbital motion of satellites contribute unique Doppler signatures at the physical layer: The second pivotal insight underpinning Spectrumize revolves around the exploitation of the orbital motion of the satellite during its contact period with a ground station, which introduces a distinctive Doppler signature unique to each satellite-ground station link. This signature is influenced by factors such as the satellite's position and velocity w.r.t. the stationary ground station. Fig 5 provides an example of two satellites, each characterized by its unique Doppler signature. These Doppler signatures effectively introduce an additional layer of physical modulation over the existing data packets transmitted. In essence, this process is akin to the codes used in CDMA. Similarly, we can harness this well-known temporally varying Doppler signature of the satellite-ground station link to enhance signal SNR and to demultiplex simultaneous transmissions.

We next detail our packet detection and collision resolution algorithm, followed by techniques to address hardware imperfections like CFO and time jitter in the packet sequence.

3.1 Packet Detection

Following the transmit schedule and command from the ground station, the satellite transmits N data packets to the assigned ground station during its contact period. Each individual data packet is preceded by a preamble to help with the detection of packet start time and subsequent decoding. Let the preamble signal be denoted by $p(t)$. Then the sequence of preambles $s(t)$ transmitted by the satellite over the course of the N data packets is:

$$s(t) = \sum_{k=0}^{N-1} h_k \; p(t - k\tau) + n(t) \tag{1}$$

where τ represents the inter-packet interval at which the satellite transmits its data packets. The wireless channel experienced by the kth data packet, denoted as h_k, and the received signal's noise, $n(t)$, comprise both thermal noise and external interference originating from terrestrial source or other satellite transmissions.

Traditional packet detection methods would attempt to find the packet start times by correlating the received signal $s(t)$ with the known preamble $p(t)$. However, in our real-world experiments, due to the overwhelming amount of noise, we could not detect any packet start times reliably. To address this challenge, Spectrumize's insight is that we can leverage the known periodicity of the transmitted signal and construct a longer virtual preamble train to correlate with the received signal. This, in turn, would allow us to boost the correlation score and would raise the peaks corresponding to the packet start times above the noise floor.

Specifically, the simulated preamble train $\tilde{p}(t)$ that we construct to correlate with the received signal $s(t)$ is:

$$\tilde{p}(t) = \sum_{k=0}^{N-1} p(t - k\tau) \tag{2}$$

where $p(t)$ is the known preamble structure. So given the virtual preamble train, our packet detection method works by correlating $s(t)$ with $\tilde{p}(t)$. That is, we want to find

$$arg\ max_{t'} \; |s(t) \; \tilde{p}^*(t - t')| \tag{3}$$

where $\tilde{p}^*$ is the complex conjugate of $\tilde{p}$. The time t' that

maximizes the above correlation would be when the preamble trains in the two signals $s(t)$ and $\tilde{p}(t)$ perfectly align. So when the two trains align, the value of the correlation score is

$$Corr_Score = \sum_{k=0}^{N-1} h_k \left|\left|p(t)\right|\right|^2 \quad (4)$$

As observed in Eq. 4, the utilization of a preamble train for correlation results in the aggregation of correlation terms from each preamble in the train. This aggregation allows the peak corresponding to the packet start times to surpass the noise floor. It's important to note, however, that the terms in the summation do not add up constructively due to the presence of the channel term h_k. The channel h_k introduces a random phase and gain variation to each term in the summation. While coherent addition of the terms in the summation would be the ideal scenario for packet detection, we still get significant gains from non-coherent combination alone. In Appendix B, we present the proof for the following Lemma.

Lemma 3.1 *Consider $a_1, a_2, \ldots, a_n$ to be complex numbers on the unit circle with random phase. That is, $a_i = e^{j\theta_i}$ where θ_i is uniformly sampled from $[0, 2\pi]$. Let s_n be defined as $s_n = \sum_{i=1}^{n} a_i$. Then we have $\mathbb{E}\left[\|s_n\|^2\right] = O(n)$.*

The above Lemma 3.1 demonstrates that the gains from non-coherent combining grows monotonically with the number of terms in the non-coherent summation. Hence, leveraging a preamble train allows us to detect the packets buried under the noise floor, and this was not possible with single preamble-based correlation techniques. This observation also aligns with the rich body of literature on non-coherent combining and beamforming of wireless signals [32, 37, 50, 51, 58].

Up to this point, we have discussed how leveraging a virtual preamble train can enhance the SNR and enable the detection of packet start times. However, this alone is not sufficient. In addition to the wireless channel's contribution represented by h_k, it is crucial to consider the presence of a Doppler signature component. As previously discussed, the orbital motion of the satellite results in a unique Doppler signature for each satellite-ground station link, which overlays the data transmissions from the satellite. This time-varying Doppler signature is denoted as $f_d(t)$. As illustrated in Fig. 5, the Doppler signatures for two satellites are depicted in green and blue. This signature essentially introduces frequency modulation on top of the sequence of data packets transmitted by each satellite. Given that the Doppler signature differs for each satellite-ground station link, we can liken them to CDMA codes. We can leverage these Doppler signatures to further enhance the SNR of the desired signal.

Specifically, we can modify our simulated preamble train $\tilde{p}(t)$ as follows

$$\tilde{p}(t) = \sum_{k=0}^{N-1} p(t - k\tau)\, e^{j2\pi f_d(t)t} \quad (5)$$

Note that the actual received signal $s(t)$ from the satellite already has the Doppler shift encoded in it, so in the correlation equation shown in Eq. 4, when we plug in our modified $\tilde{p}(t)$ from Eq. 5, the Doppler signature terms will cancel and our correlation score is going to remain the same as in Eq. 4. Hence, correcting for this Doppler signature $f_d(t)$ is essential for the correlation across preambles from different packets to add up constructively.

Packet separation: We would like to emphasize that the term $e^{j2\pi f_d(t)t}$ effectively functions as a time-varying code in this context, aiding in the separation of packets between transmissions from terrestrial sources or other satellites operating in the same frequency band. To illustrate this, let's consider the reception of transmissions from two satellites, as depicted in Fig. 5, with their respective Doppler signatures denoted as $f_d^1(t)$ and $f_d^2(t)$. Without loss of generality, let's focus on detecting the start times of packets from satellite 1, while considering packets from satellite 2 as interference. For satellite 1, employing the modified preamble train shown in Eq. 5, our correlation score achieves its highest value when the simulated preamble train with the Doppler signature aligns precisely with the train of packets present in the transmitted signal from satellite 1.

However, when considering the transmissions from satellite 2 that cause interference at the ground station, the correlation between our simulated preamble train and the signal from satellite 2 diminishes. This is because the Doppler signature encoded in satellite 2's transmissions differs from the Doppler signature we are seeking in the received signal. Specifically, the correlation score for satellite 2's transmission will be:

$$Corr_Score \sim \sum_{k=0}^{N-1} h_k \left|\left|p(t)\right|\right|^2 e^{j2\pi(f_d^2(t) - f_d^1(t))t} \quad (6)$$

Hence, as evident from the above equation, the correlation terms from different packets in the train no longer add up constructively. This occurs because the value of the term $(f_d^2(t) - f_d^1(t))$ varies with time, contributing different phase offsets to each term in the summation. Consequently, by harnessing the unique Doppler signature for each satellite, we can further effectively suppress transmissions from other sources.

3.2 Hardware Imperfections

However, it is important to note that we have not yet accounted for hardware imperfections in our technique. Due to these hardware imperfections and the lack of perfect synchronization, there will be disparities between the real signal $s(t)$ and our simulated preamble train $\tilde{p}(t)$. More specifically, the received signal $s(t)$ will exhibit a Carrier Frequency Offset (CFO) component, causing it to deviate from the simulated preamble train $\tilde{p}(t)$. Additionally, our assumption of a uniform periodic train of data packets every τ seconds may not hold true in practical scenarios. Processing delays through different layers of the network stack often introduce some jitter, causing the packets in the train not to be transmitted at precisely τ-second intervals.

It is imperative that we consider the impact of these hardware imperfections when creating the virtual preamble train used for detecting the presence and start times of packets. However, standard channel-based techniques are inadequate for estimating these offsets, as we must first locate the packet to measure these channels. Consequently, we begin with a set of initial values and iterate on these values until we observe the peaks from the data packets rising above the noise floor. Specifically, we construct the virtual preamble train with a CFO value of Δf and jitter denoted as ε. In this scenario, our modified virtual preamble takes the following form –

$$\tilde{p}(t) = \sum_{k=0}^{N-1} p(t - k(\tau + \varepsilon))\, e^{j2\pi f_d(t)t}\, e^{j2\pi\Delta f t} \tag{7}$$

The above equation for the preamble train takes into account the unique Doppler signature $f_d(t)$, the Carrier Frequency Offset (CFO) Δf, and the jitter ε in the periodic train. It is worth noting that the jitter may result in non-uniformly spaced packets, rather than just a uniformly perturbed periodicity of ε. However, for the sake of maintaining manageable computational complexity, we assume only a jitter in the periodicity of the train. Our experiments have demonstrated that this approach is sufficient to elevate the correlation score above the noise floor.

The algorithm operates as follows: We start by selecting a set of candidate CFO values $\{\Delta f_1, \ldots, \Delta f_x\}$ and a set of candidate jitter values $\{\varepsilon_1, \ldots, \varepsilon_y\}$. It is important to note that the Doppler signature is known, thanks to the TLE information from the satellite. For each pair of CFO and jitter values, we create a virtual preamble train to correlate with the received signal $s(t)$. From the collection of candidate preamble trains, we identify the CFO value Δf^* and jitter value ε^* that maximize the correlation score with $s(t)$. These chosen values of CFO and jitter may not be perfectly accurate, but our objective is to select values that are *close enough* to elevate the correlation score above the noise floor. This approach enables us to determine the position of the first packet in the signal. However, what about subsequent packets?

According to Eq.4, the correlation peaks for subsequent packets decrease as we decode and discard packets from the signal. It causes the peaks of trailing packets in the train to become increasingly obscured by noise. This makes detection as challenging as single-preamble-based detection for these trailing packets, as Eq.4 loses its summation advantage. To tackle this issue, we must restore the summation advantage by reconstructing the preamble train. To achieve this, following the initial packet detection and decoding, we remove the corresponding portion in the signal and append it to the end of the signal. We apply the same process to the virtual preamble train, with the second packet now becoming the head of the train. This procedure continues until we successfully decode all the packets in a circular queue fashion. This approach ensures the train's length is maintained, allowing us to preserve the summation advantage in Eq. 4.

3.3 Collision Resolution for Packet Decoding

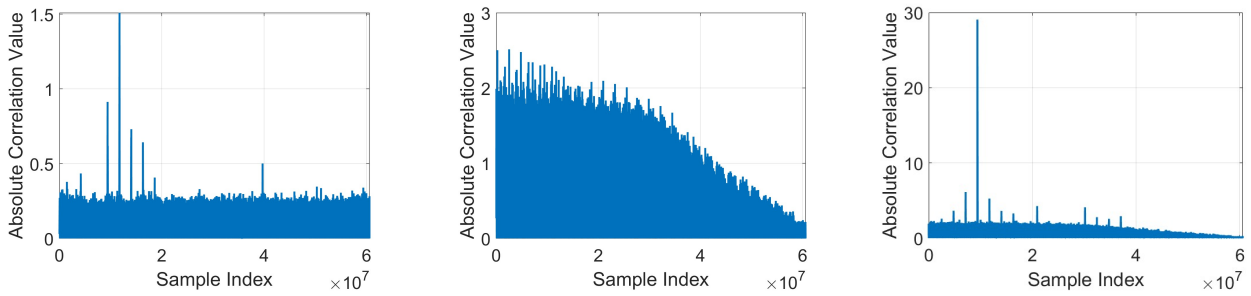
As previously mentioned, ground stations frequently find themselves within the overlapping footprints of multiple satellites due to the large coverage areas of these satellites. Although they have the potential to communicate with several satellites, current scheduling practices dictate that only one satellite can be active at a time. This precaution is taken to prevent collisions between transmissions from satellites within the same constellation, but it comes at the cost of poor spectral efficiency. As a remedy, Spectrumize facilitates simultaneous transmissions in the same frequency band from multiple satellites within the same constellation.

Spectrumize efficiently separates and decodes data symbols from the intended satellite while filtering out symbols from collided packets of other satellites. This capability is made possible by the distinct Doppler signature unique to each satellite-ground station link. Each packet, within its duration, carries a Doppler signature that differs significantly among satellites. In essence, each packet can be seen as encoded with a unique physical layer code defined by its Doppler signature, as demonstrated in our results. Multiplying the combined signals of collided packets by the complex conjugate of the relevant Doppler signature ($f_d^*(t)$) allows us to enhance the SNR of the desired packet's data symbols while attenuating data symbols from interfering packets. This approach leverages Doppler signatures in a manner akin to CDMA codes. Following this SNR enhancement, we employ a standard LoRa collision resolution algorithm to further isolate the symbols corresponding to the target packet, effectively eliminating symbols from interfering packets.

We focus on LoRa modulation for our packet decoding use case, which is prevalent in the satellite IoT industry [6, 7, 43]. Our satellite employs LoRa modulation along with 2-FSK, and the techniques we employ can also be extended to 2-FSK. Numerous studies have delved into decoding collided LoRa packets [24, 45, 46, 52, 53]. In our work, we employ the CIC algorithm [38] for decoding collided packets. It is important to note that merely applying the CIC algorithm [38] to the received signal $s(t)$ without prior Doppler signature correction, as demonstrated in Section 4, is ineffective, especially for long packets. The correction of the Doppler signature is a prerequisite for successfully decoding collided packets and extracting the desired data symbols.

4 Evaluation

We evaluate Spectrumize's performance in both simulation and real-world environments. In simulation and benchtop emulator setup, we assess the micro-benchmarks and scalability of Spectrumize, while in the real-world testbed, we evaluate its performance in actual operation settings.



(a) *Single preamble with Doppler signature* (b) *Preamble train without Doppler* (c) *Preamble train with Doppler signature*

Figure 6: *Micro-benchmark of packet detection techniques: a single preamble or a train without the Doppler signature suffering from large noise. A preamble train with the Doppler signature applied exhibits a* 13*dB gain in correlation detection power.*

4.1 Real-world Testbed

Our real-world testbed consists of our own satellite that is launched in collaboration with FOSSA Systems, along with two ground stations operated by the same collaborator, as described in Section 2. While most of the experiments are conducted using our own satellite, we also use other satellites in the constellation for certain experiments, such as those involving packet collisions. At the ground stations, we collect raw SDR recordings of satellite passes, as well as the SNR and RSSI values of the packets that are decoded by the Semtech SX126x radio. This radio is connected to the receive chain of the ground station through an RF splitter. Additionally, we utilize the Microsoft Azure Orbital ground stations to receive signals from our satellite [29].

4.2 Simulator and Benchtop Emulator Setup

For simulations, we employ MATLAB, leveraging the Communications Toolbox for 2-FSK modulation and LoRaMatlab for LoRa modulation [12]. To bring the evaluation closer to real-world conditions, we also establish a benchtop emulator setup. In this benchtop setup, we utilize three SDRs, specifically USRP B200 [11], integrated with GNU Radio and tuned to the satellite frequency plan. Two of these SDRs replicate the transmitters of two distinct satellites, while the third serves as the ground station. To introduce the Doppler effect, we calculate the Doppler signature for a satellite pass over our actual ground station location using the corresponding satellite's TLE. Subsequently, we multiply this Doppler signature with the transmission signal. It's important to note that, similar to the real-world testbed, there is no clock synchronization across the SDRs.

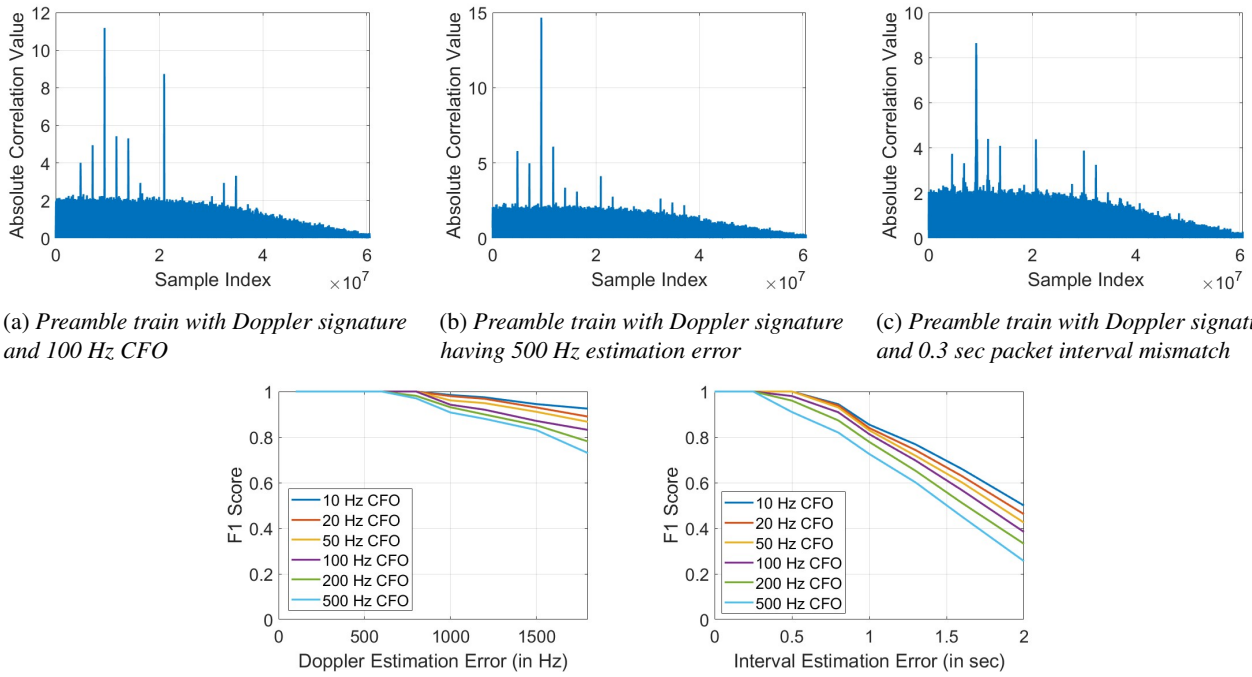
4.3 Packet Detection

In order to assess the effectiveness of Spectrumize's packet detection technique, we initially conduct a series of micro-benchmark tests to determine how different parameters impact our approach, as detailed in Section 3. Subsequently, we apply our method to the data collected from the real-world testbed, and compare its performance with the conventional technique.

4.3.1 Micro-benchmark

We first conduct the benchtop emulation to examine the impact of different correlation setups. We take a scenario where a satellite transmits 2-GFSK packets at a 9600 baud rate, every 30 seconds, with Doppler variation and Additive White Gaussian Noise (AWGN) based on real-world measurements. Our baseline approach employs classic preamble-based correlation for packet detection in the received signal. However, we encounter a lack of prominent correlation peaks at the start of packets, as illustrated in Fig. 4. To improve the correlation results, we correlate the signal with a single preamble multiplied by the Doppler signature. This results in marginally improved correlation outcomes where the peak remains relatively subtle in terms of correlation magnitude as shown in Fig. 6a. Subsequently, we experiment with correlating the signal using a preamble train aligned with the satellite's packet transmission sequence but without Doppler correction. This approach yields subpar results, with indistinct peaks that are challenging to distinguish, as demonstrated in Fig. 6b. Finally, we implement our proposed method, which involves correlating the signal with a preamble train multiplied by the Doppler signature. This approach demonstrates a significant improvement, showcasing a gain of approximately 13dB in correlation detection power, as depicted in Fig. 6c. In certain scenarios, we achieve an even higher gain of up to 25dB demonstrating the effectiveness of our technique in detecting packets despite heavy noise.

After examining our technique's performance under ideal conditions, we then investigate its robustness against real-world factors such as hardware imperfections. We begin by introducing a CFO of 100 Hz to the transmitted packets. When we use our correlation technique to detect these packets, we observe a decrease in the correlation peak (Fig. 7a), but it is still very prominent compared to the baseline results shown in Fig. 4. Next, we consider an extreme case of Doppler esti-



(a) *Preamble train with Doppler signature and 100 Hz CFO*

(b) *Preamble train with Doppler signature having 500 Hz estimation error*

(c) *Preamble train with Doppler signature and 0.3 sec packet interval mismatch*

(d) *Impact of Doppler estimation error*

(e) *Impact of interval mismatch error*

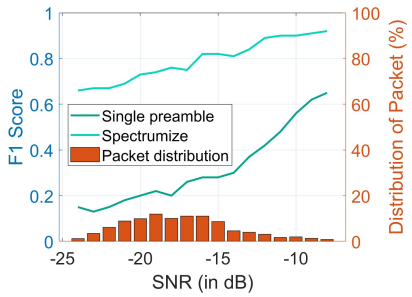
Figure 7: *Robustness of packet detection to errors: our packet detection is robust to timing or frequency mismatch.*

mation error, 500 Hz. Although the resulting correlation peak is shorter (Fig. 7b) than the ideal case, it is still very prominent. We then study the impact of packet interval mismatch by adding a 0.3 sec delay to each preamble in our preamble train. As shown in Fig. 7c, this factor has a negative impact on our performance. However, we still obtain a distinguishable correlation peak that is much higher than the baseline. For further investigation of the impact of these factors, we try out a range of values. The results are presented in Fig. 7d and 7e. In Fig. 7d, we demonstrate the effect of Doppler estimation error and CFO on the F1 score for packet detection. We observe that our technique can tolerate a Doppler estimation error of up to 1 kHz, while the CFO can reach several hundred Hz without significantly affecting the F1 score. Fig. 7e shows the impact of packet interval estimation error on the F1 score, indicating that our technique can perform well with an interval estimation error of up to one second. However, we have observed an average interval estimation error of around 0.18 sec in real-world scenarios, and our algorithm makes iterative adjustments to the preamble train to further reduce this error. Additionally, we found that when the F1 score decreases, it is mainly due to a decrease in precision rather than recall. This indicates that our technique can detect actual packets even under very adverse conditions.

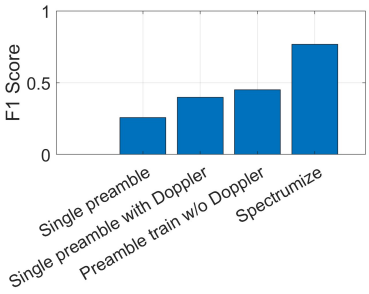
4.3.2 Real-world evaluation

In addition to emulating our technique, we also test it in real-world scenarios using our testbed. We apply our technique on SDR recording traces gathered from ground station setups, which contain packets transmitted from our satellite. We collect traces from 67 satellite passes at various times over a one-year period to ensure that we capture a range of factors related to satellite orbital motion and noise at the ground station, including weather conditions. These traces contain over 700 packets of varying lengths transmitted using 2-GFSK and LoRa modulation schemes with intervals of a few seconds to a minute between transmissions. To carry out the evaluation, we implement our technique using MATLAB and execute it in a virtual machine having 32 GiB RAM and 4 vCPUs.

The results of the real-world evaluation and comparisons among different approaches are presented in Fig. 8. Fig. 8a displays the F1 score for packet detection using Spectrumize and a classic approach of single preamble-based correlation at different SNR values (rounded off to the closest integer number). Additionally, the distribution of packets across SNR values is shown. As we only have the SNR values for the packets decoded by the SX126x radio at the ground station during a pass, we use this as a reference and calculate relative SNR values for the packets detected by our approach but not by the SX126x radio. Fig. 8a demonstrates that Spectrumize outperforms the classic approach by 5× in low SNR scenarios, where we have a higher distribution of packets since a significant amount of packets reaching the ground station suffer from low SNR. In Fig. 8b, the overall performance of Spectrumize and other approaches in detecting packets is shown. Spectrumize exhibits 3× better performance than the classic approach of single preamble-based correlation and



(a) *F1 score for packet detection using Spectrumize and classic approach along with distribution of packet by SNR*



(b) *Overall F1 score for packet detection using Spectrumize and other approaches*

Figure 8: *Real-world evaluation: performance of Spectrumize in packet detection and comparison against classic approach (single preamble correlation without Doppler correction) and other intermediate approaches.*

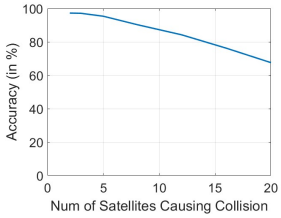
2× better performance even when we incorporate Doppler correction with a single preamble.

4.4 Collision Resolution

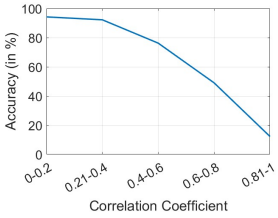
As explained in Section 2 and 3, packet collision is a significant problem when downloading data from a satellite, affecting the spectral efficiency. We now evaluate how effectively Spectrumize mitigates this issue.

4.4.1 Simulation

We first evaluate Spectrumize's ability in resolving packet collision through simulation. It further aims to assess the scalability of Spectrumize in terms of the number of satellites visible from a ground station as illustrated in Fig. 2. We select a polar region as the location for the ground station, as this is where satellites most frequently pass overhead, resulting in higher overlaps of their footprints. We simulate over 170 IoT satellites from the Swarm constellation by using their TLEs. Satellites are set to transmit packets at 2-second intervals when they come into view of the ground station. The packet length is set to 3.5 seconds to cause frequent collisions. The rest of the simulation setup is similar to that described in



(a) *Collision resolution accuracy by number of satellites in view of ground station*



(b) *Collision resolution accuracy by the correlation coefficient of Doppler signatures*

Figure 9: *Performance of Spectrumize in collision resolution and its scalability in simulation.*

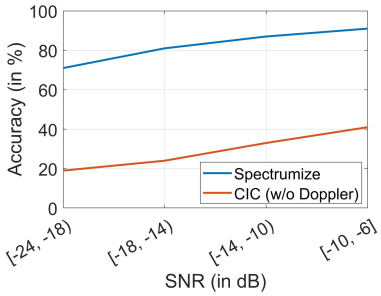


Figure 10: *Performance of Spectrumize's collision resolution in real-world. Spectrumize achieves over* 80% *accuracy on average across different SNR conditions.*

Section 4.2. As discussed in Section 3.3, we here use Spectrumize for packet detection and boosting the SNR of a packet of interest w.r.t. colliding packets using Doppler signature. We later employ CIC [38] to extract the symbols in a packet. Since the opensource source code of CIC supports only LoRa modulation, we use LoRa packets in our simulation. However, a similar technique can be translated to 2-FSK.

The performance of Spectrumize in resolving packet collisions is depicted in Fig. 9a, where we illustrate how its accuracy varies by the number of satellites that cause collisions. We begin with a few satellites from the Swarm constellation and gradually increase the number to the maximum in the simulation. We achieve an accuracy of over 90% in resolving collisions caused by up to eight satellites. The accuracy goes down as the number of satellites causing collision increases. This occurs since more satellites lead to a higher probability of similarity in Doppler signatures among the satellites. In Fig. 9b, we show the impact of similarity in Doppler signature on the accuracy, where it is measured in terms of the correlation coefficient between Doppler signatures from two satellites. The accuracy is observed to decline exponentially with the correlation coefficient.

4.4.2 Real-world evaluation

In addition to the simulation, we conduct the real-world evaluation of Spectrumize's ability to resolve packet collision using our testbed. To perform this evaluation, we use multiple satel-

lites from the same constellation launched by our collaborator. We configure two satellites to cause packet collision at the ground stations using LoRa modulation with a spreading factor of 11, a bandwidth of 125 kHz, and a code rate of 8. Given the limited number of satellites in orbit, naturally occurring collision scenarios are low. To address this, we employ our benchtop emulator to create such collisions and enrich the dataset of SDR traces. We then employ Spectrumize on the SDR traces to detect the packets and resolve the collision. We further compare its performance with the standalone CIC [38] approach proposed for terrestrial networks as a baseline. It is important to note that CIC does not use Doppler correction. The results in Fig. 10 show that Spectrumize achieves over 80% accuracy on average, with more than 70% accuracy in low SNR conditions and 90% accuracy in good SNR conditions. On the other hand, CIC performs poorly in low SNR conditions as it relies on the classic approach of single preamble correlation for packet detection. Although the performance slightly improves with the SNR, the absence of Doppler adjustment hampers its performance. We also observe that CIC struggles in decoding longer packets compared to shorter ones given the Doppler change within a packet.

5 Related Work

The quest to enhance spectral efficiency in satellite communication has garnered considerable attention in the state-of-the-art literature, with numerous techniques proposed, including MIMO and beamforming. [20, 44, 54], cognitive radio communication [39, 40], and interference mitigation and cancellation [16, 22, 22, 22, 23, 41], among others. As the coexistence of space and terrestrial networks becomes important in 5G and 6G, this field is gaining significant interest [18, 26, 34]. However, all of these techniques have been developed for broadband communication in space. This paper focuses on solving the problem of spectral efficiency in narrowband IoT satellite communication, which has unique challenges such as power constraints, lack of beamforming capability, lower frequency and bandwidth, and the use of low-cost and low-complexity connectivity hardware. As a remedy, we leverage the unique Doppler signature of satellites in packet detection and collision resolution.

5.1 Packet detection

In wireless communication, a preamble is added to the start of a packet to aid in synchronization. Various correlation-based methods have been proposed and implemented to detect the preamble in the received signal [27, 30, 56, 57]. However, in low SNR situations encountered in satellite IoT networks, single-preamble correlation techniques are not effective, as we have previously shown. To counteract the Doppler shift caused by the high relative motion of LEO satellites, Doppler compensation techniques are typically used in satellite communication, both at the hardware and software levels [13, 17, 42, 49, 55]. To the best of our knowledge, we are the initial proponents of utilizing a series of preambles multiplied with the Doppler shift of a satellite to identify packets in situations where the signal-to-noise ratio is low.

5.2 Collision resolution

The process of collision resolution in satellite communication is well researched, but not specifically in the context of narrowband IoT satellite communication. On the other hand, in terrestrial IoT networks, such as LoRa, various techniques have been proposed to address packet collision. For example, Choir [24] groups LoRa symbols based on the CFO of the transmitter hardware, while FTrack [53] generates time-frequency tracks of the symbols using sliding window Short Term Fourier Transforms (STFT) on the de-chirped signal. Similarly, mLoRa [52] and CoLoRa [46] group LoRa symbols based on the power of the received signal. NScale [45] translates the packet time offset into frequency features for improved robustness, while CIC [38] cancels out interference by combining spectra obtained from different parts of each symbol. However, the major challenges in applying these techniques to satellite IoT networks are very low SNR and RSSI, and Doppler frequency offset.

6 Conclusion

The IoT satellite industry is growing rapidly due to the low cost, low-complexity, and ease of deployment of satellite constellations in orbit. However, our experience with launching a satellite shows that the downlink from the IoT satellite to the ground station suffers from large packet losses due to their low link budget, large footprint and packet collisions, and satellite motion. This paper introduces Spectrumize, a novel approach to improve the spectral efficiency of IoT satellite downlink. Spectrumize leverages the predictable motion of the satellite and uses the temporal variation of the Doppler shift as a signature to boost signal SNR and de-multiplex simultaneous transmissions. Our evaluation involving our launched satellite and collaborator's ground stations shows that Spectrumize improves downlink communication by 3× over state-of-the-art.

7 Acknowledgement

We extend our gratitude to FOSSA Systems for their invaluable support in establishing the real-world testbed and sharing their expertise on industry practices. Our sincere appreciation also goes to the shepherd and the anonymous reviewers for their insightful feedback.

References

[1] SatNOGS – Open Source global network of satellite ground-stations. https://satnogs.org/.

[2] TinyGS – The Open Source Global Satellite Network. https://tinygs.com/.

[3] Australian IoT startup Myriota raises USD 19 million. https://tinyurl.com/mw77mzh3, 2020.

[4] Satellite launches fueling race to connect out of reach devices. https://tinyurl.com/3ehc65r8, 2021.

[5] SpaceX buys out satellite IOT startup Swarm technologies. https://tinyurl.com/5y58ar4u, 2021.

[6] Cash-rich EchoStar to take on global IoT market next year. https://tinyurl.com/2rmw8kdw, 2023.

[7] FOSSA Systems - Global Cost-Effective IoT Connectivity. https://fossa.systems/, Feb 2023.

[8] Nanosatellite Launch Forecasts - Track Record and Latest Prediction. https://tinyurl.com/2p948kwv, 2023.

[9] Swarm - Low cost global satellite connectivity for IoT. https://swarm.space/, 2023.

[10] SX1302: LoRa Core Digital Baseband Chip for LoRaWAN network gateways. https://tinyurl.com/5d8xsb7k, 2023.

[11] USRP B200. https://www.ettus.com/all-products/ub200-kit/, 2023.

[12] Bassel Al Homssi, Kosta Dakic, Simon Maselli, Hans Wolf, Sithamparanathan Kandeepan, and Akram Al-Hourani. IoT network design using open-source LoRa coverage emulator. *IEEE access*, 9:53636–53646, 2021.

[13] Irfan Ali, Naofal Al-Dhahir, and John E Hershey. Doppler characterization for LEO satellites. *IEEE transactions on communications*, 46(3):309–313, 1998.

[14] Amazon Inc. AWS Ground Station. https://aws.amazon.com/ground-station/.

[15] Anritsu. Resolving Interference Issues at Satellite Ground Stations. https://tinyurl.com/2fbdzpek.

[16] Bassel F Beidas, Hesham El Gamal, and Stan Kay. Iterative interference cancellation for high spectral efficiency satellite communications. *IEEE transactions on communications*, 50(1):31–36, 2002.

[17] HD Black and A Eisner. Correcting satellite Doppler data for tropospheric effects. *Journal of Geophysical Research: Atmospheres*, 89(D2):2616–2626, 1984.

[18] Christophe Braun, Andra M Voicu, Ljiljana Simić, and Petri Mähönen. Should we worry about interference in emerging dense NGSO satellite constellations? In *2019 IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, pages 1–10. IEEE, 2019.

[19] Tusher Chakraborty, Heping Shi, Zerina Kapetanovic, Bodhi Priyantha, Deepak Vasisht, Binh Vu, Parag Pandit, Prasad Pillai, Yaswant Chabria, Andrew Nelson, et al. Whisper: IoT in the TV White Space Spectrum. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022*, pages 401–417. USENIX Association, 2022.

[20] Dimitrios Christopoulos, Symeon Chatzinotas, Gan Zheng, Joël Grotz, and Björn Ottersten. Linear and nonlinear techniques for multibeam joint processing in satellite communications. *EURASIP journal on wireless communications and networking*, 2012:1–13, 2012.

[21] Semtech Corporation. Semtech SX1262. https://www.semtech.com/products/wireless-rf/lora-transceivers/sx1262, 2020.

[22] Cottatellucci, Laura and Debbah, Merouane and Gallinaro, Gennaro and Mueller, Ralf and Neri, Massimo and Rinaldo, Rita. Interference mitigation techniques for broadband satellite systems. In *24th AIAA International Communications Satellite Systems Conference*, page 5348, 2006.

[23] Marcos Alvarez Díaz, Nicolas Courville, Carlos Mosquera, Gianluigi Liva, and Giovanni E Corazza. Nonlinear interference mitigation for broadband multimedia satellite systems. In *2007 International Workshop on Satellite and Space Communications*, pages 61–65. IEEE, 2007.

[24] Rashad Eletreby, Diana Zhang, Swarun Kumar, and Osman Yağan. Empowering low-power wide area networks in urban settings. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 309–321, 2017.

[25] Tim Fernholz. Silicon Valley satellite startup installs ground station in Antarctica. https://tinyurl.com/4na6w7v6, Feb 2020.

[26] Ghaith Hattab, Prakash Moorut, Eugene Visotsky, Mark Cudak, and Amitava Ghosh. Interference analysis of the coexistence of 5G cellular networks with satellite earth stations in 3.7-4.2 GHz. In *2018 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6. IEEE, 2018.

[27] David L Herrick and Paul K Lee. CHESS a new reliable high speed HF radio. In *Proceedings of MILCOM'96*

IEEE Military Communications Conference, volume 3, pages 684–690. IEEE, 1996.

[28] Nozhan Hosseini and David W Matolak. Software defined radios as cognitive relays for satellite ground stations incurring terrestrial interference. In *2017 Cognitive Communications for Aerospace Applications Workshop (CCAA)*, pages 1–4. IEEE, 2017.

[29] Microsoft. Azure Orbital. https://azure.microsoft.com/en-us/services/orbital/.

[30] Sumeeth Nagaraj, Sheehan Khan, Christian Schlegel, and Marat V Burnashev. Differential preamble detection in packet-based wireless networks. *IEEE Transactions on Wireless Communications*, 8(2):599–607, 2009.

[31] NASA. State-of-the-Art of Small Spacecraft Technology. https://www.nasa.gov/smallsat-institute/sst-soa/communications, 2022.

[32] Cunhua Pan, Hong Ren, Maged Elkashlan, Arumugam Nallanathan, and Lajos Hanzo. The non-coherent ultra-dense C-RAN is capable of outperforming its coherent counterpart at a limited fronthaul capacity. *IEEE Journal on Selected Areas in Communications*, 36(11):2549–2560, 2018.

[33] Eugenio Pasqua. Satellite IoT connectivity: Three key developments to drive the market size beyond USD 1 billion. https://iot-analytics.com/satellite-iot-connectivity/, Aug 2022.

[34] Deyi Peng, Dongxuan He, Yun Li, and Zhaocheng Wang. Integrating terrestrial and satellite multibeam systems toward 6G: Techniques and challenges for interference mitigation. *IEEE Wireless Communications*, 29(1):24–31, 2022.

[35] Jason Rainbow. Spanish startup to upgrade tiny satellites to take on Global IOT Market. https://tinyurl.com/4vkm6r6z, Feb 2023.

[36] Microsoft Research. CosmicBeats-Simulator: A space simulation platform that caters to individuals with diverse research interests, including networking, AI, computing, and more. Unlike traditional simulators tied to specific research applications, our design allows for seamless integration of various space-related research verticals. https://github.com/microsoft/CosmicBeats-Simulator, 2023.

[37] François Rottenberg, Ming-Chun Lee, Thomas Choi, Jianzhong Zhang, and Andreas F Molisch. Robust non-coherent beamforming for FDD downlink massive MIMO. In *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*, pages 1–5. IEEE, 2020.

[38] Muhammad Osama Shahid, Millan Philipose, Krishna Chintalapudi, Suman Banerjee, and Bhuvana Krishnaswamy. Concurrent interference cancellation: Decoding multi-packet collisions in LoRa. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 503–515, 2021.

[39] Shree Krishna Sharma, Symeon Chatzinotas, and Björn Ottersten. Satellite cognitive communications: Interference modeling and techniques selection. In *2012 6th Advanced Satellite Multimedia Systems Conference (ASMS) and 12th Signal Processing for Space Communications Workshop (SPSC)*, pages 111–118. IEEE, 2012.

[40] Shree Krishna Sharma, SYMEON Chatzinotas, and BJORN Ottersten. Cognitive radio techniques for satellite communication systems. In *2013 IEEE 78th vehicular technology conference (VTC Fall)*, pages 1–5. IEEE, 2013.

[41] Shree Krishna Sharma, Symeon Chatzinotas, and Björn Ottersten. In-line interference mitigation techniques for spectral coexistence of GEO and NGEO satellites. *International Journal of Satellite Communications and Networking*, 34(1):11–39, 2016.

[42] Marvin K Simon and Dariush Divsalar. Doppler-corrected differential detection of MPSK. *IEEE transactions on communications*, 37(2):99–109, 1989.

[43] Swarm Technologies Inc. Application for Mobile Satellite Service. https://fcc.report/IBFS/SAT-LOA-20181221-00094, 2018.

[44] Shigenori Tani, Katsuyuki Motoyoshi, Hiroyasu Sano, Atsushi Okamura, Hiroki Nishiyama, and Nei Kato. An adaptive beam control technique for Q band satellite to maximize diversity gain and mitigate interference to terrestrial networks. *IEEE Transactions on Emerging Topics in Computing*, 7(1):115–122, 2016.

[45] Shuai Tong, Jiliang Wang, and Yunhao Liu. Combating packet collisions using non-stationary signal scaling in LPWANs. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 234–246, 2020.

[46] Shuai Tong, Zhenqiang Xu, and Jiliang Wang. Colora: Enabling multi-packet reception in lora. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 2303–2311. IEEE, 2020.

[47] David A Vallado and Paul J Cefola. Two-line element sets-practice and use. In *63rd International Astronautical Congress, Naples, Italy*, pages 1–14, 2012.

[48] Deepak Vasisht, Jayanth Shenoy, and Ranveer Chandra. L2D2: Low Latency Distributed Downlink for LEO Satellites. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 151–164, New York, NY, USA, 2021. Association for Computing Machinery.

[49] Enric Vilar and John Austin. Analysis and correction techniques of doppler shift for non-geosynchronous communication satellites. *International journal of satellite communications*, 9(2):123–136, 1991.

[50] Quang-Doanh Vu, Le-Nam Tran, and Markku Juntti. Distributed noncoherent transmit beamforming for dense small cell networks. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4599–4603. IEEE, 2019.

[51] Quang-Doanh Vu, Le-Nam Tran, and Markku Juntti. Noncoherent joint transmission beamforming for dense small cell networks: Global optimality, efficient solution and distributed implementation. *IEEE Transactions on Wireless Communications*, 19(9):5891–5907, 2020.

[52] Xiong Wang, Linghe Kong, Liang He, and Guihai Chen. mlora: A multi-packet reception protocol in lora networks. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2019.

[53] Xianjin Xia, Yuanqing Zheng, and Tao Gu. FTrack: Parallel decoding for LoRa transmissions. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, pages 192–204, 2019.

[54] Liang Yin, Ruonan Yang, Yuanzhou Yang, Li Deng, and Shufang Li. Beam pointing optimization based downlink interference mitigation technique between NGSO satellite systems. *IEEE wireless communications letters*, 10(11):2388–2392, 2021.

[55] Moon-Hee You, Seong-Pal Lee, and Youngyearl Han. Adaptive compensation method using the prediction algorithm for the Doppler frequency shift in the LEO mobile satellite communication system. *ETRI journal*, 22(4):32–39, 2000.

[56] Li Zhen, Hao Qin, Bin Song, Rui Ding, Xiaojiang Du, and Mohsen Guizani. Random access preamble design and detection for mobile satellite communication systems. *IEEE Journal on Selected Areas in Communications*, 36(2):280–291, 2018.

[57] Li Zhen, Yukun Zhang, Keping Yu, Neeraj Kumar, Ahmed Barnawi, and Yongbin Xie. Early collision detection for massive random access in satellite-based internet of things. *IEEE Transactions on Vehicular Technology*, 70(5):5184–5189, 2021.

[58] Lizhong Zheng and David N. C. Tse. Communication on the Grassmann manifold: A geometric approach to the noncoherent multiple-antenna channel. *IEEE transactions on Information Theory*, 48(2):359–383, 2002.

[59] Xiangming Zhu, Chunxiao Jiang, Linling Kuang, Ning Ge, and Jianhua Lu. Non-orthogonal multiple access based integrated terrestrial-satellite networks. *IEEE Journal on Selected Areas in Communications*, 35(10):2253–2267, 2017.

A Simulation of Orbital Dynamics

Throughout the paper, particularly in Section 2, we incorporate various statistical data related to the orbital dynamics of a LEO IoT satellite constellation. This data includes information on the satellite's footprint coverage, elevation angles, and more. To generate these statistics, we utilize the CosmicBeats Simulator [36] provided by Microsoft Research. We configure the simulator with the locations of our ground stations and the TLEs for the complete Swarm constellation.

B Proof of Lemma 3.1

Lemma: *Consider $a_1, a_2, \ldots, a_n$ to be complex numbers on the unit circle with random phase. That is, $a_i = e^{j\theta_i}$ where θ_i is uniformly sampled from $[0, 2\pi]$. Let s_n be defined as $s_n = \sum_{i=1}^{n} a_i$. Then we have $\mathbb{E}\left[\|s_n\|^2\right] = O(n)$.*

Proof: s_n is the sum of n random phasors on the unit circle with phase sampled uniformly from $[0, 2\pi]$. By leveraging the commutative property of addition of complex numbers we have

$$s_n = s_{n-1} + a_n \tag{8}$$

We know that a_n has magnitude 1 and has uniform random phase in $[0, 2\pi]$. Hence, the phase difference between the complex numbers a_n and s_{n-1} can also be treated as uniformly distributed in $[0, 2\pi]$. Let the phase difference between a_n and s_{n-1} be denoted by ϕ_n. So from a simple inner product, we have

$$\|s_n\|^2 = \|s_{n-1}\|^2 + \|a_n\|^2 + 2\ cos(\phi_n)\ \|a_n\|\ \|s_{n-1}\| \tag{9}$$

$$\|s_n\|^2 = \|s_{n-1}\|^2 + 1 + 2\ cos(\phi_n)\ \|s_{n-1}\| \tag{10}$$

$$\mathbb{E}\left[\|s_n\|^2\right] = \frac{1}{2\pi}\int_0^{2\pi} \Bigg(\mathbb{E}\left[\|s_{n-1}\|^2\right] + 1 + 2\ cos(\phi_n)\ \mathbb{E}\left[\|s_{n-1}\|\right] \Bigg) d\phi \tag{11}$$

$$\mathbb{E}\left[\|s_n\|^2\right] = \mathbb{E}\left[\|s_{n-1}\|^2\right] + 1 \tag{12}$$

We know that $s_1 = 1$. Hence, by induction we can see that

$$\mathbb{E}\left[\|s_n\|^2\right] = O(n) \tag{13}$$

The key takeaway from the above lemma is that even though we are adding up unit phasors with random phase values in $[0, 2\pi]$, the expected magnitude of their summation grows monotonically with number of terms rather than going to 0. Hence, by leveraging a virtual preamble train in Spectrumize, although we do not see the complete benefits of coherent combination, we still see enough gains that allows our system to detect satellite packets even in very low signal strength conditions.

Application-Level Service Assurance with 5G RAN Slicing

Arjun Balasingam
MIT CSAIL

Manikanta Kotaru
Microsoft

Paramvir Bahl
Microsoft

Abstract

This paper presents Zipper, a novel Radio Access Network (RAN) slicing system that provides assurances of application-level throughput and latency. Existing RAN slicing systems optimize for slice-level assurance, but these methods fail to provide predictable network performance to individual mobile apps. Extending the slice-level formulation to app-level introduces an intractable optimization problem with exploding state and action spaces. To simplify the search space, Zipper casts the problem as a model predictive controller, and explicitly tracks the network dynamics of each user. It uses an efficient algorithm to compute slice bandwidth allocations that meet each app's requirements. To assist operators with interfacing admission control policies, Zipper exposes a primitive that estimates if there is bandwidth available to accommodate an incoming app's requirements.

We implemented Zipper on a production-class 5G virtual RAN testbed integrated with hooks to control slice bandwidth, and we evaluated it on real workloads, including video conferencing and virtual reality apps. On a representative RAN workload, our real-time implementation supports up to 200 apps and over 70 slices on a 100 MHz channel. Relative to a slice-level service assurance system, Zipper reduces tail throughput and latency violations, measured as a ratio of violation of the app's request, by 9×.

1 Introduction

A rapidly growing number of mobile applications—such as mixed reality, cloud gaming, video conferencing, and cloud robotics—require predictable network connectivity (i.e., throughput and latency). The 3GPP specifications for 5G Radio Access Networks (RANs) recognized this requirement for next-generation mobile apps and introduced *network slicing* [20], a virtualization primitive that allows an operator to run multiple differentiated virtual networks, called slices, atop a single physical network. A slice can support a set of users or a set of applications[1] with similar connectivity requirements. It can span multiple network domains, including the radio access network (RAN) [15, 24, 41], core [46, 68], transport [59] and fronthaul [8]. Operators can distribute resources, like physical resource blocks (PRBs) in the RAN, amongst the slices to provide differentiated connectivity. RAN slicing is of particular interest for service assurance [13] since the last-mile wireless link is often the bottleneck for mobile apps [4, 6].

Existing approaches [7, 15, 24, 41, 72] allocate PRBs to different slices to guarantee slice-level service assurance, e.g., through service-level agreements (SLAs) for total slice throughput. However, in order to realize the vision of network slicing, where apps achieve the network performance that they require, the service assurance should be provided at application-level. Existing approaches fall short of enabling operators to provide this important capability. Slice-level service assurance does not guarantee throughput and latency to *each app* in the slice, since different users in the same slice can experience wildly different channel conditions, as we explain in §3. We need app-level service assurance in order to meet the requirements of each app within a slice. However, two key challenges arise when optimizing for app-level service assurance:

Challenge #1: Search space complexity. Prior approaches [7, 24, 41, 72] provide slice-level service assurance by tracking a state space consisting of aggregate slice-level statistics, including the average channel quality of all users in a slice, the observed slice throughput, etc. To extend these slice-level methods to support app-level requirements, one could potentially expand the state space to track the channel quality, the observed throughput, and the observed latency experienced by each app in a slice, essentially treating each app as a slice for the purposes of service assurance. However, the state space, consisting of all possible values that the tracked variables can take, grows exponentially in terms of the number of tracked variables. Thus, treating each app as a slice grows the state space exponentially in the number of apps rather than in the number of slices. Further, the control policy involves searching through this state space to determine an allocation of PRBs to slices that complies with the SLA constraints. This results in an intractable optimization problem for practical deployments, where each

[1] By "app", we refer to a single mobile application. "User" refers to a mobile device, which can run multiple apps.

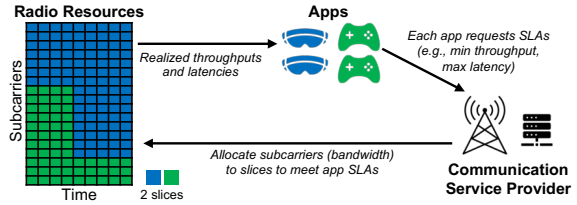


Figure 1: Apps express their connectivity requirements in terms of SLAs, and the operator provisions slice bandwidths to fulfill all SLAs.

slice accommodates tens to hundreds of apps (§3.2).

Challenge #2: Determining resource availability. To compute slice bandwidth allocations within the total available bandwidth, operators typically run admission controllers that admit or reject incoming apps according to a policy that depends on slice monetization preferences, fairness constraints, and other objectives. Algorithms for admission control have been studied widely [9, 10, 52, 62] and are not the focus of this paper. However, operators need a way to determine if the RAN has resources to accommodate the SLAs of an incoming app, in addition to the apps already admitted. We cannot adapt prior approaches [29, 37, 40], which compute required PRBs to support slice-level SLAs, because the state-space complexity precludes treating each app as a slice (§3.3).

This paper presents *Zipper*, a real-time RAN slicing system that dynamically allocates PRBs (i.e., bandwidth) to network slices to ensure app-level throughput and latency SLAs for every app in every slice.[2] As illustrated in Fig. 1, under this model, apps express their network requirements to the operator in the form of SLAs, i.e., minimum throughput and maximum latency. The operator then fulfills these SLAs over the shared wireless medium by bundling apps into slices, and then computing and allocating the PRBs required by each slice. This paradigm of operators provisioning connectivity, so that each app meets its desired network requirements, is similar to the familiar model of cloud computing—where the developer requests a combination of compute, memory, and I/O bandwidth for a particular workload, and the cloud service provider finds the right allocation of resources to reliably deliver the desired performance.

Zipper addresses the challenges in enabling app-level service assurance via three contributions:

- To manage the search space complexity, we decouple the network model and the control policy by formulating SLA-compliant PRB allocation as a **model predictive control (MPC)** problem. Zipper uses standalone predictors to forecast each of the tracked state space variables, such as the wireless channel experienced by each app. It then feeds these predictions into a control algorithm that computes a sequence of future bandwidths for each slice based on the predicted state. Our insight is that Zipper does not need to enumerate different future states within the state space, by using the well-known MPC framework (§4.1).
- We propose an **efficient control algorithm** to allocate PRBs (i.e., bandwidth) amongst the slices. Zipper efficiently prunes the search space of possible PRB allocations using the insight that app throughput and latency vary monotonically with the number of PRBs (§4.2).
- We **forecast RAN resource availability**, guided by the following question: for an incoming app A, does the RAN have enough PRBs to admit A, given the other apps already admitted? Naively applying Zipper's bandwidth estimation algorithm for a distribution of possible channel conditions experienced by the app resulted in prohibitive estimation times. We instead design a family of deep neural networks (DNNs) to predict the distribution of required PRBs. We train these neural networks on simulations of Zipper's control algorithm offline and then apply them to predict the resource availability in real time (§4.3).

We design an O-RAN-compatible [34] architecture to realize these algorithmic concepts (Fig. 4). We have implemented Zipper atop a production-class end-to-end 5G vRAN platform, implementing hooks [25] across different modules in vRAN Distributed Unit (DU) to control slice bandwidth dynamically without compromising real-time performance (§5). On a typical RAN workload consisting of video streaming, conferencing, IoT, and virtual reality apps, our real-time system can support up to 200 apps and over 70 slices on a 100 MHz channel. We find that Zipper outperforms prior schedulers and slicing frameworks (§6); relative to a slice-level service assurance scheduler [41], Zipper reduces SLA violations, measured as a ratio of the violation of the app's request, by 9×.

2 Related Work

RAN slicing. While a static allocation of PRBs to slices provides traffic isolation and simplifies radio resource management [57], it does not guarantee reliable slice performance, since throughput and latency vary with dynamic wireless channel conditions. Orion [24] and SCOPE [7] are slicing-capable RAN virtualization frameworks, and implement existing slice bandwidth schedulers like Network Virtualization Substrate (NVS) [41]. NVS, designed originally for WiMAX, allocates PRBs among slices to deliver a target aggregate slice throughput, assuming invariant Modulation and Coding Scheme (MCS) conditions. However, RAN operators adjust MCS according to time-varying channel conditions. Slice-level service assurance is also the primary focus of many other RAN slicing proposals [3, 14, 17, 45, 55].

RadioSaber [15], a recent RAN slicing system, allocates PRBs to slices in a channel-aware manner, by extending NVS to query each slice's Medium Access Control (MAC) scheduler and find the PRBs that are best-suited for each user's channels. RadioSaber is complementary to Zipper: it focuses on slice-level throughput assurance for enterprise slices, while Zipper is designed for app-level SLAs. Future work includes incorporating RadioSaber's techniques for channel-aware PRB allocation into Zipper.

LACO [72] proposes a reinforcement learning-based

[2] We focus on app-level, but our solution also generalizes to the user-level.

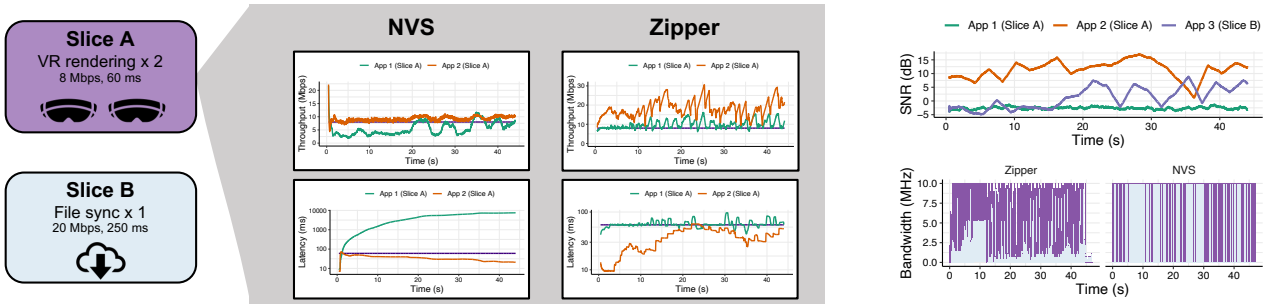


Figure 2: Zipper can efficiently manage an expressive and comprehensive state space to deliver SLAs for each app in each slice.

framework to provide latency guarantees in multi-tenancy environments by minimizing the number of bits missing the specified latency tolerance. By contrast, this paper presents a novel framework that tailors radio resource schedules for applications SLAs such as throughput and latency in the presence of dynamic wireless channel conditions.

Admission control. Admission control proposals for RAN slicing cover traffic prediction [58], load balancing [9], pricing [52,62], and game-theoretic formulations [10]. Zipper does not propose a new method for admission control; instead, we recognize that, to use Zipper on production networks, operators need to know if Zipper's slice controller has resources to accommodate the SLAs of an incoming app. Typical approaches to assess resource availability [29,40] are not compatible with Zipper's app-level formulation; we elaborate in §3.3. Recent 5G network slicing proposals [15,24,72] do not address how to estimate resource availability, which is vital for operators to use these systems in practice.

RAN virtualization. Virtual RANs serve multiple logical RANs from the same physical hardware. They have garnered significant attention [18,21,65], with a number of proposals across different compute platforms, including CPUs [21,27,61,67], DSPs [5] and GPUs [50]. Zipper leverages this body of work to dynamically allocate RAN resources among different virtual resources and expose them to the network slices.

Scheduling. Efficient RAN utilization is a key principle of mobile network design [30,32]. Canonical algorithms, such as proportionally-fair [63], round-robin, and priority-based [71] schedulers, only consider aggregate throughput and fairness.

3 Problem Setup and Challenges

In this section, we formalize the optimization problem of providing app-level throughput and latency assurance, and illustrate, through a toy example, the challenges in computing slice bandwidth allocations efficiently.

3.1 Problem formulation

Zipper allocates slice bandwidths to meet app-level throughput and latency SLAs, while (i) each app's wireless channel quality fluctuates and (ii) apps join and leave the network asynchronously. We assume that the operator configures its RAN with a set of slices, catering to different traffic types (e.g., cloud gaming, video streaming, etc.) and to different enterprise policies (e.g., separate slices for Zoom and Microsoft Teams sessions). The operator configures each each slice with a particular MAC scheduler, which is responsible for allocating PRBs to apps in each slice. Fig. 2 illustrates a RAN serving two slices: slice A for VR remote rendering and slice B for video downloads (e.g., video editing).

Formalizing SLAs. We assume that each app selects a slice, based on its specific throughput and latency requirements.[3] For example, in Fig. 2, the two VR apps each require a minimum throughput of 8 Mbps and a worst-case latency of 60 ms, while the file sync app requests a minimum throughput of 20 Mbps and a worst-case latency of 250 ms. Let x_a^{SLA} and d_a^{SLA} denote the throughput and latency SLAs for an app a.

Let $\bar{x}_a(t)$ be the average throughput over a moving window of T_w slots. App a requires that $\bar{x}_a(t) \geq x_a^{SLA}$. Similarly, let $\bar{d}_a(t)$ be the average latency over T_w. When app a expresses a latency SLA, it requires that the average latency $\bar{d}_a(t) \leq d_a^{SLA}$.

Formalizing slice bandwidth allocation. We formalize the optimization problem to compute SLA-compliant schedules. Since there can be multiple valid allocations that satisfy the SLAs, we choose the one that minimizes the total bandwidth:

$$\underset{\mathcal{S}_s(t),B_s(t)\,\forall s\in S\,\forall t}{\text{argmin}} \quad \sum_t \sum_{s\in S} B_s(t) \tag{1}$$

$$\text{s.t.} \quad \sum_{s\in S} B_s(t) \leq B \qquad \forall t \tag{2}$$

$$\bar{x}_a(t) \geq x_a^{SLA} \qquad \forall a\in A_s\ \forall s\in S\ \forall t \tag{3}$$

$$\bar{d}_a(t) \leq d_a^{SLA} \qquad \forall a\in A_s\ \forall s\in S\ \forall t, \tag{4}$$

where B is the total bandwidth available at the base station, S is the set of network slices, and $A_s(t)$ is the set of apps subscribed to slice $s\in S$ at time t. $B_s(t)$ denotes the bandwidth allocated to slice s in scheduling round t. $\mathcal{S}_s$ is the MAC schedule for slice s.

At each timestep t, Zipper must select $B_s(t)$ for each slice $s\in S$ such that the throughput and latency SLAs for all apps $a \in A_s(t)$ are satisfied, as captured by the constraints in Eqn. 3 and Eqn. 4 respectively. Eqn. 2 ensures that the sum of slice bandwidths does not exceed the bandwidth available

[3] Alternatively, the slice controller can automatically match the app to a slice already catering to apps with similar connectivity requirements. We leave app-to-slice matching to future work.

at the base station. The objective in Eqn. 1 states that Zipper must find the sequence of slice schedules and corresponding bandwidths that minimizes the overall spectral utilization.

This approach differs from previous approaches that compute the minimum bandwidth required by each slice to satisfy slice-level SLAs, such as average slice throughput. For example, Fig. 2 visualizes the results achieved by NVS [41], a widely-used slice-level service assurance system [7, 19, 24], for our toy example with two slices. Notice that NVS is *not* able to meet the throughput for App 1, and instead overcompensates for App 2. However, directly extending slice-level service assurance approaches to satisfy app-level SLAs explodes the state space.

Sometimes, the network could be at capacity, and the formulation in Eqn. 1-Eqn. 4 will not have a valid solution. To make the problem tractable, we can relax the constrains in Eqn. 3 and Eqn. 4 into two penalty functions that quantify how far—if at all—an app deviates from its throughput and latency SLAs:

$$f_x^a(t) = \left|\min\left(\bar{x}_a(t) - x_a^{SLA}, 0\right)/x_a^{SLA}\right| \quad (5)$$

$$f_d^a(t) = \left|\min\left(d_a^{SLA} - \bar{d}_a(t), 0\right)/d_a^{SLA}\right| \quad (6)$$

$f_x^a(t)$ in Eqn. 5 is nonzero only when the throughput is less than the SLA and measures the deviation as a fraction of the SLA. Similarly, $f_d^a(t)$ measures the deviation as a fraction of agreed-upon latency SLA, if that SLA is violated. So, we modify the objective of Zipper (Eqn. 1) to include a term that minimizes these penalties. In practice, penalties will remain to close to 0 most of the time, since operators admit/reject incoming apps by determining whether the RAN has sufficient capacity.

3.2 Challenge: state space complexity

Prior methods [41, 72] monitor aggregate state variables like average slice throughput [41], average channel quality across all users, and average latency across all users [72], to deliver service assurance at the slice level. The search space for such state vectors grows exponentially with the number of slices. Fig. 2 shows that considering a slice level state space could yield poor app performance. While the slice-level method meets the overall slice throughput SLAs, it violates App 1's SLA because App 1 has an inferior channel quality to that of App 2.

We could expand the state space by considering app-level characteristics, e.g., average measured app throughput, average measured app latency, channel quality of each user, etc. We could then extend slice-level service assurance approaches to meet app-level SLAs, treating each app as an individual slice for the sake of service assurance. However, the state space grows exponentially with the number of apps, rather than with the number of slices. The number of apps served by each slice in a base station could range from tens to hundreds, resulting in an intractable state space to deliver real-time performance. For example, LACO [72] trains an agent using reinforcement learning to learn a policy that selects slice bandwidths. If we adapt that architecture to a fine-grained state space, the training complexity explodes, since the agent needs to explore a more expansive search space.

3.3 Challenge: determining RAN resource availability

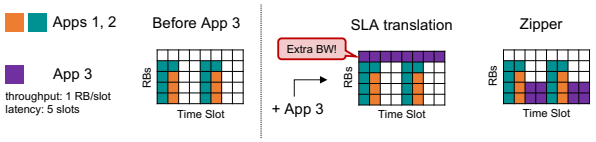


Figure 3: Translating an app's SLAs directly to required slice bandwidth can ignore schedules with greater spectral efficiency.

Recall that the slice controller's bandwidth allocations B_s cannot exceed the total available bandwidth B. As load at the RAN increases, it becomes more challenging to fulfill all SLAs under this bandwidth constraint. As a result, operators typically run admission controllers on top of their slicing systems, only admitting apps that can receive the requested SLAs. Admission control policies can depend on a variety of objectives, such as slice monetization preferences, operational costs, fairness, energy constraints, etc. Admission control for network slicing has received significant attention over the years [9, 10, 52, 62], and is not the focus of this paper. However, in order to interface the slice controller with a particular policy, we need a mechanism [52] that answers the following question: *for a pre-determined time period (i.e., contract duration), can the RAN fulfill the SLAs for an incoming app without compromising on commitments made to existing apps?*

NVS adds a buffer to each slice [41] to absorb errors that operators make in admitting apps that it cannot support. However, a constant buffer can underutilize the spectrum. Prior work [37] injects the incoming app into a separate "best effort slice" and observes whether it achieves its SLAs to determine if the RAN has resources in the desired slice to accommodate this app. However, performance in the "best effort" slice may not faithfully represent performance in the target slice. A more analytical approach [29, 40] is to translate the SLAs for the incoming app into a measure of resource blocks required to support that app in the desired slice via an analytical model or a lookup table that maps SLAs to a PRB requirement. These methods can waste spectrum.

Consider the example in Fig. 3, where a slice in the RAN initially serves Apps 1 and 2. We show the resource block schedule for these two apps; notice that the RAN allocates 4 RBs of bandwidth to the slice. Our goal is to determine how much bandwidth is required to accommodate App 3, who has a throughput SLA of 1 RB/slot and a latency SLA of 5 slots. Simply translating the 1 RB/slot throughput SLA for App 3 to RB overhead, would lead us to allocate an extra RB of bandwidth to the slice. Zipper, by contrast, accommodates App 3—along with Apps 1 and 2—without adding any more bandwidth.

SLAs are two-dimensional (i.e., throughput and latency), and a slice could have an arbitrarily complex PRB scheduler, whose behavior depends on additional factors, such as the status of app queues at the base station and changing MCS in response to the wireless channel. It is therefore challenging

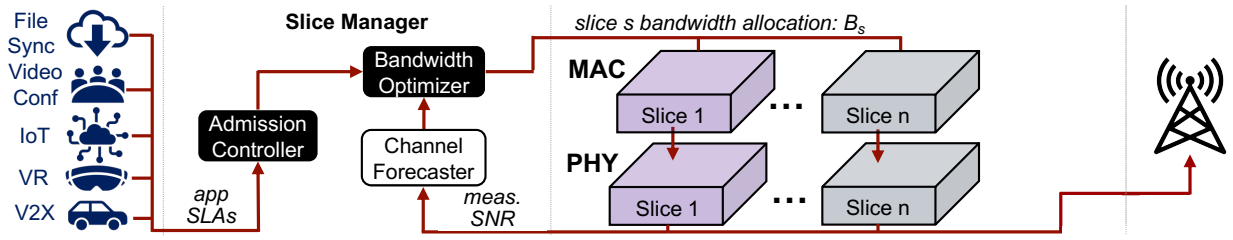


Figure 4: Zipper provisions connectivity by dynamically optimizing network slice bandwidth and resource allocation to meet app-level SLAs.

to map SLAs to a PRB differential via an analytical model.

We need a primitive to determine RAN resource availability for an incoming app that generalizes to MAC schedulers and apps with different demand patterns. Recent RAN slicing systems [15, 24] do not address how to interface their slice controllers with operators' admission control policies. Since the RAN is often the bottleneck link [4, 6], it is often oversubscribed. Thus, slicing systems are unusable in practice without a mechanism to estimate resource availability and an accompanying admission control policy.

4 Design

In this section, we describe how we design Zipper, illustrated in Fig. 4 to enable app-level service assurance. Zipper consists of a model predictive control (MPC) framework to manage the search space complexity (§4.1), uses an efficient algorithm to compute slice bandwidth allocations within this MPC formulation (§4.2), and exposes a primitive to help operators forecast RAN resource availability (§4.3).

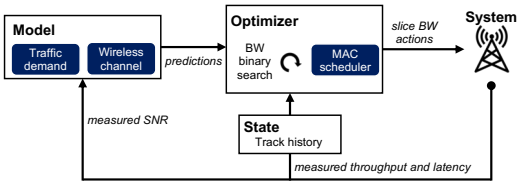


Figure 5: Zipper uses model predictive control (MPC) to compute slice bandwidths that comply with all app SLAs. With MPC, Zipper decouples prediction from control to manage the state space.

4.1 Model predictive control

MPC [26] is a framework to solve sequential decision making problems over a moving look-ahead horizon. It decouples a controller, which solves a classical optimization problem, from a predictor, which explicitly models uncertainty in the environment. MPC has proven practical in a number of real-world control problems, including in adaptive bitrate selection for video streaming [66, 70] and in robotics [64].

Fig. 5 illustrates how Zipper applies MPC to solve the optimization problem formulated in Equations 1-4. The state space consists of (i) the average throughput and average latency experienced by each app over the past T_w slots, (ii) the average signal-to-noise ratio (SNR) of each user over T_w as a measure of the channel quality, and (iii) the incoming data traffic. The action space consists of a bandwidth allocation B_s for each slice s. Independent forecasters predict how each of the state space variables evolves over a short term planning horizon. The controller uses these predictions to determine the bandwidth schedule $B_s(t)$ for each slice.

MPC allows us to use network models to explicitly predict the future states over the short term, and thus avoid searching over different future states within the state space. We describe predictive models for each of the state space variables ahead.

Forecasting the wireless channel. Zipper supports different channel predictors that forecast how each app's wireless channel (i.e., SNR)[4] will evolve over the near term. Forecasting the wireless channel is a well-researched and fundamentally challenging problem [38, 44, 47, 49]. We acknowledge this in our design, and do not aim for a perfect predictor. Instead, we quantify a desirable performance for our bandwidth allocation task, and then propose methods that meet that target.

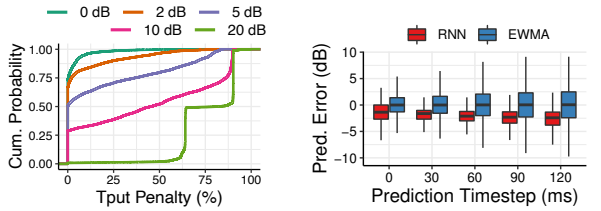
To understand the impact of SNR prediction error on Zipper's ability to meet SLAs, we run a simple experiment,[5] involving a 40 MHz channel and 10 video conferencing apps (i.e., 2 Mbps min throughput, 150 ms max latency), split across 2 slices. We randomly assign each user a 30-second SNR trace gathered on a production 5G network. To understand the effect of prediction error in the worst case, we introduce a dummy predictor that simply returns the ground truth SNR value added to some constant prediction error. Fig. 6a visualizes the results as CDFs of the throughput and latency penalties ($f_x^a(t)$ and $f_x^a(t)$ in Eqn. 5 and Eqn. 6 respectively) for different prediction errors. As expected, larger prediction errors lead to higher penalties. However, notice that a small (but not insignificant) error of 2 dB has a modest impact on penalty.

We also observe that the MAC scheduler uses the SNR forecast to determine what MCS to assign an app. The 3GPP standards define 32 MCS values [35], and MAC schedulers use a lookup table to map measured channel quality to MCS [39]. We find that this table is quantized in 2 dB steps, so a prediction error of under 2 dB may still yield the correct MCS. The results from Fig. 6a and this insight about MCS quantization show why MPC is resilient to modeling error in the context of Zipper's scheduling problem.

To forecast each user's channel, we train a sequence-to-

[4] For simplicity, we forecast SNR as an aggregate quantity over all subcarriers in a given time slot. However, Zipper can support richer predictors and schedulers [15] predict SNR at a subcarrier granularity.

[5] We evaluate Zipper more extensively against an Oracle policy in §6.



(a) Resilience to SNR error. (b) SNR prediction error.

Figure 6: Zipper is resilient to modest ∼2 dB error in forecasting SNR. Its MPC framework supports different channel forecasters. While both have small median errors, the RNN model outperforms EWMA.

sequence Recurrent Neural Network (RNN) [56], which uses an input sequence of SNR measurements over the last 1 second to predict a sequence of SNR measurements over the next 150 milliseconds. App. A.1 describes the architecture of this RNN. We train this model using a dataset of SNR traces [54] collected over a commercial network at scale. We evaluate the accuracy of this RNN on a holdout set from the same bank of traces. Fig. 6b shows the prediction error (relative to the ground truth) at different points over the 150 ms prediction horizon. Each boxplot shows a distribution of error over all traces in our holdout set. We compare the accuracy of our RNN against a simpler predictor that tracks the SNR with an exponentially-weighted moving average (EWMA). Both predictors have a suitable median performance, which falls within our target of ∼2 dB error. However, the RNN has a more consistently tight distribution. Moreover, as we describe below, prediction errors at later timesteps are less consequential, since Zipper recomputes fresh allocations at a finer granularity of 50 ms. Although we use this RNN model in our implementation, Zipper could use any other predictor—including the EWMA filter—with comparable error.

Other state space variables. Zipper tracks the average throughput and latency, $\bar{x}_a(t)$ and $\bar{d}_a(t)$ respectively, experienced by each app in the past T_w time slots. Since Zipper only tracks historical averages, there is no need for prediction. We assume that the traffic demand for each app follows the throughput SLA requested by the app. If the traffic demand from an app is higher than the agreed upon throughput SLA, Zipper only ensures that it fulfills the negotiated SLAs. Tuning slice bandwidths using more detailed traffic demand predictions is part of future work.

4.2 Tuning slice bandwidths efficiently

Given the state space, defined as each app's SNR, average throughput, average latency, and traffic demand, the slice manager must find the most spectrally-efficient slice bandwidth allocation that satisfies the SLAs, as we formalize in §3.1.

One approach to is to analytically derive a function that maps SLAs to a valid slice bandwidth. However, this is challenging, since the expected throughput and latency of any given app depends not only that app's channel quality and queue status at the base station, but also on the characteristics of the other

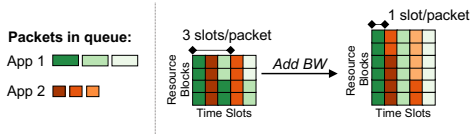


Figure 7: Exposing more bandwidth to a slice reduces packet latency.

apps contending for the same radio resources in the slice.

Monotonicity. Our insight is that both app throughput and app latency are *monotonic functions of slice bandwidth*. Fig. 7 illustrates this monotonicity property for latency with an example. Consider two apps (green and orange) with different packet sizes, and a round-robin MAC scheduler. We define latency as the difference between the times at which (i) the first byte of a packet arrives at the base station and (ii) the last byte of a packet is sent over the air. The diagram on the left visualizes a round-robin schedule for a slice with 4 resource blocks. Notice that the green app's packet is spread across multiple slots, and since the MAC is round-robin, the packet's latency is at least 3 slots. By contrast, when the bandwidth is 6 resource blocks (diagram on the right), the packet latency is just 1 slot. Thus, per-packet latency is a monotonically-decreasing function of slice bandwidth. Similarly, the app throughput increases monotonically with slice bandwidth. App. A.2 elaborates on this property.

Because app throughput and latency vary monotonically with slice bandwidth, there exists a minimum bandwidth B_s for $s \in S$ that satisfies all SLAs. Therefore, a solution that minimizes Eqn. 1 is one that minimizes the individual slice bandwidths, and Zipper can optimize each slice independently.

Computing bandwidths. Zipper treats slice MAC schedulers as a blackbox; the search algorithm does not need to know the scheduling logic. Instead Zipper uses each slice's scheduler as a building block to find the smallest bandwidth that satisfies the SLAs of all apps in the slice. In each time interval t, Zipper computes $B_s(t)$ using an iterative algorithm that *simulates* the MAC scheduler for different candidate bandwidths $\tilde{B}_s$. Zipper queries the MAC scheduler for each $\tilde{B}_s$. Zipper supplies the MAC scheduler with (i) all outstanding packets in each app's queue, and (ii) a forecast of each app's channel quality over the scheduling horizon (§4.1). Zipper evaluates the resulting schedule $\tilde{S}_s$ by computing the *penalty scores*, $f_x^a(t)$ and $f_d^a(t)$ as defined in Eqn. 5 and Eqn. 6, respectively, to determine if the schedule satisfies the SLAs. Then, amongst the set of valid schedules (i.e., when $f_x^a(t) = f_d^a(t) = 0$), Zipper chooses the one that requires the least slice bandwidth, i.e., the smallest $\tilde{B}_s$.

Zipper navigates the search space of candidate bandwidths using binary search. It starts with the entire range $0 \leq \tilde{B}_s \leq B$, and prunes the search space by half in each iteration. In the first iteration, Zipper computes a MAC schedule $\tilde{S}$ for the allocation $B/2$. For instance, if at least one app's throughput in $\tilde{S}$ is less than the throughput agreed to in the SLA, then Zipper determines that the slice needs more bandwidth to satisfy the constraint; so it continues the search in the range $(B/2, B]$. By contrast, if all performance metrics comply with the SLA constraints, then Zipper determines that the slice could possibly

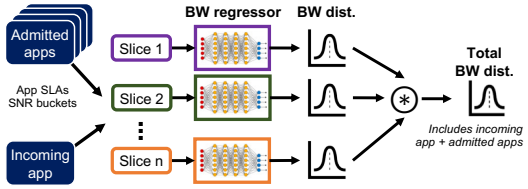


Figure 8: Zipper builds a family of DNNs that forecasts bandwidth distributions for slices consisting of different MAC schedulers and apps with different demand patterns.

meet the SLAs with less bandwidth; so it continues the search in the range $[0,B/2)$. We can apply this binary search optimization because app throughput and latency vary monontonically with slice bandwidth.

Zipper computes schedules in a cascading manner, where, in each timestep t, Zipper solves the MPC problem for a finite future horizon of T_h. It recomputes its allocation every $T_e \leq T_h$ in order to incorporate recent snapshots of user channel and app queue statuses. We use $T_h = 150$ ms and $T_e = 50$ ms[6] to ensure that Zipper is reactive but not myopic. If $B_s(t)$ violates Eqn. 2 for any slice s, Zipper resolves the conflict to ensure that the allocated bandwidth does not exceed the capacity.

Resolving conflicts. Since Zipper tunes slice bandwidths independently, the allocations could conflict, i.e., $\sum_{s\in S} B_s(t) > B$, which violates Eqn. 2. To resolve conflicts, Zipper deducts—from each slice—the excess bandwidth in proportion to the share of bandwidth that each slice was originally allocated. In practice, Zipper will not trigger this step often because it gates incoming requests with an admission controller (§4.3).

App. A.3 provides pseudocode for how Zipper computes slice bandwidth allocations. Note that Zipper only considers the number of resource blocks in aggregate when allocating resource blocks to a slice. In future work, we can extend Zipper to determine the most suitable set of resource blocks given the channel conditions, using the techniques proposed by RadioSaber [15]. We can also model methods to increase user capacity, such as beam steering [11, 28].

4.3 Forecasting RAN resource availability

To estimate if the RAN has resources to support an incoming app, Zipper answers the following question: for the contract duration, does the RAN have enough PRBs to accommodate the incoming app and to fulfill SLAs for all other admitted apps?

Predicting bandwidth statistics. Zipper estimates the distribution of bandwidths, i.e., number of PRBs, that each slice will require over a predetermined contract duration—including the incoming app. Translating the SLAs of each app in a slice to radio resource requirements [29, 40] can yield overestimates of the required bandwidth. Instead, Zipper simulates its slice manager over thousands of channel traces. Direct simulations capture how the slice MAC exploits statistical multiplexing to fulfill the SLAs without adding bandwidth to a slice (§3.3). However, running thousands of simulations for a reasonable contract duration (e.g., 5 mins) is expensive, since the slice manager computes and evaluates many MAC schedules.

All we need from the simulations is the bandwidth statistics—not the PRB schedules. To approximate the bandwidth statistics, Zipper develops a family of deep neural networks (DNNs), instead of running thousands of micro-simulations at runtime. Fig. 8 illustrates the design of this module. Since each slice caters to apps with similar network requirements (i.e., SLAs), we tailor a DNN for the traffic characteristics of each slice, similar to lookup tables in prior work [29, 40] that translate SLAs to PRB requirements. Each DNN treats a slice's MAC scheduler as a blackbox process and learns the nonlinear relationship between inputs—app demand patterns, SLAs, and channel quality—and the required bandwidth.

To create a simple and tractable input embedding for the DNN, we make a few assumptions. First, we assume that all apps in a slice have the same SLAs; this is reasonable because, in practice, network slices often isolate similar kinds of traffic [24]. Moreover, slight variations in SLAs (e.g., 4 Mbps vs. 5 Mbps video conference flows) should have negligible impact on bandwidth requirements. Second, in order to discretize the space of possible SNR values, we assign each app to an SNR bucket from the set {`poor`, `bad`, `good`, `great`}, where each bucket corresponds to a range of SNR values (e.g., -5 dB $\leq$ `bad` $<$ 2 dB).[7] Zipper drops each incoming app into the best effort slice for a brief period (e.g., 5 seconds), computes its median SNR, and assigns it an SNR bucket. For existing apps, Zipper uses SNR measured over the lifetime of each app to determine the most suitable bucket. Note that the best effort slice is only for measuring SNR, not for assessing resource availability. For each slice, Zipper generates a feature embedding consisting of the number of apps in the slice (including the incoming app, if applicable), and the number of apps in each SNR bucket. App. B.1 describes the DNN architecture.

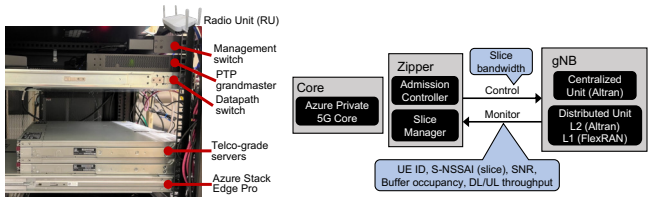
Training DNNs. We generate training data by using Zipper's slice manager as a simulator. We start by enumerating all possible feature embeddings, and run a micro-simulation of Zipper for each embedding using simulated channel traces (assuming a Rayleigh channel model) with SNR values corresponding to the SNR bucket for that embedding. We prune the space of embeddings using some simple heuristics. For instance, if we find that a simulation of 55 apps—all with `poor` SNRs—requires a maximum bandwidth of 100 MHz, we discard all embeddings with 56 or more `poor` apps, since those configurations will also require at least 100 MHz.

Estimating resource availability. Each DNN returns slice bandwidths as a probability distribution P_s for slice s. To compute a distribution of the required spectrum, we convolve[8] the slices' independent probability distributions:

[6] We found that Zipper was not very sensitive to T_h and T_e; we selected $T_e = 50$ ms because our SNR predictors are most accurate over this horizon (§4.1). A shorter horizon allows us to replan with fresh estimates of SNR.

[7] Note that we only discretize SNR into bins to estimate resource availability with the DNNs; at runtime, the slice controller forecasts SNR (§4.1).

[8] The probability distribution of a sum of independent random variables is



(a) Hardware configuration (b) Software stack

Figure 9: We implement Zipper atop a production-class 5G network.

$P = P_1 \circledast P_2 \circledast \cdots \circledast P_n$. P is the forecasted distribution of required bandwidth. Operators can choose a suitable percentile of P (e.g., p95, p99, etc.) based on their tolerance preferences. For instance, a conservative policy might deem resources available for the incoming app if the p99 bandwidth is less than the total available bandwidth B.

5 Implementation

We implemented Zipper atop an end-to-end production-class 5G network built on virtual RAN (vRAN) and Core components. Our testbed uses commercial off-the-shelf user devices.

Testbed hardware. As illustrated in Fig. 9a, our production testbed consists of two 32-core HPE ProLiant DL110 Gen10 Plus Telco-Grade servers with Intel Xeon-Gold 6338N CPUs and an Azure Stack Edge Pro device. The servers perform all baseband processing in software, except for LDPC decoding which is performed by a lookaside accelerator—Intel eASIC ACC100—to meet the stringent real-time requirements posed by baseband processing workloads. We use Foxconn RPQN-7800 5G Open Radio Units (RUs) operating on 100 MHz channels in the n78 band. We obtained FCC STA licenses to operate the radios for experimental purposes. The radio units support the popular O-RAN split Option 7.2x [51], designed to reduce the optical bandwidth required for fronthaul traffic while keeping the RU simple and inexpensive. The radio has four antennas and supports up to four spatial streams. The RU and the Telco servers are synchronized using a Qulsar Qg2 carrier-grade PTP Grandmaster [53]. The testbed additionally includes a high-speed datapath switch Arista 7050 to carry the fronthaul traffic, as well as a management switch from Netgear to facilitate remote management of the hardware devices.

Testbed software. Fig. 9b illustrates the software stack. For the L1, our testbed runs production-ready Intel FlexRAN v20.11 [18]. For the L2/L3, we use Altran 5G vRAN software from Capgemini [2]. A single Telco server hosts both the Altran and FlexRAN software, while Zipper runs on another server. We installed Azure Private 5G Core (AP5GC) [48] on the Azure Stack Edge Pro device to provide 5G core services. Both Altran vRAN and AP5GC core support standard-compliant slicing and can provide differentiated service to commercial devices. We integrated all of these systems end-to-end to realize a production-class 5G network using virtualized RAN and Core components.

Zipper can programmatically control the RAN using this virtualized setup . We implemented a vRAN data collection system to retrieve SNR of each user and buffer occupancy from the 5G vRAN software. We monitor the user throughput using diagnostic information from AP5GC. We utilize the buffer occupancy and user throughput information to estimate the latency experienced by each dataflow. We use a custom slice bandwidth controller from Altran 5G vRAN that changes the slice bandwidth according to the output from Algorithm 1. The controller is lightweight, and this API allows us to adjust slice bandwidth allocations at the granularity of a few milliseconds.

Zipper is compatible with O-RAN specifications [34]. Zipper would be hosted in the Near-Real-Time RAN Intelligent Controller (RIC). In an O-RAN deployment, Zipper would retrieve SNR, buffer occupancy and throughput from the E2 Monitor interface, and would send slice bandwidth control signals over the E2 Control interface.

Zipper slice manager. Our implementation of the slice manager (Fig. 4), which includes the MPC framework, bandwidth allocation algorithm, and resource availability module, is about 4,000 lines of Go. The slice manager is multi-threaded. It computes the bandwidth allocations for each slice in parallel. When Zipper receives a new app request, the slice manager spawns a new thread to run the admission controller. Zipper obtains RAN telemetry (i.e., SNR measurements, app buffer occupancy [43], etc.) from the vRAN via a UDP socket. Zipper populates its state—maintained over a moving horizon T_w (§3.1)—with these measurements. We implemented data buffers to support quick, thread-safe read/write access that meets the stringent slot deadlines for 5G workloads [1].

6 Evaluation

We evaluate Zipper with typical RAN workloads (§6.1) on our production-grade testbed (§6.2) and in emulated environments (§6.3-§6.5). Our evaluation highlights include:

- On our end-to-end testbed, Zipper dynamically tunes slice bandwidths every millisecond to fulfill app SLAs (§6.2).
- Compared to a slice-level service assurance scheduler, Zipper reduces tail throughput and latency penalties as a percentage of app SLAs by 9× (§6.3).
- Zipper can support 150 apps drawn from a typical workload, and incur nearly no penalty in throughput and latency (§6.3).
- In order to fulfill app SLAs, Zipper utilizes about 30% more bandwidth than RAN schedulers without SLA constraints, but 50% less bandwidth than prior slicing systems (§6.3) .
- Zipper's admission control framework can intelligently allocate unutilized bandwidth, admitting 15% more apps for a first-come-first-served policy (§6.4).
- Zipper supports 200 apps and 70 slices in real time (§6.5).

6.1 Evaluation setup

Emulation. We develop a real-time emulation framework to compare Zipper against baselines under controlled network environments. We develop a data generator to emulate realistic demand patterns for different apps, and develop a channel

the convolution of their individual distributions [31].

App Type	Min Tput	Max Latency	QCI [36]	Freq.
Video conf.	2 Mbps	150 ms	40	30%
Voice	200 kbps	100 ms	20	30%
Vehicle-to-X	200 kbps	50 ms	40	10%
Video stream.	2 Mbps	300 ms	60	20%
VR offload	10 Mbps	30 ms	68	5%
File sync	20 Mbps	—	80	5%

Table 1: Apps, SLAs, and frequencies selected for experiments.

emulator that exposes apps to real network traces. We describe both below.

Apps. For our experiments, we choose several representative applications that cover the gamut of throughput and latency requirements. Table 1 summarizes the SLAs we select for these applications, based on the definitions in the 3GPP specifications [36], and also reports the frequency of each app type (as a percentage) in our experiments. Since we do not have access to real-world cellular traces, we mimic a typical workload according to breakdowns of mobile Internet traffic published in industry technical reports [23, 60].

For file sync apps, we instrument iPerf [22] to send UDP traffic at different rates. For video conferencing, video streaming, IoT, and v2x apps, we implement a data generator in Go to send UDP packets at different sending rates and inter-packet delays. For VR remote rendering, we gather traces from a real Hololens app, and replay the packet captures over UDP.

SNR traces. To evaluate Zipper against the baselines under a controlled setting, we use a publicly available dataset of SNR traces, collected by running mobile traffic (e.g., Netflix videos, Amazon browsing, etc.) over a production 5G network in Ireland [54]. Our testbed experiment with real client devices (§6.2) evaluates Zipper on a live wireless channel.

Base station configuration. For all of our experiments, we configure our base station to have a total bandwidth of 100 MHz and 4×4 MIMO (i.e., 4 layers). For simplicity, we configure all slices to numerology $\mu = 1$ (i.e., 30 kHz subcarrier spacing). In our experiments, each slice caters to apps of the same type.[9]

6.2 End-to-end evaluation

We begin by evaluating Zipper end-to-end on our production-grade 5G vRAN testbed (§5), in order to demonstrate that Zipper can deliver reliable connectivity by dynamically adjusting slice bandwidths in real-time, while adapting to variations in channel. For this experiment, we consider a scenario where the base station has one slice that serves a single file sync app. We would like to see that Zipper (i) allocates minimal slice bandwidth such that it does not always use all 100 MHz available, and (ii) adapts the bandwidth allocation for this slice as the measured channel quality varies. We run a 17 Mbps iPerf flow on a OnePlus mobile phone that is connected to the 5G base station running Zipper. During the download, we both walk around the room and stand in a fixed location to capture a variety of channel conditions.

[9] If the operator does not know the app type ahead of time, she could match apps with similar connectivity requirements, by clustering based on SLAs (e.g., high bandwidth only, or high bandwidth and low latency).

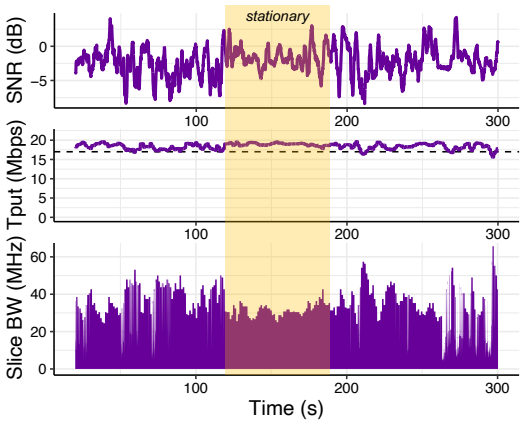


Figure 10: Zipper tunes the bandwidth allocated to a slice serving a mobile OnePlus phone running 17 Mbps iPerf flow.

Fig. 10 shows a stacked time series chart of the bandwidth that Zipper allocates to the slice (bottom), the application throughput (middle), and the SNR of the OnePlus phone (top). The segment highlighted in yellow corresponds to the segment of time during which the UE was stationary. Notice that Zipper reliably meets the target throughput of 17 Mbps—without significantly over-delivering. To do this, it adjusts its slice bandwidth allocation at a millisecond granularity; notice, in particular during the stationary period, where the measured SNR is relatively high and stable, the allocated bandwidth is accordingly lower (i.e., 30 MHz).

6.3 SLA compliance

Setup. In order to evaluate Zipper's ability to comply with SLAs, we use our emulation framework (§6.1) to compare Zipper against other schemes in settings where the wireless conditions are controlled. Like the testbed, our emulator runs in real time. We compare Zipper against the following four baseline algorithms:

- The *Single Slice policy* schedules all apps together in one slice, using a proportional fair scheduler [63], which is widely used by base stations today [4].
- The *QoS policy* [12, 71], like Single Slice, schedules all apps in one slice with a proportional fair scheduler, but additionally prioritizes each app according to its QoS Class Identifier (QCI). The 3GPP standards specify a unique QCI priority for each traffic type [36]. Table 1 shows the QCIs we use for the apps in our experiments. This policy is also common in production RAN deployments today.
- The *NVS policy* [41] is a dynamic network slicing algorithm for WiMAX, which provides slice-level QoS guarantees by multiplexing slices over time. Each slice requests an aggregate throughput (for all users). The NVS controller tracks each slice's throughput. In each time interval (e.g., 10 ms), it computes—for each slice—a priority, which is defined as the ratio of requested throughput to average throughput, and then selects the slice with the highest priority. While

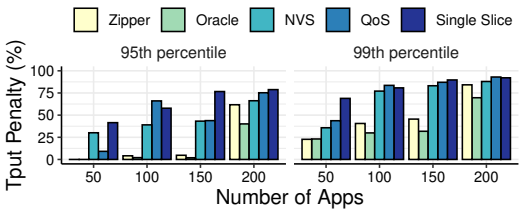


Figure 11: Tail throughput penalties for varying load. Apps scheduled by Zipper experience 95th percentile penalties close to 0%.

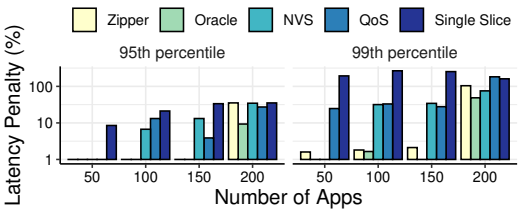


Figure 12: Tail latency penalties for varying base station loads. Apps scheduled by Zipper have low 95th and 99th percentile penalties.

the NVS paper assumes constant MCS, we implement a modified version of the algorithm that uses channel forecasts to dynamically adjust MCS. NVS is a popular benchmark among recent RAN slicing proposals [15, 19, 24].

- The *Oracle policy* simply runs Zipper's bandwidth allocation algorithm (§4.2), but instead of forecasting wireless channel, it reads the true channel quality that each user will experience from the trace of SNR values. This algorithm allows us to validate our hypothesis that Zipper should be robust to modest SNR prediction error (§4.1).

Zipper and all baselines have access to the same overall bandwidth (e.g., 100 MHz channel). These algorithms differ in how they divide up the bandwidth amongst slices.

We vary the number of apps, ensuring, for all baselines, that apps connect in the same order and that each has the same SNR trace. We create 6 slices—one for each traffic type—and run each experiment for 3 minutes. We measure the throughput and latency penalties (Eqn. 5 - Eqn. 6) for each app, after all apps have connected to the RAN (i.e., we exclude the time when apps join/leave the RAN). To create some background load at the base station, we assign twenty 20 Mbps iPerf flows to a "best effort" slice.[10] We do not use the resource availability estimator to admit/reject apps for these experiments.

Metrics of merit. We compare how well the different schemes satisfy *each app's* throughput and latency requirements, since these two quantities impact the quality-of-experience for most typical mobile apps [23, 36, 60]. In particular, for each experiment, we compute the throughput and latency penalties, defined in Eqn. 5 - Eqn. 6, and report the p95 and p99 penalties to quantify the tail performance. A lower penalty is better.

Note that we evaluate penalties (as a fraction of the requested SLAs) instead of evaluating absolute throughputs and latencies. The penalty metric allows us to directly compare app-level service assurance across slices serving apps with significantly different network requirements. Consider a slice serving a 20 Mbps file sync app A and a different slice serving a 2 Mbps video conferencing session B. A scheme that delivers 19 Mbps to A and 1 Mbps to B would yield 1 Mbps less than the requested amount for each app. But 1 Mbps is more consequential to B (50%) than it is to A (5%). The penalty metric captures this.

Throughput penalties. Fig. 11 shows the results. The Single Slice scheduler consistently incurs the highest penalty. It cannot differentiate between traffic types, and its proportional fair scheduler will attempt to maximize throughput subject to some fairness constraints. Therefore, it tends to favor bandwidth intensive apps (e.g., file sync). Notice that, because Single Slice runs a proportional fair scheduler, it does not starve any app (throughput penalty is never 100%). The QoS policy does marginally better than Single Slice. The QCI priorities only control the relative frequencies at which the scheduler allocates resource blocks to apps, but if there is a contention, an app with higher priority may not get its desired rate. Moreover, the standards pre-define the QCIs [36], and the scheduling algorithm has no ability to dynamically tune these priorities to have the desired effect on app throughput.

NVS sees a marked improvement in penalty, compared to both Single Slice and QoS. However, the throughput penalties are still high, even at low loads (e.g., p95 penalty for 50 apps is 25%). This is because it optimizes for slice-level throughput instead of for app-level. Zipper, by contrast, exhibits comparatively lower penalties at both p95 and p99.

Latency penalties. Fig. 12 shows the latency penalties (on a log-scale) for the same experiment. The Single Slice policy delivers the worst latency penalties: it tends to favor bandwidth-intensive apps, and thus, low bandwidth, latency-critical apps (e.g., v2x and voice) will suffer. These apps, in particular, experiences latencies as high as 200 ms. QoS achieves around 40% lower penalties than Single Slice at p95, since the QCI priorities allow the scheduler to explicitly prioritize the latency-critical apps with higher QCI by scheduling them more frequently. This is because NVS multiplexes slices over time, and, in each timestep, it gives all bandwidth to the slice it chooses. Even if the switching interval is low, apps can go unscheduled for long periods of times in configurations with many slices. Zipper, by contrast, maintains very low latencies up to 150 apps, after which the base station starts to become oversubscribed.

Notice that Zipper's performance is comparable to that of the Oracle. The difference in the p95 penalty is $< 5\%$ for fewer than 150 apps, which is consistent with our observations when modeling the system (§4.1). The gap widens slightly at 200 apps, since the system is more congested, and thus, assigning a suboptimal MCS would be more consequential.

RAN utilization. Fig. 11 and Fig. 12 show that Zipper reliably delivers the connectivity requested by each app. However, fulfilling SLAs for each app rather than for a slice in aggregate requires more bandwidth. We conduct an experiment to measure how much spectrum or capacity Zipper wastes at the expense of allocating resources to meet SLAs. Fig. 13 shows a 150-second time series snapshot of the aggregate RAN

[10] We assign these flows the lowest QCI priority amongst those in Table 1.

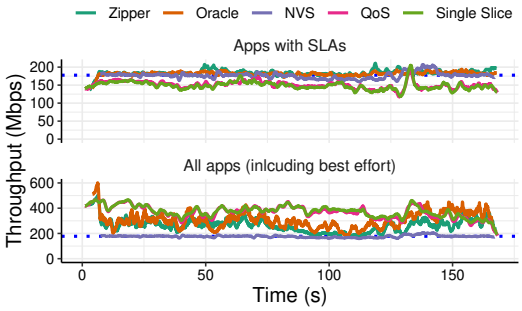
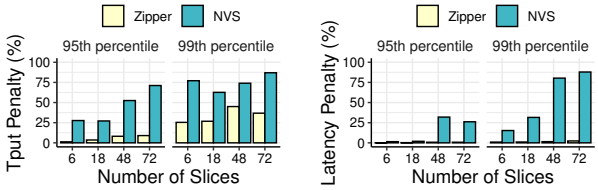


Figure 13: Throughput for 75 apps + 20 best effort apps. Zipper meets the SLAs reliably, and allocates excess capacity to best effort.



(a) Throughput penalty (b) Latency penalty

Figure 14: Zipper's performance is invariant to the number of slices.

throughput achieved by Zipper and the different baselines for 75 apps (+ 20 best effort apps). On top chart, we show the throughput amongst the 75 apps that requested SLAs, and on the bottom chart, we plot the total RAN throughput (including best effort). The dotted blue line shows the total throughput requested by the 75 apps.

Zipper, Oracle, and NVS closely track the requested throughput at the level of slice—providing reliable and consistent performance despite the wireless channels that each app experiences. Single Slice and QoS fall about 18% below the target throughput. However, as Fig. 11 shows, this drop translates to far worse in terms of throughput penalty.

When we include the best effort apps, we find that QoS and Single Slice indeed achieve the highest total RAN throughput. NVS does not schedule the best effort apps because there is no excess bandwidth when all bandwidth is allocated to a single slice in a given scheduling interval. Zipper strikes a nice balance between these extremes: in addition to meeting requested SLAs, Zipper utilizes spectrum about 50% better than NVS and 30% worse than Single Slice or QoS.

Scaling up slices. The experiments so far considered 6 slices—one for each app type. However, an operator may choose to have multiple slices for the same app type, for e.g., if Zoom and Teams want to isolate their traffic. Therefore, we would like Zipper to be invariant to the number of slices. We run an experiment similar to the setup described above; we fix the number of apps at 100 and vary the number of slices. We still dedicate each slice to serving a unique app type, and we randomly assign apps to slices when there are multiple slices of the same type. Fig. 14a and Fig. 14b show the throughput and latency penalties, respectively, for Zipper and NVS.

NVS scales poorly with the number of slices. Since NVS does not multiplex slices across frequency, slices get scheduled less frequently. Both penalties suffer with more slices, even when the switching interval is short (i.e., 10 ms). By contrast, Zipper's performance is invariant to the number of slices, since it multiplexes slices across frequency and time.

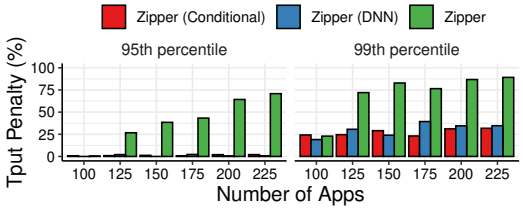


Figure 15: Both the conditional and DNN-based resource estimation methods achieve bounded (and low) penalties.

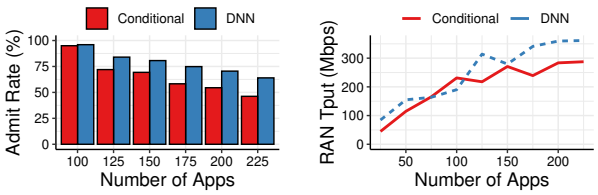
6.4 Forecasting RAN resource availability

Setup. To evaluate this module, we consider a simple first-come-first-served (FCFS) policy that admits incoming apps in the order that they arrive, as long as there is enough capacity to accommodate them without violating SLAs for other apps. Specifically, from the distribution P (§4.3), we admit an incoming app to its designated slice if the p95 bandwidth is within an $\varepsilon = 5$ MHz tolerance of the total bandwidth B, and assigns it to the best effort slice otherwise (i.e., $P < B + \varepsilon$). Note that the specific policy does not matter for this evaluation; we only want the policy to be consistent across different resource availability modules that we compare.

We compare against a "conditional" resource availability primitive: it (i) admits the incoming app into a best effort slice, (ii) measures its throughput and latency penalties for 10 seconds, and (iii) then admits in FCFS order if it incurs zero penalty or leaves it in the best effort slice otherwise. This form of probing resource availability conditionally is common in many admission control proposals [29, 37, 52]. Our goal is to evaluate if the resource availability forecasts provided by Zipper's DNN family yield better RAN utilization than this "conditional" primitive. Both use Zipper to compute slice bandwidth schedules in real-time. As in §6.3, we draw apps from the distribution in Table 1, and select app arrival times and contract durations at random.

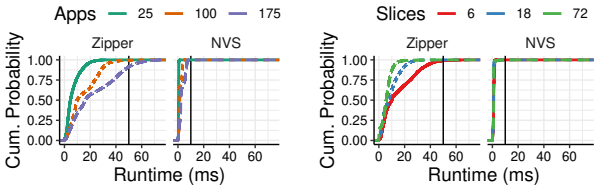
Bounded penalty. A good forecaster of resource availability should ensure that it can satisfy the SLAs of apps that it has already admitted before committing to a new app. This amounts to ensuring that the RAN maintains a bounded and low penalty, as more apps connect to the system. Fig. 15 shows the 95th and 99th percentile throughput penalties for different numbers of apps that try to connect to the RAN; the algorithm named "Zipper" uses no admission controller. Notice that, by letting each incoming app experience the network, both methods that use an admission control policy keep the penalties bounded, and, more importantly, close to 0% at the 95th percentile.

Admit rate. While the penalties are bounded, does the RAN have some unutilized capacity that it could have allocated by admitting more apps? Fig. 16a compares the admit rates for



(a) Admit rate (b) RAN utilization

Figure 16: Zipper's DNN resource estimator achieves a higher admit rate and utilization by squeezing in apps with lighter demand.



(a) Number of apps (b) Number of slices

Figure 17: Runtime of Zipper and NVS. Even though Zipper involves more computation than NVS, it is still practical for large workloads.

different resource availability modules. As we would expect, when the base station is not congested (e.g., 100 apps), the admit rate is high (around 95%) for both the conditional and DNN-based policies. However, as more apps join the system, the DNN policy has a higher admit rate—about 15% higher for 225 apps. We observe similar trends for latency.

The policy that uses the "conditional admit" forecaster rejects apps too aggressively because it conditions its decision on an app's measured penalty in the best effort slice. At higher loads, Zipper ends up allocating most—if not all—spectrum to slices serving apps with SLAs. So Zipper allocates little bandwidth to the best effort slice, and the incoming app receives infrequent air time in its initial 10 second sampling period. Its penalties are thus high, and the controller has little confidence that the app can meet its requirements in the target slice.

The DNN's admit rate is higher at greater load because it is able to differentiate between different traffic types. For instance, the DNN infers that a voice app could still achieve its light target throughput and latency because the slice's MAC scheduler could accommodate a new app without degrading the SLAs of the apps already admitted to that slice. By contrast, the "conditional admit" mechanism has little data make this inference, since the voice app gets little air time in a best effort slice.

RAN utilization. Aggressive resource availability forecasts can underutilize the RAN. Fig. 16b compares the total RAN throughput for both estimators. Notice that the two methods diverge around 150 apps, after which the DNN can better pack more "lightweight" apps.

6.5 Microbenchmarks

We profile Zipper's slice allocation and management overhead as we stress the system with more apps and with more slices. For the traffic distribution listed in Table 1, we profile the time it takes to compute the bandwidth allocations and MAC schedules. We compare Zipper with NVS. Fig. 17 shows the results as a CDF of runtimes over all scheduling intervals in each 3 minute experiment. The vertical black lines indicate the deadlines required to operate in real time. Zipper, though more complex, reliably meets processing deadlines, as the load increases with more apps or fewer slices (i.e., more apps per slice).

7 Discussion

Network APIs. Provisioning connectivity based on app SLAs creates new opportunities. For instance, a developer can split their app into multiple data streams (e.g., audio, video, and sensory for VR), and define SLAs independently for each one. Because Zipper internally estimates network capacity to forecast resource availability, an operator using Zipper could create a network API that exposes metrics like true network capacity to developers. This helps make the network more transparent.

ML components. The ideal deployment for Zipper should have a V100 GPU. However, note that Zipper is compatible with any channel predictor, and Fig. 6 shows that a simple EWMA predictor works reasonably well. Moreover, the DNNs in the resource availability estimator are lightweight and do not have real-time deadlines, unlike channel prediction; if resource constrained, operators could serve the DNN on a CPU.

Application adaptivity. An important benefit of the paradigm proposed by Zipper is that application developers no longer need to stress about making their apps reactive to the network. By design, Zipper seeks to provision the right amount of bandwidth so that each app experiences a relatively static network. As a result, interactive and adaptive apps will no longer have to adapt to changing network conditions.

8 Conclusion

We developed Zipper, the first 5G RAN slicing system for application-level service assurance. Zipper formulates the scheduling problem with MPC and develops an efficient optimization algorithm to compute SLA-compliant schedules in real-time. Zipper also introduces a primitive to forecast RAN resource availability, with which operators can interface an admission control policy. We implemented Zipper on a production-grade 5G vRAN testbed, adding critical hooks to control slice bandwidths in real time. We evaluated Zipper extensively on realistic workloads, our results showed that Zipper more reliably fulfills app-level SLAs than do QoS schedulers and slice-level service assurance systems.

There several opportunities to extend Zipper. First, to accommodate highly mobile users, we can co-optimize slicing across base stations. Second, we believe we can extend the formulation and optimization techniques we developed to other SLA types, such as energy consumption and bit error rate. Finally, we hope to explore robust economic models for admission control that build on Zipper's resource availability estimator.

Acknowledgments

We thank the reviewers, Mohammad Alizadeh, Hari Balakrishnan, Xenofon Foukas, Anuj Kalia, Radhika Mittal, and Deepak Vasisht for helpful conversations and detailed feedback.

References

[1] A. Al-Dulaimi, X. Wang, and C.-L. I. *Network Slicing for 5G Networks*, pages 327–370. John Wiley & Sons, New Jersey, USA, 2018.

[2] Altran. Capgemini altran. Technical report, Capgemini, 2023.

[3] S. Bakri, P. A. Frangoudis, A. Ksentini, and M. Bouaziz. Data-driven ran slicing mechanisms for 5g and beyond. *IEEE Transactions on Network and Service Management*, 18(4):4654–4668, 2021.

[4] A. Balasingam, M. Bansal, R. Misra, K. Nagaraj, R. Tandra, S. Katti, and A. Schulman. Detecting if lte is the bottleneck with bursttracker. In *The 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19, New York, NY, USA, 2019. Association for Computing Machinery.

[5] M. Bansal, A. Schulman, and S. Katti. Atomix: A framework for deploying signal processing applications on wireless infrastructure. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 173–188, Oakland, CA, May 2015. USENIX Association.

[6] N. Baranasuriya, V. Navda, V. N. Padmanabhan, and S. Gilbert. Qprobe: Locating the bottleneck in cellular communication. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery.

[7] L. Bonati, S. D'Oro, S. Basagni, and T. Melodia. Scope: An open and softwarized prototyping platform for nextg systems. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '21, page 415–426, New York, NY, USA, 2021. Association for Computing Machinery.

[8] N. Budhdev, R. Joshi, P. G. Kannan, M. C. Chan, and T. Mitra. Fsa: Fronthaul slicing architecture for 5g using dataplane programmable switches. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, MobiCom '21, page 723–735, New York, NY, USA, 2021. Association for Computing Machinery.

[9] P. Caballero, A. Banchs, G. de Veciana, and X. Costa-Pérez. Multi-tenant radio access network slicing: Statistical multiplexing of spatial loads. *IEEE/ACM Transactions on Networking*, 25(5):3044–3058, 2017.

[10] P. Caballero, A. Banchs, G. de Veciana, X. Costa-Pérez, and A. Azcorra. Network slicing for guaranteed rate services: Admission control and resource allocation games. *IEEE Transactions on Wireless Communications*, 17(10):6419–6432, 2018.

[11] Z. Cao, Q. Ma, A. B. Smolders, Y. Jiao, M. J. Wale, C. W. Oh, H. Wu, and A. M. J. Koonen. Advanced integration techniques on broadband millimeter-wave beam steering for 5g wireless networks and beyond. *IEEE Journal of Quantum Electronics*, 52(1):1–20, 2016.

[12] F. Capozzi, G. Piro, L. A. Grieco, G. Boggia, and P. Camarda. Downlink packet scheduling in lte cellular networks: Key design issues and a survey. *IEEE communications surveys & tutorials*, 15(2):678–700, 2012.

[13] M. Chahbar, G. Diaz, A. Dandoush, C. Cérin, and K. Ghoumid. A comprehensive survey on the e2e 5g network slicing model. *IEEE Transactions on Network and Service Management*, 18(1):49–62, 2020.

[14] C.-Y. Chang and N. Nikaein. Ran runtime slicing system for flexible and dynamic service execution environment. *IEEE Access*, 6:34018–34042, 2018.

[15] Y. Chen, R. Yao, H. Hassanieh, and R. Mittal. Channel-aware 5g ran slicing with customizable schedulers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[16] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, Oct. 2014. Association for Computational Linguistics.

[17] E. Coronado and R. Riggio. Flow-based network slicing: Mapping the future mobile radio access networks. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, 2019.

[18] I. Corporation. Flexran reference architecture for wireless. https://www.intel.com/content/www/us/en/developer/topic-technology/edge-5g/tools/flexran.html, 2022.

[19] X. Costa-Pérez, J. Swetina, T. Guo, R. Mahindra, and S. Rangarajan. Radio access network virtualization for future mobile carrier networks. *IEEE Communications Magazine*, 51(7):27–35, 2013.

[20] X. de Foy. Network slicing – 3gpp use case. Technical report, Internet Engineering Task Force, 2017.

[21] J. Ding, R. Doost-Mohammady, A. Kalia, and L. Zhong. *Agora: Real-Time Massive MIMO Baseband Processing in Software*, page 232–244. Association for Computing Machinery, New York, NY, USA, 2020.

[22] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhu. Iperf. https://iperf.fr/, 2022.

[23] Ericsson. Ericsson mobility report. Technical report, Ericsson, 2021.

[24] X. Foukas, M. K. Marina, and K. Kontovasilis. Orion: Ran slicing for a flexible and cost-effective multi-service mobile network architecture. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, page 127–140, New York, NY, USA, 2017. Association for Computing Machinery.

[25] X. Foukas, B. Radunovic, M. Balkwill, and Z. Lai. Taking 5g ran analytics and control to a new level. In *Technical Report*, December 2022.

[26] C. E. Garcia, D. M. Prett, and M. Morari. Model predictive control: Theory and practice—a survey. *Automatica*, 25(3):335–348, 1989.

[27] K. C. Garikipati, K. Fawaz, and K. G. Shin. Rt-opex: Flexible scheduling for cloud-ran processing. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 267–280, New York, NY, USA, 2016. Association for Computing Machinery.

[28] Y. Ghasempour, M. K. Haider, and E. W. Knightly. Decoupling beam steering and user selection for mu-mimo 60-ghz wlans. *IEEE/ACM Transactions on Networking*, 26(5):2390–2403, 2018.

[29] T. Guo and A. Suárez. Enabling 5g ran slicing with edf slice scheduling. *IEEE Transactions on Vehicular Technology*, 68(3):2865–2877, 2019.

[30] T. Guo and A. Suárez. Enabling 5g ran slicing with edf slice scheduling. *IEEE Transactions on Vehicular Technology*, 68(3):2865–2877, 2019.

[31] R. V. Hogg and A. T. Craig. Introduction to mathematical statistics.(5"" edition). *Englewood Hills, New Jersey*, 1995.

[32] Y. Huang, S. Li, Y. T. Hou, and W. Lou. Gpf: A gpu-based design to achieve 100 μs scheduling for 5g nr. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, page 207–222, New York, NY, USA, 2018. Association for Computing Machinery.

[33] M. Hüsken and P. Stagge. Recurrent neural networks for time series classification. *Neurocomputing*, 50:223–235, 2003.

[34] C.-L. I and S. Katti. O-ran: Towards an open and smart ran. Technical report, Open RAN Alliance, 2018.

[35] E. T. S. Institute. *Physical layer procedures for data*. ETSI 3rd Generation Partnership Project (3GPP), 06 2018.

[36] E. T. S. Institute. *System Architecture for the 5G System*. ETSI 3rd Generation Partnership Project (3GPP), 06 2018.

[37] M. Jiang, M. Condoluci, and T. Mahmoodi. Network slicing management & prioritization in 5g mobile systems. In *European Wireless 2016; 22th European Wireless Conference*, pages 1–6, 2016.

[38] W. Jiang and H. D. Schotten. Deep learning for fading channel prediction. *IEEE Open Journal of the Communications Society*, 1:320–332, 2020.

[39] M. T. Kawser, N. I. B. Hamid, M. N. Hasan, M. S. Alam, and M. M. Rahman. Downlink snr to cqi mapping for different multipleantenna techniques in lte. *International journal of information and electronics engineering*, 2(5):757, 2012.

[40] B. Khodapanah, A. Awada, I. Viering, D. Oehmann, M. Simsek, and G. P. Fettweis. Fulfillment of service level agreements via slice-aware radio resource management in 5g networks. In *2018 IEEE 87th Vehicular Technology Conference (VTC Spring)*, pages 1–6, 2018.

[41] R. Kokku, R. Mahindra, H. Zhang, and S. Rangarajan. Nvs: A substrate for virtualizing wireless resources in cellular networks. *IEEE/ACM Transactions on Networking*, 20(5):1333–1346, 2012.

[42] S. Kumar, E. Hamed, D. Katabi, and L. Erran Li. Lte radio analytics made easy and accessible. *SIGCOMM Comput. Commun. Rev.*, 44(4):211–222, aug 2014.

[43] H. Lee, S. Noghabi, B. Noble, M. Furlong, and L. Cox. Bumblebee: Application-aware adaptation for edge-cloud orchestration. In *Symposium on Edge Computing*. ACM/IEEE, December 2022.

[44] J. Lee, S. Lee, J. Lee, S. D. Sathyanarayana, H. Lim, J. Lee, X. Zhu, S. Ramakrishnan, D. Grunwald, K. Lee, and S. Ha. Perceive: Deep learning-based cellular uplink prediction using real-time scheduling patterns. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, MobiSys '20, page 377–390, New York, NY, USA, 2020. Association for Computing Machinery.

[45] J. Li, W. Shi, P. Yang, Q. Ye, X. S. Shen, X. Li, and J. Rao. A hierarchical soft ran slicing framework for differentiated service provisioning. *IEEE Wireless Communications*, 27(6):90–97, 2020.

[46] X. Li, C. Guo, L. Gupta, and R. Jain. Efficient and secure 5g core network slice provisioning based on vikor approach. *IEEE Access*, 7:150517–150529, 2019.

[47] C. Luo, J. Ji, Q. Wang, X. Chen, and P. Li. Channel state information prediction for 5g wireless communications: A deep learning approach. *IEEE Transactions on Network Science and Engineering*, 7(1):227–236, 2018.

[48] Microsoft. Azure private 5g core. Technical report, Microsoft, 2023.

[49] L. S. Muppirisetty, T. Svensson, and H. Wymeersch. Spatial wireless channel prediction under location uncertainty. *IEEE Transactions on Wireless Communications*, 15(2):1031–1044, 2015.

[50] Nvidia. Nvidia aerial sdk. https://developer.nvidia.com/aerial-sdk, 2022.

[51] O-RAN. O-ran specifications. Technical report, O-RAN Alliance, 2023.

[52] M. O. Ojijo and O. E. Falowo. A survey on slice admission control strategies and optimization schemes in 5g network. *IEEE Access*, 8:14977–14990, 2020.

[53] Qulsar. Qulsar qg2. https://qulsar.com/Products/Systems/Qg_2.html, 2022.

[54] D. Raca, D. Leahy, C. J. Sreenan, and J. J. Quinlan. Beyond throughput, the next generation: A 5g dataset with channel and context metrics. In *Proceedings of the 11th ACM Multimedia Systems Conference*, MMSys '20, page 303–308, New York, NY, USA, 2020. Association for Computing Machinery.

[55] D. A. Ravi, V. K. Shah, C. Li, Y. T. Hou, and J. H. Reed. Ran slicing in multi-mvno environment under dynamic channel conditions. *IEEE Internet of Things Journal*, 9(6):4748–4757, 2022.

[56] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[57] O. Sallent, J. Perez-Romero, R. Ferrus, and R. Agusti. On radio access network slicing from a radio resource management perspective. *IEEE Wireless Communications*, 24(5):166–174, 2017.

[58] V. Sciancalepore, K. Samdanis, X. P. Costa, D. Bega, M. Gramaglia, and A. Banchs. Mobile traffic forecasting for maximizing 5g network slicing resource utilization. In *2017 IEEE Conference on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*, pages 1–9, Atlanta, GA, 2017. IEEE.

[59] N. Shahriar, S. Taeb, S. R. Chowdhury, M. Zulfiqar, M. Tornatore, R. Boutaba, J. Mitra, and M. Hemmati. Reliable slicing of 5g transport networks with bandwidth squeezing and multi-path provisioning. *IEEE Transactions on Network and Service Management*, 17(3):1418–1431, 2020.

[60] C. Systems. Cisco annual internet report (2018 - 2023). Technical report, Cisco Systems, 2020.

[61] K. Tan, H. Liu, J. Zhang, Y. Zhang, J. Fang, and G. M. Voelker. Sora: High-performance software radio using general-purpose multi-core processors. *Commun. ACM*, 54(1):99–107, jan 2011.

[62] S. Troia, A. F. R. Vanegas, L. M. M. Zorello, and G. Maier. Admission control and virtual network embedding in 5g networks: A deep reinforcement-learning approach. *IEEE Access*, 10:15860–15875, 2022.

[63] D. Tse and P. Viswanath. *Fundamentals of Wireless Communication*. Cambridge University Press, Cambridge, UK, 2005.

[64] P. Wieber. Trajectory free linear model predictive control for stable walking in the presence of strong perturbations. In *2006 6th IEEE-RAS International Conference on Humanoid Robots, Genova, Italy, December 4-6, 2006*, pages 137–142, Genova, Italy, 2006. IEEE.

[65] J. Wu, Z. Zhang, Y. Hong, and Y. Wen. Cloud radio access network (c-ran): a primer. *IEEE network*, 29(1):35–41, 2015.

[66] F. Y. Yan, H. Ayers, C. Zhu, S. Fouladi, J. Hong, K. Zhang, P. Levis, and K. Winstein. Learning *in situ*: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, Santa Clara, CA, feb 2020. USENIX Association.

[67] Q. Yang, X. Li, H. Yao, J. Fang, K. Tan, W. Hu, J. Zhang, and Y. Zhang. Bigstation: Enabling scalable real-time signal processingin large mu-mimo systems. *SIGCOMM Comput. Commun. Rev.*, 43(4):399–410, aug 2013.

[68] Q. Ye, J. Li, K. Qu, W. Zhuang, X. S. Shen, and X. Li. End-to-end quality of service in 5g networks: Examining the effectiveness of a network slicing framework. *IEEE Vehicular Technology Magazine*, 13(2):65–74, 2018.

[69] W. Yin, K. Kann, M. Yu, and H. Schütze. Comparative study of CNN and RNN for natural language processing. *CoRR*, abs/1702.01923, 2017.

[70] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference*

on Special Interest Group on Data Communication, SIGCOMM '15, page 325–338, New York, NY, USA, 2015. Association for Computing Machinery.

[71] Y. Zaki, T. Weerawardane, C. Görg, and A. Timm-Giel. Multi-qos-aware fair scheduling for LTE. In *Proceedings of the 73rd IEEE Vehicular Technology Conference, VTC Spring 2011, 15-18 May 2011, Budapest, Hungary*, pages 1–5, Budapest, Hungary, 2011. IEEE.

[72] L. Zanzi, V. Sciancalepore, A. Garcia-Saavedra, H. D. Schotten, and X. Costa-Pérez. Laco: A latency-driven network slicing orchestration in beyond-5g networks. *IEEE Transactions on Wireless Communications*, 20(1):667–682, 2021.

A Allocating Slice Bandwidth in Zipper

A.1 Forecasting the wireless channel with an RNN

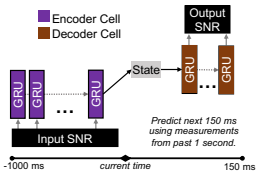


Figure 18: Architecture of RNN model to forecast wireless channel.

To forecast each user's channel, we train a sequence-to-sequence Recurrent Neural Network (RNN) [56], which uses an input sequence of SNR measurements over the last 1 second to predict a sequence of SNR measurements over the next 150 milliseconds. Each RNN cell is a Gated Recurrent Unit (GRU) [16], a lightweight mechanism that learns both short-term and long-term trends in a signal. GRUs are popular in temporal prediction tasks, like time-series prediction [33] and natural language models [69]. Zipper's RNN model includes two types of GRU cells—encoder and decoder—allowing the model to develop two distinct skills: (i) to build a model of the current state by looking at past values and (ii) to understand the current state to predict future values. Our implementation uses 50 hidden neurons in each layer of the encoder and decoder. Fig. 18 illustrates the architecture of this RNN.

RNNs are emerging a popular method to forecast timeseries, including wireless channel [38, 44, 47, 49]. We develop and train a model, but note that Zipper can support any predictor of wireless channel. §4.1 characterizes the requirements for suitable predictor.

A.2 Monotonicity of throughput and latency

An app's instantaneous RAN throughput depends on (i) the number of resource blocks it is allocated in each slot and (ii) the MCS scheme used to modulate data onto those resource blocks [4, 42]. If a scheduler assigns an app more resource blocks (i.e., bandwidth) in a slot, then the app will experience a higher throughput. Therefore, app throughput is a monotonically-increasing function of slice bandwidth.

§4.2 explains how latency is a monotonically-decreasing function of slice bandwidth. The intuition is that adding more bandwidth to a slice gives the scheduler more space to fit packets for an app and therefore reduce its latency.

A.3 Algorithm

Algorithm 1 specifies (in pseudocode) how Zipper computes slice bandwidth allocations. `SearchBandwidth()` is a recursive function that evaluates candidate bandwidths, pruning the search space with binary search, using the property that app throughput and latency vary monotonically with slice bandwidth.

Algorithm 1 Allocating slice bandwidth in Zipper

```
1:  procedure ZIPPER
2:      for each scheduling round t do
3:          for slice s ∈ S do
4:              B_s ← FINDBANDWIDTH(s, B)
5:          Resolve conflict if Σ_{s∈S} B_s > B
6:          Update app x̄_a(t) and d̄_a(t) and run MAC/PHY
7:  function FINDBANDWIDTHSLICE(s, B)
8:      Grab snapshot of app queues, throughput, and latency from s
9:      Forecast SNR for apps in s
10:     return SEARCHBANDWIDTH(s, 0, B, NULL)
11: function SEARCHBANDWIDTH(s, B_min, B_max, best)
12:     B̃_s ← (B_min + B_max)/2                    ▷ Find midpoint bandwidth.
13:     schedule ← RUNMAC(s, B̃_s)
14:     throughput ← ISTHROUGHPUTVALID(schedule)
15:     latency ← ISLATENCYVALID(schedule)
16:     if throughput ∧ latency then
17:         best = B̃_s                              ▷ Save best bandwidth.
18:         decrease = true
19:     if B̃_s ≤ B_min or B̃_s ≥ B_max then
20:         return best
21:     if decrease then
22:         return SEARCHBANDWIDTH(s, B_min, B̃_s, best)
23:     else
24:         return SEARCHBANDWIDTH(s, B̃_s, B_max, best)
```

B Estimating resource availability in Zipper

B.1 DNN architecture

Zipper builds a family of DNNs to estimate resource availability. Each DNN caters to different slice types. The input embedding consists of (i) the number of apps in the slice (including the incoming app, if applicable) and (ii) the number of apps in each SNR bucket. There are four possible SNR buckets.

Each DNN is a fully-connected network with 5 hidden layers, ranging from 512 to 10 neurons. The final layer has 7 output values for different percentiles of the probability distribution, i.e., p10, p25, p50, p75, p90, p95, p99.

CHISEL: An optical slice of the wide-area network

Abhishek Vijaya Kumar*, Bill Owens¶, Nikolaj Bjørner†, Binbin Guan†, Yawei Yin†, Paramvir Bahl†, Rachee Singh*

*Cornell University, †Microsoft, ¶NYSERnet

Abstract

Network slicing reserves a portion of the physical resources of radio access networks and makes them available to consumers. Slices guarantee traffic isolation, strict bandwidth and quality of service. However, the abstraction of slicing has been limited to access networks. We develop CHISEL, a system that dynamically carves slices of the wide-area network (WAN), enabling an *end-to-end* network slicing abstraction. CHISEL creates *optical slices* between WAN endpoints to avoid queueing and congestion delays inherent in packet switched paths in WANs. CHISEL incrementally allocates optical spectrum on long-haul fiber to provision slices. This task is made challenging by the co-existence of data-carrying channels on the fiber and numerous physical constraints associated with provisioning optical paths *e.g.,* spectrum contiguity, continuity and optical reach constraints. CHISEL leverages the empirical finding that cloud WANs have abundant optical spectrum to spare — 75% of optical spectrum on 75% of fiber spans is unused. CHISEL can optimally allocate terabits of slice requests while consuming minimal optical spectrum within seconds without increasing spectral fragmentation on fiber. CHISEL trades-off optimality of slice bandwidth allocation for faster run-time, provisioning slices within 2% of optimal in less than 30 seconds in a commercial cloud WAN. Finally, CHISEL reduces the latency of provisioning optical slices on hardware by 10X. Compared to IP tunnels of equivalent capacity, CHISEL consumes 3.3X fewer router ports.[1]

1 Introduction

Next generation mobile wireless networks are undergoing a technological revolution. The 3rd Generation Partnership Project (3GPP) has played a key role in this change by defining international standards that govern the capability and implementation of 5G technology [1]. Recently, 3GPP defined a *network slice* as "A logical network that provides specific network capabilities and network characteristics." Network slices guarantee traffic isolation, bandwidth and quality of service (QoS) by dynamically creating logical network instances that share the underlying physical network.

A case for end-to-end network slices. Slicing is primarily limited to access networks in the mobile network architecture

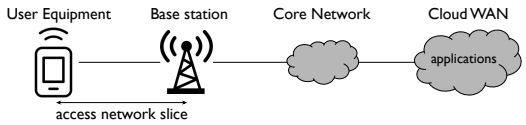


Figure 1: Access network slices.

(Figure 1) [30]. In this setting, traffic from user equipments (UEs) can be allocated a network slice for guaranteed QoS on the access network. However, the remaining network path from the radio access network (RAN) to the traffic destination does not support the slicing abstraction. Many applications (*e.g.,* voice, live-video, gaming) require strong network performance guarantees for reliable user experience. We aim to enable this by extending network slices from user endpoints all the way to the application server for an end-to-end QoS guarantee. The strict performance requirements of service-oriented traffic flows in next-generation mobile networks compound the need for end-to-end network slices [2].

Slices of the wide-area network. Next-generation mobile networks like 5G, offer a unique opportunity to implement end-to-end network slicing. Unlike the *monolithic* implementation of previous generations of mobile wireless, 5G disaggregates the access and packet core capability. The disaggregated access and packet core can be implemented as network functions on cloud-hosted servers and accelerators (Figure 2). The resulting *cloudification* of 5G networks has extended the placement of mobile network functions from operator base stations to cloud edges and datacenters [8]. Enabling end-to-end network slices from user equipment to cloud edges and datacenters requires a mechanism for slicing the cloud WAN.

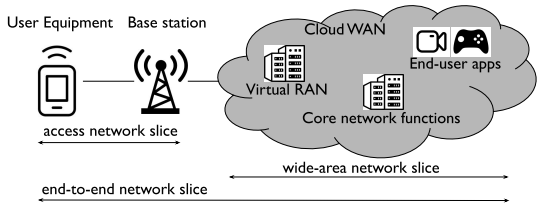


Figure 2: Access network slices vs. end-to-end network slices.

Optical slicing. In this work we undertake the task of slicing the wide-area network to implement end-to-end slices for network transport in the era of disaggregated and cloud-hosted mobile wireless networks. At their core, network slices provide dedicated bandwidth to their clients with strict QoS guar-

[1] CHISEL's code and data is at: http://opticalslice.network/.

antees. While tunneling abstractions (*e.g.,* MPLS tunnels [11], IP-in-IP [35]) make similar promises regarding performance, they fall short in practice (§2). By virtue of being built over packet-switched networks, such tunneling abstractions suffer from queuing delays and congestion at packet switches. These phenomena weaken the latency and throughput guarantees offered by tunneling, making them unsuitable for WAN slicing. Instead, we propose to slice the wide-area network at the physical or the *optical* layer.

We propose CHISEL, a system that provisions on-demand *optical slices* of the WAN. CHISEL's slices expose a network connection of dedicated bandwidth and strict performance guarantees to *network tenants* (*e.g.,* mobile access providers) of the cloud WAN. CHISEL slices provide *one-hop connections* between a source and destination router in the WAN by provisioning an all-optical path between them. CHISEL's slices achieve traffic isolation and performance guarantees on shared network infrastructure by optically bypassing all electrical packet switching between the endpoints of the slice.

Challenges. Provisioning on-demand optical WAN slices is challenging in practice for several reasons. First, while cloud providers are financially incentivized to provide an end-to-end slicing abstraction, the deployed network infrastructure (optical switches, fiber, amplifiers *etc.*) must support existing inter-datacenter traffic. Thus, carving on-demand optical slices requires accommodating slice requests in the partially occupied optical spectrum on fiber without disrupting existing traffic. Second, carving all-optical slices of the spectrum challenges physical limits like optical reach of signal transmission on fiber. Finally, cloud operators use static *channel maps* to assign optical spectrum to packet switches, making dynamic spectrum slicing slow and potentially disruptive.

Contributions. We tackle these technical challenges by first examining the utilization of spectrum in the optical backbone of a large commercial cloud provider. We find that nearly 75% of optical fiber spans in the cloud backbone have more than 75% of their spectrum freely available, showing that sufficient spectrum is free for CHISEL to carve wide-area optical slices (§3). We make the following contributions to leverage the available optical spectrum for enabling WAN slices:

Optimal slice allocation. CHISEL formulates the problem of dynamically allocating optical spectrum in cloud WANs as an optimization. CHISEL's optimization algorithm encodes physical constraints of optical signal transmission — limited optical reach, wavelength continuity and contiguity. CHISEL's algorithm can efficiently allocate optical slices in planet-scale WANs in seconds without fragmenting the spectrum (§4, §5).

Rapid and hitless provisioning of slices. CHISEL develops tools to implement optimal slice allocations computed by the algorithm on WAN optical switches. CHISEL can programmatically provision a slice of spectrum within 10 seconds without disrupting the existing channels on fiber (§6).

Field and lab experiments for evaluation. We build a hardware testbed to mimic a point-to-point wide-area fiber span of 2,600 km to proof-of-concept optical slicing. Additionally, we evaluate CHISEL in the field by provisioning a slice in an educational WAN located in New York, USA. Both laboratory and field experiments show that allocating on-demand optical spectrum is not disruptive to existing traffic and CHISEL can provision an optical slice in tens of seconds (§6).

Results. We show that CHISEL can compute slice allocations for terabits of bandwidth within a few seconds. Moreover, CHISEL takes tens of seconds to allocate an on-demand slice on optical hardware. CHISEL's allocations can be efficiently packed with existing wavelengths without increasing spectrum fragmentation. Not only do CHISEL-provisioned optical slices enable better performance guarantees and isolation than packet-switched tunnels, they offer these benefits at a lower price point. We show that CHISEL slices consume 3.3X fewer router ports compared to traffic engineering (TE) tunnels of the same bandwidth (§7).

General-purpose abstractions for optical circuit switching. We note that slicing optical spectrum on-demand is a general idea that is broadly applicable to other settings where network tenants require strict performance guarantees that go beyond the ability of packet switched networks. We believe that cloudified 5G network operators would benefit from CHISEL's optical slices in the near term (§8).

2 CHISEL Design

CHISEL creates on-demand optical slices of the WAN, in response to user requests. CHISEL's users are the tenants of the WAN, like cloudified access network operators [8]. Optical slices are *optical tunnels*, similar in function as tunnels at higher layers of the networking stack (*e.g.,* IP-in-IP, MPLS, GRE) — they provide a bandwidth pipe between the two ends of the slice. Unlike traditional tunnels, optical slices provide guaranteed network bandwidth and quality of service between endpoints. These strict guarantees are made possible by dynamic reservation of physical resources on the optical WAN. CHISEL's slices eliminate packet switching on the slice since bulk of network delays, packet drops and performance jitter are caused at the router hops in the path [4].

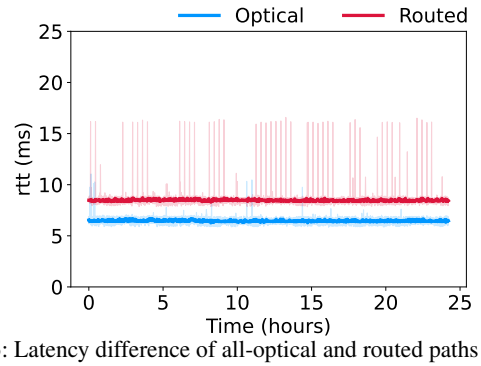


Figure 3: Latency difference of all-optical and routed paths between the same endpoints in a WAN.

Why optical slicing? We quantify the benefit of an optical slice vs. a packet switched path in a production WAN. We describe the detailed setup for this experiment in Appendix A. We use isochronous round-trip testing or *irtt* measurements at every 500 ms over the span of 24 hours to compare the latency between the same endpoints in the WAN over an all-optical path (like the ones enabled by CHISEL) and a routed path. We find that despite keeping all factors the same, *e.g.,* , the underlying fiber path, the endpoint equipment, time of day *etc.*, the all-optical path outperforms the routed path in terms of latency and jitter by a significant amount (Figure 3). Motivated by this experiment and the intuition that optical paths offer more predictable performance, we implement network slicing in the wide-area at the optical layer.

What about statistical multiplexing? CHISEL slices the WAN by carving chunks of optical spectrum, dynamically. This is similar to circuit switching at the physical layer. One might argue that packet switched paths offer a multitude of benefits to networks, like statistical multiplexing [29] over network resources and CHISEL is decidedly avoiding packet switched long-haul paths. Moreover, circuit switching necessitates the reservation of resources which can lead to inefficient utilization of the network. However, the optical layer of cloud WANs is *massively over-provisioned* by design (§3), making abundant optical spectrum available for reservation with no downside. Therefore, by using CHISEL, we are not making the network inefficient in carrying existing traffic and are additionally saving network tenants from the downside inherent to routed paths — unpredictable performance.

Why is this new? Network operators incrementally provision capacity on long-haul fiber (§3). At first glance, these well-known network capacity provisioning mechanisms seem sufficient to achieve CHISEL's goals [31]. However, CHISEL must *dynamically* carve bandwidth on fiber at both line-rate (§6) and sub line-rate granularity (§8) in operational networks without causing side-effects to existing traffic. These requirements pose harder technical challenges in terms of system design, hardware implementation and algorithmic complexity. We focus on these challenges in this work.

Our contribution. CHISEL relies on the availability of optical spectrum in operational cloud networks. We first show that optical spectrum is available in abundance in large parts of a cloud network (§3.3), making it feasible to carve it for optical slicing. Spectrum allocation on an operational network is subject to a number of complex physical constraints of signal transmission on fiber and hardware limitations. We formulate the problem of efficient spectrum slicing by encoding these constraints as an optimization objective (§4) and solve it in a few seconds for large WANs. Finally, implementing the optimal allocation of optical slices computed by CHISEL's optimization requires careful software design that works around the limitations of current optical hardware. CHISEL accelerates this by programmatically creating all-optical slices within seconds on off-the-shelf optical hardware without disrupting existing data-carrying channels (§6).

How will CHISEL enable end-to-end network slicing? Radio-access network slices are created and managed by a Network Slice Subnet Management Function (NSSMF) as per the 3GPP standard [26]. Similarly, CHISEL will interface with the *transport NSSMF* to provision WAN transport slices. Together with the RAN and core NSSMFs, CHISEL will enable end-to-end slices from the user end point to the traffic destination in next-generation mobile wireless networks.

3 Optical spectrum in long-haul fiber

Long-haul connectivity in wide-area networks is built with optical wavelengths on fiber. Cloud providers provision their WANs by laying fiber between cloud datacenters. Such fiber, often referred to as "dark fiber", is acquired by expensive manual effort of laying fiber under the ground. Once acquired, long-haul fiber is incrementally utilized for several decades.

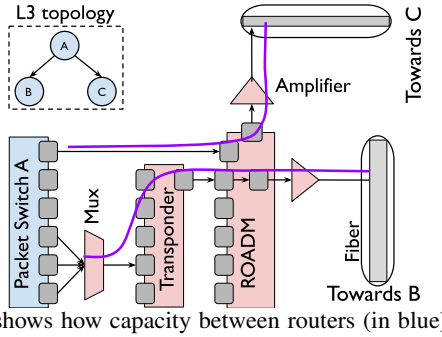


Figure 4: shows how capacity between routers (in blue) is provisioned by adding wavelengths via optical hardware (in pink). Provisioned capacity translates to spectrum allocation on optical fiber.

Capacity is provisioned incrementally. Based on the estimated capacity required between routers, cloud operators provision *wavelengths* on the fiber. Provisioning a wavelength requires connecting ports on a packet router to an optical switch or multiplexer which can *add or drop* an optical wavelength onto connected fiber (Figure 4). A wavelength is a unit of *optical spectrum* on fiber that is encoded with bits egressing from router ports. Provisioning a wavelength incurs bulk of the hardware cost of acquiring network capacity (*e.g.,* router ports, transceivers, optical switch ports) and is done incrementally as the need for more capacity arises.

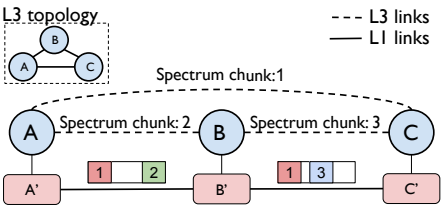


Figure 5: Spectrum allocation to provision router edges $A-B$, $A-C$ and $B-C$. Edge $A-B$ uses spectrum chunk 2 on optical span $A\prime-B\prime$. Edge $B-C$ uses chunk 3 on span $B\prime-C\prime$. Edge $A-C$ uses chunk 1 on both spans $A\prime-B\prime$ and $B\prime-C\prime$.

Optical spectrum is limited. A wavelength on fiber can have variable *spectral widths*, corresponding to the size of optical spectrum they occupy. Advances in optics and signal processing have enabled transmitting over 100 wavelengths on a single strand of fiber. However, the overall spectrum on fiber is limited to the frequencies in the *C-band* of the electromagnetic spectrum. Optical signals with frequencies belonging to the C-band and nearby suffer the lowest loss as they propagate. Therefore, most long-haul signal transmission in WANs is limited to the C-band. Figure 5 shows how multiple wavelengths occupy the optical spectrum on fiber.

3.1 Spectrum translates to network capacity

Higher layers in the networking stack abstract network connections as edges between routers, annotated with capacity. The physical network underlying this graph abstraction translates edge capacities into allocations of optical spectrum in fiber (Figure 5). The Shannon capacity [33] of a wavelength *i.e.,* the theoretical upper bound on the wavelength's capacity, is proportional to the width of spectrum allocated to the wavelength. Thus, allocating larger chunks of spectrum leads to higher data rates and vice versa.

Modulation format. Modulating the wavelength with *higher-order* modulation formats (*e.g.,* 8-QAM, 16-QAM) packs more bits per *symbol* of the wavelength, resulting in higher data rates or capacity of the wavelength. Overall, the choice of spectral width and modulation format decides the data rate of router ports that are the endpoints of a newly provisioned network edge in the logical wide-area network.

Signal quality. Sustaining higher order modulation formats requires a sufficiently high signal-to-noise-ratio or SNR — the second factor that determines the Shannon capacity of a data channel. Signal quality can be engineered by the operator only to a certain extent. Factors outside the control of the operator affect the signal quality *e.g.,* impairments to fiber, noise and power levels of amplifiers *etc.*Thus, the signal quality can often limit the achievable data rates of wavelengths.

Optical reach. A consequence of the selected modulation format is the *reach* of the optical signal. Optical reach is the longest distance a signal can travel on fiber before it needs to be *regenerated*. After traversing distances higher than the optical reach on fiber, it is essential to regenerate signals to recover data bits from the errored bits. Signal regenerations are expensive in terms of hardware and power since they require conversion of optical signals into electrical signals followed by a conversion back into optical signals. Higher order modulation formats are more susceptible to signal attenuation during transmission and therefore have shorter optical reach.

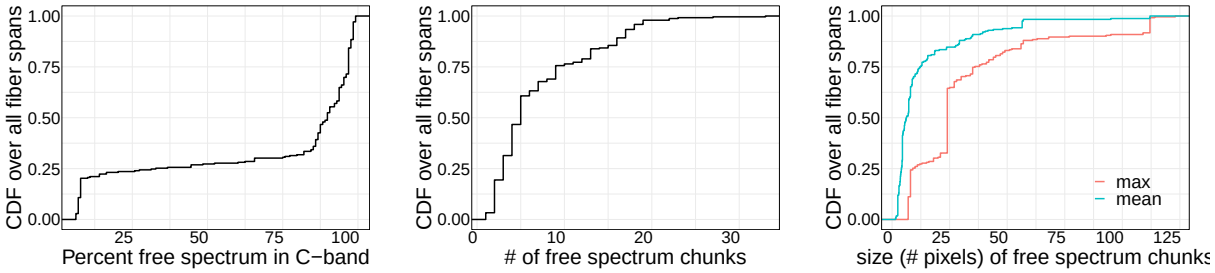
3.2 Spectrum usage in the wild

Operational networks provision wavelengths manually today. Network operators keep offline *channel maps* that map capacity between router ports to wavelengths and their corresponding spectral widths on all fiber spans in the WAN. For instance, in Figure 4 three router ports are multiplexed to feed bits into the chunk of spectrum allocated on fiber towards node B whereas a single port feeds bits to the smaller chunk of spectrum on fiber towards node C.

Ideally, spectrum channel maps should be *correct*, *efficient* and *performant*. Correct channel maps ensure that different router ports are not allocated overlapping spectrum. Efficient channel maps reduce the amount of spectrum used to provision the network as electromagnetic spectrum, especially the C-band, is limited. Performant channel maps ensure that the allocated spectrum meets the network capacity and optical reach goals. For example, an operator intends to augment the capacity between router ports in Austin, TX and New York City, NY. In this case, the provisioned wavelength should have an optical reach roughly equivalent to the distance between the two cities $\approx 2,800$ km. In Figure 5, to connect A and C *directly* at the Network layer, an operator must provision the same wavelength on both fiber spans $A\prime - B\prime$ and $B\prime - C\prime$. Maintaining *continuity* of wavelength allocations across spans is essential in fiber transmission to connect endpoints without regenerations at intermediate hops, like B.

Regenerations or OEO conversions. Cloud providers operate *point-to-point* optical networks [37]. In point-to-point optical networks, physically adjacent locations in the WAN are also adjacent electrically. In other words, all wavelengths undergo optical to electrical to optical (OEO) conversion at every hop in the network. Operators run point-to-point networks for flexibility and operational simplicity. The OEO conversion forces correction of errors that happen during transmission of signals. In such networks, capacity between router ports in Austin, TX and New York City, NY can be provisioned using different wavelengths on multiple fiber spans connecting the two cities. For instance, in Figure 5, A and C have indirect connectivity via B. This connection is enabled by two different wavelengths, 2 on $A\prime - B\prime$ and 3 on $B\prime - C\prime$.

Fully-loaded line systems. Cloud providers operate *fully-loaded* line systems [14]. In such line systems only a fraction of the optical spectrum is provisioned and connected to router ports. However, the *entire* spectrum has optical channels traversing it. The optical channels that are not connected to router ports simply carry *noise* generated by specialized equipment. Network operators fully-load their optical line systems to ensure that the network will function as needed in the future, when high network demands necessitate the allocation of the entire spectrum. The steady-state of optical spectrum in an operational cloud network appears fully utilized since operators use fully-loaded line systems. Therefore, it is important to differentiate between noise-carrying wavelengths and *data-carrying* wavelengths. Data-carrying wavelengths are the ones that contribute to spectrum utilization since they are allocated for provisioning capacity in the network.



(a) Percentage unused spectrum. (b) Number of unused spetrum chunks. (c) Size of unused spectrum chunks.

Figure 6: Figure 6a shows the percentage of optical spectrum pixels that are free across all spans in a commercial cloud provider's WAN. Figure 6b shows the number of contiguous unused chunks of spectrum across all spans in the cloud provider's network. Figure 6c shows the size of unused regions of spectrum on fiber.

3.3 Available spectrum

We measure the state of optical spectrum on fiber spans of a large commercial cloud provider. We collect which wavelengths (or frequencies) map to different ports of wide-area optical switches or reconfigurable optical add/drop multiplexers (ROADMs). Since the cloud provider runs a fully-loaded line system, the data collected from optical switches alone will not accurately inform us of the true spectrum allocation. Instead, we augment the Layer-1 data collected from ROADMs with Layer-3 network capacity information that maps router ports to ROADM ports. If we find that no router ports map to a given part of the optical spectrum, we mark that spectrum *available*. In this manner, we differentiate between noise-carrying vs. data-carrying optical channels in the WAN.

The C-band consists of frequencies in the range 191100 – 195900 Ghz. We discretize the frequency band into 128 *pixels*, each corresponding to 37.5 GHz of frequency. Using the cross-layer data, we mark which pixel is free vs. used by data-carrying wavelengths for all spans in the planet-scale cloud WAN. Since CHISEL's goal is to allocate unused portions of spectrum to slice requests, we quantify the unused spectrum on optical spans of the WAN. Figure 6a shows that 75% of optical spans have over 75% of their spectrum available. The free pixels in the spectrum are concentrated in a small number of *contiguous chunks* of free spectrum. Figure 6b shows that free pixels in over 70% of spans are concentrated in less than 10 contiguous chunks. Finally, majority of spans have free spectrum chunks that are less than 25 pixels wide (Figure 6c).

Implications for CHISEL. Our empirical analysis shows that the physical layer of wide-area networks is heavily over-provisioned in terms of optical spectrum on fiber. The main reason for the low utilization of optical spectrum is the large difference between switching capacity of state-of-the-art network routers and data-carrying capacity of modern optical fiber. Fiber can carry tens of terabits of traffic, which is a magnitude higher than the total port capacities of a router linecard. Moreover, operators incrementally deploy more router ports to provision capacity but fiber is typically acquired by multiple strands in the same conduit. Large providers can fill the the spectrum over the next several years from the time they first acquire fiber. This fits well with CHISEL's agenda — there is sufficient free optical spectrum on fiber for CHISEL to carve slices without impacting existing traffic.

4 Bandwidth slices in optical WANs

We formalize the problem of carving optical slices on *operational* optical cloud WANs. Our goal is to establish end-to-end optical slices for access network operators using the available optical spectrum in the cloud backbone.

4.1 Switching in the optical domain

CHISEL's goal is to dynamically carve optical spectrum on end-to-end wide-area paths. This task is subject to physical considerations of signal transmission on fiber. Moreover, while CHISEL's algorithm will be a part of a Layer 1 software-defined controller [9, 12, 20, 34] optical switches should be capable of efficiently implementing decisions made by CHISEL. We briefly discuss the signal transmission and hardware considerations that shape CHISEL's algorithm design.

Discrete units of spectrum. The entire spectrum on fiber is limited to the C-band and parts of the L-band. Roughly, there is 4,800 GHz of usable spectrum on fiber. We discretize the spectrum into chunks of 37.5 GHz. While fiber carries wavelengths, the decisions of routing wavelengths on different fiber paths are made by the optical switches or ROADMs. Each ROADM has a *grid* of *pixels* to represent parts of the spectrum. A pixel is one unit of spectrum that the ROADM can provision towards a wavelength that connects ROADM ports to router ports. We denote the grid of an optical switch (v) using S^v where S^v_i represents the ith pixel's state. If S^v_i is 1, it means the corresponding part of the optical spectrum is already in use and $S^v_i = 0$ implies the corresponding portion of the spectrum is available to carve slices using CHISEL.

Slicing requests. CHISEL receives a set of slicing requests from clients. Each request specifies the desired bandwidth between a pair of source and destination WAN routers. We represent every slice using the source and destination pair (sd) of the slice. We represent the set of fiber paths between source

s and d using P_{sd}. We compute the set of fiber paths using Yen's k-shortest path algorithm [40]. Each fiber path $p \in P_{sd}$ is an ordered set of optical circuit switches or ROADMs and $v \in p$ means that path p contains switch v. The variable y^p_{sd} is an indicator that path p is chosen to carve slice sd,where $p \in P_{sd}$. We use x_{sd} to refer to the vector of spectrum pixels allocated to the sd slice. The vector x_{sd} is indexed at position j by x^j_{sd}. x^j_{sd} is binary.

4.2 Optimal slice allocation

In this section, we discuss the physical, hardware and operational constraints that impact optical slicing with CHISEL.

Pick only one path for a slice request. This constraint ensures that of the possible fiber paths that can be selected for carving the sd slice, CHISEL should select only one.

$$\sum_{p \in P_{sd}} y^p_{sd} = 1, \qquad \forall sd \tag{1}$$

Using available parts of the spectrum for slices. The spectrum allocated to a slice on a fiber path must not overlap with the spectrum that has already been allocated for carrying inter-datacenter traffic in the cloud WAN. S^v is a binary vector that represents the initial state of spectrum utilization on switch v. S^v is a result of the existing capacity provisioned in the cloud network before CHISEL can carve slices. S^v is an input to CHISEL. The decision variable x_{sd} in CHISEL's optimization is a binary vector of size M, where M is the total pixels in every optical switch. x_{sd} represents pixels allocated to the sd slice. The following constraint ensures that CHISEL allocates only the available portions of spectrum to slices:

$$(y^p_{sd} = 1) \implies x_{sd} \cdot (S^v)^\intercal = 0, \qquad \forall sd, p \in P_{sd}, v \in p \tag{2}$$

Non-overlapping use of spectrum slices. The spectrum allocated to different slices by CHISEL should not overlap. We encode this constraint as:

$$\sum_{sd} \sum_{p \in P_{sd}, p \ni v} y^p_{sd} \cdot x_{sd} \leq \mathbf{1}, \qquad \forall v \in V \tag{3}$$

The constraint selects switches (x_{sd}) that are on the chosen path (y^p_{sd}) for a slice (sd) and ensures that a pixel on each switch is allocated across all slices at most once. We use $\mathbf{1}$ to present a vector of length M consisting of all 1s.

Spectral width. The spectral width of an optical slice is the portion of the spectrum assigned to it by CHISEL. We denote the spectral width of the sd slice with w_{sd}.

$$\sum_{j=1}^{j=M} x^j_{sd} = w_{sd}, \qquad \forall sd \tag{4}$$

Slice bandwidth. The bandwidth allocated to a slice is decided by two factors: (1) the chosen path for the slice and (2) the spectral width of the slice. The chosen path determines the signal quality of the received signal. The signal quality, in turn, determines which modulation format can the signal be modulated with. Higher order modulation formats can pack more bits on the channel, increasing their bandwidth. We represent the modulation format of a optical path (p) with $mod(p)$. The data rate of a slice is the product of the spectral width of the slice and the modulation format of optical path.

$$(y^p_{sd} = 1) \implies w_{sd} \cdot mod(p) \leq B_{sd}, \qquad \forall sd, p \in P_{sd} \tag{5}$$

Wavelength contiguity. Optical spectrum allocation is subject to the physical constraint of *contiguity*. As per this constraint, CHISEL must assign every slice *contiguous* parts of the spectrum on fiber. Wavelength contiguity ensures efficient use of the spectrum since every data channel on the fiber must be separated from the next with *guard bands*. Chopping the spectrum in non-contiguous smaller chunks increases the number of guard bands, wasting the spectrum available for carrying data. By allocating contiguous parts of the spectrum to slices, CHISEL minimizes the number of guard bands.

To represent this constraint we track the first pixel that transitions from used to unused states. We call this state the *toggle* state and denote it by the binary variable t^j_{sd}. Pixels after the toggle state should not be used in a slice allocation. We encode the contiguity constraint using three inequalities:

$$x^{j-1}_{sd} \leq x^j_{sd} + t^j_{sd}, \qquad \forall sd, j \in \{2, \ldots M\} \tag{6}$$

$$t^{j-1}_{sd} \leq t^j_{sd}, \qquad \forall sd, j \in \{2, \ldots M\} \tag{7}$$

$$t^j_{sd} + x^j_{sd} \leq 1, \qquad \forall sd, j \in \{1, \ldots M\} \tag{8}$$

The first inequality enforces the toggle to be 1 whenever the j'th pixel transitions from used to unused. Inequality (7) ensures that the toggle remains 1 after it is set, and (8) prevents pixels to be used after the first transition from used to unused.

What about wavelength continuity? A key physical constraint for spectrum allocation on optical fiber is called *wavelength continuity* constraint. This constraint ensures that the pixels assigned to all switches on the path for a slice should be the same. For instance, if pixels 2 and 3 are assigned to allocate a slice on one optical span of the selected path, all other spans should also use pixels 2 and 3 for the slice. A discontinuous allocation of spectrum across optical spans necessitates a conversion of the signal to the electrical domain to change the wavelength of the channel. OEO conversions are expensive and CHISEL avoids them. Instead of adding spectrum continuity as a constraint to the optimization, CHISEL embeds it in the formulation by assigning one spectrum allocation decision variable (x_{sd}) for all spans on the path of a slice. This reduces the number of decision variables and constraints, making CHISEL's optimization scale to large problem sizes.

Objective. CHISEL's goal is to maximize the bandwidth provisioned towards slices. We express the overall slice bandwidth as the total bandwidth allocated to all slices by CHISEL:

$$\sum_{sd} \sum_{p \in P_{sd}} y^p_{sd} \cdot w_{sd} \cdot mod(p) \tag{9}$$

Algorithm 1 Optical slicing with CHISEL

Inputs:

$G\langle V,E\rangle$:	optical network G, optical switches V and fiber links E
S^v:	binary vector indicating initial spectrum utilization on the optical switch v
P_{sd}:	set of fiber paths between optical terminals s, d
B_{sd}:	bandwidth request for the sd slice
$mod(p)$:	modulation format supported on path p
M:	total pixels available on optical switches

Outputs:

y^p_{sd}	$\in \{0,1\}$	1 if fiber path p in P_{sd} is selected to carve the sd slice, 0 otherwise
x_{sd}	$\in \{0,1\}^M$	binary vector represents spectrum allocated to sd
t_{sd}	$\in \{0,1\}^M$	binary vector represents toggle of x_{sd}
w_{sd}	$\in \{1,\dots M\}$	spectral width of the sd slice

Maximize: $\sum_{sd}\sum_{p\in P_{sd}} y^p_{sd}\cdot w_{sd}\cdot mod(p)$

subject to $\forall sd, p\in P_{sd}, v\in p$:

(1) $\sum_{p\in P_{sd}} y^p_{sd} = 1$

(2) $(y^p_{sd}=1) \implies x_{sd}\cdot (S^v)^\intercal = 0$

(3) $\sum_{sd}\sum_{p\in P_{sd}, i\ni v} y^p_{sd}\cdot x_{sd} \le \mathbf{1}$

(4) $\sum_{j=1}^{j=M} x^j_{sd} = w_{sd}$

(5) $(y^p_{sd}=1) \implies w_{sd}\cdot mod(p) \le B_{sd}$

(6) $x^{j-1}_{sd} \le x^j_{sd} + t^j_{sd}$ $\quad j\in\{2,\dots M\}$

(7) $t^{j-1}_{sd} \le t^j_{sd}$ $\quad j\in\{2,\dots M\}$

(8) $t^j_{sd} + x^j_{sd} \le 1$ $\quad j\in\{1,\dots M\}$

Managing spectral fragmentation. An important consideration for CHISEL is to allocate slices in a way that does not increase *spectral fragmentation* on fiber spans. Higher spectral fragmentation makes it challenging to find large enough contiguous free pixels in the spectrum. The lack of large chunks of contiguous spectrum can hamper provisioning capacity in the network. We encourage CHISEL to reduce spectral fragmentation by pushing the allocated slices to the extreme left of the spectrum on any fiber span.

$$\sum_{sd}\sum_{p\in P_{sd}} y^p_{sd}\cdot w_{sd}\cdot mod(p) + \varepsilon \sum_{sd}\sum_{p}[y^p_{sd}\cdot mod(p)\cdot \sum_{j=1}^{j=M} t^j_{sd}] \tag{10}$$

Key insight. Equation 10 sums the toggle variables for a slice on every span. Recall that the toggle variables (t^j_{sd}) are set for all pixel counts higher compared to the one where the spectrum allocation of a slice ends. Therefore, larger sum of toggle variables means the slice allocation is to the left of the spectrum since that will allow more toggle variables to be set. By encouraging the allocations to be shifted to the left of the spectrum, we aim to close gaps in allocations of CHISEL.

We trade-off the two goals *i.e.,* allocating spectrum for most slice requests and reducing spectral fragmentation using the parameter ε. We set the value of epsilon empirically to achieve high slice allocations with less spectral fragmentation in §5.4.

Implementation. Algorithm 1 summarizes CHISEL's optimization formulation for allocating slices in the WAN. We implement Algorithm 1 in Python 3 and solve it using the commercial solver, Gurobi [16]. We evaluate CHISEL on an Ubuntu 22.04.1 server with 128 cores and 1024 GB RAM.

5 Slice allocation with CHISEL

We first evaluate our implementation of CHISEL's optimization formulation (Algorithm 1) on the optical WAN of a large commercial cloud provider. We then show that our results are robust on different publicly available network topologies.

Network topologies. We work with a large commercial cloud provider to collect optical topology information in their WAN. The physical topology consists of optical switches or ROADMs and fiber spans that connect them. In addition to the cloud provider's network topology, we also evaluate CHISEL on publicly available topologies in the Topology Zoo [39]. These topologies ($G\langle V,E\rangle$) are input to an instance of Alg 1. We compute k-shortest paths between all pairs of nodes in the topology to provide possible fiber paths for slices (P_{sd} in Alg 1). We set $k=4$ for the evaluation.

Initial spectrum state. We infer the spectrum utilization on fiber spans of the commercial cloud WAN (§3.3). For each fiber span, we construct a bitmap (S^v) of 128 pixels (M in Alg 1), each pixel representing 37.5 GHz of the 4,800 GHz of spectrum in the C-band. We collect spectrum channel maps for all fiber spans in the cloud network and mark bits in the bitmap to 1 if they are allocated to a data-carrying channel. Figure 6a in §3.3 shows the percentage of unset bits in the bitmap across all fiber spans. While Topology Zoo provides several real network topologies, it does not have spectrum utilization information for them. Instead, we use the bitmaps from the cloud provider network as a distribution of spectrum allocations. For every edge in a network from Topology Zoo, we randomly sample a bitmap from the distribution to populate the spectrum bitmaps of edges in all network topologies.

Modulation formats and fiber lengths. We use geographic distance between the end points of individual fiber spans as a proxy for the lengths of fiber spans. We derive the length of fiber paths input to CHISEL by adding the lengths of individual fiber spans that compose the path. Modulation formats and the corresponding data rates of wavelengths are a function of the path length (Table 1). Thus, we assign modulation formats to all paths input to CHISEL based on their fiber path length.

Mod. format	QPSK	8-QAM	16-QAM
Data rate	100 Gbps	150 Gbps	200 Gbps
Optical reach	5,000 km	2,500 km	800 km

Table 1: Data rates and optical reach of sigal modulation on fiber.

Hardware pixel counts. State-of-the-art flexgrid ROADMs can allocate spectrum to wavelengths in increments of 6.25 GHz. Ideally, 6.25 GHz should be the width of one pixel for CHISEL. However, optical line-side ports on ROADMs operate at coarser granularity of spectrum, traditionally 37.5 GHz and above. Moreover, very fine-grained allocation of spectrum increases the number of variables that CHISEL has to contend with (x_{sd} in Algorithm 1). Given CHISEL solves a mixed-integer program, a class of problems well known for being NP-Hard, increasing the width of a pixel also improves the scalability of CHISEL. Therefore, we set each pixel to have a width of 37.5 GHz while evaluating CHISEL. We note that this is not a fundamental limitation of CHISEL but a hyperparameter that the operator can set for their network.

5.1 Efficiency of slice allocations

We generate slice requests on the cloud WAN to test what fraction of them can get allocated by CHISEL. We scale the number of slice requests input to CHISEL and observe what fraction of overall bandwidth demands can be successfully allocated on the cloud WAN.

Granularity of slice allocations. In practice, customers of CHISEL can request a variety of slice bandwidths, similar to numerologies in 5G parlance. To test this, we allow each slice request to select a bandwidth value from 50 Gbps to 200 Gbps in discrete increments of 50 Gbps. For each slice request, we randomly sample a bandwidth value from the set [50, 100, 150, 200] Gbps. Figure 7 shows the percentage of overall slice bandwidth requests were successfully allocated by CHISEL on the cloud WAN. The shaded region shows the std. deviation from the mean allocation across five instances of the experiment for each data point. For smaller number of slice requests, CHISEL can allocate nearly 100% of the requests but as we increase the number of slices beyond 50, CHISEL is limited by the availability of contiguous spectrum in the WAN. Interestingly, we find that only a handful of fiber spans are bottlenecks in allocating more spectrum.

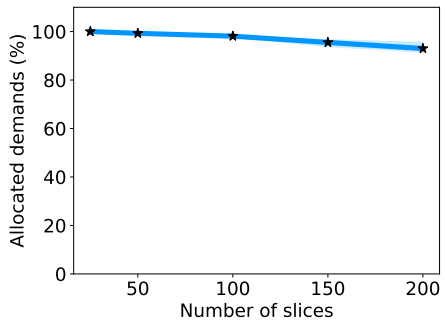


Figure 7: shows the percentage of slice bandwidth requests CHISEL can optimally allocate on the cloud wide-area network.

5.2 Scalability of the algorithm

The mixed integer program (MIP) in Algorithm 1 forms the core of CHISEL. MIPs are difficult to scale to large problem sizes. We pay attention to these scaling challenges while both designing the MIP and providing it inputs. For instance, we obviate the need to encode continuity constraints (§4) for slices and decide appropriate pixel widths to reduce the number of decision variables in Algorithm 1. For MIPs, commercial solvers like Gurobi allow setting a *MIP gap* which is the difference between the solution found by Gurobi and the optimal solution. Gurobi estimates the optimal by relaxing the integer program to a linear program which can be solved quickly. The MIP gap parameter tells Gurobi to find solutions that are close to the optimal. We set the MIP gap to 2%, a small percentage away from optimal, for the next set of experiments. We evaluate the time it takes for the optimization to converge to a solution within 2% of optimal.

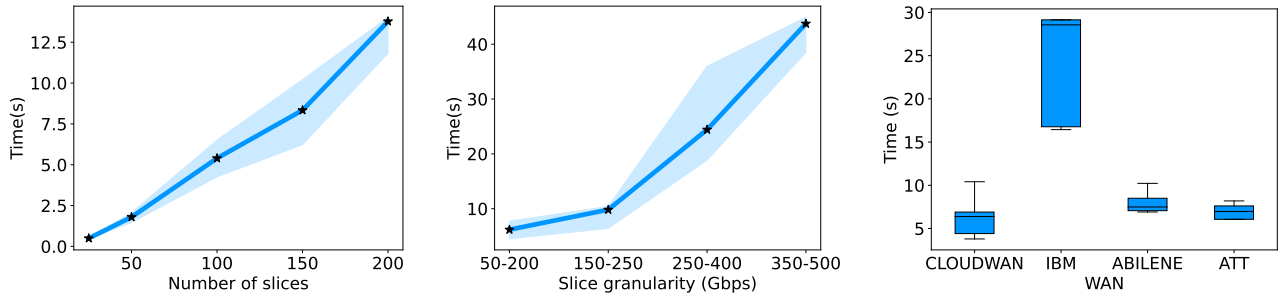
Scaling with the number of slice requests. Figure 8a shows the time needed to solve Algorithm 1 as we increase the number of slice requests. We find that CHISEL's optimization can be solved within 12 seconds on average for up to 200 slice demands, allocating optical spectrum for up to 40 Tbps across all slices. To put this in context, a well-known resource allocation problem in WANs, traffic engineering (TE), routes traffic on an existing Layer 3 network graph. The time budget to compute optimal routes in TE is nearly five minutes [24].

Scaling with slice granularities. Next, we show how long CHISEL takes to allocate slices at different granularities. We allow slices to request bandwidth from ranges [50, 100, 150, 200] Gbps, [150, 200, 250, 300] Gbps, [250, 300, 350, 400] Gbps and [350, 400, 450, 500] Gbps. Figure 8b shows that it takes longer for CHISEL to allocate large slices close to 500 Gbps per slice. This is because it is harder to find large contiguous chunks of spectrum that already do not have data-carrying channels on it vs. allocating smaller chunks needed for small slice granularity. We show the modulation formats of CHISEL-provisioned slices in Appendix B.

Scaling with network sizes. Figure 8c shows how well our findings generalize to other network topologies aside from the one of the commercial cloud provider. We find that CHISEL can scale to different realistic network topologies and slice counts while staying within 2% of the optimal solution. For all networks in Figure 8c, CHISEL takes less than 30 seconds to converge to a solution that is 2% away from the optimal.

5.3 Bounded sub-optimality for fast runtime

We evaluate how CHISEL performs in tight time budgets of cloud operators who need to provision optical slices rapidly. To do this, we give Gurobi small time budgets and evaluate the quality of solutions found for Algorithm 1 within these time budgets. We find that for small slice bandwidths, CHISEL can find solutions with less than 2% MIP Gap within 5 seconds. So, we focus on the time vs. MIP gap tradeoff for larger slice bandwidths in Figure 9. It shows how far the solution is from the optimal when the solver is given a time budget. We find that CHISEL converges to within 5% of optimal in thirty seconds even for large slice bandwidth requests (*e.g.,*



(a) Solver time vs. slice count. (b) Solver time vs. slice granularity. (c) Solver time vs. network topology.

Figure 8: Figure 8a the time taken by CHISEL to allocate different number of slices. CHISEL can efficiently compute slice allocations within 2% of optimal in less than 10 seconds for up to 200 slices. Figure 8b measures the effect of slice granularities on how long CHISEL takes to compute solutions. We show that CHISEL takes longer to find large slice ($350-500$ Gbps per slice) allocations. Figure 8c shows how the time to compute slice allocations with CHISEL changes for different network topologies. All experiments are repeated five times with different randomly sampled slices bandwidths. The shaded region around data points show the 25 and 75 percentile of results.

300–750 Gbps per slice). While there is a small improvement in MIP Gap when the time budget is increased to 60 seconds, increasing the budget past 90 seconds is not needed since the problem has converged to optimal.

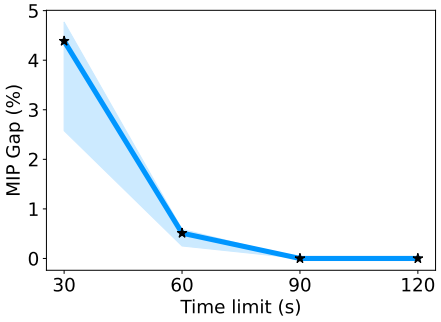


Figure 9: shows how CHISEL can tradeoff optimality for faster runtime in practical deployments.

5.4 Spectral fragmentation

In §4 we formulate two goals for CHISEL. The first (Equation 9) objective maximizes the bandwidth allocation across all slice requests. The second (Equation 10) objective maximizes the bandwidth allocation across slices while attempting to push all allocations on the low end of the optical spectrum. This formulation aims to reduce the fragmentation of spectrum by incentivizing the solver to allocate all slices in the lower frequencies. We evaluate the effect of both objectives on the amount of fragmentation and the allocated bandwidth.

We increase the value of ε in Equation 10 from 0.01 to 1 to vary the importance of fragmentation awareness in CHISEL. To show the effect of fragmentation clearly, we start with empty spectrum allocation on all spans of the cloud network and then allocate 200 slices in the range $50-200$ Gbps. Figure 10 compares the fragmentation index (Eq 11) for all spans in the network and finds that operators can set ε to minimize fragmentation while allocating all slice requests.

$$\text{F} = 1 - \frac{\text{largest block of free spectrum}}{\text{Total free spectrum}} \tag{11}$$

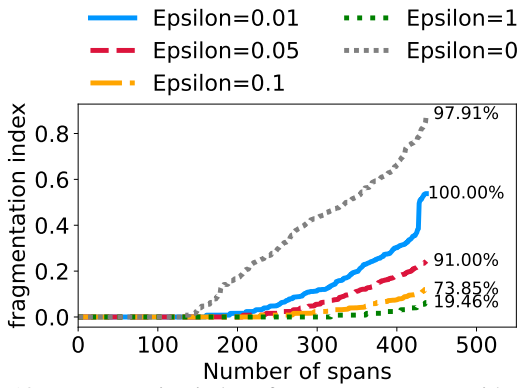


Figure 10: Fragmentation index of spectrum on spans with various degrees of fragmentation awareness in CHISEL. Each curve is annotated with percentage of allocated bandwidth across slices.

6 Hardware Evaluation

Once CHISEL computes slice allocations (§5), it must program these allocations on to the hardware. In this section, we discuss the process of physically allocating spectrum on optical fiber. We experiment with slice allocation on a laboratory hardware testbed that mimics a single optical span in a production network. We then repeat these experiments in the field in the optical WAN of an ISP in NY state.

Lab experiment setup. In the lab we replicate a typical optical span in cloud WANs using the optical line system equipment from the equipment manufacturer, VENDOR-A. The span connects two optical switches with 2,600 km of optical fiber. The two end points of the span have optical switches or ROADMs which can add/drop wavelengths on the fiber. ROADMs come in the form of linecards that fit into optical shelves which are multiple rack unit chassis devices. The 2,600 km fiber span is constructed using fiber spools. Like in a typical production setting, we connect an amplifier node

at every 80–100 km of fiber distance on this span. Overall, our testbed has 26 nodes in as many shelves where node 0 and node 25 have ROADM cards. Figure 11a shows one of the 26 shelves in our hardware testbed. We loaded this span with three bi-directional channels added and dropped at the ROADMs at the endpoints. The channels are roughly positioned at 25% 50% and 75% points of the C-band spectrum.

Field experiment setup. We verify our findings by repeating the lab tests in the production network of a regional ISP connecting universities, ISPWAN, in New York state. We were given access to ROADMs in two locations in ISPWAN. These two ROADMs are connected by a multi-hop optical path spanning $\approx$ 300 km. ISPWAN's optical line system is from the equipment manufacturer, VENDOR-B. Figure 11b shows one of shelves of VENDOR-B line system equipment used in ISPWAN.

6.1 Adding optical channels on fiber spans

We begin our experiments in the lab testbed. The optical line system in the lab consists of VENDOR-A equipment. Modifications to the VENDOR-A line system can be done using a GUI (Figure 16 in Appendix C.1). To provision a channel, we first create a software port and assign it a center frequency and spectral width. In all our lab testbed experiments, we use channels of spectral width 37.5 GHz, same as the pixel width in CHISEL's algorithm. We repeat the same steps for configuring ports on the ROADM at the far end of the span.

Equalizing power levels. After provisioning the new channel, the channel has to be built by the line system. Figure 16 in Appendix C.1 shows the GUI status while a channel is being built. Building the channel requires the proprietary vendor line system controller to communicate with all nodes in the span, including the amplifiers and the ROADMs about the newly provisioned channel. Finally, the power levels of all amplifiers on the span are *equalized* to ensure that the new spectrum on the fiber receives adequate amplification power. We note that building of the channel and equalizing its power consumes the bulk of the time to provision the channel. Depending on the vendor ecosystem and number of nodes on the span, it takes roughly five minutes to provision a channel.

6.2 Effect on existing data channels

Previous work has expressed concerns about the adverse effects of adding new wavelengths on existing channels in fiber spans [42]. Since CHISEL must provision channels to meet slicing requests, we set out to measure how addition and removal of channels on the testbed fiber span impacts the quality factor (Q-factor) of existing channels. Recall that the hardware testbed has 3 existing channels (six bidirectional channels) on it. We remove one of the channels by deleting the software port and tearing down the L1 connection between the software ports. We verify, by measuring receive power of the deleted channel at the endpoints of the span, that the channel has indeed been removed. We add the channel back soon after it has been removed. We repeat this experiment with a second channel. We find that the remaining channels on the span, regardless of their position in the spectrum, remain unaffected with both the addition and the removal of new channels (Figure 11c).

Finally, we repeat this experiment in the field. We note that ISPWAN uses a different optical line system than the one in our lab. However, this line system is also primarily programmed using a GUI. We add a wavelength at the ROADM located in CITY-B and drop the wavelength at the ROADM in CITY-A. During the process, the existing telemetry mechanism in the production network is gathering the performance statistics. Figure 11d shows the Q-factor of existing channels on the span (not including the wavelength we added). Confirming the findings from our lab experiment, the figure shows that all wavelengths on the path remain unaffected by the addition of the new channel.

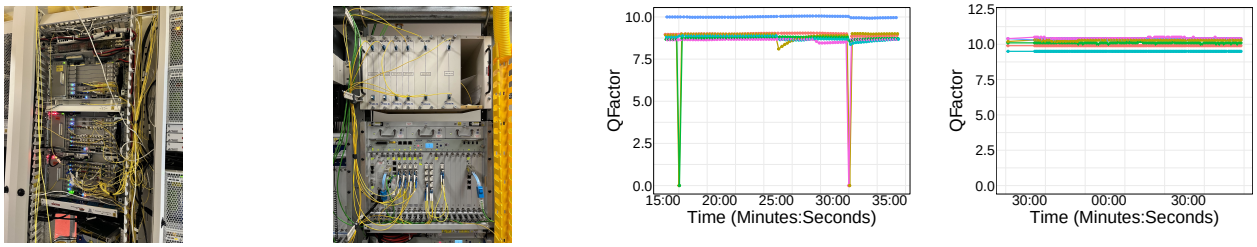
6.3 Latency of provisioning channels

We used the equipment from two major vendors of optical line systems in WANs and found that both use GUI-based management tools for creating and deleting channels. These tools only allow configuring channels and equalizing them serially at every node on the span. In our experiments, the process of creating and equalizing channels took $\approx$5 minutes for spans with moderate number of nodes on the path. We set out to investigate if creating channels programmatically would significantly reduce the delay of provisioning wavelengths.

Benefits of automating provisioning. VENDOR-B provides a REST API for configuring their line system. Seldom used in practice by network operators, we leverage the REST interface to programmatically add optical channels in a lab setup of the VENDOR-B line system. Our tool first provisions the software port on the ROADM for a new channel and configures its position in the spectrum and spectral width. Our tool uses channels of 50GHz spectral width since it was the only available width on this ROADM. It then equalizes the power on nodes in the path. We repeat the provisioning of a channel rapidly using our tool and find that on average it takes 8 seconds seconds to add a new channel. Therefore automation shaves off most of the observed latency of provisioning channels seen by network operators.

Parallel equalization of power. In an attempt to reduce the latency of provisioning a channel, CHISEL's automatic provisioning tool equalizes the channels in parallel instead of sequentially. We find equalizing in parallel can further reduce the latency of provisioning wavelengths significantly in practice. We release our tool to aid operators in rapid provisioning of channels in their networks.

Contrast with heavily-loaded spans. Our findings in §6.1 are from networks with lightly loaded spectrum, both in the lab (Figure 11c) and in the field (Figure 11d). Large portions of the spectrum in these settings were free. To ensure that our findings hold in heavily loaded optical networks, we use



(a) Optical shelf (VENDOR-A). (b) Optical shelf (VENDOR-B). (c) Q-factor (lab experiment). (d) Q-factor (field experiment).

Figure 11: Figure 11a one shelf of equipment that forms one node in the 26 node hardware testbed used by CHISEL. Figure 11c shows that the addition and removal of channels does not affect the Q-factor of existing wavelengths on the same fiber. Figure 11d shows the same effect in the field. Q-factor of the channel that is being added is not shown in Figure 11d.

CHISEL's tool to programmatically provision wavelengths that fill the optical spectrum on fiber and monitor the effect of adding these wavelengths on existing channels on the span. We provide details of this setup in Appendix C.1. Over time, we kept adding wavelengths to the span and monitoring the two channels that are connected to data sources. Figure 12 shows that even with increasing spectrum utilization, addition of new waves does not disrupt signal transmission on the existing wavelengths on the span. Figure 17 shows the fully-loaded spectrum at the end of our experiment. In summary, our experiments show that it is safe to provision wavelengths incrementally as instructed by CHISEL's algorithm.

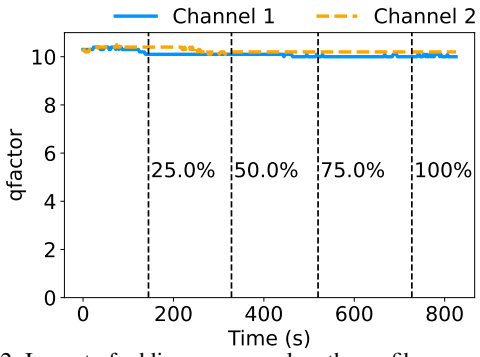


Figure 12: Impact of adding new wavelengths on fiber as a function of percentage spectrum occupied by existing channels.

7 CHISEL vs. wide-area traffic engineering

An alternative to optical slicing with CHISEL is using Layer 2 or Layer 3 tunnels for allocating bandwidth between end-points in the WAN. This approach is used by WAN traffic engineering (TE) [18, 19, 23]. Cloud providers deploy TE to efficiently use their network infrastructure [18, 19, 23]. TE systems take demands between sources and destinations as input to solve a multi-commodity flow problem that allocates demands on network *tunnels*. Network tunnels are built by tunnelling protocols like MPLS [32], IP-in-IP [35], GRE [25] *etc.*At the optical layer, cloud WANs are point-to-point which means that the tunnel abstraction of Layer-2 and above is enabled by a series of one-hop optical links [37].

We contrast the two approaches of allocating bandwidth, CHISEL vs. WAN TE, along the axis of marginal hardware expense needed to provision the bandwidth. We assume that CHISEL and network TE take the same hop paths in the network with one difference — CHISEL constructs an end-to-end optical slice over the path while TE constructs a packet switched tunnel over the path. We compare the hardware cost of allocating slices using CHISEL with that of allocating equal bandwidth using TE. We implement the most commonly used TE formulation that maximizes network throughput [18, 23]. Both the TE algorithm and CHISEL allocate the bandwidth requests and we compute the number of router ports required to provision the slice requests for both. TE tunnels carry traffic that undergoes OEO conversion at every hop by design and thus consumes router ports not only for adding (dropping) traffic at the source (destination) router but also at all intermediate hops. In contrast, an all-optical slice from CHISEL only needs router ports at the slice source and destination router.

Figure 13 shows the number of network ports used in provisioning different number of slices in the cloud WAN. We find that allocating bandwidth using CHISEL's slices not only improves performance under strict QoS requirements (Figure 3) but is also significantly cheaper in terms of the hardware cost of provisioning network bandwidth compared with cloud WAN TE. On average, CHISEL-provisioned slices consume $2.6-3.3$X fewer router ports than an equivalent TE tunnel.

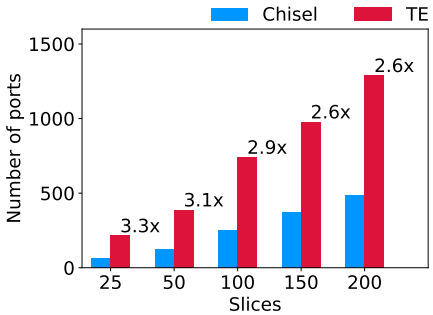


Figure 13: Hardware cost reduction using CHISEL vs. TE.

8 Giving tenants access to optical slices

Once CHISEL computes optimal allocations to slice requests (§5), it provisions the slices on the optical line system (§6).

The final step of provisioning an end-to-end slice is to give the network tenants access to the provisioned optical slice. In this section we discuss the mechanism CHISEL uses to furnish optical slices to its clients. CHISEL exposes the slice's allocated bandwidth using ports on WAN routers. Since slice allocations are dynamic, CHISEL needs a mechanism to map the slice to router ports dynamically. If this mapping is static, CHISEL would need to reserve router ports and potentially waste the allocated ports if they were not needed.

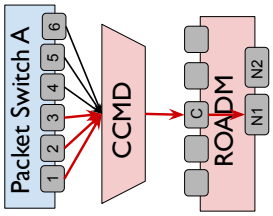


Figure 14: Mapping router ports to optical spectrum.

Dynamically mapping router ports to slices. CHISEL uses *colorless*, *directionless* multiplexers and demultiplexers (CCMDs) which connect multiple ports of WAN routers to the client port of the ROADM using coherent optical transponders. Each of these transponders are tuned to generate signals of specific wavelengths. Once CHISEL has provisioned the slice, it can program the ROADM to select wavelengths incident on the client port that correspond to the appropriate router ports. For instance, Figure 14 shows that the ROADM can select ports 1, 2, 3 to drop traffic from those ports into the allocated optical slice. While the remaining router ports (4, 5, 6) are connected to the same CCMD and ROADM client port *C*, the ROADM can select alternate routes for traffic from those ports by dropping their channels on a different network port of the ROADM (*e.g., N*2). CCMDs are common and relatively inexpensive equipment that can be leveraged by CHISEL to make allocated spectrum available on router ports.

8.1 Sub-rate optical slices with OTN switching

Our evaluation of CHISEL (§6) allocates slices at the granularity of 37.5 GHz of the optical spectrum. A slice of this spectral width can carry up to 200 Gbps of traffic depending on the signal quality of the channel in fiber, length of the fiber path and modulation format with which the channel is encoded. While we expect cloudified access networks require large optical slices for service-oriented traffic classes (*e.g.,* voice over IP, tele-presence), some clients of CHISEL might want to carve smaller chunks of the optical spectrum.

We discuss whether CHISEL can be used to to carve smaller bandwidth slices. We note that the CHISEL spectrum allocation algorithm (Algorithm 1) is agnostic to pixel width and can compute allocations of spectral widths as small as 6.25 GHz. This is in line with the the capability of state-of-the-art flexgrid ROADMs that can only allocate spectrum in multiples of 6.25 GHz pixels. The challenge of allocating spectrum at this fine granularity lies in the lower bound on the spectrum expected at the client port of a ROADM (Figure 14). The minimum width of spectrum that the client port of a ROADM can add or drop on the fiber is restricted by the minimum spectrum of light produced by optical transponders that connect router ports to ROADM client ports. This lower bound is 37.5 GHz for the line system equipment we work with.

However, the problem of carving *sub-rate* slices *i.e.,* slices that need bandwidth lower than that one a single wavelength on fiber, can be solved using *Optical transport networking* (OTN) switches. OTN switches can multiplex sub-rate channels from router ports in the time domain to feed a larger wavelength to the ROADM client port. The use of OTN switches will not change how the CHISEL algorithm functions but will change how sub-rate slices are made available to clients.

9 Related Work

We now place CHISEL in the context of related work:

Optical reconfigurability in WANs. Researchers have developed techniques to reconfigure the optical backbone to improve throughput [38] and reliability [42]. For instance, RADWAN [38] adapts data rates of long-haul links in response to changes in signal quality. ARROW [42] recovers from fiber cuts by migrating wavelengths to alternate paths. CHISEL uses similar ideas of optical reconfiguration as these systems but solves a very different challenge: that of optical spectrum slicing. Previous work in networked systems has largely ignored the problem of spectrum management, especially at the scale and time budget demanded by network slicing. OWAN [21] schedules transfers on existing optical circuits in the WAN to meet long-running bulk transfers. In contrast, CHISEL establishes new circuits in operational networks for slicing without impacting engineered traffic. The resources allocated by OWAN are optical circuits whereas the resource allocated by CHISEL is optical spectrum, similar to radio spectrum allocation in RAN slicing. As a result, challenges faced by CHISEL are around fast spectrum allocation, spectral fragmentation and hardware support for dynamic spectrum allocation in WANs.

Optical network design. ISPs have studied the design of optical networks in depth [5, 6, 7, 10, 15, 27, 28]. Shoofly [37] solves a similar problem in the context of cloud providers. Shoofly uses optical bypass to lower the cost of provisioning capacity in cloud WANs. In contrast, CHISEL is solving a runtime problem and not a design-time problem.

Spectral allocation. Optical communication researchers have studied the problem of spectral allocation and fragmentation in fiber [41]. However, they approach spectral allocation as a network capacity provisioning problem that is done infrequently. Authors of [31] discuss the problem of assigning wavelength to circuits in a traffic demand matrix without considering existing traffic on the network. In contrast, CHISEL considers existing traffic on the network and also models the hardware constraints into the algorithm. In contrast, CHISEL works in tandem with traffic engineering

and allocates spectrum at fine granularity for optical slices. Our goal with CHISEL is to not replace existing network provisioning or network traffic engineering. Instead, CHISEL introduces a new way of rapidly allocating bandwidth at the physical layer for clients who need it.

Elastic spectrum slicing. Researchers have established elastic bandwidth circuits using optical frequency division multiplexing (OFDM) modulation format and variable bandwidth cross connects (OXC) [22]. This work builds a mechanism to dynamically change the bandwidth associated with a circuit in the spectrum domain. It is complementary to our work since CHISEL's algorithm determines how much bandwidth to allocate to each circuit and can be modified to work with a different circuit switch's characteristics.

Wide-area traffic engineering TE is a related problem that allocates bandwidth along tunnels in WANs using a centralized, software-defined controller [19, 23, 36].

RAN slicing. There has been a lot of work on slicing the radio access network and algorithms for slicing with high efficiency [13]. CHISEL is complementary to RAN slicing and in fact extends the reach of RAN slicing to WANs.

10 Conclusions

We develop CHISEL, a system that creates slices of optical spectrum on fiber. CHISEL's algorithm computes bandwidth optimal slice allocations that consume $2.6-3.3$X fewer router ports compared to Layer-3 traffic engineering. CHISEL dynamically allocates spectrum slices while limiting spectral fragmentation. CHISEL programs spectral allocations automatically within seconds without impacting existing data-carrying channels on fiber. We have released CHISEL's experimental data, implementation and automation tools [3].

References

[1] 3GPP. 3GPP TS 23.501 version 16.6.0 Release 16. Technical report, 2020.

[2] 3GPP. Service requirements for enhanced V2X scenarios. Technical report, 2020.

[3] Anonymous. Chisel Code and Data. http://opticalslice.network.

[4] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. Toward formally verifying congestion control behavior. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 1–16, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Ajay Kumar Bangla, Alireza Ghaffarkhah, Ben Preskill, Bikash Koley, Christoph Albrecht, Emilie Danna, Joe Jiang, and Xiaoxue Zhao. Capacity planning for the google backbone network. 2015.

[6] M. Birk, G. Choudhury, B. Cortez, A. Goddard, N. Padi, A. Raghuram, K. Tse, S. Tse, A. Wallace, and K. Xi. Evolving to an sdn-enabled isp backbone: key technologies and applications. *IEEE Communications Magazine*, 2016.

[7] A. Brzezinski and E. Modiano. Dynamic reconfiguration and routing algorithms for ip-over-wdm networks with stochastic traffic. *Journal of Lightwave Technology*, 23(10):3188–3205, 2005.

[8] Microsoft News Center. AT&T to run its mobility network on Microsoft's Azure for Operators cloud, delivering cost-efficient 5G services at scale, 2021.

[9] Mayur Channegowda, Reza Nejabati, and Dimitra Simeonidou. Software-defined optical networks technology and infrastructure: Enabling software-defined optical network operations (invited). *J. Opt. Commun. Netw.*, 5(10):A274–A282, Oct 2013.

[10] Angela L Chiu, Gagan Choudhury, George Clapp, Robert Doverspike, Mark Feuer, Joel W Gannett, Janet Jackel, Gi Tae Kim, John G Klincewicz, Taek Jin Kwon, et al. Architectures and protocols for capacity efficient, highly dynamic and highly resilient core networks. *IEEE/OSA Journal of Optical Communications and Networking*, 2011.

[11] Cisco. What is MPLS - Multiprotocol Label Switching. https://www.cisco.com/c/en/us/products/ios-nx-os-software/multiprotocol-label-switching-mpls/index.html, (Accessed on 2021-01-20).

[12] N. Cvijetic, A. Tanaka, P. N. Ji, K. Sethuraman, S. Murakami, and T. Wang. SDN and OpenFlow for dynamic flex-grid optical access and aggregation networks. *Journal of Lightwave Technology*, 32(4):864–870, Feb 2014.

[13] A. Destounis, G. Paschos, S. Paris, J. Leguay, L. Gkatzikis, S. Vassilaras, M. Leconte, and P. Medagliani. Slice-based column generation for network slicing. pages 1–2, 2018.

[14] Mark Filer, Hacene Chaouch, and Xiaoxia Wu. Toward transport ecosystem interoperability enabled by vendor-diverse coherent optical sources over an open line system. *Journal of Optical Communications and Networking*, 10(2):A216–A224, 2018.

[15] Jennifer Gossels, Gagan Choudhury, and Jennifer Rexford. Robust network design for ip/optical backbones. *IEEE/OSA Journal of Optical Communications and Networking*, 11(8):478–490, 2019.

[16] Gurobi. GUROBI Optimization. https://www.gurobi.com/, (Accessed on 2019-10-02).

[17] Pete Heist. IRTT (Isochronous Round-Trip Tester) . https://github.com/heistp/irtt, 2021.

[18] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with Software-driven WAN. *SIGCOMM*, 2013.

[19] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined wan. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 74–87, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Chris R. Jackson, Reza Nejabati, Fernando Agraz, Albert Pagès, Michael Galili, Salvatore Spadaro, and Dimitra E. Simeonidou. Demonstration of the benefits of SDN technology for all-optical data centre virtualisation. *Optical Fiber Communication Conference*, page Tu3L.3, 2017.

[21] Xin Jin, Yiran Li, Da Wei, Siming Li, Jie Gao, Lei Xu, Guangzhi Li, Wei Xu, and Jennifer Rexford. Optimizing bulk transfers with software-defined optical WAN. *SIGCOMM*, 2016.

[22] Bartłomiej Kozicki, Hidehiko Takara, Yukio Tsukishima, Toshihide Yoshimatsu, Kazushige Yonenaga, and Masahiko Jinno. Experimental demonstration of spectrum-sliced elastic optical path network (slice). *Opt. Express*, 18(21):22105–22118, Oct 2010.

[23] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. Decentralized cloud wide-area network traffic engineering with BLASTSHIELD. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 325–338, Renton, WA, April 2022. USENIX Association.

[24] Umesh Krishnaswamy, Rachee Singh, Paul Mattes, Paul-Andre C Bissonnette, Nikolaj Bjørner, Zahira Nasrin, Sonal Kothari, Prabhakar Reddy, John Abeln, Srikanth Kandula, Himanshu Raj, Luis Irun-Briz, Jamie Gaudette, and Erica Lan. OneWAN is better than two: Unifying a split WAN architecture. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 515–529, Boston, MA, April 2023. USENIX Association.

[25] Tony Li, Dino Farinacci, Stanley P. Hanks, David Meyer, and Paul S. Traina. Generic Routing Encapsulation (GRE). RFC 2784, March 2000.

[26] Affirmed Networks. Network slice management . Technical report, 2020.

[27] P. Papanikolaou, K. Christodoulopoulos, and E. Varvarigos. Joint multi-layer survivability techniques for ip-over-elastic-optical- networks. *IEEE/OSA Journal of Optical Communications and Networking*, 9(1):A85–A98, 2017.

[28] P. Papanikolaou, K. Christodoulopoulos, and E. Varvarigos. Optimization techniques for incremental planning of multilayer elastic optical networks. *IEEE/OSA Journal of Optical Communications and Networking*, 10(3):183–194, 2018.

[29] Larry L. Peterson and Bruce S. Davie. *Computer Networks, Fifth Edition: A Systems Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[30] Petar Popovski, Kasper F. Trillingsgaard, Osvaldo Simeone, and Giuseppe Durisi. 5G Wireless Network Slicing for eMBB, URLLC, and mMTC: A Communication-Theoretic View. In *IEEE Access*, 2018.

[31] R. Ramaswami and K.N. Sivarajan. Routing and wavelength assignment in all-optical networks. *IEEE/ACM Transactions on Networking*, 3(5):489–500, 1995.

[32] Eric C. Rosen, Arun Viswanathan, and Ross Callon. Multiprotocol label switching architecture, January 2001. RFC 3031.

[33] Claude E. Shannon. Two-way communication channels. 1961.

[34] D. Simeonidou, R. Nejabati, and S. Azodolmolky. Enabling the future optical Internet with OpenFlow: A paradigm shift in providing intelligent optical network services. *International Conference on Transparent Optical Networks*, pages 1–4, June 2011.

[35] W. Simpson. IP in IP Tunneling. RFC 1853, October 1995.

[36] Rachee Singh, Sharad Agarwal, Matt Calder, and Paramvir Bahl. Cost-effective cloud edge traffic engineering with cascara. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 201–216. USENIX Association, April 2021.

[37] Rachee Singh, Nikolaj Bjørner, Sharon Shoham, Yawei Yin, John Arnold, and Jamie Gaudette. Cost-effective capacity provisioning in wide area networks with shoofly.

In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 534–546, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Rachee Singh, Manya Ghobadi, Klaus-Tycho Foerster, Mark Filer, and Phillipa Gill. Radwan: Rate adaptive wide area network. ACM SIGCOMM, August 2018.

[39] The Internet Topology Zoo . The Internet Topology Zoo. http://www.topology-zoo.org/dataset.html, (Accessed on 2013-03-02).

[40] Jin Y. Yen. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quarterly of Applied Mathematics*, 27:526–530, 1970.

[41] Yawei Yin, Mingyang Zhang, Zuqing Zhu, and S. J. B. Yoo. Fragmentation-aware routing, modulation and spectrum assignment algorithms in elastic optical networks. In *Optical Fiber Communication Conference/National Fiber Optic Engineers Conference 2013*, page OW3A.5. Optica Publishing Group, 2013.

[42] Zhizhen Zhong, Manya Ghobadi, Alaa Khaddaj, Jonathan Leach, Yiting Xia, and Ying Zhang. Arrow: Restoration-aware traffic engineering. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 560–579, New York, NY, USA, 2021. Association for Computing Machinery.

A Optical vs. Electrical paths

In this section we provide an in-depth detail about the setup for Figure 3 in §2. For this experiment, we gained access to two machines connected to two routers in ISPWAN, a NY state based WAN. We established two routes between the two routers, one that goes over a routed path and one that is an all-optical path. The routed path has 4 intermediate routers. The all-optical path bypasses conversion to the electrical domain entirely. The length of the fiber path in between these routers is the same — roughly 500 km.

We set out to contrast the performance of the two paths using isosynchronous RTT measurements [17]. We start an irtt server on one of the machines. We start measurement of both paths from the other machine at the same time and run it for roughly 24 hours. We measure every 500 ms and track the mean and standard deviation of RTT between the machines. Figure 3 in §2 shows that the routed path performs worse compared to the all-optical path consistently. Moreover, the jitter on the routed path is significantly higher than the optical path. This paves the way for CHISEL's design. CHISEL implements slicing in the wide-area by carving chunks of the optical spectrum.

B CHISEL algorithm evaluation

In this section we characterize the slices carved by CHISEL. This is in addition to the experiments in §5.

B.1 Modulation formats of slices

Figure 15 shows the modulation formats of slices allocated by CHISEL. Most slices are short enough to have the modulation format of 8-QAM (150G), some slices with shorter fiber paths have the modulation format 16-QAM (200G) and some have the modulation format of QPSK (100G).

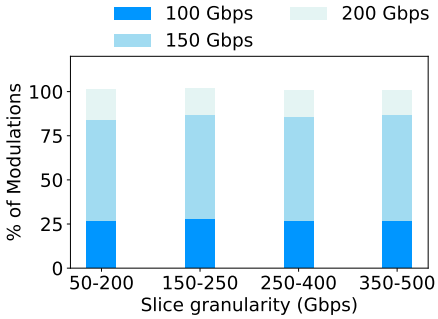


Figure 15: Modulation formats of slices.

B.2 Comparison with TE

We provide a detailed description of the setup for comparing CHISEL with wide-area traffic engineering in §7. We evaluate TE as an alternative to CHISEL on the same network graph with the same set of slice requests. The slice requests between router pairs and their corresponding bandwidths form the demand matrix for TE. We compute k-shortest paths for both TE and CHISEL. For TE, we solve the most common optimization formulation — the max throughput edge formulation [18].

TE computes allocations along paths it considers tunnels and CHISEL allocates spectrum along paths it considers optical slices. While both can meet the demands, we compare the number of router ports the two approaches will use across the entire network. We show in Figure 13 that TE requires a significantly larger number of router ports. This large difference in ports consumed by CHISEL and TE arises from the fact that TE tunnels are constructed over point to point optical links in the WAN. Therefore, traffic carried over TE tunnels consumes router ports at ingress as well as egress at each intermediate router hop in the tunnel.

C Hardware Evaluation

We evaluated CHISEL using the optical line system from two major equipment vendors: VENDOR-A and VENDOR-B. We found that line systems from both vendors were primarily programmed in the field using graphical interfaces.

C.1 VENDOR-A hardware testbed

The VENDOR-A lab testbed consisted of a point-to-point optical link typical of cloud WANs. The two end points of

the link were ROADMs with add/drop capability. In between the two ROADMs were amplification sites. Overall, the two ROADMs were connected using fiber spools of 2,600 km. The testbed took 26 shelves in our lab and Figure 11a shows one of the shelves. Configuration on the VENDOR-A line system was done over a GUI. Figure 16 shows the status of the GUI as we added a channel between the ROADMs and waited for the line system to *build* the channel.

C.2 VENDOR-B hardware testbed

We used the VENDOR-B line system in our field experiment with sysname. This line system had a GUI similar to VENDOR-A's GUI for configuration but it also provided a REST API to make the same configuration.

We also used VENDOR-B line system in the lab setting to test automatic addition and removal of optical slices through code. To do this, we created an optical span in the lab with two amplifiers and ROADM ports on each end of the span. This span had two legitimate light sources and configured channels. Our goal was fully populate the spectrum on this span. To do this, we used an amplifier as a source of noise channels to the ROADM. Once we had the amplifier setup as a light source, we developed the software tool to populate the entire spectrum on the span.

C.3 Fully loading the spectrum

We automated the creation of slices on VENDOR-B ROADM cards using the REST interface exposed by the card. The creation of a slice involves three main steps, first, we create a channel on the frequency picked by CHISEL in either the *add/drop* or *passthrough* modes, second, we enable the components of the channel so that they start receiving or sending signals and finally, we equalize the channel. We configure the terminal ROADMs in the setup in add/drop mode. the passthrough mode is for the intermediate ROADMs in the path to enable light to pass through them.

We now describe the process in detail for a single ROADM. The channel creation step invokes a POST request with the mode of the channel which can either be add/drop or passthrough and the spectrum frequency picked by CHISEL. In the second step we issue a GET request to list the components of the channel and enable them to be in service. Finally, we make a GET request to start the equalization process. During this, we periodically (1s) monitor the power level of the newly provisioned wavelength using a different GET endpoint. The first two steps complete within 2 seconds and the third step completes within 5-7 seconds.

We run the script in a loop to completely fill the spectrum of the ROADM form VENDOR-B. We can see in figure 17 that all the channels that have been provisioned are equalized to have power higher than -16dBm.

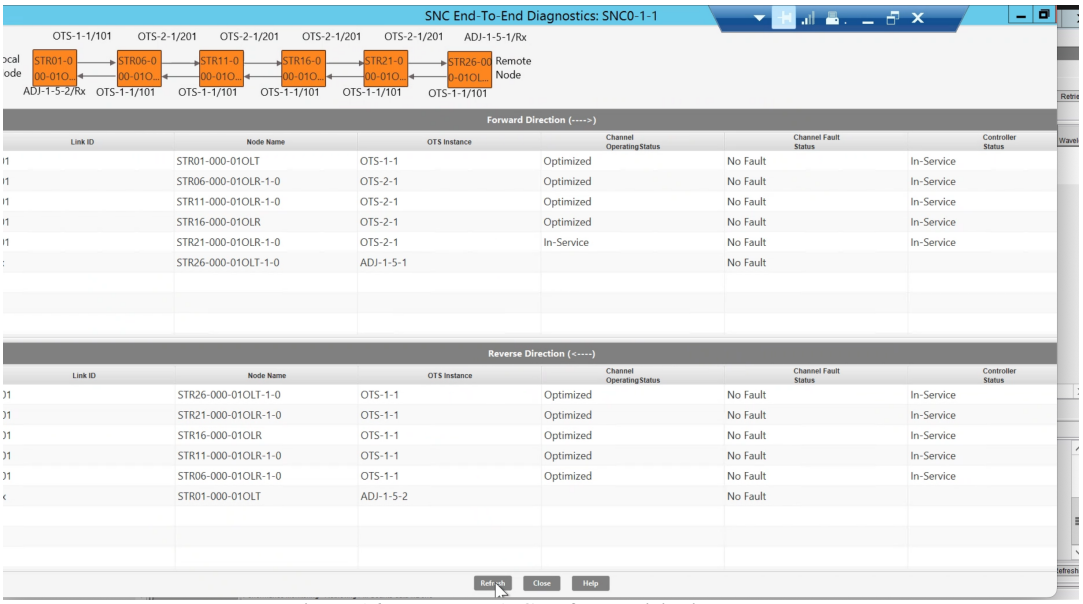


Figure 16: VENDOR-A GUI for provisioning waves.

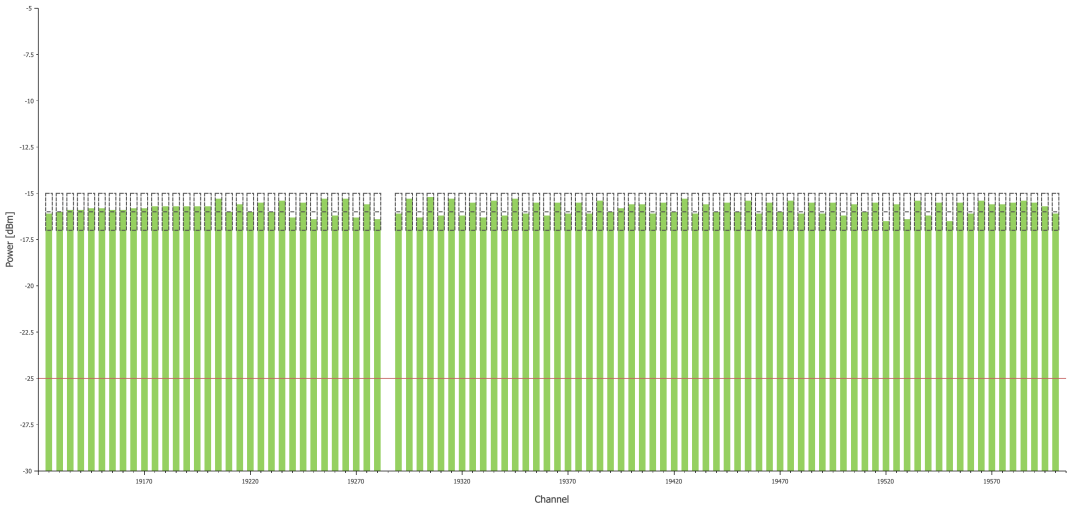


Figure 17: GUI on the ROADM showing that the complete spectrum is filled. The power levels of all the created channels are around -16dBm. The power levels of existing channels carrying data are not affected throughout the experiment.

LuoShen: A Hyper-Converged Programmable Gateway for Multi-Tenant Multi-Service Edge Clouds

Tian Pan[†*], Kun Liu[†*], Xionglie Wei[†*], Yisong Qiao[†*], Jun Hu[†], Zhiguo Li[†], Jun Liang[†], Tiesheng Cheng[†], Wenqiang Su[†], Jie Lu[†], Yuke Hong[†], Zhengzhong Wang[†], Zhi Xu[†], Chongjing Dai[†], Peiqiao Wang[†], Xuetao Jia[†], Jianyuan Lu[†], Enge Song[†], Jun Zeng[†], Biao Lyu[‡†], Ennan Zhai[†], Jiao Zhang[◇], Tao Huang[◇], Dennis Cai[†], Shunmin Zhu[⋆†□]

[†]*Alibaba Cloud* [‡]*Zhejiang University* [◇]*Purple Mountain Laboratories* [⋆]*Tsinghua University*

Abstract

Edge clouds are expected to be a key revenue growth driver for cloud vendors in the next decade; however, simply replicating the network infrastructure for the public cloud to the edge experiences deployment issues. At the edge, the challenge for cloud network design is to deliver the required performance under the stringent restrictions of hardware budget and deployment footprints, while retaining functionality equivalence. To this end, we propose *LuoShen*, a hyper-converged gateway for multi-tenant multi-service edge clouds by consolidating the entire cloud network infrastructure into a 2U server switch with a P4-centric architecture. At the data plane, LuoShen conducts pipeline folding and fits the original overlay and underlay devices into the switch pipeline via meticulous on-chip resource budgeting. At the control plane, LuoShen relies on BGP peering to ensure inter-component reachability. LuoShen achieves 1.2Tbps throughput and reduces the upfront cost, deployment size and power usage by 75%, 87%, 60%, compared with the original cloud network architecture. It has been deployed in Alibaba Cloud at hundreds of edge sites.

1 Introduction

The past decade has witnessed the rise of the public cloud to reshape the global IT infrastructure [10]. The success of the public cloud lies in a win-win economic model: cloud vendors save money by bulk purchasing computing resources at lower costs, while cloud customers utilize the shared resources without investing in their own expensive hardware in a pay-as-you-go manner [28, 32, 49]. Following this model, we have built 28 public cloud regions globally for worldwide service coverage [6]. Recently, however, we have seen a surge in customer requests to provide cloud infrastructure close to their sites to address evolving needs, such as ultra-low-latency applications like machine learning inference [56], local data processing of large volumes of data [52], data residency due to security or data sovereignty [43] and the cloudification of telecom infrastructure [23, 29]. These requirements can certainly be addressed by on-premises data centers; however, not everyone can afford to build their own. To meet local processing needs, we start to build edge clouds near our customers with products like Local Region [2] and Cloud Box [1, 3].

To offer customers the same product experience as provided by the public cloud and reduce the time-to-market cycle of our edge clouds, replicating the existing public cloud architecture to the edge is a wise strategy. In our public cloud, to manage traffic bursts from various cloud services and accommodate large forwarding tables due to multi-tenancy, we deploy different roles of gateway clusters for handling different cloud services [41]. However, when extending such "role-splitting" gateway architecture to the edge, we've encountered several deployment issues. The first is how to fit the entire network infrastructure within a constrained space. For example, the Cloud Box condenses the entire cloud infrastructure into a 42U server cabinet. If the network infrastructure consumes too much space, there will be little room left for the server payload, which will reduce the available VMs for sale as well as the vendor's revenue. The second is how to save upfront and operational costs without economies of scale [49]. A typical public cloud region can have tens of thousands of servers, thus the costs of the network infrastructure can be spread over the vast number of servers. However, such economies of scale diminish in a single edge cloud as the network infrastructure constitutes a substantial proportion of its upfront cost due to the much reduced server payload. Moreover, as the number of edge clouds grows rapidly, such cost inefficiency will be magnified many times. The third is how to provide the required stable performance in extreme cases. Although the traffic volumes and the table sizes are significantly reduced at the edge, high-bandwidth traffic and heavy-hitter flows may still occur in some edge cloud use cases, *e.g.*, high-bandwidth traffic directed from the local edge cloud to the remote public cloud, IoT traffic aggregation into a single heavy-hitter flow after being tunneled. Besides, the cloud network infrastructure in production should be durable for a longer service time to save development expenses and adapt to future traffic growth.

[*]These authors contributed equally to this work. [□]Corresponding author.

To address the above issues, we propose *LuoShen*, a hyper-converged programmable gateway for multi-tenant multi-service edge clouds, which *provides the required performance under the stringent restrictions of hardware budget and deployment footprints, while retaining functionality equivalence*. Unlike the expensive role-splitting gateway architecture that uses separate underlay/overlay devices to handle different cloud services like VM-VM (same VPC), VM-VM (different VPC), VM-Cross-region-VM, VM-IDC/IDC-VM, VM-Internet/Internet-VM and SLB, LuoShen fits the entire cloud network infrastructure into a *2U* server switch with a "*P4-centric*" design. That is, all stateless cloud network functions are converged into the Tofino pipeline, with the remaining stateful processing handled by the CPU and accelerated by the FPGA. All traffic will traverse through the Tofino pipeline before being forwarded to the external networks or distributed to the CPU/FPGA by the converged underlay devices. At the data plane, we propose a novel pipeline layout with pipeline folding and carefully budget the on-chip resources to house all the major cloud network functions. At the control plane, we achieve resource isolation, efficient table provisioning and inter-component BGP peering for coexistence of different components. To save the development costs, retain the system stability and shorten the time-to-market cycle, we extensively reuse mature code from the existing public cloud infrastructure. LuoShen has been deployed in Alibaba Cloud for over two years at hundreds of edge sites. We share the experiences and lessons from its development and deployment.

Our major contributions are summarized as follows.

- LuoShen is the world's first hyper-converged gateway disclosed for multi-tenant multi-service edge clouds. It follows a p4-centric architecture and achieves a good balance of performance, costs and deployment footprints.
- At the data plane, we propose techniques such as pipeline folding, pipe/table bypass, on-chip resource budgeting to maximize the table convergence density in the Tofino.
- At the control plane, we reserve multiple configuration channels, and conduct BGP peering with hot standby for inter-component reachability and high availability.
- LuoShen achieves 1.2Tbps throughput and reduces the upfront cost, deployment size and power usage by 75%, 87%, 60%, compared with the role-splitting architecture.

2 Background and Motivation

In this section, we introduce the cloud network infrastructure in Alibaba Cloud over the years, followed by the issues we've encountered when mirroring the infrastructure at edge clouds.

2.1 VPC Network Infrastructure

Virtual private clouds in the public cloud. The public cloud vendors serve a massive number of tenants with a shared infrastructure, where tenant isolation is essential to ensure that their resources are segregated and secure, *e.g.*, one tenant's traffic should be invisible to any other tenant [40]. In addition, some tenants own lots of VMs distributed worldwide but still want to manage them in a unified network address space. To this end, the cloud vendors are expected to virtualize a flat address space for each tenant, hiding the physical network intricacies [26]. To satisfy the isolation and virtualization needs, virtual private clouds (VPCs) [55] are created in the public cloud. The VPC multiplexes the underlying resources and offers a logically isolated address space, and each VPC is uniquely identified by its VNI. Today, overlay protocols, such as VXLAN [37], NVGRE [24], GENEVE [27], are used for VPC implementation. They leverage tunneling to stretch virtualized networks over an underlying L3 network within or across geo-distributed data centers.

Networking requirements of VPCs. A tenant's VMs need to cooperate for delivering scalable cloud services and this produces the VM-to-VM traffic. These VMs can reside in one VPC as the simplest deployment case. They can also reside in multiple VPCs within the same region as some tenants wish to isolate different parts of their infrastructure into different VPCs for more precise access control. A tenant's VPCs can even reside in multiple regions as top tenants deploy VMs across geo-distributed data centers for global service delivery. To this end, the cloud vendors need to address the communication requirements between VMs in the same VPC, in different VPCs within the same region, and across regions. For enterprise customers who use public cloud resources, the connectivity between their on-premises data centers (IDCs) and their VPCs needs to be established. For cloud services through the Internet, to allow Internet traffic to enter into VPCs and vice versa, the connectivity between the Internet and the selected VPCs is also needed. Besides, to handle the massive scale and rapid growth of cloud traffic, either from the Internet (north-south traffic) or from within the data centers (east-west traffic), horizontal scaling of VMs is required and the incoming traffic needs to be directed to the load balancers first, before reaching the backend VMs [18, 39, 42, 57].

Roles of gateways/load balancers in the VPC network. As mentioned, traffic and address space isolation is achieved by assigning one or more VPCs to each tenant, where traffic is isolated within a VPC by default, unless the routes to the external networks are explicitly added to the cloud gateway [31] at the VPC border. Therefore, the first role of a gateway is to route outbound traffic from the local VPC to the external networks and vice versa. To achieve this, the gateway will query its routing table with the VNI of the local VPC and the destination VM IP as the key to obtain the VNI of the next-hop VPC/tunnel if the external network is another virtual network. Besides, traffic will be forwarded across the underlying physical network when being tunneled between two adjacent virtual network devices (*e.g.*, vSwitch [44] or cloud gateway). Therefore, the second role of a gateway is a virtual tunnel endpoint (VTEP), where the outer header encapsulation or decapsulation is conducted when relaying traffic onto or off a physical network domain. When tunneling a packet from the local gateway to a remote VTEP across a physical network,

Table 1: Major cloud services and the corresponding traffic routes in Alibaba Cloud's VPC network infrastructure.

Cloud services	Traffic routes
VM-VM (same VPC)	VM-vSwitch-VGW-vSwitch-VM
VM-VM (different VPCs)	VM-vSwitch-VGW-vSwitch-VM
VM-Cross-region-VM	VM-vSwitch-TGW-Cross-region-TGW-vSwitch-VM
VM-IDC	VM-vSwitch-TGW-CSW-IDC
IDC-VM	IDC-CSW-TGW-vSwitch-VM
VM-Internet	VM-vSwitch-IGW-Internet
Internet-VM	Internet-IGW-vSwitch-VM
Internet-LB-Service	Internet-IGW-SLB-vSwitch-VM
VM-LB-Service	VM-vSwitch-VGW-SLB-vSwitch-VM

the outer SIP will be the gateway's physical IP, the outer DIP will be the remote VTEP's physical IP, and the VNI of the tunnel will also be encapsulated. Furthermore, as most legacy IDCs are non-virtualized (*e.g.*, bare-metal servers) [11, 58], protocol translation is needed for connecting IDCs and VPCs. Similarly, address translation is also required when the gateway provides external connectivity to the Internet. For traffic to be load balanced, it will be delivered to the gateway first, then the gateway will route the traffic to the load balancer. For a better understanding of our design, we elaborate on our gateway's different use cases (see Table 1) as follows.

①VM-VM (same VPC): If two VMs are in the same VPC, inter-VM traffic will first query the VXLAN routing table, confirming that the destination VM is in the local VPC. Then, traffic will be forwarded to the physical server hosting the destination VM with the server IP obtained by querying the VM-NC mapping table using the local VNI and the VM IP.

②VM-VM (different VPCs): If two VMs are in different VPCs within the same region, inter-VM traffic will query the VXLAN routing table twice. In the first pass, we will obtain the VNI of the VPC containing the destination VM. In the second pass, we use the obtained VNI to query the routing table again and the remaining procedure is the same with ①.

③VM-Cross-region-VM: For cross-region VM communication, traffic from the source VM will first query the VXLAN routing table at the local gateway to obtain the VNI of the cross-region tunnel and the physical IP of the remote gateway. After that, traffic will be tunneled to the remote gateway, where its VXLAN routing table will be queried for the VNI of the remote VPC containing the destination VM. The remaining procedure is the same with the second pass of ②.

④VM-IDC: For communication between a virtual network and a non-virtualized network, a specialized device is placed at the border of the public cloud to decapsulate the packets from the virtual network and make the VXLAN-to-VLAN translation if needed. On receiving the outbound traffic from VMs to IDCs, the gateway will first query the VXLAN routing table to tunnel the traffic to the specialized device.

⑤IDC-VM: For inbound traffic from IDCs to the public cloud, the specialized device at the border of the cloud will encapsulate the incoming packets with VXLAN headers and then tunnel them to the gateway of the destination VM. The remaining procedure is the same with ②.

⑥VM-Internet: The VMs in the cloud use private addresses while accessing the Internet needs public addresses. Therefore, the outbound traffic to the Internet needs to query the SNAT table (key: VM IP, value: EIP) at the local gateway to obtain the public address EIP. Then, the gateway will replace the VM IP (*i.e.*, inner SIP) with the EIP and decapsulate the tunnel header before sending the traffic to the Internet.

⑦Internet-VM: The inbound traffic from the Internet will be tunneled to the VM via the gateway. The tunnel encapsulation requires the destination VM IP, the VNI of the VPC containing the VM and the physical address of the server hosting the VM, which can be obtained by querying the DNAT table at the gateway using the EIP (*i.e.*, DIP of the packet).

⑧Internet-LB-Service: Traffic to be load balanced will first hit the gateway before reaching the load balancer. For traffic from the Internet, it will query the DNAT table at the gateway and then be encapsulated and tunneled to the load balancer.

⑨VM-LB-Service: For traffic from the VMs in the cloud (*e.g.*, to use database hosted on servers behind a load balancer), it will query the VXLAN routing table and VM-NC mapping table at the gateway and then be tunneled to the load balancer.

Alibaba Cloud's solution for scalable VPC networking. The challenges of building a scalable VPC network infrastructure lie in many aspects. In a typical cloud region, there are tens of thousands of servers, equipped with 25G/100G NICs, contributing to dozens of Tbps traffic to the gateway. The gateway contains several major tables including the VXLAN routing table, the VM-NC mapping table and the SNAT/DNAT table. In our cloud region, there are O(1M) VPCs and O(1M) VMs, leading to a very large VXLAN routing table, VM-NC mapping table and SNAT/DNAT table. Fitting these tables into an x86 server may not be a serious problem; however, if you want to make hardware acceleration to combat the rapid growth of cloud traffic, the hardware's on-chip memory can easily be exhausted [41]. In Alibaba Cloud, we adopt a *centralized* gateway model that the traffic from the source VM will first be tunneled from its vSwitch to the gateway, where the next-hop forwarding decision will be made. That is, the gateway needs to handle all kinds of traffic as listed in Table 1, leading to a sophisticated packet processing pipeline under ultra-high traffic rates. Moreover, as traffic volumes and cloud services change rapidly, the infrastructure needs to be elastic and flexible. Finally, as the central hub of east-west and north-south traffic, the stability of the gateway is equally important as its performance since failures of the gateway will affect a wide range of tenants and their services. To address these challenges, we have the following architectural designs.

The first design is *separation of underlay and overlay network devices*. Fig. 1 shows the VPC network infrastructure in Alibaba Cloud. There are underlay devices such as SW, LSW, BSW and overlay devices such as vSwitch, SLB and gateways (vendor-specific acronyms are defined in Table A1 in §D). The underlay devices provide underlying network connectivity on top of which overlay tunnels are built. The traffic through the overlay tunnels carried by the underlay devices includes ①east-west traffic between VMs within a region (via

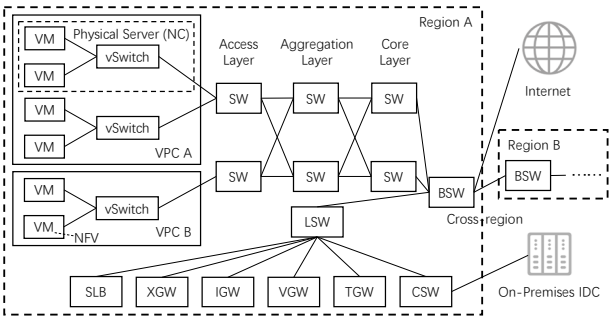


Figure 1: In Alibaba Cloud, different roles of gateway clusters are deployed to forward traffic of different cloud services.

SW); ②traffic routed to the gateways/SLB (via SW, BSW and LSW); ③traffic routed to the external networks (*e.g.*, Internet, via LSW and BSW). The overlay devices provide tenant isolation, virtual routing/forwarding, and tunnel encap/decap. The separation of the underlay and overlay enables rapid cloud service iteration. When cloud services need a change, we only need to update the overlay devices without reconstructing the underlay infrastructure. Besides, caching VM-to-VM routes at vSwitches enables bypassing overlay gateways for faster routing directly through the scalable underlay SW fabric [54].

The second design is *deploying different roles of gateway clusters for different cloud services*. To handle cloud-scale traffic and huge forwarding tables caused by multi-tenancy, the logically centralized gateway is further split into multiple gateway clusters, each dealing with a particular cloud service. For example, *VGW* handles VM-to-VM traffic within a region, *TGW* handles VM-to-VM traffic across regions as well as traffic between VMs and IDCs, *IGW* handles Internet traffic, *SLB* is for server load balancing, *CSW* is a specialized device for VXLAN-to-VLAN translation and vice versa, *XGW* holds all the gateway tables for fallback traffic processing. In the early days, all gateways are based on x86 servers and clustered for scalable performance. To prevent CPU overload due to traffic bursts, some gateways are accelerated via programmable switches [41]. To increase elasticity and flexibility, some stateful gateways, such as NAT and VPN, reside on servers in the form of NFV instances [47]. All the gateways will advertise their VIPs to the underlay network and the vSwitch will tunnel the VM traffic to different gateways by encapsulating the traffic with different outer DIPs based on the VNIs and the inner DIPs. Table 1 shows some major cloud services and their corresponding traffic routes. In the "role-splitting" gateway architecture, failures will be isolated within a single gateway cluster. If horizontal splitting of tenants [41] is enabled, failures will be further isolated within a single physical gateway in the cluster. Similarly, the development, deployment and update of different gateway clusters can also be decoupled.

2.2 Rise of Edge Clouds

Extending public cloud services to the edge. Alibaba Cloud has built 28 public cloud regions across the globe. However, we are increasingly requested by more and more customers to provide cloud infrastructure close to their locations for the following use cases. The first is ultra-low-latency applications such as cloud gaming, live streaming, manufacturing control and machine learning inference [56], which cannot afford the large latencies of task processing in the remote cloud. The second is local data processing needs for large volumes of data generated at the edge, *e.g.*, data uploaded by IoT nodes, which can save network bandwidth usage between the edge and the remote cloud. The third is data residency needs due to security or data sovereignty [43], *e.g.*, financial services. Another potential market is the cloudification of telecom infrastructure, *e.g.*, cloud-native 5G core and RAN [23, 34]. To meet the above needs, we replicate our public cloud infrastructure to the edge, expecting to continue its past success. In this way, there is no need for our customers to build and operate their own on-premises IDCs and they can use the same tools and APIs that they use in the public cloud to manage their edge clouds. Specifically, we have launched two edge cloud products, namely, *Local Region* [2] and *Cloud Box* [1, 3]. The Local Region is a local public cloud infrastructure serving nearby customers in the same city while the Cloud Box is a highly-converged public cloud infrastructure within a 42U server cabinet. They both offer a user experience akin to the public cloud but with significantly reduced capacities, *e.g.*, a Local Region usually has dozens of servers, and a Cloud Box contains even fewer. As a comparison, our largest public cloud region houses hundreds of thousands of servers.

Deployment constraints of edge clouds. When extending the VPC network infrastructure to the edge, we've experienced the following deployment issues. ① *Fitting the entire public cloud into limited space.* Edge clouds have small footprints but full functionality of the public cloud. For example, for Cloud Box, the entire cloud infrastructure (including compute, storage, network, power supply, cooling system) will be packed into a 42U server cabinet, leaving limited space to accommodate servers running tenants' VMs. In addition, if the network occupies too much cabinet space, the server payload will be further compacted, which reduces the available VMs for sale as well as the cloud vendor's revenue. If we pack the 9 different types of underlay and overlay devices (as shown in Fig. 1) with 1:1 backup into the 42U cabinet, there will be little room left for the server payload. ② *Cost disadvantages without economies of scale.* In a public cloud region, the VPC network infrastructure consists of a few clusters of underlay and overlay devices, serving tens of thousands of servers. The upfront and operational costs of the VPC network infrastructure can be spread over the huge number of servers due to the economies of scale [49]. However, for a single edge cloud, the situation has changed as the network infrastructure will occupy a significant proportion of its upfront cost due to the much reduced server payload. Moreover, the "role-splitting" architecture is not conducive to operational cost reduction. Last but not least, as the edge clouds are placed near the customer sites, they will have a huge site number compared with the public clouds. For example, Alibaba Cloud now has only

28 public cloud regions but the number of edge clouds grows rapidly. That is, the cost inefficiency of the original VPC network architecture in a single edge cloud will be magnified thousands of times when edge clouds become more popular.

Possible solutions and their limitations. The traffic load is significantly reduced at the edge due to the fewer tenants served. Therefore, the 3-tier switch fabric for underlay traffic processing can be simplified and the role-splitting gateway clusters for overlay traffic handling are indeed overkill. Accordingly, in our first generation of edge cloud products, we simply use a few ToR switches for the underlay and a few x86 server-based gateways and load balancers for the overlay. Specifically, for the Local Region, we use separate x86 servers to build XGW and SLB. As the XGW holds all the gateway tables, it can easily replace all the single-role gateways, significantly saving the upfront and operational costs as well as the deployment footprints. For the Cloud Box, to maximally save the cabinet space, we further converge the XGW and the SLB into a single x86 server, using different groups of CPU cores to perform different tasks. To handle potential traffic growth in the future, horizontal scaling [41, 50] is still leveraged.

Although extensive horizontal scaling will cause cost inefficiency and large deployment footprints of the VPC network infrastructure, we believe this will not happen soon at the edge. However, two real edge cloud use cases change our minds. The first is about the high-bandwidth requirement at the edge. In this case, our customer expects to migrate his data from the local IDC to the remote public cloud through the nearest point of presence (PoP). However, the nearest PoP is in another city hundreds of miles away from the customer IDC and a leased line from an ISP is needed for the IDC-PoP connection. To save expenses and time, our customer decides to route his data to the public cloud directly through the Local Region in his city, which produces hundreds of Gbps traffic and floods the XGWs in the Local Region. The second is about CPU overload by a heavy-hitter flow. In this case, our customer uses the Cloud Box to manage his IoT devices. The data collected by the IoT devices is sent through a tunnel to the Cloud Box. The tunnel aggregates the traffic into a single flow that reaches dozens of Gbps. At the XGW, the heavy-hitter flow is hashed to a CPU core via the RSS mechanism [25], which easily overloads the CPU core and causes packet drops.

3 LuoShen's Architecture

In this section, we list the design goals of LuoShen, followed by its architectural innovation to achieve these goals.

3.1 Design Goals

①*Small deployment footprints*. The VPC network infrastructure at the edge should have small footprints to leave more room for the server payload, maximizing the VM capacity.

②*Complete VPC network functions*. At the edge, our customers want ultra-low latency but do not want to compromise the consistent product experience of the public cloud. The VPC network inside the 42U server cabinet needs to provide the same functions as provided by the public cloud.

③*Cost efficiency*. Considering the large edge cloud number, the upfront and operational costs of the network infrastructure in each edge cloud should be controlled. Besides, network architectures with smaller energy footprints are preferred.

④*Performance stability*. In a multi-tenant cloud, failures of the shared network infrastructure will affect a large number of tenants. As the gateway is the central hub of the cloud traffic, its performance stability needs to be strengthened to avoid being overloaded by either high-bandwidth traffic or heavy-hitter flows. Once a failure occurs, traffic should be taken over by the backup component as soon as possible.

⑤*Elasticity and flexibility*. Different edge cloud use cases have different network customization needs in terms of traffic scale and network functionality, so the edge cloud network infrastructure should be elastic and programmable to quickly respond to the changing service requirements.

⑥*Avoid reinventing the wheel*. We have invested tremendous person-months into developing the network infrastructure for the public cloud. The stability of the system has stood the test of time. As the edge clouds inherit all the functions of the public cloud, to save the development costs, retain the system stability and shorten the time-to-market cycle, we'd better reuse as much code as possible from the existing systems.

3.2 Hyper-Converged Gateway

Opportunities for infrastructure convergence. Programmable switches [15, 16] have been proved to work well even under cloud-scale traffic bursts [41]. They are suitable for stateless forwarding which covers a majority of cloud gateway functions. However, some cloud services are stateful (*e.g.*, SLB, NAT) and better to be processed by the CPU due to the large memory footprints as well as the high processing complexity. To deal with high-bandwidth traffic, the stateful processing can also be offloaded to the FPGA. In the following, we discuss how to converge the core functions of the VPC network infrastructure into a Tofino chip [5] and how to converge the remaining functions into the CPU with high-performance functions accelerated by the FPGA.

①*Converge different gateway functions sharing the same table*. As you may notice in §2.1, many gateway functions in our cloud share the same forwarding tables, *e.g.*, the VXLAN routing table is involved in all gateway functions except Internet-related services. Hence, we can converge these functions from different gateways into the same Tofino pipeline stages. Actually, in past deployments, we have already made efforts in this direction, *e.g.*, VGW handles VM-VM (same VPC), VM-VM (different VPCs) and VM-LB-Service traffic within a region and TGW handles all the cross-region traffic between two VMs and between VMs and IDCs. Actually, VGW and TGW also share the VXLAN routing table, that is, they can be further converged into a single gateway. Besides, given that Tofino is programmable, we can also converge the VXLAN-to-VLAN translation function into it, saving space originally for CSW. Finally, in the Tofino, we converge VGW, TGW and CSW into a converged gateway called *CGW*.

②*Converge different gateway functions without table overlapping*. In edge clouds, the gateways serve tenants a few miles away, so there is a significant reduction in the number of table entries compared to the public cloud. As a result, it is possible to converge different gateway functions into the same Tofino chip even though their tables have no overlap. Specifically, we can place different functions in different pipeline stages. Packets will be sequentially processed by each function when they traverse through the pipeline. In this way, we can further converge *CGW* and *IGW* into the Tofino chip.

③*Converge underlay and overlay devices*. After we converge overlay devices (*i.e.*, CGW, IGW) into the Tofino, there are still many standalone underlay devices such as SW (for switching east-west traffic), LSW (for routing traffic to gateways) and BSW (for connecting Internet/remote regions), which need additional costs as well as space for placement. Given that Tofino has ultra-high throughput, if its on-chip resources are still available, we can further converge the underlay devices together with the overlay devices into the same gateway. Specifically, we converge the original SW and LSW into a *new* SW, which is further converged with the CGW and IGW into the Tofino (BSW is still standalone). In this way, one Tofino handles both overlay and underlay traffic.

④*Process fallback traffic and stateful forwarding at the CPU*. The P4 switch pipeline cannot handle all types of cloud traffic. Some traffic needs fallback processing. For example, the VM-to-VM route caching at vSwitches requires special treatment of the first packet of the VM-to-VM traffic within a region. If a packet from a VM cannot hit the route cache at the vSwitch, it will be forwarded to the gateway [54]. After querying the VM-NC mapping table at the gateway, a reply packet containing the mapping relationship will be constructed and sent back to the vSwitch for route cache population. However, the reply packet construction is a bit complicated for the P4 pipeline and better to leave to the CPU. Besides, some cloud services (*e.g.*, NAT, VPN) require stateful processing at the gateway with large session tables. These tables are too large to fit into the Tofino and better to use CPU to process them.

⑤*Offload high-bandwidth stateful forwarding to the FPGA*. Although we can use kernel-bypass techniques (*e.g.*, DPDK) [13] to accelerate CPU-based stateful packet processing, for some stateful cloud services, high-bandwidth traffic can easily overload a CPU core [36, 41, 48]. One example is east-west traffic load balancing. Sometimes, a massive number of VMs want to access the cloud service hosted on a server cluster behind a load balancer. The load balancer may face huge traffic pressure since the east-west traffic in a data center is usually not rate-limited. As a comparison, the Internet-related and cross-region traffic are often rate-limited based on tenants' spending. To handle high-bandwidth stateful forwarding, we can use FPGA with large HBM to meet both search speed and memory capacity requirements [57].

The server-switch hardware. According to the above infrastructure convergence principles, we build LuoShen, a hyper-converged programmable gateway for multi-tenant multi-service edge clouds. LuoShen fits the entire VPC network infrastructure into SNA [8], a 2U server switch developed by Alibaba Cloud's infrastructure team, with full public cloud network functionality retained. Inside LuoShen, the original single-role gateways (including VGW, TGW, CSW, IGW) and the underlay devices (including SW and LSW) are converged into a Tofino chip, handling both overlay and underlay traffic; the XGW and SLB are converged into a CPU, handling fallback traffic and conducting traffic load balancing, respectively; the SLB+ in an FPGA conducts traffic load balancing hardware acceleration. LuoShen essentially follows a "P4-centric" architecture inherited from our centralized gateway model in the public cloud (§2.1), because most of the gateway functions are now converged into the P4 switch pipeline. In the public cloud, the cloud gateways are the central hub of the east-west and north-south traffic. Now, for edge clouds, the Tofino in LuoShen becomes the new central hub of the edge cloud traffic. In LuoShen, the 64×100G ports of Tofino are split into different purposes. Some ports directly connect to servers to receive VM traffic, some ports connect to BSW for Internet access and cross-region communication, some ports connect to on-premise IDCs, and some ports connect to the CPU and FPGA for fallback and stateful traffic processing (the CPU sits on a NIC for hardware offloading/traffic rate-limiting, while the FPGA is plugable so that we can also select other cards for fast adapting to the diverse edge cloud service needs). Besides, a large number of ports that are not shown explicitly in Fig. 2 are used internally for pipeline folding [41], which will be detailed in 4.1. As a comparison, in the public cloud, all ports of the P4-accelerated gateways are connected to the upstream switches (*i.e.*, LSW). Since we converge the 8 standalone overlay and underlay devices into a 2U box, LuoShen has superior advantages in terms of deployment footprints, upfront and operational costs, and power consumption for edge cloud deployment.

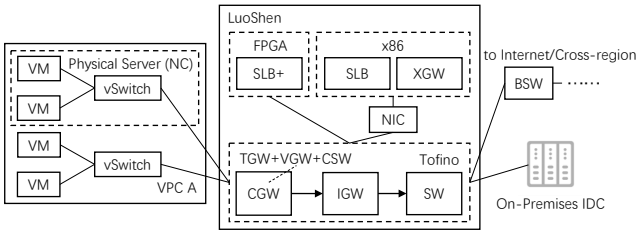


Figure 2: LuoShen fits the entire VPC network infrastructure into a 2U server switch with full functionality retained.

Functionality equivalence analysis. To provide tenants with the same product experience as that of the public cloud, we need to achieve *functionality equivalence* after convergence. We compare the two architectures before and after convergence (Fig. 1 and Fig. 2). First, all the gateways/SLB functionality is retained because VGW, TGW, CSW, IGW are now in the P4 pipeline while XGW and SLB are now in the CPU/FPGA. Second, the functionality of east-west traffic switching as well as traffic routing to

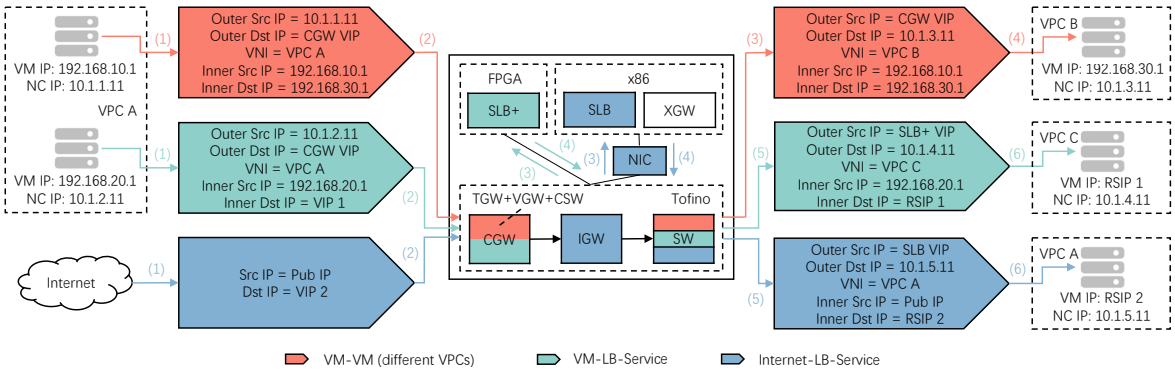


Figure 3: Packet journeys of three different cloud services converge at LuoShen.

gateways/SLB/Internet/IDCs/remote regions provided by the original underlay device SW and LSW is also retained after they are converged into the new SW in LuoShen. Specifically, the original SW needs to route the VM traffic to the gateways/SLB through LSW, however, this function is no longer needed since physical servers are now directly connected with LuoShen. It also needs to switch the east-west traffic which hits the route cache at vSwitches and this function is still retained at the new SW. The original LSW needs to route traffic to gateways/SLB/Internet/IDCs/remote regions. Since VGW, TGW, CSW, IGW are now in the Tofino which is directly connected with physical servers, there is no need to explicitly route traffic to these gateways. However, the function to route traffic to XGW/SLB/Internet/IDCs/remote regions of the original LSW is still retained at the new SW, which will forward the corresponding traffic through the P4 switch ports to these external destinations outside the Tofino.

To summarize, in LuoShen's P4-centric architecture, the core functions of the VPC network infrastructure are offloaded to the P4 pipeline. All incoming traffic will go through CGW, IGW and the new SW sequentially in the pipeline before being distributed to CPU/FPGA or external networks by the SW. Since all traffic experiences the "distribute after deep pipelining" packet flow, the P4 pipeline becomes the traffic aggregation point. Thus, the traffic of different cloud services will be queuing together in the pipeline. However, there is no need to worry about it at present as the high-capacity pipeline is sufficient to accommodate the traffic from the edge.

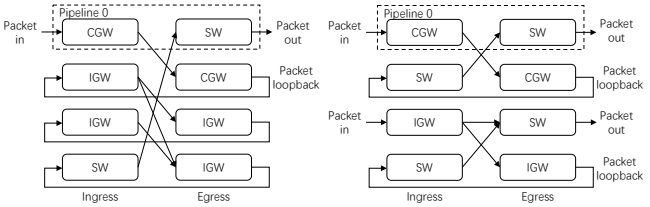
Packet journeys in LuoShen. Fig. 3 shows the packet journeys of three different cloud services converging at LuoShen. The first is VM-to-VM traffic across VPCs within a region, where the outer IP is set to CGW VIP by the vSwitch due to a route cache miss. Then, the packet will be routed to the CGW in LuoShen and query the VXLAN routing table and VM-NC mapping table to obtain the VNI of the destination VPC as well as the server IP of the destination VM. Finally, the packet will query the FIB in SW and be forwarded to the destination VM. The second is VM traffic load balancing for accessing the cloud service behind SLB, which is more complicated as the Tofino needs to interact with the SLB+ in the external FPGA (we use FPGA for SLB acceleration). The packet containing VIP 1 (*i.e.*, the IP of the cloud service) as the inner DIP will be routed to CGW in LuoShen to query the VXLAN routing table and VM-NC mapping table to obtain the VIP of SLB+. After that, the packet will be forwarded by SW to the SLB+, where server load balancing is performed by querying a session table to find the real server (identified by RSIP 1 and 10.1.4.11 in Fig. 3) using the 5-tuple. Then, the packet will be reencapsulated with the real server address as the inner/outer DIP and routed back to the Tofino (because the FPGA has no other connection to the network). Finally, the packet will be forwarded by SW to the selected real server behind SLB+. The third is Internet traffic load balancing for accessing the cloud service behind SLB. The packet flow is almost the same with the second case, except that the traffic from the Internet will be routed to IGW first to query the DNAT table using the DIP to obtain the VIP of SLB, and the server load balancing is performed in the CPU this time.

4 Data Plane

In this section, we show how to fit the core functions of the VPC network infrastructure into a single Tofino chip through sophisticated pipeline layout and on-chip memory budgeting.

4.1 Tofino Pipeline Layout

Pipeline folding for CGW/IGW/SW convergence. The Tofino has 4 pipelines and each pipeline consists of an ingress pipe and an egress pipe, connected by a traffic manager (TM). Each pipeline has limited SRAM/TCAM memory resources [30], distributed equally in 12 stages of that pipeline. Each pipeline stage is shared by the ingress pipe and the egress pipe, *e.g.*, if we fit a large table into stage 0 of the ingress pipe, there will be little space left for stage 0 of the egress pipe. When converging the core functions of the VPC network into the Tofino, we should first evaluate if the stage/memory resources are sufficient to hold the deep pipeline of CGW, IGW, and SW. Actually, in our P4-based single-role gateways for the public cloud [41], the stage/memory occupancy of some major tables, such as the VM-NC mapping table, has already exceeded the resources of a single pipeline. Besides, other tables, such as SNAT/DNAT table for Internet access and meter tables for rate-limiting, will also have large memory footprints. Although the table sizes will shrink at the edge, a single Tofino pipeline still cannot hold all the converged core functions of the VPC network.



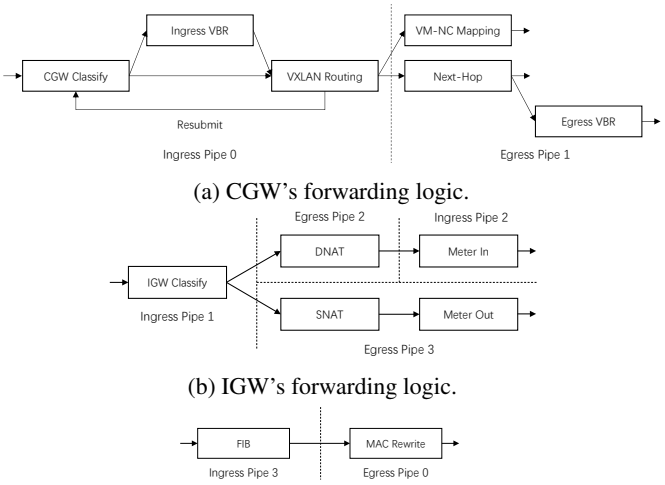
(a) LuoShen's pipeline layout. (b) Another layout option.

Figure 4: Two optional ways of pipeline folding.

To this end, we conduct pipeline folding to assemble the four independent pipelines into a 48-stage deep pipeline for CGW/IGW/SW convergence (as shown in Fig. 4a). The traffic of different cloud services will sequentially traverse through the deep pipeline and be processed at the corresponding stages. For example, the VM-VM (same VPC) traffic will go through Ingress Pipe 0, Egress Pipe 1, Ingress Pipe 1, Egress Pipe 3, Ingress Pipe 3, and Egress Pipe 0, while the Internet-VM traffic will go through Ingress Pipe 0, Egress Pipe 1, Ingress Pipe 1, Egress Pipe 2, Ingress Pipe 2, Egress Pipe 3, Ingress Pipe 3, and Egress Pipe 0. Notice that there is one explicit pipe branch at Ingress Pipe 1 with either Egress Pipe 2 or Egress Pipe 3 as the next-hop pipe to separately process Internet-VM or VM-Internet traffic. There is also an inexplicit pipe brach at Ingress Pipe 0 (not shown in Fig. 4a) with either Egress Pipe 1 or Ingress Pipe 0 itself as the next-hop pipe as some cloud services need to query the VXLAN routing table twice (*e.g.*, VM-to-VM traffic across VPCs). Last but not least, as the FIB in SW is embedded in Ingress Pipe 3, it will select the next-hop ports from Egress Pipe 0 according to the FIB lookup results (the 16 ports of Egress Pipe 0 are connected to the external destinations outside the Tofino such as XGW/SLB in CPU, SLB+ in FPGA, physical servers, Internet/remote regions through BSW, and on-premises IDCs).

Major tables in the pipeline. We show how the major tables of CGW, IGW and SW are distributed in the pipeline.

①CGW (Fig. 5a): CGW contains all the major tables of VGW, TGW and CSW. As the VXLAN routing table and the VM-NC mapping table are the two largest, we separate them into different pipes. To determine whether an incoming packet will be processed locally or bypass the current pipe, we place a CGW Classify at the front of Ingress Pipe 0. In this way, Internet traffic will directly bypass CGW (we detail the bypass logic later). The packet flows of different cloud services related to CGW are listed as follows. VM-VM (same VPC): CGW Classify->VXLAN Routing->VM-NC Mapping; VM-VM (across VPCs, same region): CGW Classify->VXLAN Routing(resubmit)->VM-NC Mapping; VM-Cross-region-VM (sender): CGW Classify->VXLAN Routing(resubmit)->Next-Hop; VM-Cross-region-VM (receiver): CGW Classify->VXLAN Routing(resubmit)->VM-NC Mapping; VM-IDC: CGW Classify->VXLAN Routing(resubmit)->Next-Hop->Egress VBR(VXLAN-to-VLAN); IDC-VM: CGW Classify->Ingress VBR(VLAN-to-VXLAN)->VXLAN Routing(resubmit)->VM-NC Mapping. Notice that we use Ingress/Egress VBR for VLAN-to-



(a) CGW's forwarding logic.

(b) IGW's forwarding logic.

(c) SW's forwarding logic.

Figure 5: Distribution of major tables in the pipeline.

VXLAN translation and vice versa to deal with IDC traffic. Besides, we use Next-Hop to obtain the physical address of the remote device in either VM-Cross-region-VM (sender) or VM-IDC case. Finally, the resubmit ability of Tofino enables circular lookups of the VXLAN routing table in several cases.

②IGW (Fig. 5b): IGW contains SNAT for VM-Internet traffic and DNAT for Internet-VM traffic, as well as their rate-limiting tables (*i.e.*, Meter Out and Meter In). The packet flows are: VM-Internet: IGW Classify->SNAT->Meter Out; Internet-VM: IGW Classify->DNAT->Meter In. Similar to CGW Classify, IGW Classify is used to filter the traffic that should be processed locally. Besides, it will also decide whether the traffic should be sent to SNAT or DNAT according to packet header fields. Notice that Meter In occupies the entire pipe while Meter Out shares its pipe with the SNAT table. The reason is that, most rate-limiters for the VM-Internet direction are installed at vSwitches while all rate-limiters for the opposite direction are installed at the gateway, so that Meter Out is much smaller than Meter In at the gateway.

③SW (Fig. 5c): SW is responsible for underlay traffic forwarding according to the outer DIP. The possible destinations can be XGW/SLB in CPU, SLB+ in FPGA, physical servers, Internet/remote regions through BSW, and IDCs. SW conducts FIB lookup at Ingress Pipe 3 to select the next hop and then rewrites the MAC headers at Egress Pipe 0.

Pipe/table bypass logic. Although each packet will sequentially pass through CGW, IGW and SW, not all tables have to be queried, *e.g.*, Internet traffic will go through the CGW pipes but there is no need to query the VXLAN routing table. Hence, we can make an early judgment to determine whether the packet will be processed by the local pipe or even the local table to reduce unnecessary processing overhead. Usually, the judgment is placed at the front of a pipe, *e.g.*, in CGW Classify, we set flags in the metadata according to the VNI and DIP in the packet header to classify traffic into different cloud services and the flags can be carried by the metadata across pipeline stages. Fig. 6 shows the P4 code framework

```
struct metadata_t {
    bit<1> flag; /* whether to bypass the current pipe */
    bit<4> subflag; /* whether to bypass the current table */
    ...
}
control Ingress( ... ) {
    action tbl_ac1() { flag = 0; }
    action tbl_ac2() { flag = 1; subflag = 0; }
    action tbl_ac3() { flag = 1; subflag = 1; }
    ...
    table tbl {
        /* to distinguish different cloud services */
        key = { ... }
        /* to take different bypass actions */
        actions = {
            tbl_ac1; /* bypass the current pipe */
            tbl_ac2; /* enter the current pipe, query tb2 */
            tbl_ac3; /* enter the current pipe, query tb3 */
            ... } }
    table tb2 { ... }
    table tb3 { ... }
    ...
    apply {
        tbl.apply();
        if (flag == 1) {
            if (subflag == 0) { tb2.apply(); }
            else if (subflag == 1) { tb3.apply(); }
            ...
        }
        ... /* flag is 0, bypass the current pipe */
    } }
```

Figure 6: P4 code framework for pipe/table bypass.

for pipe/table bypass. Specifically, we use flag for pipe bypass and subflag for table bypass in the local pipe. For components that span across multiple pipes (*e.g.*, IGW), we can place separate judgment logic at the front of each pipe, or let the judgment results from the first pipe pass down to the subsequent pipes. The former consumes additional stage occupancy while the latter increases metadata bridge usage [41].

Rationale behind pipeline design.

①*Two optional ways of pipeline folding*. In designing the pipeline layout, we have also considered another option in Fig. 4b, which has separate pipelines folded for CGW and IGW, achieving 3.2Tbps total bandwidth. However, it has limitations: (1) Traffic balance is hard to achieve between two pipelines. (2) Table occupancy balance is also hard to guarantee between two pipelines. (3) Although more ports are exposed to servers, there is also tight coupling between the cloud services and the ports, which adds constraints for server placement. (4) Our selected pipeline layout maximally reuses the existing code by porting the entire IGW codebase from the existing P4-based IGW gateway [41] for rapid deployment.

②*Component sequence in the pipeline*. Someone may wonder why the sequence is CGW->IGW->SW. In fact, we can place either CGW or IGW in the front. However, the overlay devices (CGW and IGW) must be placed ahead of the underlay device (SW) because the gateway always conducts outer IP modification based on the VNI and the inner IP first (*e.g.*, via VXLAN route lookup), then uses the outer DIP to query the FIB in SW to determine the next-hop forwarding port.

③*Next-hop selection at the ingress pipe*. You may notice that all next-hop selection happens at the ingress pipe (*e.g.*, VXLAN routing table resubmit at Ingress Pipe 0, SNAT/DNAT selection at Ingress Pipe 1, FIB lookup in SW at Ingress Pipe 3). This is because we must rely on the TM of Tofino for packet switching and it is necessary to determine where the packet will go before it is sent to the TM.

④*Performance issue and the coping strategy*. The current pipeline layout also has limitations: (1) The original 4 parallel pipelines are folded into one, decreasing the bandwidth from 6.4Tbps to 1.6Tbps. As 4 ports are connected with the CPU and FPGA, the actual gateway bandwidth is only 1.2Tbps. (2) Lots of Tofino ports are used for internal loopback, thereby greatly reducing the number of ports for connecting physical servers. However, for edge clouds, as the 1.2Tbps throughput is more than enough and the server payload in a Cloud Box is restricted, the above-mentioned will no longer be a problem. In real deployment, hot standby is leveraged which doubles the available gateway bandwidth as well as server payload.

4.2 On-Chip Resource Budgeting

As Tofino has scarce SRAM/TCAM, pipeline stages, and PHV resources, we need to carefully budget the on-chip resource occupancy to fit multiple gateway functions into the chip.

Table placement for balanced resource occupancy. In LuoShen, forwarding tables are distributed in the 8 cascaded pipes based on the service logic as well as balanced pipeline resource occupancy. As mentioned in 4.1, the SRAM/TCAM memories in each pipeline stage are shared by the ingress pipe and the egress pipe. That is, for each pipeline, we can have different resource occupancy combinations like ingress-heavy, egress-heavy or balanced, but we can never have ingress-heavy and egress-heavy simultaneously. When fitting the tables into the Tofino, we also follow this principle to balance the resource occupancy of the 4 pipelines. For example, Pipeline 0 is ingress-heavy as its ingress pipe stores the VXLAN routing table, therefore, its egress pipe can only store very small tables for MAC rewrite. Similarly, Pipeline 1 is egress-heavy due to the VM-NC mapping table at Egress Pipe 1, Pipeline 2 is balanced because its ingress pipe has Meter In while its egress pipe has DNAT, Pipeline 3 is egress-heavy due to the SNAT and Meter Out at its egress pipe.

Stage occupancy compression. After pipeline folding, all tables of the cascaded CGW/IGW/SW are fit in the 48 stages. The stages are consumed mainly in two ways: (1) table dependency, (2) large table occupancy. The table dependency means the packet processing behavior of the current stage depends on the table lookup results of the previous stage, *e.g.*, CGW Classify occupies the first stage of Pipeline 0 to set bypass flags for subsequent stages. Clearly, in a hyper-converged gateway, table dependencies occur more often. The large table occupancy means the large table that cannot be fit in one stage will span across multiple stages. In LuoShen, for (1), we rely on the P4 compiler to generate the dependency graph; for (2), we shrink some large tables (*e.g.*, the VM-NC mapping table) to reduce their stage occupancy for edge clouds.

PHV usage optimizations. PHV is a set of containers that carry the headers and metadata along the pipeline. The first PHV optimization is to reduce the metadata bridging as much as possible by meticulous table placement because metadata bridging not only decreases the switching throughput through the TM [41] but also consumes more PHV resources at both

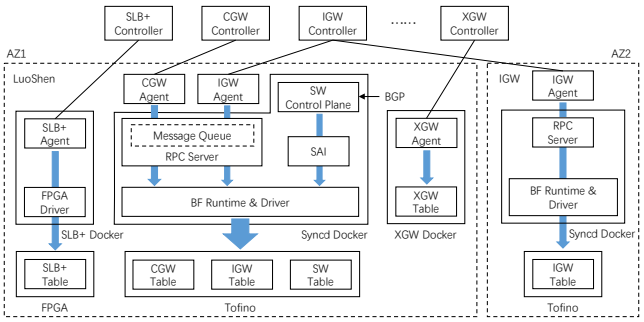


Figure 7: Multi-component table configuration channels.

ingress pipe and egress pipe. The second PHV optimization is to overlay two or more header/metadata fields in the same PHV container if their lifetimes do not intersect by using the *pa_alias* pragma to instruct the compiler. Moreover, we break the large header/metadata field into smaller pieces of the same size intentionally to increase the possibility of PHV sharing.

5 Control Plane

Resource isolation. Multiple components run on LuoShen's CPU. They are (1) DPDK-based forwarding instances such as XGW and SLB, (2) control plane agents of the data plane components in Tofino (CGW/IGW/SW), FPGA (SLB+) and CPU (XGW/SLB). Although the underlying CPU resources are shared, these components have different performance requirements (*e.g.*, line-rate forwarding) and do not want to be interrupted by others. To achieve performance isolation, we dockerize [38] these components and bind XGW and SLB to dedicated CPU cores. As the number of CPU cores is limited, we also allow components (*e.g.*, CGW and IGW agents) to share the CPU cores while using cgroups [46] for resource isolation. Except for CPU, resource isolation for memory/disk is also needed to contain memory leaks and core dump files.

Multi-component table configuration. Unlike the single-role gateways, LuoShen converges multiple components in its data plane, thereby requiring multiple table configuration channels (Fig. 7). For each channel, a control plane agent at the CPU receives table update requests pushed from its remote controller and installs them into the corresponding data plane component through the underlying interface (*e.g.*, BF Runtime). For rapid deployment of LuoShen, we reuse the agent code directly from the single-role gateway so that the remote controller can talk directly to LuoShen without any modification (*e.g.*, in Fig. 7, the IGW controller controls LuoShen and IGW). The channels to Tofino, FPGA and CPU are physically isolated to avoid table configuration collisions. For the Tofino, the channel to CGW/IGW and the channel to SW are also separate (*i.e.*, through RPC and SAI). Although CGW and IGW share the same channel, the complete separation of their tables in the Tofino pipeline makes their table configuration entirely lockless. Besides, to speed up table configuration, we use batch to assemble multiple table update requests and issue them in one shot to the BF Runtime.

Inter-component BGP peering. LuoShen converges the role-splitting VPC network infrastructure of the public cloud into one device, where internal components still need to communicate with each other (*e.g.*, SW needs to route traffic to the next-hop components). To exchange the reachability information between components, we set up BGP speakers at the control plane for inter-component BGP peering so that a component can learn the routes to others and it can also advertise its reachability to others. Specifically, we set up separate BGP speakers for XGW, SLB, SLB+ with both learning and advertising capabilities. For CGW, IGW, SW, we set up one BGP speaker for all of them. For CGW and IGW, as there is only one path in the pipeline, there is no need for them to learn BGP routes and their agents only perform advertising. For SW, as it is the final component in the pipeline, its control plane only needs to learn the routes for next-hop selection. With BGP peering, LuoShen achieves high availability based on component-level ECMP load balancing and fast failure recovery, which are discussed in §A due to page limitation.

6 LuoShen's Performance

We illustrate LuoShen's performance under pressure test and in production. The pressure test topology is shown in §B.

On-chip memory occupancy. Fig. 8 shows the memory usage in ingress/egress pipes. Generally, LuoShen consumes more SRAM than TCAM because most major tables (*e.g.*, VM-NC, SNAT/DNAT, meters, counters) are based on exact match. Besides, as the Tofino contains more SRAM than TCAM, we use ALPM [51] to reduce the TCAM usage at the cost of additional SRAM usage [41]. Except for the VXLAN routing table, Classify tables, ACLs, and FIB in SW will also consume TCAM. Fig. 9 and Fig. 10 show how we achieve balanced pipeline resource occupancy via meticulous table placement. For Pipeline 0, we place the VXLAN routing table in the later stages in the ingress pipe as it has dependencies with the tables in its front (*e.g.*, CGW Classify). As each stage is shared by the ingress and egress, we have to place the tables for MAC rewrite in the early stages in the egress pipe for balanced memory usage. For Pipeline 2, the pipeline resources are shared in a good balance by the ingress (Meter In) and egress (DNAT). Fig. 11 shows the memory usage of CGW/IGW/SW. The SRAM/TCAM occupancy is exactly as expected. However, even in a hyper-converged gateway, the memories are not fully exhausted. The reason is that table dependencies make stages exhausted earlier than memories. Fig. 12 shows the maximum PHV usage in different pipelines. After PHV optimizations, the PHV has not been exhausted in all pipelines. Pipeline 2 has the lowest PHV usage since it contains only two major tables (*i.e.*, DNAT and Meter In).

Performance under pressure test. Fig. 13 shows table configuration speedup via batching. As CGW and IGW share the same download channel, the improved table configuration efficiency will not only reduce the gateway's cold start time but also alleviate download channel congestion. Fig. 14 shows the throughput and latency of the VM-to-VM traffic within the same VPC. LuoShen can achieve 1.2Tbps line-rate forwarding with 256B packets and the latency is bounded within

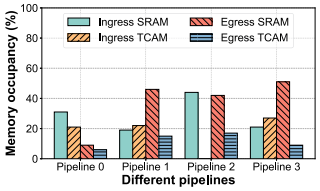


Figure 8: SRAM/TCAM usage in ingress/egress pipes.

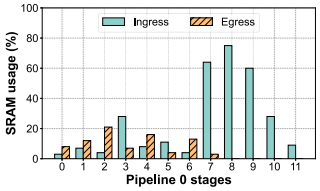


Figure 9: SRAM usage distribution over stages (Pipeline 0).

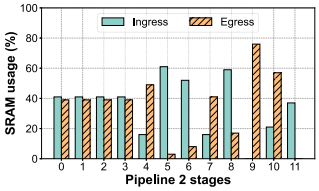


Figure 10: SRAM usage distribution over stages (Pipeline 1).

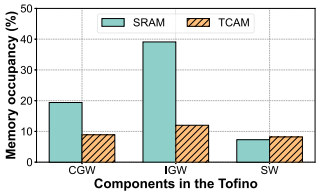


Figure 11: SRAM/TCAM usage of CGW/IGW/SW.

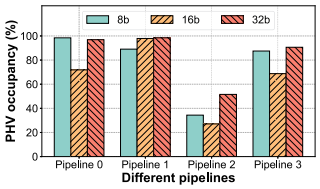


Figure 12: PHV usage in different pipelines (the worst case).

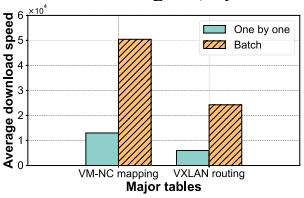


Figure 13: Table configuration speedup via batching.

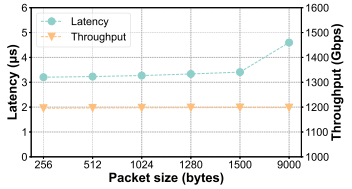


Figure 14: Throughput/latency of VM-to-VM traffic (same VPC).

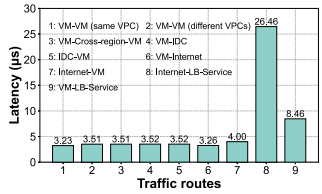


Figure 15: Latency of different traffic routes (512B packets).

5μs even with 9000B packets. Fig. 15 shows the latency of different cloud traffic routes with 512B packet size. The traffic of VM-VM (same VPC) and VM-Internet has the lowest latency due to the shortest route inside the Tofino (only through 6 pipes) without a resubmit at Ingress Pipe 0. The traffic of cross-VPC, cross-region and IDC has a slightly higher latency due to the resubmit operation. The traffic of Internet-VM has an even higher latency due to its longer route through 8 pipes. The traffic of the remaining cloud services will be routed to the external CPU and FPGA, producing the longest latency. Fig. 16 shows the throughput of different cloud traffic routes with 512B packet size. The traffic traversing through the external CPU and FPGA has the lowest throughput, exactly as expected. However, we notice that the traffic of VM-IDC and VM-Internet also has slightly reduced throughput. The reason is that they both experience decapsulation of outer headers.

Performance in production. Fig. 17 shows the converged traffic throughput at an edge site, which is around 50Gbps. As edge clouds have just been rolled out, most edge sites do not have heavy traffic. However, we still select the P4-centric architecture as it not only addresses the extreme cases but also reserves a large performance margin for future traffic growth, which makes the architecture durable. Fig. 18 shows the traffic throughput of different cloud services of the edge site. It can be inferred that most edge cloud traffic is IDC/Internet-related. Most of the VM-to-VM traffic within the same VPC is shortcut by the underlay network due to route caching. Cross-VPC traffic is currently very limited. Fig. 19 shows the CPU utilization of different components. SLB has the highest CPU usage as a data plane function. The control plane agents of CGW/IGW/SW also consume considerable CPU time.

Advantages of LuoShen at the edge. We compare LuoShen with the role-splitting gateway architecture in the public cloud in upfront cost, deployment size and power consumption. We assume that the gateway/load balancer/switch clusters in the public cloud will be reduced to separate devices in edge clouds with one device for each role. Table 2 shows that LuoShen reduces the upfront cost, deployment footprints and power consumption by 75%, 87% and 60%, respectively. Due to page limitation, we discuss the calculation in §C.

Table 2: LuoShen vs role-spliting in cost, size and power.

	Cost (unit)	Size (U)	Power (W)
LuoShen	15	2	∼1000
Role-splitting	61	15	>2500

7 Experiences and Lessons

Step-by-step deployment. To decompose the complexity from infrastructure convergence, we deploy LuoShen in a step-by-step way in production. We deploy the components in the Tofino first since CGW and IGW cover the majority of cloud network functions while SW provides the next-hop selection ability. XGW/SLB/SLB+ are implemented by reusing the existing x86 devices in the public cloud. As CGW/IGW/SW are highly coupled, we have to deploy them together in one pass. After they become more stable, we start to add XGW, SLB and SLB+ one by one for complete deployment.

Function upgrade. As Tofino, CPU and FPGA are physically separated, they can be upgraded independently. By contrast, as CGW, IGW and SW are highly coupled, we have to upgrade them as a whole, which will involve cross-team collaboration as they are managed by different teams. The function isolation issue in Tofino has also been discussed in [19, 53]. In 2018, Barefoot announced Tofino Fast Refresh [14] to address the hot upgrade issue. The feature can reset an entire P4 pipeline within 50ms. However, as LuoShen deeply relies on pipeline folding with one component spanning across pipelines and two components possibly sharing the same pipeline, simply using Tofino Fast Refresh for component upgrade may have consistency issues. For example, when CGW resets Pipeline 0 and Pipeline 1 for an upgrade, the state of SW and IGW in the same two pipelines will disappear for a short time and their table query/update requests on the fly will be discarded.

Capacity expansion. The gateway capacity expansion includes the upgrade of performance and table size. For performance, CPU is more likely to become the system bottleneck. Some of the solutions we've tried include (1) adjusting the number of CPU cores used by different services, (2) migrating workloads from LuoShen to external x86 servers, (3) horizon-

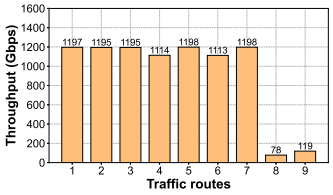


Figure 16: Throughput of different traffic routes (512B packets).

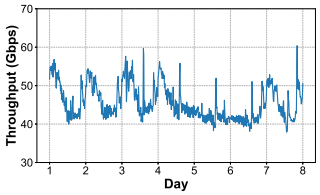


Figure 17: Converged traffic throughput at an edge site.

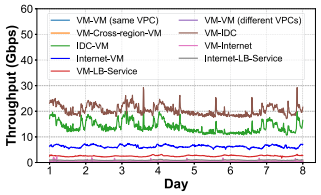


Figure 18: Traffic throughput of different cloud services.

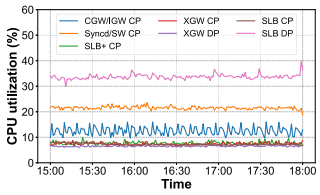


Figure 19: CPU utilization of different components.

tal scaling with more LuoShen gateways. In real deployment, we prefer (3) for stability and ease of maintenance. For table size, we borrow the horizontal table splitting technique from Sailfish [41] to share large tables among more gateways.

Failure isolation. In the public cloud, different gateway clusters are physically separated. In each cluster, tenants' entries are horizontally split into multiple gateways. Therefore, failures will be isolated within a gateway. However, as each gateway is associated with a huge number of tenants, the blast radius of a failure will still be large. In edge clouds, LuoShen serves all tenants miles away. That is, the failure of LuoShen will affect all tenants at the edge site, although the number will not be so large. In the Tofino, as CGW/IGW/SW are cascaded in a folded pipeline, the stability of each component is critical and we reuse the mature code from the single-role gateways. At the CPU, we rely on docker and cgroup for failure isolation. Finally, hot standby is our last resort to guarantee a failsafe.

Telemetry and debugging. As packet flows of different cloud services converge at LuoShen, for network-wide telemetry, we should probe all traffic routes in the gateway [59]. To achieve this, we generate probes with pre-defined header fields for all cloud service coverage. For network anomaly debugging, we need to pinpoint the exact anomaly locations. To achieve this, we collect and export telemetry data to the CPU at each component/pipe/stage along the anomaly forwarding path.

Elastic NFV deployment. For LuoShen, if we deploy NFV instances in its CPU, the elasticity will be restricted as the CPU has already been excessively used (with XGW, SLB and control plane agents). But, if we deploy NFV instances in x86 servers, the CPU cores for tenants' VMs will be occupied (some tenants are concerned about this as the server payload is valuable at the edge). We develop an NFV framework that can autoscale across LuoShen's boundary by horizontally expanding table entries onto x86 servers during peak workloads.

8 Related Work

Building hybrid CPU/ASIC/FPGA network systems is a common tactic [9, 11, 35, 41, 57] to combat traffic growth that goes far beyond Moore's law [45]. However, few systems succeed in fitting the entire multi-tenant cloud network infrastructure with multi-service packet flows into a 2U box deployed in production. For example, ServerSwitch [35] takes advantage of both commodity servers and switching ASICs to perform flexible and performant underlay traffic processing in DCNs. Tiara [57] proposes a hardware-accelerated L4 load balancer with a 3-tier architecture of P4/FPGA/CPU. While Tiara fits the SLB function for the public cloud into a 5U box, LuoShen fits the entire VPC network infrastructure (including SLB) into a 2U box tailored for edge clouds. Sailfish [41] uses Tofino to accelerate the single-role cloud gateways (*e.g.*, VGW). While Sailfish and LuoShen both use pipeline folding to provide sufficient stage resources, LuoShen considers additional constraints. In addition to fitting table entries within Tofino, it must also consolidate multi-service packet flows, achieve equivalent implementation of underlay/overlay devices, expose ports to interact with external CPU/FPGA, and maximize codebase reuse. Bluebird [11] uses Tofino to accelerate the bare-metal cloud services which have similar packet flows with our VM-IDC, IDC-VM scenarios with VXLAN-to-VLAN translation and vice versa. Comparatively, LuoShen is an "all-in-one" gateway with a more sophisticated pipeline layout to consolidate almost all cloud network functions.

Different from our centralized gateway model, other cloud vendors distribute the VPC network functions to end-hosts [12, 17, 20–22, 31, 33]. For example, VFP [21] installs virtual routing, tunnel encap/decap, and load balancing into a host stack, which is scalable and fault-tolerant as there is no single point of failure. The host stack performance can further be accelerated by SmartNICs [22] or by a remote DPU pool [12] to deal with high-bandwidth traffic bursts. Such distributed cloud network architecture could also be a good fit for edge clouds. Considering the consistency with our existing infrastructure, LuoShen takes an alternative approach with different design tradeoffs. It offloads almost all VPC network functions to the gateway to maximally free up valuable CPU cores for tenants' VMs. SmartNIC acceleration is not necessary which saves upfront cost. The IDC/cross-region traffic is also handled by the same gateway which saves cost and space. To address single point of failure, LuoShen relies on Tofino to absorb traffic bursts and reuses mature code for stability.

Some vendors build white box server switches without service logic (*e.g.*, Accton's CSP-7551 [4]). LuoShen's idea can be extended to them to build other hyper-converged systems.

9 Conclusion

We propose LuoShen, a hyper-converged gateway for edge clouds, which fits the entire VPC network infrastructure of the public cloud into a 2U server switch with a P4-centric architecture. To adapt to the new design constraints of performance, costs and deployment footprints, we rearchitect the data plane and control plane. LuoShen has been deployed in Alibaba Cloud for over two years at hundreds of edge sites.

Acknowledgements: We thank the anonymous reviewers and our shepherd Anirudh Sivaraman for constructive feedback.

References

[1] Alibaba cloud launches cloud box - extending public cloud services to local devices. https://www.alibabacloud.com/blog/alibaba-cloud-launches-cloud-box---extending-public-cloud-services-to-local-devices_596688, 2020.

[2] Alibaba cloud launches local regions. https://infrastructure.aliyun.com/local-regions, 2021.

[3] Alibaba cloudbox - product page. https://www.aliyun.com/product/ecs/cloudbox, 2021.

[4] Csp-7551, hyper network appliance. https://www.accton.com/product-csp-7551/, 2021.

[5] Tofino: P4-programmable ethernet switch asic that delivers better performance at lower power. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html, 2021.

[6] Alibaba cloud's global infrastructure. https://www.alibabacloud.com/global-locations, 2022.

[7] Cascade lake: Overview. https://www.intel.com/content/www/us/en/products/platforms/details/cascade-lake.html, 2022.

[8] The journey towards predictable network in alibaba cloud. https://opennetworking.org/wp-content/uploads/2022/05/Dennis-Cai-Final-Slide-Deck.pdf, 2022.

[9] Mina Tahmasbi Arashloo, Pavel Shirshov, Rohan Gandhi, Guohan Lu, Lihua Yuan, and Jennifer Rexford. A scalable vpn gateway for multi-tenant cloud services. *ACM SIGCOMM Computer Communication Review*, 48(1):49–55, 2018.

[10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[11] Manikandan Arumugam, Deepak Bansal, Navdeep Bhatia, James Boerner, Simon Capper, Changhoon Kim, Sarah McClure, Neeraj Motwani, Ranga Narasimhan, Urvish Panchal, et al. Bluebird: High-performance {SDN} for bare-metal cloud services. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 355–370, 2022.

[12] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, et al. Disaggregating stateful network functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1469–1487, 2023.

[13] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 5–16. IEEE, 2015.

[14] Antonin Bas. Leveraging stratum and tofino fast refresh for software upgrades. *Accessed: Jul*, 4:2021, 2018.

[15] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[16] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[17] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX symposium on networked systems design and implementation (NSDI 18)*, pages 373–387, 2018.

[18] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, 2016.

[19] Yong Feng, Zhikang Chen, Haoyu Song, Wenquan Xu, Jiahao Li, Zijian Zhang, Tong Yun, Ying Wan, and Bin Liu. Enabling in-situ programmability in network data plane: From architecture to language. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 635–649, 2022.

[20] DANIEL FIRESTONE. A virtual switch platform for host sdn in the public cloud. *USENIX Open Access Policy*, page 6.

[21] Daniel Firestone. {VFP}: A virtual switch platform for host {SDN} in the public cloud. In *14th USENIX*

Symposium on Networked Systems Design and Implementation (NSDI 17), pages 315–328, 2017.

[22] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking:{SmartNICs} in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.

[23] Xenofon Foukas and Bozidar Radunovic. Concordia: Teaching the 5g vran to share compute. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 580–596, 2021.

[24] Pankaj Garg and Y Wang. Nvgre: Network virtualization using generic routing encapsulation. Technical report, 2015.

[25] Stephen D Goglin and Linden Cornett. Flexible and extensible receive side scaling, September 1 2009. US Patent 7,584,286.

[26] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 51–62, 2009.

[27] Jesse Gross, Ilango Ganga, and T. Sridhar. Geneve: Generic Network Virtualization Encapsulation. RFC 8926, November 2020.

[28] Rolf Harms and Michael Yamartino. The economics of the cloud. *Microsoft whitepaper, Microsoft Corporation*, 3:157, 2010.

[29] Antonios Katsarakis, Zhaowei Tan, Matthew Balkwill, Bozidar Radunovic, Andrew Bainbridge, Aleksandar Dragojevic, Boris Grot, and Yongguang Zhang. rvnf: Reliable, scalable and performant cellular vnfs in the cloud. Technical report, Technical Report MSR-TR-2021-7, Microsoft, 2021.

[30] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 90–106, 2020.

[31] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, et al. Network virtualization in multi-tenant datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 203–216, 2014.

[32] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. Cloudcmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 1–14, 2010.

[33] Anthony Liguori. The nitro project–next generation aws infrastructure. In *Hot Chips: A Symposium on High Performance Chips*, 2018.

[34] Yunzhuo Liu, Hao Nie, Hui Cai, Bo Jiang, Pengyu Zhang, Yirui Liu, Yidong Yao, Xionglie Wei, Biao Lyu, Chenren Xu, et al. X-plane: A high-throughput large-capacity 5g upf. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2023.

[35] Guohan Lu, Chuanxiong Guo, Yulong Li, Zhiqiang Zhou, Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, and Yongguang Zhang. {ServerSwitch}: A programmable and high performance platform for data center networks. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[36] Jianyuan Lu, Tian Pan, Shan He, Mao Miao, Guangzhe Zhou, Yining Qi, Shize Zhang, Enge Song, Xiaoqing Sun, Huaiyi Zhao, et al. Cloudsentry: Two-stage heavy hitter detection for cloud-scale gateway overload protection. *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[37] Mallik Mahalingam, Dinesh Dutt, Kenneth Duda, Puneet Agarwal, Lawrence Kreeger, T Sridhar, Mike Bursell, and Chris Wright. Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. Technical report, 2014.

[38] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux j*, 239(2):2, 2014.

[39] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[40] Jayaram Mudigonda, Praveen Yalagandula, Jeff Mogul, Bryan Stiekes, and Yanick Pouffary. Netlord: a scalable multi-tenant network architecture for virtualized datacenters. *ACM SIGCOMM Computer Communication Review*, 41(4):62–73, 2011.

[41] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, et al. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 194–206, 2021.

[42] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. *ACM SIGCOMM Computer Communication Review*, 43(4):207–218, 2013.

[43] Zachary NJ Peterson, Mark Gondree, and Robert Beverly. A position paper on data sovereignty: The importance of geolocating data in the cloud. In *3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 11)*, 2011.

[44] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open {vSwitch}. In *12th USENIX symposium on networked systems design and implementation (NSDI 15)*, pages 117–130, 2015.

[45] Lawrence G Roberts. Beyond moore's law: Internet growth trends. *Computer*, 33(1):117–119, 2000.

[46] Rami Rosen. Resource management: Linux kernel namespaces and cgroups. *Haifux, May*, 186:70, 2013.

[47] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24, 2012.

[48] Enge Song, Nianbing Yu, Tian Pan, Qiang Fu, Liang Xu, Xionglie Wei, Yisong Qiao, Jianyuan Lu, Yijian Dong, Mingxu Xie, et al. Mimic: Smartnic-aided flow backpressure for cpu overloading protection in multi-tenant clouds. In *2022 IEEE 30th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2022.

[49] George J Stigler. The economies of scale. *The Journal of Law and Economics*, 1:54–71, 1958.

[50] Luis M Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45–52, 2011.

[51] Henry Wang. Algorithmic longest prefix matching in programmable switch, December 17 2019. US Patent 10,511,532.

[52] Jianyu Wang, Jianli Pan, Flavio Esposito, Prasad Calyam, Zhicheng Yang, and Prasant Mohapatra. Edge cloud offloading algorithms: Issues, methods, and perspectives. *ACM Computing Surveys (CSUR)*, 52(1):1–23, 2019.

[53] Tao Wang, Xiangrui Yang, Gianni Antichi, Anirudh Sivaraman, and Aurojit Panda. Isolation mechanisms for {High-Speed}{Packet-Processing} pipelines. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1289–1305, 2022.

[54] Chengkun Wei, Xing Li, Ye Yang, Xiaochong Jiang, Tianyu Xu, Bowen Yang, Taotao Wu, Chao Xu, Yilong Lv, Haifeng Gao, et al. Achelous: Enabling programmability, elasticity, and reliability in hyperscale cloud networks. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 769–782, 2023.

[55] Timothy Wood, Prashant J Shenoy, Alexandre Gerber, Jacobus E van der Merwe, and Kadangode K Ramakrishnan. The case for enterprise-ready virtual private clouds. In *HotCloud*, 2009.

[56] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE international symposium on high performance computer architecture (HPCA)*, pages 331–344. IEEE, 2019.

[57] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, et al. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1345–1358, 2022.

[58] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. High-density multi-tenant bare-metal cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 483–495, 2020.

[59] Shunmin Zhu, Jianyuan Lu, Biao Lyu, Tian Pan, Chenhao Jia, Xin Cheng, Daxiang Kang, Yilong Lv, Fukun Yang, Xiaobo Xue, et al. Zoonet: a proactive telemetry system for large-scale cloud networks. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*, pages 321–336, 2022.

Appendices

A High Availability with Hot Standby

In a production environment, we deploy at least two LuoShen gateways in a hot-standby mode to achieve high availability, as shown in Fig. A1. The same components of the two LuoShen gateways will advertise the same VIP to the outside so that the upstream switches/routers can conduct ECMP load balancing to split the incoming traffic equally between the two components. With such design, we can easily scale out our gateway system to handle the edge cloud traffic growth in the future. For example, in Fig. A1, two CGW components advertise the same VIP to the upstream BSW, which will conduct ECMP load balancing on the traffic towards CGW. Except for traffic load balancing, the same components of the two LuoShen gateways can also back up for each other for fast failure recovery. For example, in Fig. A1, two IGW components advertise the same VIP to the upstream BSW, which will conduct ECMP load balancing on the traffic towards IGW. However, at a certain time, if there is a link or component failure of the IGW in one gateway, it will trigger BGP route withdrawal in the upstream BSW. After that, traffic will be forwarded by the upstream BSW to the remaining IGW. According to our measurement, the BGP route withdrawal will be quickly completed in milliseconds during a link failure.

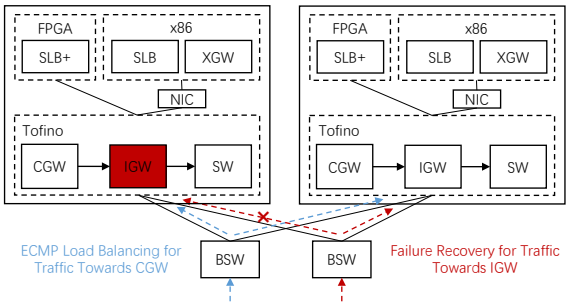


Figure A1: In production, LuoShen achieves load balancing and high availability with hot-standby deployment.

B Performance Test Topology

In order to pressure test LuoShen's performance, we build a test topology as shown in Fig. A2. As mentioned in §4.1, the Tofino chip in LuoShen only exposes the 16 ports of its Pipeline 0 after pipeline folding and 4 of them are connected with the CPU and FPGA. Therefore, we use a traffic generator to inject 1.2Tbps traffic through 12 optical fibers into the remaining 12×100G ports. By changing the traffic header fields (*e.g.*, VNI, DIP), we can conduct pressure tests on different traffic routes inside LuoShen for different cloud services.

C Calculation of Cost, Size and Power

When deploying the role-splitting gateway architecture of the public cloud at the edge, due to the reduced traffic volumes at the edge, we reduce the original role-based clusters of devices

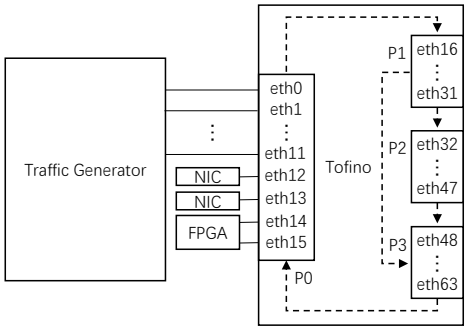


Figure A2: Performance test topology.

to multiple devices accordingly, with one device for a particular cloud function. In this way, we will have a device list including an x86 server attached with an FPGA (2U) to implement SLB+, two x86 servers (2*2U) to implement XGW and SLB, three P4-based gateways (3*2U) to implement VGW, IGW and TGW, three switches (3*1U) to implement CSW, LSW and the original SW. Then, we calculate the cost, size and power of LuoShen and the role-splitting architecture, respectively. ① Upfront cost: As the exact cost numbers of FPGA, x86 server, P4-based gateway and LuoShen are confidential, we normalize them to 1:10:10:15 according to our experience (for the sake of simplicity, we omit the cost of switches). Accordingly, the cost of LuoShen is 15 while the cost of the role-splitting is (1+10)+2*10+3*10=61. ② Deployment footprints: LuoShen will occupy 2U while the role-splitting will occupy 2U+2*2U+3*2U+3*1U=15U. ③ Power consumption: Generally, an FPGA consumes 100W [11, 57], an x86 server consumes 500W [57] and its CPU consumes 200W [7], a P4-based gateway consumes 300W [39]. Therefore, the power consumption of the role-splitting architecture is at least (100W+500W)+2*500W+3*300W=2500W (we still omit the power consumption of switches). LuoShen contains one P4 switch, two CPUs and one FPGA. Besides, the optical modules, fans, disks will additionally consume around 200W. Therefore, the power consumption of LuoShen is around 300W+2*200W+100W+200W=1000W.

D Acronym Definition

Table A1: Definition of vendor-specific acronyms.

Acronym	Expansion
SLB	Server Load Balancer
XGW	eXtendable Gateway
IGW	Internet Gateway
VGW	Virtual Private Gateway
TGW	Transit Gateway
CGW	Cloud Gateway
vSwitch	Virtual Switch
NC	Node Controller
SW	Switch
CSW	Customer Switch
LSW	Integrated Access Switch
BSW	Border Switch
IDC	Internet Data Center
VBR	Virtual Border Router

Sprinter: Speeding Up High-Fidelity Crawling of the Modern Web

Ayush Goel[1] Jingyuan Zhu[1] Ravi Netravali[2] Harsha V. Madhyastha[3]
[1]*University of Michigan* [2]*Princeton University* [3]*University of Southern California*

Abstract—**Crawling the web at scale forms the basis of many important systems: web search engines, smart assistants, generative AI, web archives, and so on. Yet, the research community has paid little attention to this workload in the last decade. In this paper, we highlight the need to revisit the notion that web crawling is a solved problem. Specifically, to discover and fetch all page resources dependent on JavaScript and modern web APIs, crawlers today have to employ compute-intensive web browsers. This significantly inflates the scale of the infrastructure necessary to crawl pages at high throughput.**

To make web crawling more efficient without any loss of fidelity, we present Sprinter, which combines browser-based and browserless crawling to get the best of both. The key to Sprinter's design is our observation that crawling workloads typically include many pages per site and, unlike in traditional user-facing page loads, there is significant potential to reuse client-side computations across pages. Taking advantage of this property, Sprinter crawls a small, carefully chosen, subset of pages on each site using a browser, and then efficiently identifies and exploits opportunities to reuse the browser's computations on other pages. Sprinter was able to crawl a corpus of 50,000 pages 5x faster than browser-based crawling, while still closely matching a browser in the set of resources fetched.

1 Introduction

To make the most of the enormous trove of information available on the web, all of us today rely upon a range of efforts. Web search engines help users find pages relevant to their needs. Data from the web serves as input to smart assistants such as Siri and Alexa, and is used to train generative AI models that can answer our questions. Web archives store repeated snapshots of web pages to document changes over time and to preserve the content of deleted pages. Researchers continually study the web to help improve its performance and security.

A key enabler for all of the above is a capability that we take for granted today: the ability to crawl the web at scale. Web crawlers have traditionally crawled a page by first downloading the page's HTML, and then recursively fetching all embedded links to images, CSS stylesheets, scripts, etc. If one deploys many such so called static crawlers [32, 28] across a fleet of machines, the rate of crawling is limited by the network bandwidth of each machine.

Given that web crawlers have existed for over three decades, why revisit this topic now? Because, static crawlers no longer suffice. On today's web, the URLs of many of the resources on a page are determined at runtime, rather than being statically embedded in the page's source. To discover and fetch such resources, modern "dynamic" crawlers [5, 2, 4] leverage web browsers such as Chrome, Firefox, or Edge. However, due to the compute overheads associated with JavaScript (JS) execution and with browsers in general, the rate at which one can crawl pages drops by an order of magnitude relative to static crawling (Figure 1). Consequently, dynamic crawlers need to be deployed across a much larger number of servers in order to sustain the same crawling throughput as that feasible with static crawlers.

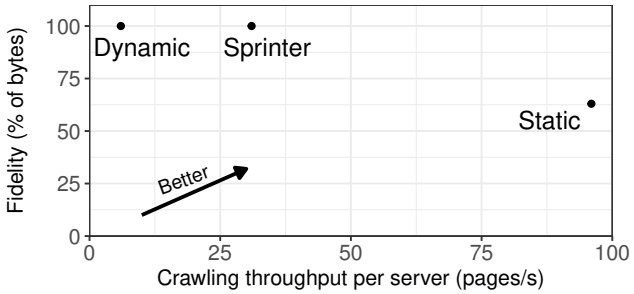


Figure 1: **Tradeoff between fidelity and performance with different crawlers.**

Thus, anyone seeking to crawl the web today has to make do either with the poor performance of dynamic crawlers or the incompleteness of static crawlers. Unfortunately, there is no easy fix. One could try to augment a static crawler with a lightweight JavaScript execution engine, but keeping up with constantly evolving web APIs is a challenge best left to the developers of widely used browsers. On the other hand, proposals that attempt to mitigate the impact of client-side web computations on user-perceived web performance have little utility in the context of crawling. For example, overlapping the browser's computations with its network activity [41, 24, 56] or parallelizing the browser's execution of JavaScripts [40] can reduce page load times, but crawling throughput remains unchanged since the total amount of client-side computation is the same.

We address this undesirable status quo with Sprinter, a new crawler which judiciously combines browser-based and browserless crawling. Sprinter crawls pages at a much faster rate than dynamic crawlers while matching them in the resources fetched. Our main observation is that large-scale web crawling workloads typically include many pages from each site and there is significant potential to reuse client-side computations across pages (§3.1).To exploit this property, our design of Sprinter is based on three key principles.

First, when Sprinter crawls a page using a browser, it strives to minimize the amount of JS code executed. For every script file on a page, Sprinter attempts to reuse the

browser's execution of that file on a previously crawled page. In user-facing page loads, execution of the same file is seldom exactly identical across multiple pages. In contrast, Sprinter can reuse JS execution at such a coarse granularity because it can skip executing a JS file as long as the URLs of the resources that file would fetch match those fetched during a prior execution of that file.

Second, even if none of the JS files on a page are executed, crawling the page with a browser imposes significant compute overhead. Therefore, on any site, Sprinter crawls the vast majority of pages on the site without a browser. To realize browserless crawling that does not sacrifice fidelity, we implement a lightweight page instrumentation framework that tracks the web APIs used on any page without support for executing these APIs. When it crawls a page without a browser, Sprinter uses this instrumentation to identify whether it can safely reuse JS executions from pages that it previously crawled with a browser.

Lastly, to maximize the fraction of pages that can be crawled without a browser, Sprinter crawls the pages on a site in a carefully chosen order. For any given site, Sprinter efficiently identifies a subset of pages such that most of the scripts seen on other pages are fetched as part of this subset. Sprinter crawls these pages first using a browser and captures the effects of JS executions. Most of the remaining pages can then be crawled without a browser, since Sprinter can identify all resources to be fetched on those pages without executing any JS code or web APIs.

We used Sprinter to crawl a corpus of 50,000 pages spread across a diverse collection of 100 sites. It offered a 5x speedup in crawling throughput compared to existing dynamic crawlers. When we recrawled the same corpus a week later, the rate at which Sprinter crawls pages improved by a further 78%. Importantly, Sprinter preserves almost all resource fetches issued by a browser-based crawler, and it is compatible with legacy web servers. Sprinter's source code is available at https://github.com/goelayu/Sprinter.

2 Background and Motivation

We begin by describing common web crawling workloads and quantifying the limitations of existing strategies for supporting these workloads.

2.1 Target workloads

Web crawlers take as input a seed list of URLs to pages that need to be crawled. The input configuration to the crawler can specify a range of options such as timeout per page, retry policy, politeness constraints (i.e., time gap between crawls of pages on the same site), and whether other pages discovered while crawling the seed list should also be recursively crawled. Some crawlers provide the option of saving page screenshots [2] and triggering user interactions (e.g., scrolling or clicking) on rendered pages [4]. In this work, we focus on supporting the common need for crawlers to save the content of resources that are fetched on every page that is crawled. To not make any assumptions about what the crawls will be used for, we aim to fetch and save all page resources requested by a browser such as Chrome, rather than a subset that may suffice for a particular use case.

We focus on supporting workloads where pages are crawled from a large number of sites. This is the case in any large-scale system that relies on web crawls, e.g., to support web search, ChatGPT, and Siri, their providers aim to crawl the entire web. Even in more focused crawls, it is common to crawl many sites and many pages in each site. For example, after every presidential term in the US, the Internet Archive captures a snapshot of 1.3 million government websites, crawling roughly 700 pages on average per site [14]. Similarly, research studies attempting to understand the web's security vulnerabilities [44] have crawled roughly 2500 pages per site. When pages are crawled from a single site (e.g., a research study of pages on Facebook), the rate at which pages can be crawled is constrained by the rate limits imposed by the site being crawled.

2.2 Shortcomings of static crawling

As mentioned earlier, web crawling has traditionally relied on static crawlers, which identify all the resources to fetch on every page by extracting links embedded in the page's source. To demonstrate and quantify why static crawlers are now insufficient, we compile $Corpus_{10k}$, a collection of 10,000 pages comprising 100 randomly sampled pages from each of 100 sites: roughly 33 sites chosen at random from three ranges – [1, 1000], [1000, 100k], and [100k, 1m] – from Alexa's site rankings. This corpus spans a diverse collection of sites and is representative of real-world crawls in that it includes a large number of pages per site crawled.

On a server which has a 16-core 2.1 GHz Intel Xeon CPU, a 1 Gbps network connection, and a 500 GB SSD disk, we crawl every page in $Corpus_{10k}$ using a custom crawler which loads every page in Google Chrome but also fetches all URLs, both absolute and relative, that are embedded in text-based resources (i.e., HTML, CSS, and JS). We record all responses using a web record and replay tool [9]. We then separately crawl every page from our recorded copy once using Chrome and once using our custom static crawler (which mimics wget2 [12], a state-of-the-art static crawler), with network caching enabled in both cases, i.e., across all pages, each unique resource was only fetched once. Comparing the two types of crawlers in this manner eliminates any differences that might arise due to server-side nondeterminism [36].

First, the *"Dynamic - Static"* line in Figure 2 shows that a static crawler fails to fetch 32% of bytes on the median page. This is because, on a modern web page, which resources are served to a client are often determined only when the client executes the scripts included on the page. Since a static crawler can identify the URLs of a page's resources

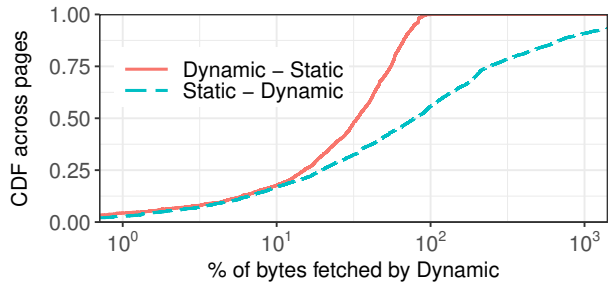


Figure 2: **Compared to a dynamic (i.e., Chrome-based) crawler, a static crawler both fails to fetch some resources and fetches many additional resources. Distribution shown is over 10,000 pages spread across 100 sites. Note logscale on x-axis.**

9297.js

```
var EA = fetch("crazyegg.com/usnews.com.json")
// json contents:  {
      script_url: "crazyegg.com/commonscripts/759.js"
}
```

utag.js

```
const n = document.createElement("script");
n.src = EA.script_url;
const r = document.getElementsByTagName("script")[0];
r.parentNode.insertBefore(n, r);
```

Figure 3: **Snippet of JS code from www.usnews.com. The browser first fetches a JSON file, and then requests a JS file referenced inside the JSON.**

only by parsing the source code of the page, it is blind to such resource fetches. Figure 3 shows an example.

Second, Figure 2's *"Static - Dynamic"* line shows that, on the median page, the static crawler fetches 93% more bytes than fetched by Chrome; on some pages, this overhead is as high as 200x. These extra resource fetches arise because, within a single page, web developers often embed resources that are applicable across a large number of client device types, expecting the client browser to download the resources applicable to it. Examples include multiple resolutions of the same image, or different font files for the same HTML text. To enable the client to pick the appropriate version of any particular resource, modern pages either use media queries [17] or CSS selectors [11]; see Figure 4 for examples. A static crawler is unable to evaluate media queries and does not know which CSS selectors are dynamically applied during JavaScript execution. Therefore, it offers no control on whether to fetch only resources applicable to the machine used for crawling a page or to fetch every resource that might be requested in any load of the page.

2.3 Compute overheads of browser-based crawling

Given the shortcomings of static crawlers, state-of-the-art web browsers are often employed to crawl pages [2, 5, 4]. We observe that Chrome is the most widely used in browser-based crawling frameworks because of its better support for web APIs [18] and for automation capabilities [8]. For this

index.html

```
<picture>
    <source srcset="ct.img/600x338" media="(min-width:768px)">
    <source srcset="ct.img/400x225" media="(min-width:0px)">
</picture>
```

style.css

```
.icon-calendar
      {font-family: {src: url("fonts/icomoon.woff")}}
```

widget.js

```
if (body.firstChild.hasAttr("data-widget")){
      var inode = document.createElement("i");
      inode.class = "icon-calendar";
      body.firstChild.insertBefore(inode)
}
```

Figure 4: **Code snippet from www.chicagotribune.com showing the two causes for a static crawler's extra resource fetches. (a) It will fetch both versions of the `ct.img` image, irrespective of the width of the client device's display. (b) It will fetch the font file `fonts/icomoon.woff`, whether or not the CSS selector `.icon-calendar` is used in the rest of the page. The CSS selector is only added if the HTML code contains a `data-widget` attribute.**

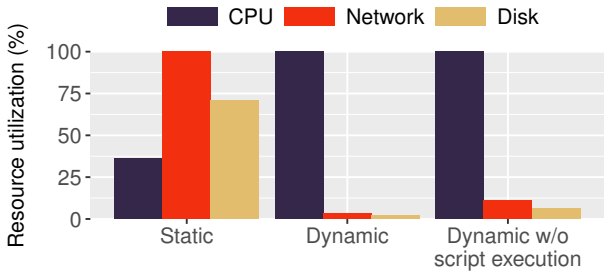


Figure 5: **A comparison of average CPU, network, and disk utilization by static and dynamic crawlers.**

reason, in the rest of this paper, we refer to Chrome[1] when discussing overheads of browser-based crawling.

We observe that the average number of pages that we could crawl per second with Chrome was only 12% of that achievable with the static crawler. The cause for this significant drop in crawling throughput is shown in Figure 5, which plots the average utilization of CPU, network, and disk with either crawler. Unlike the static crawler, which was limited by network bandwidth, the dynamic crawler ended up saturating all CPUs. If we were to use a 10 Gbps network, more than 5000 CPU cores would be necessary for the dynamic crawler to fully utilize the network, which is infeasible to accommodate on a single server.

We break down the reasons behind Chrome's high CPU usage using data from Chrome's in-built profiler [7]. We find three primary contributors: 1) the JavaScript engine, which is responsible for parsing and interpreting JS code, 2) inside the rendering engine, computation of the layout tree which

[1]We use Chrome in a headless mode as it is known to be more compute efficient [49, 21].

specifies on-screen positions for page content, and 3) time spent inside Chrome's internal code, into which the profiler has no visibility. Together, these three sources of computation account for 96% of the compute delays on the median page, with JavaScript execution alone accounting for about half. Given the complex inter-dependencies between these three tasks, none of them can be simply eliminated to reduce Chrome's computation overheads. For example, JavaScript execution queries layout information when scripts inspect the position of elements on the screen.

2.4 Minimizing browser's computation delays

The observation that the amount of client-side computation needed to load a web page has increased in recent times is not new. A large body of prior work [57, 42, 40, 41] has focused on addressing the impact of this overhead on user-perceived web performance. However, those solutions have little utility in the context of web crawling for two reasons.

First, many proposals for reducing the impact of client-side computation on page load times aim to either increase the overlap between the browser's use of the client CPU and network [41, 46] or parallelize the browser's execution of scripts on a page [40]. Such solutions can reduce the end-to-end latency of individual crawls, but the total amount of computation that the crawler needs to perform, and thus the crawling throughput, will remain unchanged.

Second, others [42, 57, 1] rely on server-/proxy-side support to ship processed versions of pages so as to minimize the amount of JavaScript that clients need to execute. Notwithstanding the fact that such solutions are not usable until they are adopted by millions of domains, we estimate their best case utility by crawling pages in $Corpus_{10k}$ with script execution in Chrome disabled. A comparison of *"Dynamic"* and *"Dynamic w/o script execution"* in Figure 5 shows that the latter marginally reduces the gap between CPU and network utilization. However, client-side computation remains a significant bottleneck, thereby limiting crawling throughput to still be only 17 pages per second.

Alternatively, one could attempt to build a lightweight browser from scratch which only supports crawling, but does not enable users to visit web pages, i.e., has no graphical interface, does not support user interactions, etc. However, significant engineering effort would be required to constantly keep up with updates in HTML, CSS, and JavaScript APIs. For example, when we load the landing pages of the top 1000 Alexa sites using a version of Chrome from five years ago (v65), it fails to fetch 16% of the resources fetched by the most recent version of Chrome (v114). This is because certain JavaScript APIs that are commonly used today were not supported by Chrome v65, e.g., support for optional chaining [15] was only added in v80. It would be best for web crawlers to rely on widely used browsers which are well-maintained, instead of having to replicate the effort in a lightweight browser dedicated to crawling.

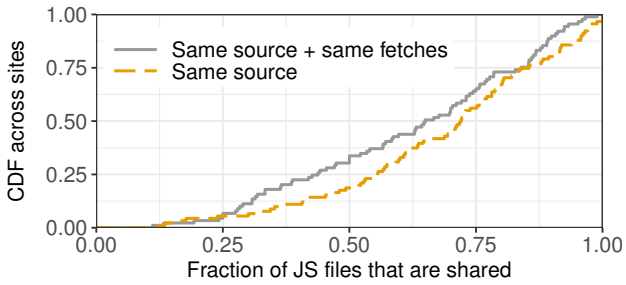


Figure 6: **For the sites in $Corpus_{10k}$, most JavaScript files appear on multiple pages and a script typically fetches the same resources on all the pages which include that script.**

3 Overview

The takeaway from the previous section is that, today, operators of web crawlers are stuck with having to choose between two less than ideal options: use static crawlers and miss out on some resources, or make do with the poor performance of dynamic browser-based crawlers. We seek to resolve this quandary by enabling high-fidelity crawling at high throughput. We do so while respecting two constraints. First, we make sure to crawl all the resources on a page that a browser would fetch, but make it configurable whether to crawl only the resources relevant to the machine on which the crawler is executed. Second, to make our crawler compatible with the legacy web, we require no changes to web pages and the servers that host them.

3.1 Observations and approach

The high-level observation that guides our approach is that, on any site, there typically is significant overlap across pages both in the JavaScript code that they include and JavaScript-initiated fetches when a browser loads them. Figure 6 demonstrates this property on the pages in $Corpus_{10k}$.

First, for every site, out of all the unique JS files seen on at least one of the 100 pages on that site, we compute the fraction which are included in multiple pages; here, we consider the combination of a file's URL and a hash of its source code to be a unique identifier for a file. The *"Same source"* line plots the distribution of this fraction across the 100 sites in our corpus. For the median site, 72% of JS files were shared across multiple pages.

Next, we examine the likelihood that a JS file fetches the same set of resource URLs when it is executed on different pages. For this, we consider a script file's execution uniquely by the file's URL, the hash of its source, and the set of URLs it fetches. When we consider only those executions which result in at least one fetch, the *"Same source + same fetches"* line in Figure 6 shows that, on the median site, 65% of unique file executions – at least with respect to resource fetches – are repeated across multiple pages.

The takeaway from these observations, coupled with the property that web crawling workloads typically crawl a large number of pages per site (§2.1), is that there exists signifi-

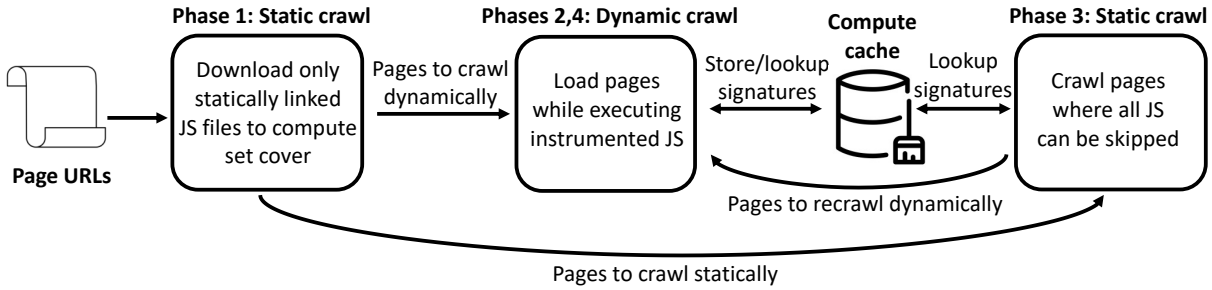


Figure 7: **Sprinter crawls the pages on any site in four phases which alternate between browserless and browser-based crawling.**

cant redundancy in a dynamic crawler's execution of JS files. When the browser used by the crawler executes a JS file that it had previously executed on a different page on the same site, the numbers from Figure 6 indicate that the same set of resources are often requested as on the previous page. The browser's network cache will ensure that it does not have to waste network bandwidth in re-downloading those resources. But, the browser will still execute every JS file in its entirety just to identify these resources.

To improve crawling performance by reducing the crawler's computations, our approach aims to first eliminate redundant execution of JS files. Specifically, whenever our crawler, Sprinter, crawls any page, it skips executing a JS file if a) it has already executed that file while crawling a different page on the same site, and b) it identifies that, if executed, the file will fetch the same resources as it did on the previously crawled page. However, as observed earlier (§2.4), a browser imposes high compute overhead even when it is used to load pages with execution of scripts disabled. Therefore, second, on pages where it can reuse the executions of *all* JS files, Sprinter does not even employ a browser to crawl those pages. Put together, Sprinter uses a browser to crawl only a small subset of pages in each site and minimizes the browser's execution of JavaScripts.

3.2 Challenges

Realizing the above approach requires us to answer the following three questions.

- Whenever a script appears on multiple pages, it is not guaranteed to initiate the same resource fetches on all pages; in our corpus, 48% of repeated scripts had at least one execution where they fetched a different set of URLs than what they fetched in their first execution. Prior to executing a script, how can Sprinter efficiently identify that the script's execution will match a prior execution, and it is safe to skip executing it?
- Classic memoization involves storing the results of execution and using them to skip future executions of the same code in the same runtime context. In contrast, when Sprinter crawls a page without a browser, how can it reuse the browser's prior computations on other pages? Mimicking the entire browser runtime will significantly increase complexity and degrade performance.
- Finally, on each site, which subset of pages should Sprinter crawl using a browser? To minimize Sprinter's compute overheads, it is key that the subset be small. However, for Sprinter to crawl all the remaining pages on the site without a browser, we must ensure that the script executions on the pages crawled using a browser suffice to skip executing all the JS files on the remaining pages.

4 Design

As shown in Figure 7, Sprinter crawls a corpus of pages from any particular website in four phases. In the first phase, Sprinter identifies the subset of pages that need to be crawled with a browser. It crawls those pages in the second phase while skipping JS executions whenever feasible. Next, Sprinter crawls the remaining pages on the site using its augmented static crawler. Finally, it recrawls some of the pages from the third phase using a browser. We present our design of Sprinter by first describing its operation in phases 2 (§4.1) and 3 (§4.2), and lastly, phases 1 and 4 (§4.3).

4.1 Memoizing JavaScript execution

Sprinter maintains a *compute cache* in order to take advantage of the opportunities to reuse JS executions across the pages on a site. On any page that Sprinter crawls with a browser, prior to executing JS on the page, the browser looks up the compute cache to determine whether the execution can be skipped. Upon a cache miss, the browser executes the JS code and logs a summary of its execution in the compute cache, for use on other pages.

Execution signatures to enable reuse. When JS code runs within a browser, it can read from or write to the JavaScript heap and HTML DOM object. It can also read the return values from various web APIs. Therefore, to enable reuse of JS executions without violating correctness, we associate the execution of every block of JS code with a *signature* which includes the values at the start of executing that block of code for all state from the heap or DOM that is read within that block. When the browser executes any block of JS code, its execution is guaranteed to result in the same externally visible effects (i.e., writes to the DOM and heap, and URL fetches) as a prior execution which had the same signature, if the block does not invoke any non-deterministic APIs (e.g., `Date`, `Random` or `Performance`). Figure 8 shows an example block of code and the corresponding signature.

vendor.js

```
var gKey = window.grumi.key; // “bfd2-4adc”
fetch(`https://www.geoedge.com/${gKey}/grumi-ip`)
var NYTD = {PageViewID:‘mubjhislka7867’};
window.NYTD = NYTD;
```

$\text{Signature}_{\text{vendor.js}}$

```
{
      Reads: [“window.grumi.key”,”bfd2-4adc”],
      Writes: [“window.NYTD”,”{PageViewID:‘mubjhislka7867’}”]
      Fetches: [“https://www.geoedge.com/bfd2-4adc/grumi-ip“]
}
```

Figure 8: **JavaScript code from www.nytimes.com which reads a global variable using the *window* object and, based on the property read, fetches a URL. It also writes to the window object. Signature for this includes the global state read and written (both the keys and the values) and the fetches initiated.**

However, to construct code signatures, modern browsers provide no APIs to extract the necessary runtime information about JavaScript execution. To remedy this, Sprinter uses a custom JS instrumentation framework, similar to the ones used in prior work [31, 40, 42]. This instrumentation framework runs inside a man-in-the-middle (MITM) proxy which sits in front of the browser. For every new JS file requested by the browser, the proxy statically analyzes the code in the file and rewrites it by injecting code that tracks the state and APIs that are accessed when the file is executed. Sprinter's instrumentation tracks variables on the heap which are in either 1) the global scope, which is accessible using the `window` object, or 2) the closure scope, which is created within a function but persists after the function's execution if there exists a nested function declared in the same enclosed scope. For the DOM object, Sprinter tracks all APIs that can read from (e.g., `getElementById`) or write to (e.g., `appendChild`) the DOM.

Once the browser finishes loading a page, Sprinter's injected JS code compiles signatures for the scripts on the page and stores them in the compute cache which is co-located with the proxy. These signatures include both the above-mentioned information needed to identify the opportunity for reuse, and the writes to the heap and DOM that need to be executed when the corresponding code is skipped, along with any fetches initiated; see Figure 8. When a previously cached JS file is fetched in future page loads, the proxy embeds stored signatures for the code in this file directly into the file. When processing each JS file, the browser uses the embedded signatures to determine if any code within the file can be skipped.

Granularity of JS execution reuse. Given our results from §3.1, it is natural to try and reuse the browser's JS executions at the granularity of entire files, i.e., prior to processing any script file, the browser uses cached signatures for that file to determine whether to skip all the code in the file or execute all of it. However, as shown in the *"Full signature"* line in Figure 9, the cache hit rate is pretty poor. On the median site in $Corpus_{10k}$, only 40% of JS file executions can be skipped.

Figure 9: **Cache hit rate for JavaScript files that initiate fetches for other URLs.**

To improve the hit rate, our key insight is that, unlike in user-facing page loads, we do not need to restrict Sprinter's skipping of a JS file's execution only when it is guaranteed to execute in a manner *exactly* identical to a previous execution of the same file. Rather, as long as we can guarantee that the code will fetch the same resource URLs, we can skip it. A crawler does not need to preserve other aspects of JavaScript execution, such as visual changes by modifying the DOM or functional changes by adding event handlers that allow users to interact with the page.

This observation enables us to trim file signatures and only include state that influences resource fetches. To identify this state, we turn to dynamic taint tracking [47]. Our instrumentation of any JS file marks all statements that initiate URL fetches (such as `XMLHTTPRequest.send`) and all DOM nodes with a *src* property as sinks. We also mark all control-flow statements as sinks. At the end of any file's execution, we include in the file's signature only those reads which propagate values to any of the sinks.

The *"Trimmed signature"* line in Figure 9 shows that trimming the signatures stored in Sprinter's compute cache improves the cache hit rate on the median site to over 80%. This is because a large fraction of reads performed by JS on the web does not influence the set of URLs fetched. Furthermore, we see that the cache hit rate with Sprinter is close to the best achievable hit rate, which we obtain via post-hoc analysis of JS executions to identify when the set of URLs fetched matched a prior execution. The gap between *"Trimmed signature"* and *"Oracle"* is due to Sprinter's conservative tracking of all control-flow dependencies, instead of only the ones that influence the URLs fetched.

4.2 Statically crawling pages

So far, we have discussed how Sprinter reuses execution across pages. However, as mentioned in §2.3, JS execution is only a part of the total compute overhead of web browsers. To maximize Sprinter's performance, we now discuss how it crawls pages without a browser in phase 3.

Crawling without a browser. We observe that the primary utility of crawling a page within a browser is its implementation of the JavaScript heap and the DOM object, and its

support of various APIs. However, if we are able to skip executing a file, we only need to compile the read state in its signature, for which we need a log of all the writes performed by previously executed or skipped JS files. We do not need to apply these writes to the browser's heap and DOM since there are no user interactions at the time of crawling.

Based on this insight, Sprinter's static crawler maintains a shadow heap, which is a key-value map from the properties of the heap to the corresponding values. It also maintains a shadow DOM, which it constructs by parsing the page's HTML at the start of every page load and offers the same read and write APIs as the ones provided by the browser.

For every page that it statically crawls in phase 3, Sprinter fetches the page's HTML, extracts all embedded resource URLs, and recursively fetches them. For every JS file fetched, the static crawler looks up the shadow heap and DOM to construct the file's signature. Upon a successful cache hit, Sprinter logs the writes included in the file's signature to the shadow heap and shadow DOM, and issues any resource fetches included in the signature. It repeats this process until all resources on the page have been fetched. Whenever there is a cache miss for a JS file, the static crawler is unable to execute the file, and it defers these pages for browser-based crawling in phase 4 (§4.3).

Handling additional fetches. Crawling pages as described above has the downside of fetching additional resources that a browser would not (as described in §2.2). For *Corpus*$_{10k}$, this increases the total number of bytes fetched by 3.5x. Unlike during dynamic crawling, when the network is severely underutilized (Figure 5), these additional fetches significantly degrade overall throughput when crawling without a browser.

If the input configuration to Sprinter specifies that only the resources relevant to the machine executing the crawler be downloaded, its static crawler does so by leveraging the browser's processing of pages crawled earlier in phase 2. First, during every page load executed within a browser, Sprinter adds to its compute cache the media queries evaluated and the corresponding value (true or false). For any media query encountered during browserless page loads, the static crawler fetches the corresponding URL if the compute cache either returns a true value or does not contain any entry for that media query. Similar to our observation of similarity in JS executions across pages, we find that, for the median site in *Corpus*$_{10k}$, 92% of all media queries occur on more than one page. Second, the static crawler uses the cached signatures for JS files to identify which selectors were applied when the browser executed those files. It fetches only the URLs contained within these selectors.

4.3 Scheduling page crawls

Given the high compute overhead of loading pages in a browser and extracting signatures, we must minimize the number of pages that Sprinter crawls using a browser. However, Sprinter can crawl a page without a browser only if it is able to skip executing *every* JS file on that page. Hence, the subset of pages on any site that Sprinter crawls without a browser in phase 3 should ideally be such that all of the JS files that appear on any of these pages also appear in at least one of the pages previously crawled with a browser in phase 2. This does not guarantee that the static crawler will find a compute cache entry with a matching signature for every JS file, but at least makes it possible.

Figure 10: **Approximate set cover captures a large fraction of JS files ("JS"), while the number of pages in the set cover ("Pages") are a small fraction of the total corpus size.**

Need for scheduling. Since the set of JS files on any page is not known apriori, Sprinter could use a browser to crawl the pages on any site in a random order and switch to browserless crawling once the set of JS files converges, i.e., once the union of JS files remains unchanged for n consecutive pages crawled. But, we find that there is no value of n that offers a good tradeoff between compute overheads and coverage of JS files. For example, with $n = 2$, we would need to crawl only 8% of pages on the median site in *Corpus*$_{10k}$ with a browser, but only 49% of the JS files seen across all the pages on this site appear on those pages. With $n = 10$, the fraction of JS files covered by browser-based loads increases to 82%; however, 38% of pages now need to be crawled using a browser.

Efficient identification of set cover. Sprinter takes an alternate approach of carefully selecting which subset of pages on each site to crawl using a browser in phase 2. Though we cannot predict which JS files are on the remaining pages, we leverage our finding from §3.1 that the same JS file often fetches the same resources across pages of a site. Therefore, instead of finding a subset of pages that includes *all* the JS files used on that site, we find a subset that includes all the JS files that are statically embedded in the remaining pages. When these files are executed as part of the browser-based loads, all the JS files that are dynamically fetched on this site's pages will likely be fetched and executed.

Thus, in phase 1, Sprinter crawls all pages using a static crawler which only fetches the JS files that are directly linked. We then have a set of JS files for every page, and Sprinter computes the set cover, i.e., the subset of sets whose union matches the union of all sets. Since computing the op-

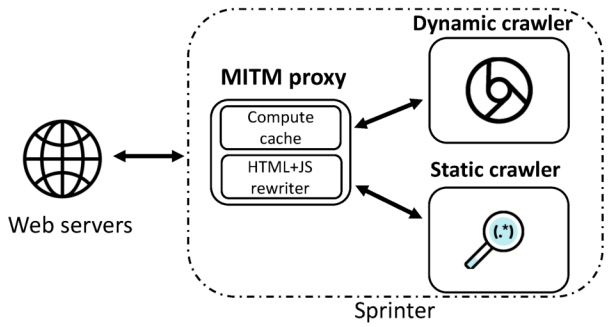


Figure 11: **Overview of our Sprinter implementation.**

timal set cover is NP-complete [29], we use a greedy heuristic which runs in polynomial time and is known to closely approximate the optimal [50]. Figure 10 shows that, on the median site in $Corpus_{10k}$, Sprinter selects only 7% of pages to be crawled using a browser, yet these pages cover over 80% of all the JS files seen across all pages on the site.

Given this methodology for choosing which pages to crawl with a browser in phase 2, there are multiple reasons why Sprinter's static crawler may not find a matching compute cache entry for every JS file that it fetches. First, since the set cover is only based on statically linked JS files, some dynamically fetched JS files may not have been encountered in the browser-based loads. Second, even for files that were executed by the browser, those executions may not have had the same signature as that expected by the static crawler. If the static crawler runs into either issue on any page, Sprinter recrawls that page using a browser in phase 4.

5 Implementation

Our implementation of Sprinter has three components, which work together as shown in Figure 11.

Dynamic crawler. Sprinter's dynamic crawler is written in 1150 lines of NodeJS code. This dynamic crawler allows Sprinter to load pages using Chrome in phases 2 and 4. To automate Chrome, the crawler uses the Puppeteer library [16]. At the end of every page load, it uses Chrome's DevTools protocol [8] to collect runtime information, and it then sends the per-file signatures that it compiles to the man-in-the-middle (MITM) proxy.

Static crawler. The static crawler, used in phases 1 and 3, is written in 920 lines of Golang. It uses the goquery library [13] to create a virtual DOM for every HTML file and the cascadia library [6] to parse and query CSS selectors.

MITM proxy. The MITM proxy is an HTTP proxy written in 350 lines of Golang. It intercepts requests and responses for every resource fetched by the static and dynamic crawlers. The proxy also statically analyzes and instruments JS files, for which it uses a static analyzer written in 1200 lines of NodeJS code. The static analyzer uses Babel [3], a JS transpiler, to create the abstract syntax tree for every JS file. While instrumenting JS files, the static analyzer enables tracking of the JS heap and DOM. Ideally *all* web APIs should be tracked to achieve perfect correctness since their return values can potentially influence URL fetches. However, our evaluation shows that the subset of APIs that our implementation currently supports – all values accessible from the `window` object directly, e.g., `window.navigator.userAgent` – largely suffices. Our finding is in accordance with prior work which studied the impact of web APIs on URL fetches [36]. The proxy also runs a gRPC server to receive signatures from the dynamic crawler.

6 Evaluation

We evaluate Sprinter with respect to the fidelity with which it crawls pages and its performance in terms of crawling throughput. We also estimate the effort that would be required to maintain its implementation over time as web APIs evolve. We compare it to various options that exist for web crawling today. Our key findings are as follows:

- When compared to dynamic browser-based crawlers, Sprinter improves crawling throughput by 5x while preserving over 99% of the bytes fetched.
- Even in comparison to prior web accelerators that rely on assistance from web servers to reduce in-browser computations when loading pages, Sprinter delivers 2.4x higher crawling throughput.
- Sprinter's performance benefits significantly improve as more pages are crawled per site, e.g., its crawling throughput increases by 2.1x when the number of pages per site goes up from 100 to 500.
- Sprinter's performance improves as the same corpus is crawled repeatedly over time, e.g., the second crawl of our corpus is 78% faster than the first run 1 week earlier.

6.1 Evaluation setup

Workload. We expand our corpus of pages to include 500 randomly sampled pages in each of 100 sites. This new corpus, which we refer to as $Corpus_{50k}$, retains the same properties as $Corpus_{10k}$: diverse set of sites, and representative of real-world crawling workloads in having a large number of pages per site. As described in §2.2, we crawl every page in this new corpus using a custom crawler which fetches the resources downloaded by either dynamic or static crawlers, and we use a record-replay tool [9] to record all request-response headers along with the corresponding payloads.

Hardware configuration and crawling methodology. We store the recorded pages in an SSD drive of a Linux server which hosts 450 web servers to concurrently service HTTP requests with the appropriate recorded content. The crawlers run on a different Linux server with a 16-core 2.1 GHz Intel Xeon CPU, 128 GB RAM, and a 1 Gbps Ethernet connection to the server housing recorded pages. Crawling performance in this setup matches what we see when crawling pages from the live web, but eliminates the impact of any server-side effects on our evaluation of performance and fidelity.

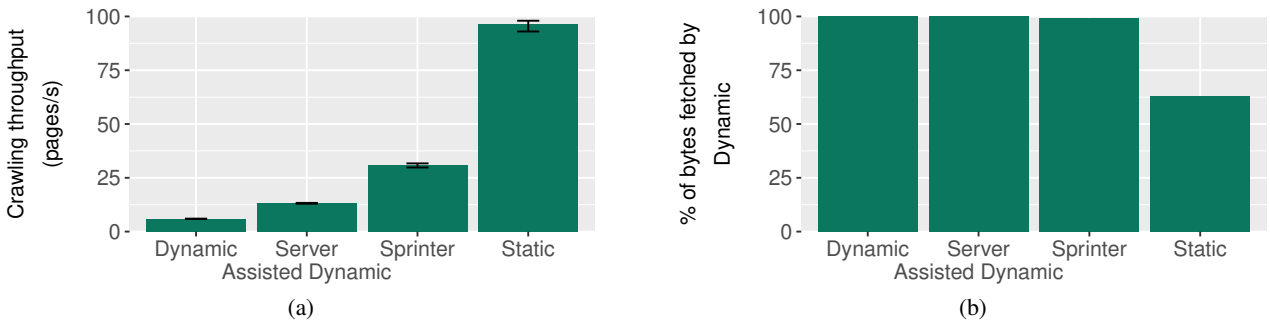


Figure 12: **Comparison of (a) crawling throughput and (b) fidelity of Sprinter against the three baselines.**

Baselines. Our primary baselines represent existing static and dynamic crawlers. For the static approach, we port wget2, a popular open-source crawler, to be compatible with our proxy-based setup; we verified that our static crawler is identical to wget2 in terms of fetched content. For the dynamic approach, we first considered three popular open-source crawlers: Archivebox [2], Browsertrix [4], and Brozzler [5]. However, our benchmark results for each revealed substantial performance drawbacks, likely because their primary goal was high fidelity, not necessarily high throughput. Specifically, undue overheads stem from spawning a new browser instance for each crawled page, using a CPU-intensive MITM proxy, and relying on an outdated Chrome automation framework. Therefore, we instead built an in-house Chrome-based crawler that achieves 20%, 33%, and 250% higher throughput than Archivebox, Browsertrix, and Brozzler, respectively. We verified that our custom crawler fetches the same set of resources as Archivebox when used to crawl the landing pages for the 100 sites in $Corpus_{50k}$.

Our third baseline is representative of prior server-/proxy-assisted solutions to reduce client-side computations in user-facing page loads [42, 57]. To the best of our knowledge, none of these systems are open sourced, and we are unaware of any domains that have adopted these techniques. Therefore, to evaluate Sprinter against this prior work, we consider the best case outcome of these systems, where *all* client-side JS execution is eliminated. We mimic such a scenario by using our Chrome-based crawler to crawl a version of every page wherein we include links to all the resources fetched by JS files in the page's main HTML. The browser loads this modified HTML with JS execution disabled. We refer to this baseline as *server assisted dynamic* crawling.

Metrics. We measure the crawling throughput of each crawler as the average number of pages it can crawl per second on a single server. For each crawler, we run a sufficiently large number of instances so as to saturate either the CPU or the network. We expect crawling throughput to linearly increase with the number of servers. We run 5 trials for each experiment and plot the median value, with error bars plotting the minimum and the maximum values.

We consider the default goal of crawling to be to match a Chrome-based crawler. Therefore, we measure the fidelity offered by a crawler as the fraction of bytes it fetches of all the resources fetched by Chrome when crawling the same pages. When the goal is to crawl all resources that are relevant to any client device, we measure fidelity as the fraction of bytes fetched out of the union of the resources fetched by the static and dynamic crawlers.

6.2 Throughput and Fidelity

6.2.1 *Comparison with baselines*

To compare Sprinter with the three baselines, we load pages in $Corpus_{50k}$ using each of the four crawlers separately. We monitor the resources fetched by each crawler on every page, and the total time taken to finish crawling the entire corpus. We also monitor the CPU and network utilization to identify the bottleneck for each crawler.

Figure 12(a) plots the crawling throughput achieved with each crawler, and Figure 12(b) shows the fidelity achieved.[2] Static crawler achieves the best crawling throughput by far of 96 pages per second. However, it misses out on 37% of the bytes fetched by the dynamic crawler. In contrast, the dynamic crawler could only crawl at a rate of 6 pages per second. Since CPU utilization was at 100% throughout the entirety of the crawl with the dynamic crawler, throughput increased to 13 pages per second with the server-assisted dynamic crawler, which does not execute any JS.

Sprinter offers a significant additional speedup, improving crawling throughput to 31 pages per second, a 5x improvement relative to the dynamic crawler. Importantly, it does so without requiring any changes to the web and while preserving 99.2% of the bytes fetched by the dynamic crawler. The 0.8% of bytes that went unfetched stem from the incomplete support for all web APIs in our current implementation. 50% of these unfetched bytes correspond to JavaScript files, 27% to images, and 17% to HTMLs, with the remaining accounted for by CSS and other content types. While no resources went unfetched on the median page, the 90th percentile page was missing 1 resource.

[2] We see no variation across runs in the resources fetched by each crawler because all of our crawls rely on one snapshot of every page crawled from the live web.

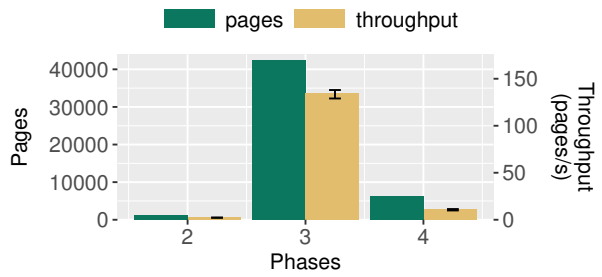


Figure 13: **Number of pages crawled during each of the different phases of Sprinter and the corresponding throughput achieved in each phase.**

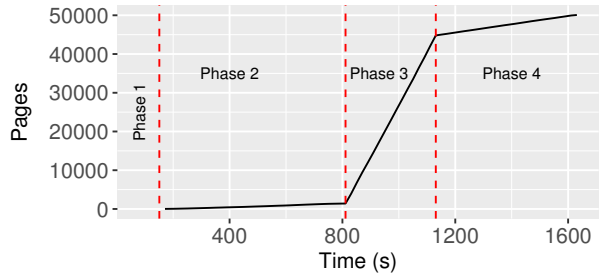


Figure 14: **A timeline of Sprinter's crawl of *Corpus*$_{50k}$, showing the duration and number of pages crawled in each phase.**

6.2.2 *Throughput in each phase*

Sprinter's crawling throughput varies widely across phases. Figure 13 plots the number of pages crawled in each phase and the corresponding throughput. Whereas, Figure 14 shows a timeline of how Sprinter's crawling of the pages in *Corpus*$_{50k}$ proceeds over time.

- No page is fully crawled in phase 1; Sprinter only statically crawls the HTML files and embedded JavaScript files for every page so as to identify the subset of pages to be crawled with a browser in phase 2. Therefore, phase 1 finishes in 151s, the quickest of all four phases.
- Phase 2 is the slowest since Sprinter not only has to crawl pages with a browser, but it also has to incur the overheads of statically analyzing and rewriting every JavaScript file, executing these instrumented files inside Chrome, and processing the information it collects to generate and store per-file signatures. In this phase, Sprinter crawls 1413 pages in 620s, resulting in a crawling throughput of a little over 2 pages per second.
- Sprinter crawls the vast majority of pages in phase 3: 42497 pages in 316s. The average throughput of 135 pages per second in this phase is even higher than what a static crawler can achieve (96 pages per second, as shown in Figure 12(a)). This is because, unlike a static crawler, Sprinter leverages browser-based execution of media queries and CSS selectors in phase 2 to eliminate fetches of resources relevant only for other client types.
- In Phase 4, Sprinter recrawls the remaining 6090 pages with a browser; about a quarter of these are because they contained a JS file not executed in phase 2, and the remaining pages incurred at least one compute cache miss. The crawling throughput of 11 pages per second in this phase is better than in phase 2 because significantly fewer JS files need to be instrumented.

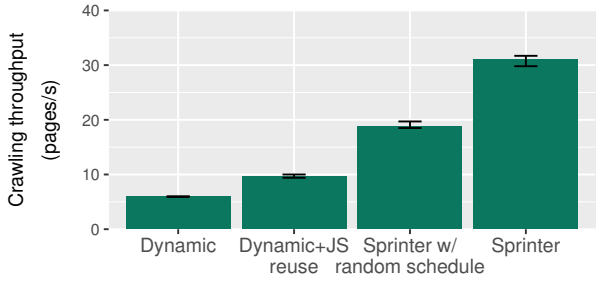


Figure 15: **Incremental benefit offered by each of the techniques used in Sprinter.**

At the end of phase 4, Sprinter's compute cache had 3089 entries. The cache hit rate of 95.6% is the key enabler of Sprinter's throughput improvements as it could crawl a large fraction of pages in phase 3, without requiring a browser. We cannot further reduce the total crawl time by immediately spawning a browser to crawl any page that incurs a cache miss in phase 3 because both phases 3 and 4 are bottlenecked by the CPU.

6.2.3 *Contribution of techniques*

To understand the performance benefits of each of the techniques used in Sprinter, we incrementally add them to the dynamic crawler and measure crawling throughput.

First, we evaluate the benefits of only using JS memoization (§4.1) in Chrome, loading all pages in the corpus in a random order. Figure 15 shows that *"Dynamic+JS reuse"* provides a roughly 66% speedup over *"Dynamic"*.

Next, we crawl some of the pages with a browser and the rest using Sprinter's augmented static crawler (§4.2). To determine which pages to crawl using a browser, we consider the strawman approach (§4.3) wherein we transition to browserless crawling once the union of JS files remains unchanged for n consecutive pages. For *Corpus*$_{50k}$, we observe that $n = 25$ results in browser-based loads fetching the same fraction of all JS files as that covered by Sprinter's chosen set cover. Even this unsophisticated combination of dynamic and static crawling – *"Sprinter w/ random schedule"* in Figure 15 – roughly doubles the crawling throughput.

Finally, by efficiently choosing a carefully chosen subset of pages to crawl with a browser, Sprinter crawls 88% fewer pages using a browser in phase 2, resulting in a further 1.6x improvement in throughput.

6.3 Sensitivity to crawling parameters

We evaluate the impact of the following three configuration parameters on Sprinter's crawling throughput: 1) the number of pages crawled per site, 2) the time gap between repeated

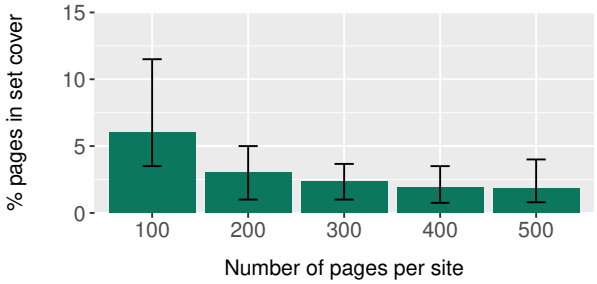


Figure 16: **Percentage of pages selected by Sprinter for browser-based crawling as a function of number of pages crawled per site. Bars show value for median site, with error bars for the 25th and 75th percentiles.**

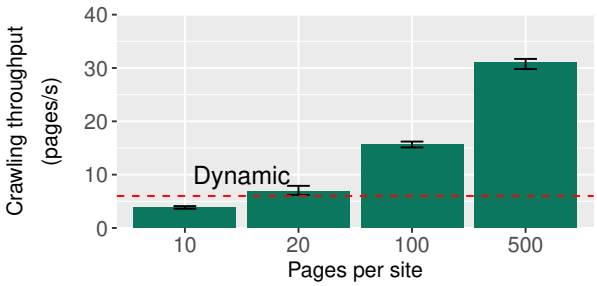


Figure 17: **Sprinter's crawling throughput as a function of the number of pages per site.**

crawls, and 3) whether fetching all statically embedded resource URLs is desired.

6.3.1 *Number of pages per site*

The key to Sprinter's high crawling throughput is its judicious partitioning of pages, crawling a small fraction using a browser and the remaining without. We examine how the fraction chosen for browser-based crawling varies as a function of the number of pages being crawled per site. For 5 different values of the number of pages per site, Figure 16 plots this fraction for the 25th, median, and 75th percentile sites. The percentage of pages in Sprinter's carefully selected "set cover" for the median site goes down from 6% with 100 pages per site to 1.6% with 500 pages per site. As a result, Sprinter is able to crawl a corpus of 10k pages at an average rate of 15 pages per second. But, for a 50k page corpus, its throughput improves to 31 pages per second (Figure 17). Akin to how a static crawler benefits more from network caching with more redundant resource fetches, Sprinter's compute cache enables it to reuse more client-side computations when it crawls more pages per site.

On the flip side, lower the number of pages per site, lower Sprinter's throughput. Figure 17 shows that, with 10 pages per site, Sprinter crawls 4 pages per second on average, which is slower than the dynamic crawler. For Sprinter to offer any benefit, we see that it must be asked to crawl at least 20 pages per site. As a result, workloads that only crawl landing pages of sites [10] will not benefit from Sprinter.

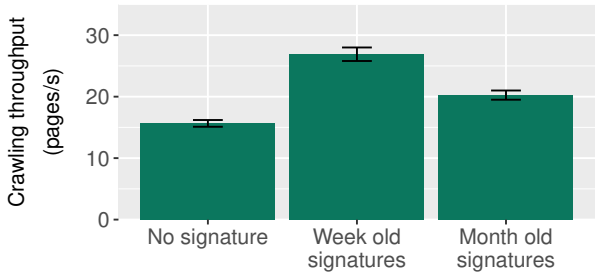


Figure 18: **Sprinter can crawl pages faster by leveraging signature information from previous crawls of the same corpus.**

6.3.2 *Repeated crawling*

In many web crawling workloads, the same corpus of pages is repeatedly recrawled. For example, a web search engine must ensure that its search index reflects the latest content on every page, and web archives must track changes to page content over time. In such cases, Sprinter will crawl the entire corpus in 4 phases the first time. However, when the corpus is recrawled, Sprinter can directly jump to crawling pages statically in phase 3, leveraging JS execution signatures from the previous crawls. Pages where no compute cache entry was found for at least one JS file would have to be recrawled with a browser in phase 4.

To measure the crawling throughput with Sprinter when the same corpus is crawled multiple times, we recrawl $Corpus_{10k}$ once three weeks after our initial crawl, and again a week later. We then use Sprinter in our replay setup to crawl pages from our last copy of the corpus. We run Sprinter once starting with an empty compute cache, once using signatures from the crawl a week before, and once using signatures from the crawl a month before.

Figure 18 shows that reusing signatures from a week ago improves Sprinter's throughput by 78% as compared to when no prior crawl existed. Reusing month-old signatures also speeds up Sprinter. But, since the compute cache entries are more stale and more previously unseen JS files are fetched, the benefits are significantly lower.

6.3.3 *Preserving static fetches*

Thus far in our evaluation, we have considered the goal of crawling to be to fetch the same resources on every page as a dynamic crawler. However, in some cases, it might be desirable to also crawl all resources that would be fetched by a static crawler. For example, web archivists might want to preserve all versions of every image on a page, so as to be able to accurately render the preserved page irrespective of the client device used to visit this page in the future.

In these cases, Sprinter can be configured to not eliminate fetches using the techniques mentioned in §4.2. The resultant throughput of Sprinter drops to 28 pages per second which, though 9% lower than when it only tries to match the dynamic crawler, is still 4.6x faster than the dynamic crawler. This drop in throughput is because of Sprinter's static crawler having to fetch additional bytes in phase 3.

Chrome version	Lightweight browser		Sprinter	
	# of APIs added	# of files added/-modified	# of APIs added	LOC added
v108	4	41	1	9
v109	3	70	1	13
v110	4	58	1	6
v111	7	109	0	0
Total	18	278	3	28

Table 1: **Comparison of number of APIs that need to handled by Sprinter and a lightweight browser.**

Note that the impact of this configuration option on Sprinter's throughput depends on the number of pages crawled per site. With more pages per site, phase 3 is able to achieve higher crawling throughput due to the benefits of network caching.

6.4 Maintainability

Web APIs and their specifications are constantly updated [38]. Web crawlers need to be correspondingly updated over time to ensure that web pages using the latest APIs are accurately crawled. Dynamic crawlers leveraging web browsers such as Chrome and Firefox simply need to update to the latest version of the browser, as these browsers are well-maintained and constantly updated to support most of the latest web APIs.

To get a measure of the effort that would be needed to maintain Sprinter or a lightweight browser such as phantomJS, we look at all the APIs added in the 4 most recent versions of Chrome (v108 to v111). For each API, we manually read its specification. Only a subset of these would need to be implemented by a lightweight browser designed for the purpose of crawling, e.g., any API that takes effect only during user interactions (such as $webRTC$ APIs to enable video conferencing or $navigator.credentials$ API to enable secure logins) would not have to be handled. Sprinter's instrumentation of JS code would need to keep track of an even smaller subset of APIs, only those which influence execution signatures, i.e., any API that can read from or write to the global state.

Table 1 compares the number of APIs that need to be tracked and implemented by Sprinter versus a lightweight browser designed for crawling. Across the four versions, a lightweight browser would be required to implement 18 APIs; in Chrome's source, these APIs touch 278 files (Chrome's commit history only shows files added/modified, not the number of lines of code). In contrast, Sprinter needs to handle only 3 of these APIs, requiring 28 lines of code.

7 Related Work

Scalable web crawling. The engineering issues associated with web crawling are well studied [33, 58, 19, 37, 22, 23]. Some of these crawlers [37, 23, 22] are able to achieve a crawling throughput of upwards of 1000 pages per server. However, all of these crawlers only download the HTML file for every page URL. In contrast, Sprinter downloads all the resources which would be fetched by browser-based crawlers such as Archivebox [2], Brozzler [5], and Browsertrix [4].

Incremental crawling A large amount of prior work [27, 39, 25, 52] has focused on incremental web crawling, i.e., how to efficiently recrawl pages. These techniques are helpful only when the same set of pages are crawled multiple times. Sprinter, on the other hand, eliminates redundant computations across pages even within a single crawl.

Resource bottlenecks of large-scale distributed systems. Prior work has studied the bottlenecks in scaling various distributed data processing workloads such as sorting [45], data analytics [43], and distributed deep learning [54, 48, 20]. These efforts first identify the hardware resource (CPU, GPU, network, or disk) that constrains overall performance, and then propose solutions to optimize the utilization of that resource. To the best of our knowledge, we are the first to study the compute bottleneck in browser-based web crawling and propose a solution to reduce its impact.

Web performance optimization. The negative impact of a web browser's computations on user-perceived latency while loading web pages is well-known [55, 56]. As discussed earlier (§2.4), proposals to lower page load times either do not reduce the total amount of computation that web clients need to perform [41, 46, 40] or require server-side changes [42, 57]. Sprinter is backward compatible with the legacy web and reduces the total amount of client-side processing by memoizing and reusing computations across pages on the same site.

Compute memoization. Memoization is widely used across different kinds of application. Prior work has leveraged such techniques to reduce compile-time latency [51, 34], improve runtime performance [30, 53], minimize scheduling overheads [26], and enable faster auditing of web applications [35]. Sprinter uses similar memoization techniques to reduce the amount of client-side computation required to crawl pages, and it maximizes its benefits by selectively identifying the state that influences URL fetches.

8 Conclusion

Over the years, crawling web pages with high fidelity has evolved from a workload that is limited by network bandwidth to a CPU-intensive one. In this paper, we showed that the key to mitigating this new bottleneck is to strategically minimize the use of the web browser and its execution of JavaScripts. Our design of Sprinter does so by efficiently identifying and exploiting opportunities to safely reuse the browser's computations across the pages on any site. We hope that our work will spur a new wave of innovation in scalable web crawling, a task that underlies many important systems in today's society.

REFERENCES

[1] Amazon silk. https://docs.aws.amazon.com/silk/latest/developerguide/what-is-silk.html.

[2] Archivebox. https://github.com/ArchiveBox/ArchiveBox.

[3] Babel. https://babeljs.io/.

[4] Browsertrix crawler. https://github.com/webrecorder/browsertrix-crawler.

[5] Brozzler. https://github.com/internetarchive/brozzler.

[6] Cascadia. https://github.com/andybalholm/cascadia.

[7] Chrome cpu profiler. https://developer.chrome.com/docs/devtools/performance//.

[8] Chrome devtools protocol. https://chromedevtools.github.io/devtools-protocol/.

[9] Chrome web page replay. https://chromium.googlesource.com/catapult/+/HEAD/web_page_replay_go/README.md.

[10] Common crawl. https://commoncrawl.org/.

[11] CSS selectors. https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors.

[12] Gnu wget2. https://github.com/rockdaboot/wget2.

[13] Goquery. https://github.com/PuerkitoBio/goquery.

[14] Internet archive end of term 2020 web crawls. https://archive.org/details/EndOfTerm2020WebCrawls.

[15] Optional chaining. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Optional_chaining.

[16] Puppeteer. https://pptr.dev/.

[17] Using media queries. https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries/Using_media_queries.

[18] Web APIs. https://caniuse.com/?compare=chrome+114,firefox+113&compareCats=all.

[19] F. Ahmadi-Abkenari and A. Selamat. An architecture for a focused trend parallel web crawler with the application of clickstream analysis. *Information Sciences*, 2012.

[20] T. Akiba, K. Fukuda, and S. Suzuki. Chainermn: Scalable distributed deep learning framework. *arXiv preprint arXiv:1710.11351*, 2017.

[21] A. S. Bale, N. Ghorpade, S. Rohith, S. Kamalesh, R. Rohith, and B. Rohan. Web scraping approaches and their performance on modern websites. In *International Conference on Electronics and Sustainable Communication Systems*, 2022.

[22] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience*, 2004.

[23] P. Boldi, A. Marino, M. Santini, and S. Vigna. Bubing: Massive crawling for the masses. *ACM Transactions on the Web*, 2018.

[24] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *NSDI*, 2015.

[25] J. Cho and H. Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *VLDB*, 2000.

[26] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *OSDI*, 2010.

[27] J. Edwards, K. McCurley, and J. Tomlin. An adaptive model for optimizing performance of an incremental web crawler. In *WWW*, 2001.

[28] M. Erdélyi, A. A. Benczúr, J. Masanés, and D. Siklósi. Web spam filtering in internet archives. In *International Workshop on Adversarial Information Retrieval on the Web*, 2009.

[29] U. Feige. A threshold of ln n for approximating set cover. *Journal of the ACM*, 1998.

[30] A. Goel, V. Ruamviboonsuk, R. Netravali, and H. V. Madhyastha. Rethinking client-side caching for the mobile web. In *HotMobile*, 2021.

[31] A. Goel, J. Zhu, R. Netravali, and H. V. Madhyastha. Jawa: Web archival in the era of JavaScript. In *OSDI*, 2022.

[32] G. Gossen, E. Demidova, and T. Risse. ICrawl: Improving the freshness of web collections by integrating social web and focused web crawling. In *JCDL*, 2015.

[33] A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. In *WWW*, 1999.

[34] M. Johnson. Memoization of top down parsing. *arXiv preprint cmp-lg/9504016*, 1995.

[35] T. Kim, R. Chandra, and N. Zeldovich. Efficient patch-based auditing for web application vulnerabilities. In *OSDI*, 2012.

[36] R. Ko, J. Mickens, B. Loring, and R. Netravali. Oblique: Accelerating page loads using symbolic execution. In *NSDI*, 2021.

[37] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov. Irlbot: scaling to 6 billion pages and beyond. *ACM Transactions on the Web*, 2009.

[38] J. Li, Y. Xiong, X. Liu, and L. Zhang. How does web service API evolution affect clients? In *IEEE International Conference on Web Services*, 2013.

[39] G. S. Manku, A. Jain, and A. Das Sarma. Detecting near-duplicates for web crawling. In *WWW*, 2007.

[40] S. Mardani, A. Goel, R. Ko, H. Madhyastha, and R. Netravali. Horcrux: Automatic javascript parallelism for resource-efficient web computation. In *OSDI*, 2021.

[41] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *NSDI*, 2016.

[42] R. Netravali and J. Mickens. Prophecy: Accelerating mobile page loads using final-state write logs. In *NSDI*, 2018.

[43] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *NSDI*, 2015.

[44] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou. CSPAutoGen: Black-box enforcement of content security policy upon real-world websites. In *CCS*, 2016.

[45] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. Tritonsort: A balanced large-scale sorting system. In *NSDI*, 2011.

[46] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *SIGCOMM*, 2017.

[47] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *CGO*, 2008.

[48] A. Sergeev and M. Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[49] S. M. Shariff, H. Li, C.-P. Bezemer, A. E. Hassan, T. H. Nguyen, and P. Flora. Improving the testing efficiency of selenium-based load tests. In *International Workshop on Automation of Software Test*, 2019.

[50] P. Slavík. A tight analysis of the greedy algorithm for set cover. In *STOC*, 1996.

[51] A. Suresh, E. Rohou, and A. Seznec. Compile-time function memoization. In *International Conference on Compiler Construction*, 2017.

[52] Q. Tan and P. Mitra. Clustering-based incremental web crawling. *ACM Trans. Inf. Syst.*, 2010.

[53] Y. Tang and J. Yang. Secure deduplication of general computations. In *USENIX ATC*, 2015.

[54] C. Unger, Z. Jia, W. Wu, S. Lin, M. Baines, C. E. Q. Narvaez, V. Ramakrishnaiah, N. Prajapati, P. McCormick, J. Mohd-Yusof, et al. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *OSDI*, 2022.

[55] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with wprof. In *NSDI*, 2013.

[56] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How speedy is SPDY? In *NSDI*, 2014.

[57] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding up web page loads with Shandian. In *NSDI*, 2016.

[58] Q. Zheng, Z. Wu, X. Cheng, L. Jiang, and J. Liu. Learning to crawl deep web. *Information Systems*, 2013.

Hairpin: Rethinking Packet Loss Recovery in Edge-based Interactive Video Streaming

Zili Meng[1,2,3], Xiao Kong[1,3], Jing Chen[1,3], Bo Wang[1], Mingwei Xu[1],
Rui Han[3], Honghao Liu[3], Venkat Arun[4], Hongxin Hu[5], Xue Wei[3]
[1]*Tsinghua University,* [2]*Hong Kong University of Science and Technology,*
[3]*Tencent,* [4]*UT Austin,* [5]*University at Buffalo, SUNY*

Abstract

Interactive streaming requires minimizing stuttering events (or deadline misses for video frames) to ensure seamless interaction between users and applications. However, existing packet loss recovery mechanisms uniformly optimize redundancy for initial transmission and retransmission, which still could not satisfy the delay requirements of interactive streaming, but also introduces considerable bandwidth costs. Our insight is that in edge-based interactive streaming, differentiating retransmissions on redundancy settings can often achieve a low bandwidth cost and a low deadline miss rate simultaneously. In this paper, we propose Hairpin, a new packet loss recovery mechanism for edge-based interactive streaming. Hairpin finds the optimal combination of data packets, retransmissions, and redundant packets over multiple rounds of transmissions, which significantly reduces the bandwidth cost while ensuring the end-to-end latency requirement. Experiments with production deployments demonstrate that Hairpin can simultaneously reduce the bandwidth cost by 40% and the deadline miss rate by 32% on average in the wild against state-of-the-art solutions.

1 Introduction

Edge-based interactive video streaming is coming to age. Emerging interactive streaming services, such as cloud gaming, has been attracting considerable attention from both academic [24, 28, 44, 53, 63] and industrial communities [3, 8, 9]. These services run applications (e.g., video games) on remote servers, stream the contents (e.g., screens) to users, and interact with users in real time. To provide a seamless interaction between servers and users, the streaming needs to be delivered with low latency. To reduce the network round-trip time (RTT) for interactive streaming, operators deploy servers on edge computing nodes. For example, cloud gaming applications render gaming scenes on remote servers and deliver these scenes to users over the Internet, where these servers could be deployed near the users to shorten the physical distance of the streaming. By deploying interactive streaming services at the edge, the *median* network RTT of these edge-based applications could be reduced to 10-20ms.

To provide a satisfactory user experience, controlling *the ratio of frames violating the latency requirement* is critical for interactive streaming applications [56]. This is due to the continuous interactions between users and applications: Interactive streaming users will start to experience stutters when the end-to-end delay is above 50-200ms (i.e., the deadline of delivering a frame). However, even if only 0.1% of the frames stutter, users may experience stutters every one minute at a frame rate of 24fps. Thus, to improve user experience, we need to reduce the ratio of deadline misses in an interactive streaming service [57].

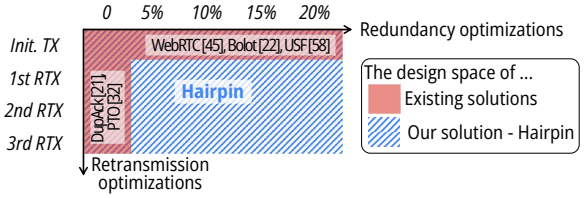


Figure 1: An illustration of the design space of existing solutions and Hairpin. By co-designing the redundancy and retransmission at the transport layer, Hairpin is able to break the existing trade-off between bandwidth cost and deadline miss rate.

In this case, a major challenge to control the deadline misses comes from the high instantaneous loss rate on the Internet. Due to the spatial dependency within video frames and temporal dependency between video frames, interactive streaming expects packets to be reliably delivered [57]. However, from our measurement of our edge-based cloud gaming service in production with O(10,000) users, sessions can experience a drastically high *instantaneous* loss rate. Although the average loss rate is considerably low by mechanisms such as proper rate control, our measurement observes that more than 2% of video frames suffer from an instantaneous loss rate of 20% or higher (§2.1). It indicates that those lost packets are concentrated on a few frames. Thus, although the network RTT can be very low with edge deployments, retransmissions of lost packets take additional time and will consequently violate the deadline. Thus, it is essential to optimize the loss recovery mechanisms to control the deadline miss rate (DMR) of video frames.

Unfortunately, existing solutions to recover packet losses cannot meet the stringent DMR requirements with a reasonable bandwidth cost. As shown in Figure 1, one line of research efforts (the vertical dimension) is devoted to quickly retransmitting lost packets, such as probe timeout (PTO) [33], from the transport layer. However, merely retransmitting lost packets cannot meet the requirement of interactive streaming – the DMR is much higher than 0.1% (§4.3). Another line of effort (the horizontal dimension) is devoted to adaptive forward error correction (FEC) so that the client might be able to recover packets based on redundant packets without retransmission [5]. Yet, redundancy-based solutions come with the price of a considerable bandwidth cost of 20% or more due to the high instantaneous loss rate. For content providers, such a high bandwidth cost will drastically increase operating expenses and

degrade users' video quality. To the best of our knowledge, none of the existing solutions *jointly optimized* retransmission and redundancy. Such an orthogonal design of redundancy and retransmission, even when adopted together, still cannot meet the needs of bandwidth cost and DMR for interactive streaming.

Our key insight is to break the trade-off by differentiating retransmission packets. Edge-based interactive streaming services can achieve an average RTT of 10-20ms between application servers and users by deploying the servers on the edge [27, 58, 89]. In this case, limited times of retransmissions (but not too many) are tolerable for applications that have a deadline of 50-200ms (§2.1). But the strategy for retransmission packets must be different for the initial transmission packets. The volume of retransmission packets is much less than initial transmission packets since packet loss is always the minority. Yet, retransmission packets have a much tighter time requirement since they have already consumed time. This brings new changes to reduce the bandwidth cost and the DMR at the same time (§2.4). By differentiating the strategies for initial transmission and retransmission packets, we can break the trade-off between bandwidth cost and DMR, and improve the performance significantly (§4.4).

We then propose Hairpin[1], a new packet loss recovery mechanism to jointly optimize packet retransmission and redundancy for edge-based interactive streaming (§3.3). However, as later elaborated in §3.2, to further analytically optimize the performance, we still face the challenge of (1) the dependency of decisions and future states, (2) the multi-dimensionality of decisions, and (3) the convoluted goal of DMR and bandwidth cost. In response, Hairpin further formulates the problem into a Markov decision process (MDP), which is known for efficiently optimizing the temporal dependency [79]. We then encodes the decisions and states into nodes of MDP to reduce the complexity and achieve the optimal result.

We conduct a week-long packet-level measurement campaign on Tencent START cloud gaming service to motivate the design of Hairpin (§2.3 and §2.4). We then implement Hairpin and evaluate it with both trace-driven simulators and real-world deployments in production (§4.1). Experiments demonstrate that Hairpin could significantly push forward the Pareto frontier [1] by reducing the DMR by 67%-80% and achieve comparable bandwidth costs simultaneously compared with state-of-the-art baselines (§4.3). Preliminarily testing Hairpin in Tencent START cloud gaming in production also shows significant and consistent performance improvements in different types of networks (§4.7). We will release the simulation codes of Hairpin.

Our main contributions are summarized as follows:

- We motivate the need for joint optimization of retransmission and redundancy through the operating experiences of a production edge-based interactive streaming service (§2).
- We present challenges in the joint optimization over retransmissions and redundancy for edge-based interactive streaming, and then propose Hairpin with MDP formulation (§3).
- We implement and integrate Hairpin in a cloud gaming application in production, and extensively evaluate its performance with trace-driven simulation and real-world deployments (§4).

[1] In badminton, a hairpin shot is played when the shuttle is very near to the ground and the net (the deadline of a shot) [76].

2 Background and Motivations

We introduce the interactive streaming (§2.1), present our measurement of packet losses (§2.2), analyze why existing solutions are insufficient (§2.3), and motivate the design of Hairpin (§2.4).

2.1 Interactive Video Streaming

Interactive streaming applications are increasingly attracting interest in many scenarios. Examples include cloud gaming [3, 8, 9], remote driving [2, 54], cloud phone / PC [13, 18, 39], and regional videoconferencing [4], forming a considerable market value of billions of dollars. Compared with legacy live video streaming, with the intensive deployment on edge nodes (or content generators in VR), the network delay over the wide-area network could be reduced for interactive streaming (e.g., an average RTT of 10-20ms [27, 58, 89]). With the recent emergence of the metaverse and so on, these interactive video streaming applications are going to be increasingly dominant on the Internet. Edge-based interactive streaming imposes specific requirements on transport, as summarized below.

Stringent deadline requirements. Since interactive streaming applications continuously interact with humans, controlling end-to-end delay is critical for a seamless user experience. For example, videoconferencing may expect an end-to-end delay of $<$130ms for network [49, 56], while cloud gaming would argue for a latency of $<$96ms [50][2]. In practice, server- and client-side processing usually take $\approx$30 ms [14, 42, 74, 83]. Therefore, the end-to-end round-trip delay *for the network* should not exceed 50-150ms (depending on scenarios), which is the deadline required by the application [11, 75].

This also corroborates our measurement study with users in our production cloud gaming service. We measure our cloud gaming service in production for one week (details in Appendix A), with O(10,000) users every day, and collect a variety of metrics. Unless other specified, the analysis using online data in this paper is also from this measurement campaign. We categorize the measured round-trip interaction delay of each video frame into several intervals. We present the appearance distribution of the position of those frames in a flow for each category in Fig. 2, where the x-axis is the position of that frame in a session normalized by the length of that session. Compared to the uniform distribution of low-delay frames (solid lines), frames with an end-to-end delay of $>$100ms (dashed lines) have a higher probability to appear around the end of a flow. We hypothesize that this is because users tend to exit a session if they have a high end-to-end delay. User's exiting behavior is a critical metric for user's experience in real-time video streaming [32]. In the meantime, setting a deadline for the delivery and reducing the fraction of higher than that specific value has also been widely adopted

[2] Based on the statistics of the majority of people.

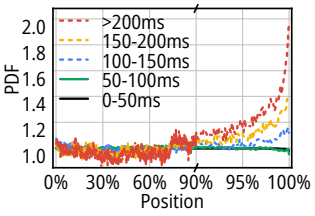


Figure 2: Probability density function (PDF) of the position of frames with different total delays.

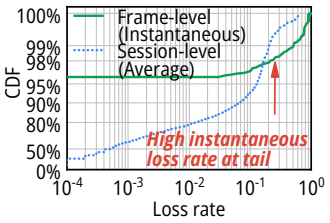


Figure 3: Session-level and frame-level loss rate distributions (both axes are log-scaled).

*Note: Unless otherwise specified, all measurements in §2 and §3 are from the production in the wild (measurement details in §4.1).

in real-time video streaming [56, 57]. The similarity between the 50ms and 100ms curve in Fig. 2 also indicates that, as long as packets could be delivered within the deadline (∼100ms in this case), faster delivery barely improves the user's experience.

Thus, we should minimize the *deadline miss rate* (DMR) to enable a seamless experience for users in interactive streaming, where in our cloud gaming service, the deadline for interaction delay is around 100ms. For interactive streaming, it is essential to minimize the occurrence of deadline misses for frames to an ultra-low level. For example, even a DMR of 10^{-3} still leads to a poor experience every 1000 frames (17 seconds at 60 fps), which drastically degrades the user's experience [11].

Reliable delivery. Meanwhile, interactive streaming also requires reliable delivery for each frame. For commercial video codec, failing to deliver a part of the frame will lead to severe image quality degradation. Moreover, the loss of one frame would also lead to blurring for the subsequent frames due to the dependency between frames[3]. Therefore, existing interactive streaming services usually try their best to reliably deliver frames. For example, industrial frameworks (e.g., WebRTC) [5, 46] and academic efforts [22, 40, 61] propose to employ forward error correction (FEC) to recover lost packets at the receiver if possible, and will retransmit lost packets if the recovery fails [7].

Low bandwidth cost. The bandwidth cost is still one of the largest operating expenses in our and other cloud gaming service [17]. Moreover, to achieve a satisfactory user experience, interactive streaming must stream with high video resolution and frame rate (e.g., 60fps and >1080p for cloud gaming), which requires high goodput to support. Given the requirements of low operating expenses and high video quality for users, we need to control the bandwidth cost in packet loss recovery.

2.2 Packet Losses in Edge-based Interactive Streaming

Our observation from our cloud gaming service is that although the median loss rate is as low as 10^{-3}, the *instantaneous* loss rate could be very high. In our measurement campaign as described in §2.1, we also calculate the *session-level loss rate*, which is the ratio of total lost packets in one user session (minutes to hours, containing at least O(10,000) frames), to reflect the average loss rate over a long timescale. We then calculate *frame-level loss rate*, which is the ratio of lost packets within one frame (tens of milliseconds), to show the instantaneous loss rate over a short timescale. For example, if a session has 1M packets and 10 of them are lost, the session-level average loss rate is 0.01%. Meanwhile, if these 10 packets belong to the same video frame which has 50 packets in total, the frame-level instantaneous loss rate will be 20% for that frame and 0% for other frames.

As shown in Fig. 3, the session-level loss rate is 0.05% at the median, which is comparable to similar measurements [45]. However, the instantaneous frame-level loss rate could be very high: 2% frames lose more than 20% of their packets within one frame. Such a high instantaneous packet loss poses a great challenge in controlling the deadline miss rate to 10^{-3} or lower – we can no longer ignore these transient behaviors and have to deliver video frames in time even when the instantaneous loss rate is high.

Moreover, these packet losses cannot be easily mitigated by reducing the sending rate. To achieve a low latency, most CCAs in interactive streaming use *delay* as the signal to reduce the sending rate (e.g., BBR [25], Copa [15], GCC [26]). In this case, congestion losses rarely happen since the sending rate has already been reduced based on an increasing delay in advance, which has also been measured in related work [26]. Our online measurements unveil similar observations: our cloud gaming service has already adopted a delay-based CCA similar to GCC [26], which is widely deployed in interactive streaming applications such as Chrome and Stadia. We further demonstrate the weak correlation between RTT[4] increases and packet losses in our measurement in §2.4. As shown in Fig. 3, losses are still outstanding at the tail, indicating that merely controlling the bit rate or frame rate is still insufficient to avoid packet losses for edge-based interactive streaming.

2.3 Why Existing Solutions Fail?

As we discussed in §1, packet losses contribute a lot to deadline misses. Thus, we investigate why existing packet loss recovery mechanisms are insufficient for edge-based interactive streaming. Existing solutions mainly fall into two categories as follows.

Retransmissions. Existing transport protocols (e.g., TCP) rely on retransmissions to cope with packet losses. Merely relying on retransmissions is insufficient to achieve an extremely low DMR for interactive streaming frames at the magnitude of 0.1% or lower. For example, when the packet loss rate is instantaneously 20%, there would still be 0.16% packets lost even after 3 retransmissions. Note that since there could be tens to hundreds of packets per frame, being unable to deliver even one packet would violate the deadline requirement of that frame since interactive streaming requires all packets to be reliably delivered (§2.1). Thus, the DMR of frames is still considerably high when relying on retransmissions and rate controls only. Our evaluation in §4 also demonstrate the performance degradation.

Redundancy-based algorithms. There are also several solutions

[3] Mechanisms such as scalable video coding (SVC) allow limited packet losses, yet reduce the bandwidth efficiency and require client support [70].

[4] In this paper, we use RTT to represent the delay at the network layer that does not contain the time of retransmission. We use application delay to refer to the delay at the application layer that contains the retransmissions.

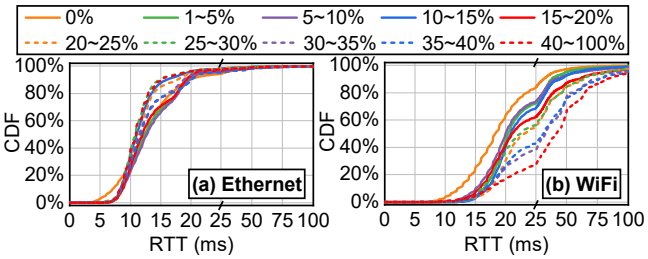


Figure 4: RTT distributions measured in production, categorized by the frame-level loss rate. Note that retransmissions are not counted.

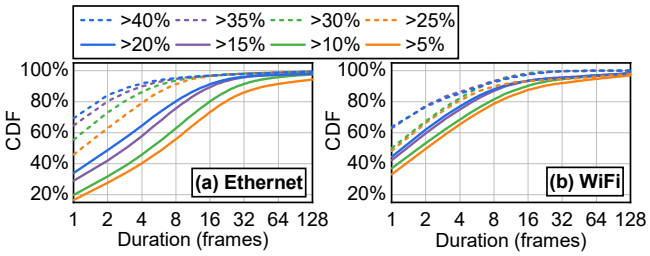


Figure 5: The distribution of the duration of each loss event measured in production. We measure the duration of each time when the loss rate is larger than different thresholds (5%, ..., 40%). Loss rates are measured at the frame level. The network type is reported from our cloud gaming clients. Better viewed in color.

in interactive streaming with redundancy mechanisms such as FEC. However, existing adaptive FEC solutions from both the industry [5,46] and academia [22,40,61] optimize the FEC parameters only for the initial transmission. They adjust the number of FEC packets according to loss rate and retransmit packets as usual when packet loss occurs. Note that packet losses are not deterministic: when the transient loss probability increases to 20%, it does not mean precisely one packet loss every five packets. In this case, to achieve an extremely low DMR of 10^{-3} or lower, FEC rates need to be much higher than the loss rate, leading to severe bandwidth cost (§4). For example, WebRTC, a state-of-the-art interactive streaming framework, will send 100% redundant packets during this short timescale of high instantaneous loss rate for initial transmissions. In this case, there will be considerable bandwidth cost while the DMR might still not be satisfied. We further evaluate the performance of other baselines in §4.2.

2.4 Motivations

Therefore, with the reduced RTT, retransmissions are tolerable to some extent for edge-based interactive streaming. In this case, we have the following observations on how and what to retransmit.

RTT being much lower than the deadline enables the joint optimization of redundancy and retransmission. As we discussed before, with an RTT of 10-20 ms and a deadline of 50-150 ms, multiple retransmissions are tolerable to some extent. This enables the joint optimization of redundancy and retransmission, which results in benefits in two folds:

- Reduce the deadline miss rate. In existing FEC mechanisms, many of the deadline misses come from the packet losses in the retransmissions. When adding redundancy packets over retransmission packets, we could effectively avoid the loss of retransmission packets and further reduce the deadline miss rate.
- Save bandwidth costs. To achieve the same DMR, the bandwidth cost of adding redundancy to retransmissions is significantly lower than that of only adding redundancy to initial transmissions. This is because retransmission packets are always the minority in bandwidth consumption – redundifying retransmissions will only introduce a little bandwidth cost, but could have significant DMR improvements.

When more rounds of retransmissions are tolerated (e.g., with smaller RTTs), the joint optimization will have more significant benefits (later presented in §4.5). We are thus motivated to utilize the retransmission chances enabled by edge deployments and jointly optimize the redundancy and retransmission mechanisms.

Loss recovery adaptions at the server are possible. Dynamically optimizing the tail cases of high instantaneous loss rate needs quick adaption. According to our measurement, the feedback loop between the server and client is smaller than the duration of loss events, making the joint optimization of redundancy and retransmission practical. This comes in two folds:

- The feedback loop does not inflate with the increase in the loss rate. We measure the RTT of our cloud gaming service and categorize them into different frame loss rate intervals. As shown in Fig. 4(a), the distribution of RTT does not significantly vary with the frame loss rate. The RTT in WiFi increases with the increase of frame loss rate (e.g., due to retransmissions at the link layer [31]). Nevertheless, even when the frame-level loss rate is 30% (the dashed green curve in Fig. 4(b)), 60% of those acknowledged packets have an RTT of less than 25ms. This indicates (i) the server is able to quickly detect the network condition changes, and (ii) there are still multiple transmission chances when the instantaneous loss rate increases.
- The duration of loss events is transient but still longer than several feedback loops. We measure the duration of lossy frames in our cloud gaming service and present the results in Fig. 5. According to our measurements, most loss events span multiple RTTs. For example, 70% of frames with a frame-level loss rate of >10% will last more than 2 frames in Ethernet sessions, which is several times the median RTT (12ms) at the frame rate of 60fps. Therefore, the reaction from the server is still effective to alleviate packet losses by adjusting the redundancy parameters.

3 Hairpin Optimizer

As we discussed above, edge-based interactive streaming needs to reduce the deadline miss rate and bandwidth cost. For clarity, we first present the formula of frame deadline miss rate (DMR) and bandwidth cost (BWC):

$$\begin{aligned} DMR &= \frac{\text{\#Frames arrive after the deadline}}{\text{\#Total frames}} \\ BWC &= \frac{Redundancy_{byte} + Retransmission_{byte}}{Data_{byte}} \end{aligned} \quad (1)$$

A higher DMR or BWC means more frequent stutters or higher operating expenses respectively, both of which interactive streaming service providers will try to avoid. Note that pushing

DMR to an extremely low level is critical since the lower it is, the better user's experience is going to be.

In this section, we first summarize some intuitions in the design space of joint optimization of redundancy rate and retransmission and present a strawman solution (§3.1). We then present the design challenges in the joint optimization of retransmission and redundancy (§3.2). We address these challenges by providing a Markov chain-based optimization algorithm to efficiently improve both the DMR and BWC (§3.3). We finally discuss how Hairpin handles the inaccuracy in measurement, the overhead in online deployment, and other practical issues in §3.4.

3.1 Basic Idea and Strawman Solution

Differentiating retransmissions from initial transmissions. The most important insight in this paper is to understand the significance of differentiating retransmissions from initial transmissions. In other words, we want an adaptive redundancy rate based on the planning of multiple transmission chances. The short RTT of edge-based interactive streaming enables packets to have more than one transmission chance without violating the deadline. The ratio of RTT and remaining time t indicates the potential number of (re)transmissions. For example, when the current RTT is 20ms and packets still have 40ms towards their deadline, the ratio follows $\frac{t}{RTT} = \frac{40ms}{20ms} = 2$, indicating that these packets could be approximately transmitted twice before the deadline. Packets with more transmission chances could better utilize the potential retransmissions to deliver packets before the deadline, which has already been discussed in §2.4. Therefore, our basic idea is to take future transmission chances into consideration when optimizing the redundancy rate. When one batch of packets has more foreseeable transmission chances (i.e., the deadline is still far away), we could reduce the redundancy rate to save bandwidth costs. When the remaining time of these packets is getting closer to the deadline due to retransmissions, we could further increase the redundancy rate to avoid deadline misses.

Strawman solution: RTT-aware adaptive FEC algorithm. Therefore, a strawman solution is to (i) add redundancy to both initial transmissions and retransmissions, and (ii) consider the remaining transmission chance in the optimization of the redundancy rate. Since there have already been existing solutions on the redundancy rate based on network conditions [5, 22, 61], we could introduce a multiplier controlled by the transmission chance over the existing redundancy rate optimizations, i.e. a strawman solution is to reduce the redundancy rate when there are many transmission chances, and increase it when transmission chances are few. Thus, we could enhance these algorithms by introducing a factor over the results from existing algorithms.

FEC consists of two parameters (d,k), where d data packets and k redundant packets are sent as a *block*. Block is composed to the convenience of FEC encoding. If there are up to k packets lost in an FEC block (d,k), an ideal FEC decoder can recover all data packets with any remaining packets [64, 65, 85]. We denote $\beta = \frac{k}{d}$ as the FEC redundancy rate, and d as the FEC block size.

Specifically, given a packet loss rate α and bitrate B, assume one of the state-of-the-art solutions has already determined that $\beta_0(\alpha,B)$ should be the optimized redundancy. We could then increase or decrease the redundancy rate $\beta_0(\alpha,B)$ based on the remaining transmission chance $\frac{t}{RTT}$, i.e.:

$$\beta(\alpha,B,RTT,t) = k \cdot \frac{RTT}{t} \cdot \beta_0(\alpha,B) \tag{2}$$

where k is a coefficient to adjust how aggressive the strawman solution is going to increase or decrease the redundancy rate.

In fact, according to our evaluation in §4.6, such a strawman solution is enough to push the Pareto frontier of DMR-BWC forward. However, it confronts a series of shortcomings, which prevents the operator from further improvements in performance. We will elaborate on these challenges in the following section.

3.2 Design Challenges

Although we have presented a heuristic RTT-aware adaptive FEC algorithm as above, it is still challenging to optimize these parameters due to the following reasons.

Temporal dependency: cascading decision-making between transmission rounds. When considering multiple transmission chances, the decision of FEC parameters of one round of transmission would cascadingly affect the optimization of the next round. For example, if we aggressively add a high redundancy rate to a group of packets, the number of packet losses will then be decreased. On the contrary, a low redundancy rate for the same group of packets would probabilistically increase the number of packet losses under the same network condition. However, these packet losses bring more packets to retransmit in the next round. If we consider all actions for F packets for the foreseeable L rounds of transmission, the action space will be extremely large: Since for each redundancy decision, there are F possible scenarios of the number of packets to transmit in the next round (depending on how many packets are lost), the number of variables that we need to optimize will be $O(F^L)$[5]. Therefore, in the enlarged action space over multiple retransmissions, it is challenging to efficiently optimize. Moreover, the conditional probability between scenarios is not linear (e.g., hypergeometric for individually independently and identically distributed losses). Therefore, using traditional optimization methods such as integer programming in an extremely large action space is impractical. We need to coordinate the choices in different rounds of transmission to achieve optimal performance.

Spatial dependency: redundancy rate and block size are tightly coupled. Even in a single round, different variables (e.g., redundancy rate, block size, etc.) still have complicated dependencies on each other. This goes to the following aspects:

(a) Number of packets to transmit in one round affects redundancy rates. The number of packets to transmit in the different rounds is varying, depending on how many data packets are lost

[5] For a frame with 50 packets (F=50), and 5 potential transmission rounds (L=5, e.g., RTT is 20ms and deadline is 100ms), this turns into 10^8 variables.

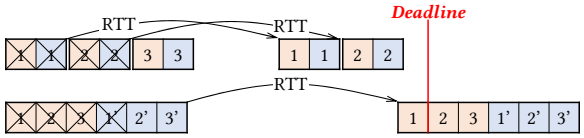
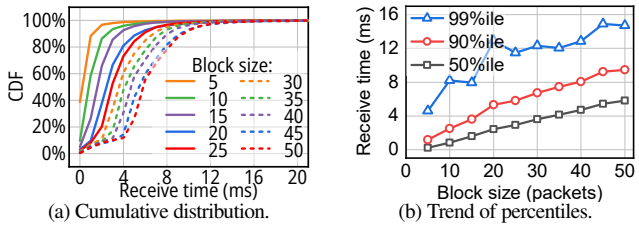


Figure 6: Smaller block sizes in one frame could have better performance. Scenarios above and below represent using small and large blocks. Data and FEC packets are shaded orange and blue.



(a) Cumulative distribution. (b) Trend of percentiles.

Figure 7: Block receiving time with different block sizes. FEC blocks are burstily sent out at the server side. Fig. 7(b) is processed from Fig. 7(a). Measurement details in §4.2. Better viewed in color.

during the last transmission. The penalty of redundancy rate on BWC also varies according to the number of packets to retransmit. For example, when there are few packets to retransmit, even adding a redundancy rate of 100% for retransmissions would not consume too much bandwidth, as also discussed in §2.4. Therefore, fewer data packets to retransmit would encourage a more aggressive redundancy rate. The strawman solution is not aware of the dependency here, leading to its suboptimal result.

(b) Dispersion of blocks might lead to deadline misses when using larger blocks. Due to the bandwidth limit at the bottleneck link, packets sent out at the same time could be dispersed [47] and arrive at the receiver one by one. In this case, constructing large blocks will increase the delay to wait for all packets at the receiver. Since packet losses can only be determined after the completion of one block, smaller blocks may know earlier whether they need retransmission and enjoy additional transmission chances before the deadline. For example, in Fig. 6, due to the early determination of packet loss, the retransmission of data packets for small blocks could arrive at the receiver before the deadline, while no packets could arrive before the deadline for large blocks. We quantify the influence of block size by measuring the receiving time of FEC blocks from our service online with different block sizes. As we can see in Fig. 7 and 7(b), with a block size of 50 packets, more than 10% blocks could span 10ms at the receiver, which is even comparable to the RTT. Also, smaller block sizes might also be beneficial when the loss rate is higher than the redundancy rate. As illustrated in Fig. 6, when the first four packets are lost during the transmission, data packet #3 could still be successfully delivered for a small block size (the case above in Fig. 6). For large blocks, there is no way to recover any lost packet if the loss rate is larger than the redundancy rate.

Convoluted goal: deadline miss rate and bandwidth cost. Unlike latency or throughput which we can directly measure, the estimation of the expected deadline miss rate needs to consider multiple potential rounds of transmission. In this way, the strawman solution, without explicitly estimating whether that frame is going to miss the deadline or not, will have suboptimal results. For example, the relationship between the packet loss rate and the success rate of delivering a video frame with tens of packets in a single round is *hypergeometric*, even under the identical and independent distribution (i.i.d.) assumption. Considering multiple future rounds together will only make the relationship between deadline miss rate and network conditions more convoluted. Moreover, some applications or even the same application in different operating regions may have different preferences over deadline miss rate v.s. bandwidth cost. The traffic cost in some regions might be higher than in another, and some applications may give it all for the user's experience while others may not. Therefore, we need to explicitly optimize towards the goal to achieve the optimal result.

3.3 Model Formulation and Optimization

We have the following designs to address the challenges above.

Encode the temporal dependency in multi-round planning into edges in Markov chain. Markov chain is widely used in the optimization of the sequential decision-making process (e.g., reinforcement learning [79]). With the Markov chain, we can formulate the loss detection between two rounds of (re)transmission into the transition between two Markov nodes. In this case, by only focusing on the optimal parameters between the transition of the current state and its potential states in the next round, we could decouple the cascading effects of the transitions between neighbor nodes, which reduces the action space significantly. We further show in Appendix B.1 that, in such a Markov chain, locally focusing on the neighbor nodes could still have globally optimal results.

Encoding the spatial dependency between variables into nodes in Markov chain. To ensure the number of packets to transmit is considered in the optimization, we build a 2-D Markov chain, with two dimensions as the transmission chance and the number of packets to transmit. We present the state transition of our Markov chain in Fig. 8. Each node is represented by (d,l), where d denotes the number of remaining data packets to transmit, and l represents the remaining transmission chance for those packets. Our goal is to find out the optimal redundancy rate for node (B,L), where B is a given block size, and L is the remaining transmission chance from Eq. 3. In this case, both the temporal dependency and spatial correlation between variables could be formulated into this 2-D Markov chain.

Explicitly optimize deadline miss rate and bandwidth cost with Markov chain formulation. We finally provide an explicit expression of the deadline miss rate and bandwidth cost for multi-round optimization within the formulation of MDP. We inversely calculate the DMR and BWC at different states from the last chance to transmit (as the last layer of the Markov chain), to the first chance to transmit (as the first layer of the Markov chain). In this way, the transition probabilities between states could be directly iterated. We further decouple the optimization of redundancy rate and block size to improve the optimization efficiency.

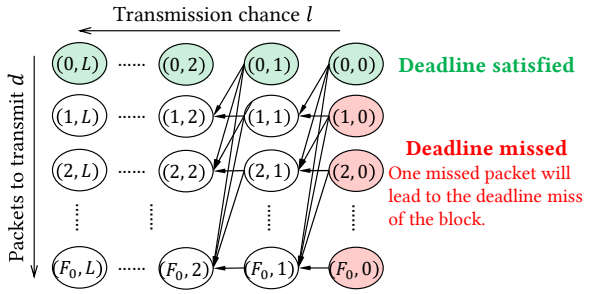


Figure 8: The absorbing Markov chain in redundancy rate optimization at given loss rate and frame size. l is the estimated remaining transmission chances for the packets to transmit.

We present the analytical model and the algorithm below. In interactive streaming, frames are continuously generated and sent out from the server. There are thousands to millions of frames within one stream, depending on the specific application, where the retransmission of previous frames overlaps with the transmission of subsequent frames. Therefore, similar to the finite element analysis in mechanics [10], we pick one frame from the stream, and analyze the expected DMR and BWC of that frame. The expected DMR and BWC of one frame should be consistent with the DMR and BWC of a stream. We list all notations that will use in Table 2 in Appendix B.1. Specifically, Hairpin optimizes the FEC parameters as follows:

Step 1: Calculating remaining transmission chance. Given current network RTT, the remaining time towards deadline T, the bottleneck bandwidth Θ, and a certain block size d, the remaining transmission chance L could be calculated as:

$$L = \frac{T - d/\Theta}{RTT} \tag{3}$$

Step 2: Generating absorbing Markov chain. We then calculate the optimal redundancy rate given the current loss rate α and frame size F. We iteratively calculate the absorbing Markov chain from layer $l-1$ to layer l. For the node (d,l), at a certain redundancy rate β,we respectively calculate the $DMR(d,l;\beta)$ and $BWC(d,l;\beta)$ based on the DMR and BWC from the $l-1$-th layer. We leave the detailed equations to Appendix B.1. Then, we calculate the optimal β for (d,l):

$$\beta_{opt}(d,l) = \text{argmin}_\beta utility(DMR(d,l;\beta), BWC(d,l;\beta)) \tag{4}$$

Here, $utility(DMR, BWC)$ is the utility function to balance preference for low DMR and low BWC. For simplicity, we adopt a linear combination of DMR and BWC as the optimization goal:

$$utility(DMR,BWC) = DMR + \lambda \cdot BWC \tag{5}$$

Note that Hairpin does not fall into the same trade-off between DMR and BWC as baselines, but improves both DMR and BWC, as we will evaluate later in §4.3. In practice, service providers can adjust the coefficient λ to balance stuttering events and bandwidth costs in different scenarios. A lower λ indicates that users prefer the deadline miss rate more than bandwidth costs. We also evaluate performance with different utility functions in §4.5.

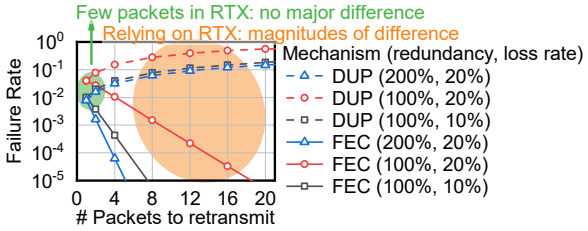


Figure 9: A theoretical illustration of the failure rate of retransmitting different numbers of packets by per-packet duplication or constructing FEC blocks. The failure rate of DUP increases with the number of packets to retransmit, since we need to ensure every data packet is delivered. We vary the redundancy rate and loss rate.

Therefore, the redundancy rate for (B,L) could be optimized accordingly. After calculating all nodes at the layer l, we could then calculate the DMR and BWC at the layer $l+1$, until the node (B,L) has been calculated. Since the iterations between nodes are linear, as long as the utility function is monotonic to DMR and BWC (e.g., linear relationship), the optimality still holds.

We set $DMR(d,0)$ to 1 for $d > 0$ since one missed packet would lead to the miss of the block (shaded green). We also set all $DMR(0,l)$ to 0 since there is no remaining packet to transmit. The BWC for all these boundary nodes is set to 0. Note that different block sizes and remaining transmission chance could multiplex the same chain to accelerate the optimization, since the chain only depends on loss rate α and frame size F.

Step 3: Calculating optimal block size. We enumerate the possible block sizes from 1 to the frame size, calculate the DMR and BWC for each block according to the chain in Step 2, and finally find the optimal block size in terms of a given utility function. We leave the mathematical details to Appendix B.2. According to our evaluation in §4.6, not surprisingly, when the bottleneck bandwidth is high (i.e., the dispersion is insignificant), the optimal block size for most scenarios is the frame size. Nevertheless, when the dispersion is significant, constructing smaller blocks could achieve better DMR. Operators could optimize the block size for improvements at the last mile.

During the optimization of block sizes, we also optimize the trade-off of when a loss has been detected, whether to retransmit that packet as soon as possible or wait for other packets to formulate an FEC block. On recovery ability, constructing several lost packets into one FEC block might be more effective than individually retransmitting (or duplicating, if with redundancy) each packet. We calculate the failure rate of delivering these packets when there are different numbers of packets to retransmit at different redundancy rates and loss rates and present the results in Fig. 9. When there are few packets that need retransmission, whether duplicating or constructing FEC blocks has no major difference (dashed line and solid lines shaded green). However, when optimizing at the *tail* for interactive streaming, there could be multiple packet losses within one frame. Therefore, considering each frame could contain tens of packets, it is possible to suffer losses of 4 packets or more at the tail. Constructing FEC blocks for these retransmission packets could reduce the failure rate of delivering packets by several magnitudes.

Step 4: Getting the optimal parameters. Finally, based on network conditions and remaining time towards a deadline, Hairpin can calculate the optimal block size based on Step 3, and the optimal redundancy rate with the block size based on Step 2.

3.4 Deployment Discussions

In §3, we analytically optimize the FEC parameters given certain network conditions. The reality might be more complicated than the theoretical model. In this section, we discuss several practical concerns of Hairpin based on our operational experiences. Our further trace-driven simulation and deployments in production in §4 also demonstrate the effectiveness of Hairpin in the wild.

Reducing computational overhead online. Hairpin adopts an optimization-based algorithm, which might not scale to production-scale deployments in terms of computational overhead. Since the optimization needs to run frequently (approximately every frame) and scale to tens of thousands of users simultaneously, it should be computation-efficient and time-efficient. In response, we do an offline step of enumerating the state space and solving each specific instance. Then, in the online step, the algorithm will be reduced to a simple table lookup towards pre-computed optimized redundancy parameters. We enumerate the state space of Hairpin as below.

1. Remaining transmission chance: 1 to 10.
2. Loss rate: 0% to 50% with quantization of 1%.
3. Frame size: 5 to 60 packets with quantization of 5 packets.
4. Number of packets to (re)transmit: 5 to 60 packets with quantization of 5 packets.

Hairpin then stores the best redundancy rate and block size under different conditions. We found that the benefits of finer quantization are marginal. Our further evaluation in the real world in §4.7 shows that the memory consumption (2MB) and table lookup time are negligible for online deployment.

Handling network fluctuations. We discuss how Hairpin handles the fluctuations in network conditions. For RTT, as presented in Fig. 4, RTT does not increase too much – the median RTT always allows Hairpin to have 3-5 transmission chances no matter the loss rate. Moreover, we further measure the network conditions in Hairpin with a short sliding window to make sure Hairpin has the most recent network conditions. We set the measurement window to 2 frames and evaluate the sensitivity of this parameter in §4.5. In this case, the transient fluctuation of RTT could be reflected in the optimization results immediately. We later demonstrate in §4 that Hairpin behaves well with real-world traces and production deployments.

Handling various loss patterns. In this paper, when given a certain loss rate, Hairpin assumes the pattern of packet losses is identically and independently distributed (in the transition probability of Eq. 6). Note that the duration of a certain loss rate still follows the results of the online measurement in Fig. 5. In practical deployment, working with FEC codecs that could recover from different loss patterns (bursty or arbitrary) [65], Hairpin could also handle different loss patterns since Hairpin only focuses on how many packets within a block are lost. Since our data is collected frame by frame, if the burstiness spans over several frames, it will be directly reflected on the value of loss rates. If the burstiness spans within the frame, no matter how the pattern changes, the number of lost packets will not change, which does not affect the recovery efficiency of the FEC codec. For example, when there are 4 packet losses in one block, no matter whether these losses are consecutive or separated in the block, as long as there are 4 additional FEC packets in the same FEC block, the client would be able to recover these packet losses. Therefore, Hairpin does not rely on the assumption of underlying loss patterns, but only focuses on the number of lost packets. Packet losses might be consecutive across several frames. In this case, due to the short feedback loop enabled by edge deployments, Hairpin should have already timely reacted as analyzed in §2.4.

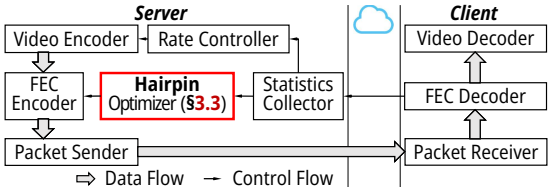


Figure 10: Overview of Hairpin implementation.

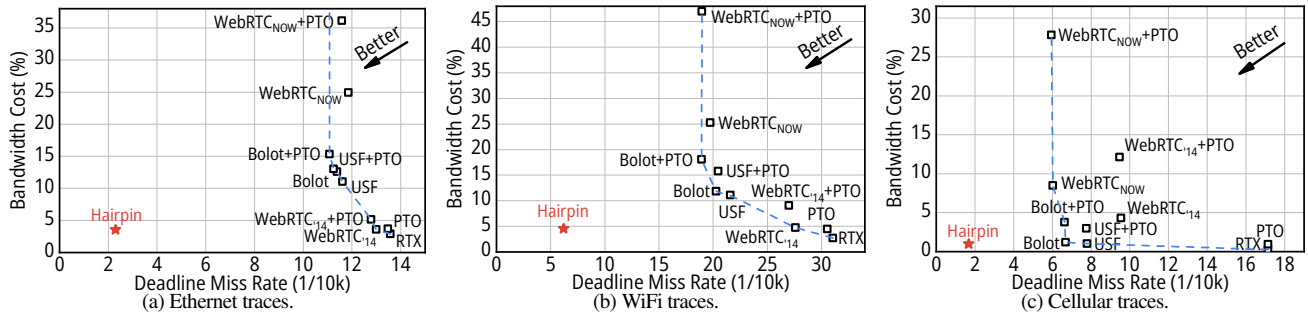
4 Evaluation

We introduce the implementations in §4.1 and experiment settings in §4.2. We further answer the following questions:

- How does Hairpin perform under real-world traces? We demonstrate that Hairpin could push forward the Pareto frontier of baselines on DMR and BWC (§4.3).
- Is Hairpin sensitive to the settings of parameters? We investigate the performance variation of Hairpin with different parameters, and demonstrate that Hairpin has performance improvements in a wide range (§4.5).
- Which design in Hairpin contributes to the performance improvement against other baselines? In §4.4, we break down the performance improvements of Hairpin.
- How does Hairpin make decisions and how each optimization affects the user's experience? We further dive into the details of Hairpin's design in §4.6.
- How does Hairpin perform well in the wild? Finally, we deploy Hairpin in production servers and find that Hairpin significantly improves both DMR and BWC in the real world (§4.7).

4.1 Hairpin Implementation

We implement Hairpin in both an `ns3`-based WebRTC simulator [88] and our cloud gaming application in production. We present the workflow of Hairpin in the network stack in Fig. 10. Without Hairpin, interactive contents are first encoded with `Video Encoder` by the application, and then sent out at the transport layer frame-by-frame. Then the video frames could be received by the protocol stack at the client. `Packet Sender` and `Packet Receiver` abstract the network stack at the transport layer for connection management. After that, `Video Decoder` decodes the streaming contents and displays



(a) Ethernet traces. (b) WiFi traces. (c) Cellular traces.

Figure 11: Trace-driven simulation. The blue dashed line is the envelope of all baselines on the Pareto frontier.

them to users. Meanwhile, network conditions (e.g., RTT, packet loss events) will be measured at the server, collected by the `Statistics Collector`, and reported to `Rate Controller` to adaptively adjust the streaming bit-rate according to network conditions [26]. Hairpin inserts between the existing application layer and transport layer, and optimizes the redundancy parameters based on current network conditions, as shown in Fig. 10. The network statistics is still passed to the congestion controller (rate controller) without modification. The underlying transport protocol in our cloud gaming service is a customized version of the RTP protocol [69] based on UDP to allow the loss of redundant packets without modifying the kernel at the client. We implement Reed-Solomon FEC due to its recovery performance when the redundancy rate is $<$100% [65], and implement a customized FEC codec for the redundancy rate of $>$100%. Note that Hairpin could also work with other codecs (e.g., XORFEC, FlexFEC, etc.) as long as their parameters are exposed to Hairpin. We discuss FEC codecs in Appendix C.

4.2 Experiment Setup

Traces. As for simulation traces, we collect a dataset in one production server in the wild on our cloud gaming service in two weeks in January and August 2021, resulting in more than 100M video frames and more than 600 hours of playtime. This also supports our measurements in §2 and §3. Users access the service via either Ethernet, WiFi, or cellular connection, which we collect from our cloud gaming client. The cloud gaming service streams at the frame rate of 60fps and the bit rate ranging from 2Mbps to 30Mbps. The network conditions are recorded on the server of our cloud gaming service, including the average RTT, average bit rate, and loss rate at the frame level (approximately every 16 ms). The traces contain 1,995 Ethernet gaming sessions, 741 WiFi sessions, and 572 cellular sessions in total, each lasting from minutes to hours. To the best of our knowledge, we are the first to collect online traces from an interactive streaming service for weeks at both the frame level and packet level.

Baselines. We orthogonally review the public adaptive FEC mechanisms and retransmission mechanisms with deployments in practice. On the axis of retransmission optimization, we implement the following baselines.

- `Out-of-order`. Traditionally, packet losses are detected by checking the out-of-order packets, such as TCP duplicated ACK [21]. We use it as our default loss detection mechanism.
- `Probe timeout (PTO)`. Besides, to quickly detect packet losses of tail packets, recent researchers also propose an aggressive timeout-based loss detection mechanism [33].

On the axis of redundancy parameter optimization, we implement the following mechanisms:

- `WebRTC'14` comes from the research paper published by Google in 2014 [46].
- `WebRTC`$_{NOW}$ is the adaptive FEC mechanism used in WebRTC now (adopted by Google Stadia [35], Meet [23], etc.), replacing the `WebRTC'14`. The difference is that `WebRTC'14` is aware of RTT and will reduce the redundancy rate when RTT is low, while `WebRTC`$_{NOW}$ is more aggressive on adding redundancy. We migrate the implementation of the m88 version of Chromium released in December 2020 [7].
- `Bolot` [22] and `USF` [61] are two heuristic adaptive FEC algorithms from the research community. Unlike Hairpin, they do not add redundant packets for retransmissions.
- `RTX` adds no redundancy, but fully relies on retransmissions.

Note that none of these baselines optimize the redundancy for retransmissions here. Since these two lines of work are orthogonal to each other, we combinatorially implement 2 (retransmission) $\times$ 5 (redundancy) $=$ 10 baselines.

Hairpin Setup. In our simulation, we set the coefficient in the utility function in Eq. 5 to $\lambda = 10^{-4}$, the measurement window of network conditions to 2 frames, and the deadline to 100ms. We evaluate the sensitivity of these parameter settings in §4.5.

4.3 Trace-driven Simulations

To evaluate the performance of Hairpin in dynamic network conditions, we simulate Hairpin over real-world traces as introduced in §4.2. We emulate the collected traces of loss rate and RTT with `ns-3`, and evaluate whether Hairpin could capture the network dynamics of loss and RTT variations and effectively adapt in real traces. We first present the trade-off between DMR and BWC over three sets of traces in Fig. 11.

As shown in Figure 11, `RTX` has the lowest bandwidth cost since `RTX` only retransmits a packet after it is lost. However, it also has the highest deadline miss rate among all baselines. Meanwhile, `WebRTC`$_{NOW}$ working with `PTO` has the lowest DMR among all baselines but also the highest BWC. Other baselines stay on the Pareto frontier in the trade-off between DMR and BWC. In contrast, Hairpin could break the trade-off

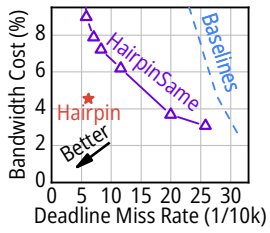


Figure 12: Breaking down the performance improvements of Hairpin.

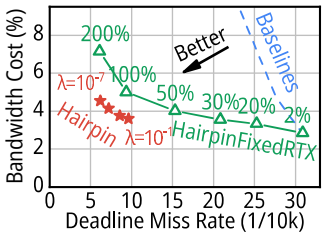


Figure 13: Parameter sensitivity of λ in Hairpin, and the effectiveness of differentiating retransmission.

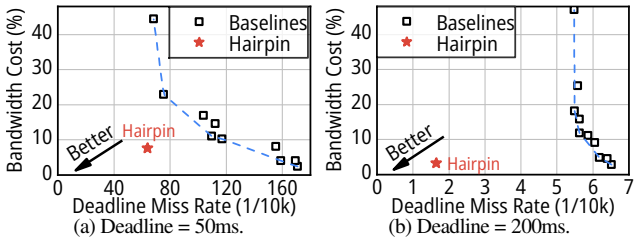


Figure 14: The performance of Hairpin and all baselines (labels omitted for brevity) on WiFi traces when the deadline requirement from the application is different.

and achieve a much better DMR and BWC, as the red stars denoted in Figure 11: 67%-80% lower than the lowest DMR ($\texttt{WebRTC}_{NOW}$), and comparable BWC as RTX. Thus, as we analyzed above, Hairpin could effectively improve both DMR and BWC significantly compared with all other baselines.

Note that the traces here are collected from our production servers, including the network RTT and instantaneous loss rate, with a fined granularity of every 16ms. The results in WiFi traces are worse than in Ethernet traces since WiFi traces have higher loss rates and RTTs, as measured in §2.4. Results over cellular traces are surprisingly good. This is because, during our online measurements, we just started to provide cloud gaming service for cellular users and had admission control over network conditions during that time. In all, Hairpin could significantly push forward the Pareto frontier of existing baselines in all traces.

4.4 Performance Breakdown

As discussed before, the performance improvement of Hairpin comes from two aspects: a carefully crafted optimization model, and the design space of differentiating retransmissions from initial transmissions. To investigate the contribution of these two components, we set up another candidate: HairpinSame. HairpinSame adopts the same Markov Chain-based model as Hairpin, but enforces the redundancy rate of all rounds to be the same. We further sweep the choice of λ from$\{10^{-4},10^{-2},10^{-1},1,5,10\}$ for HairpinSame to show the trade-off between bandwidth cost and deadline miss rate in this non-differentiating scenario. We also present the envelope of all baselines from Figure 11(b) as the blue dashed line in Figure 12.

As shown in Figure 12, even without differentiating retransmissions, HairpinSame is still able to significantly push the frontier forward. For example, when the bandwidth cost is 4.5%, Hairpin is able to reduce the deadline miss rate from to 0.28% to 0.17%, as shown in the gap between the purple solid line and blue dashed line. We are not going to argue that the proposed Markov chain algorithm outperforms all the baselines due to the optimization algorithm. We hypothesize that the improvement is due to the change of the optimization goal. Previous baselines focus on the optimization of tail latency, which is different from deadline miss rate as discussed in §2. In fact, we further present the latency distribution of Hairpin and all baselines in Appendix D (Figure 22), and the results of Hairpin and baselines are comparable for most percentiles below the 99.9th. Therefore, with the focus mostly on the extreme tails after the 99.9th percentile, Hairpin is able to significantly reduce the deadline miss rate, or reduce bandwidth cost for the same deadline miss rate.

Nevertheless, with the new design space of differentiating retransmissions, Hairpin can further reduce the deadline miss rate from 0.17% to 0.06% when the bandwidth cost is 4.5%, as shown in the gap between the red star and purple line in Figure 12. This further demonstrates the importance of differentiating retransmissions. Both designs are critical to performance improvement.

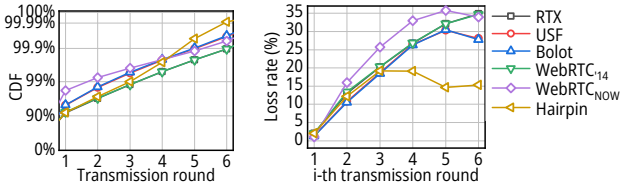
We further want to see how far a naive algorithm, which does nothing but differentiates retransmissions, will go. To this end, we have HairpinFixedRTX, which only adds FEC packets to retransmissions with a fixed ratio, and never adds FEC packets to initial transmissions, in contrast to all existing solutions in §4.2. We vary the static redundancy rate for retransmissions from 2% to 200%. As shown in the gap between the green solid line and the blue dashed line in Figure 13, HairpinFixedRTX significantly improves the trade-off between DMR and BWC against existing baselines. The series of red stars will later be explained in §4.5. This demonstrates that differentiatedly adding FEC over initial transmission and retransmission packets can effectively improve performance. As we discussed in §3, even naively differentiating the retransmissions with another fixed redundancy rate would already be helpful, illustrating the necessity of differentiating retransmissions.

4.5 Parameter Sensitivity

We also evaluate how Hairpin performs with different parameters.

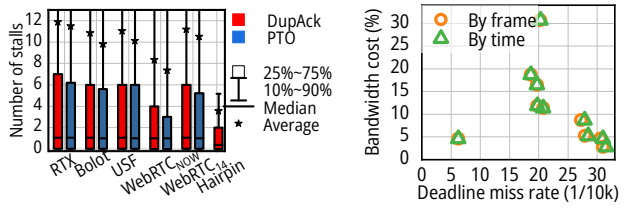
Utility coefficient λ. For the utility coefficient λ in Eq. 5, as introduced in §4.2, it could adjust the preference over the trade-off between the DMR and BWC. A higher λ indicates that users prefer the BWC more, while a lower λ indicates that the DMR is outweighing the BWC. Therefore, we change λ from 10^{-1} to 10^{-7}, and present the DMR and BWC of Hairpin with different λ over WiFi traces in Fig. 13. Note that Fig. 13 is zoomed in from Fig. 11(b). As shown in the red stars in Fig. 13, the BWC is decreasing with the increase of λ, while the DMR is increasing by a little. Thus, operators could adjust λ to balance the DMR and BWC according to the requirements of applications.

The setting of the deadline. In the evaluation in §4.3, the deadline is set to 100ms. We also investigate how Hairpin performs when the deadline is shorter or longer. Thus, we present the results of DMR and BWC of Hairpin and baselines over WiFi traces when the deadline is set to 50ms (Fig. 14(a)) or 200ms



(a) Number of rounds. (b) Loss rates by round.

Figure 15: The loss rate in each transmission round.



(a) Number of stalls per session. (b) DMR by frame and by time.

Figure 16: The effect of DMR on other metrics.

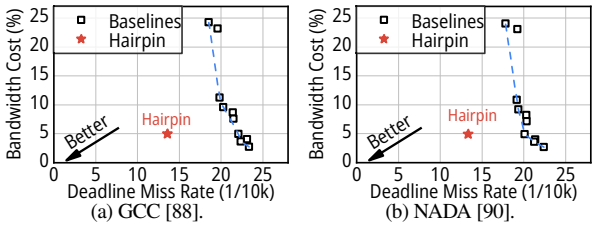
(Fig. 14(b)). As presented in Fig. 14, given the same trace, when the deadline is shorter (50ms), the advantages of Hairpin over baselines are a little less than when the deadline is 100ms. This is because the retransmission chance is less and the design space is smaller when the deadline is shorter. Nevertheless, Hairpin is still much better than all existing baselines. When the deadline is longer, the benefits are even larger due to the larger design space in retransmission. Results over other sets of traces are similar.

4.6 Hairpin Deep Dive

We further provide a deeper understanding of Hairpin in the following aspects.

Understanding Hairpin's decisions. In Appendix D, we present the redundancy rate and block size results of Hairpin to provide a deeper understanding of how Hairpin optimizes in different scenarios. Besides, we present the number of transmission rounds of Hairpin and baselines in Fig. 15(a). When Hairpin gradually increases the redundancy rate in future transmission rounds, most frames could therefore be delivered. Thus, the 99.9th percentile of the number of transmission rounds in Hairpin is less than all other baselines by more than one. Similarly, when we inspect the loss rate in each round as shown in Fig. 15(b), Hairpin also successfully maintains the lowest loss rate when the transmission round goes up. Note that the loss rate here is significantly high due to the survivorship bias – only lost packets will have another transmission round, while loss has already indicated a degraded network performance. This also indicates that the loss is not i.i.d. but bursty in the experiments.

Optimizing towards extremely low DMR. We further illustrate why we need to achieve an extremely low DMR and how it affects user's experience. As analyzed in §3.1, a lower DMR approaching zero directly indicates fewer stall events in a gaming session. We measure the number of *stall events* in each gaming session, where stall event is only counted once if there are multiple missed frames in one second or if it lasts longer than one second. As shown in Fig. 16(a), Hairpin can reduce the



(a) GCC [88]. (b) NADA [90].

Figure 17: The performance of Hairpin and all baselines (labels omitted for brevity) on WiFi traces with different deadline requirements.

average and median number of stall events (which is also critical for user's opinion scores [67]) by a half or more against baselines. By having a DMR of 0.06%, Hairpin is able to reduce the 75th percentile number of stalls in a session to 2. Considering the duration of a gaming session (minutes to hours), this will considerably improve the user's experience.

We also show the difference of calculating DMR *by frame* and *by time* in Fig. 16(b). In this paper, we do not argue using a new metric (DMR by frame) is better – we calculate DMR by the number of missed frames over total frames because of the simplicity in the formulation in §3.3. Calculating DMR by time is almost equivalent to DMR by frame since the stalled time is the number of stalled frames (missed frames) times the interval between frames. Therefore, we replot Fig. 11(b) using two different DMRs. As shown in Fig. 16(b), the results are almost the same with each other.

Integrating with congestion control. To further investigate the performance of Hairpin when interacting with the CCA, we integrate the Hairpin with two CCAs in the WebRTC framework, GCC [26] and NADA [90], in our simulation. We then replay the collected traces by setting their bandwidth, RTT, and loss rate to the link in ns-3. The bandwidth ranges from 2Mbps to 30Mbps. As shown in Fig. 17, Hairpin could still achieve significant advantages over all existing baselines.

4.7 Real-World Experiments

Finally, we deploy the Hairpin in a production server in our cloud gaming service. We conduct an A/B test in production of Hairpin against the $\texttt{WebRTC}_{NOW}$ baseline. The bit rate of the cloud gaming service also supports the range of 2-30Mbps as simulated in §4.3. The A/B test runs for one week in September 2021, covering 17k sessions in total, all of which have a duration of at least 4 minutes. Hairpin has been integrated into the UDP-based connections of our cloud gaming service since then. Since other optimizations are also deployed into our service after we deploy Hairpin, to make a fair comparison, we only present the results from the controlled A/B test in September 2021.

Performance. As shown in Table 1, Hairpin is able to improve both the average DMR and the average BWC compared to $\texttt{WebRTC}_{NOW}$. Specifically, for Ethernet sessions, Hairpin could improve the DMR by 32% while also reducing the BWC by 40% against $\texttt{WebRTC}_{NOW}$. For WiFi sessions, the improvements on DMR and BWC are 30% and 43%. We also measure the ratio of sessions with an average DMR of larger than 1%, i.e.

Ethernet	DMR	BWC	P(DMR>1%)	#Session
WebRTC$_{NOW}$	0.34%	30.4%	6.9%	8380
Hairpin	0.23%	3.0%	4.6%	7306
WiFi	DMR	BWC	P(DMR>1%)	#Session
WebRTC$_{NOW}$	0.72%	31.8%	19.3%	652
Hairpin	0.51%	3.0%	15.3%	613

Table 1: Real-world experiment results. P(DMR>1%) denotes the ratio of sessions with an average DMR of larger than 1%.

tail sessions. Hairpin could also reduce the tail sessions by 34% and 21% for Ethernet and WiFi sessions respectively compared to WebRTC$_{NOW}$. Note that the DMRs in real-world experiments are a little higher than those in simulations (§4.3). This might be because of other external factors (e.g., user devices) that could affect the DMR. Nevertheless, Hairpin could significantly improve the users' experiences on both the DMR and BWC compared to WebRTC$_{NOW}$.

Overhead. We further measure the overhead of the optimization of Hairpin. As introduced in §3.4, to accelerate the optimization online, we precompute the optimized FEC parameters and store the result table for online look-up. At our quantization granularity of the table, it takes 1.98MB to store the table, which is negligible on servers since the table is static and could be shared by all connections. Moreover, according to our measurements, the time of looking up the table is always less than 1ms, which is also negligible since the table is looked up at the granularity of the frame.

5 Related Work

We first discuss the limitations in Appendix E, and then discuss some pieces of related work here.

Deadline-aware optimization. Optimizing transport protocols for deadline-aware flows has been intensively studied in the networking community. Research efforts have been devoted to the optimization of transport protocols under the assumption of delivery deadline in video streaming [72, 87], space network [73], and others. There are also deadline-aware optimization in datacenters for flow completion time [30,78,82] and job completion time [36]. These research efforts mainly focus on the priority-aware scheduling between different packets, jobs, or flows. Instead, Hairpin is orthogonal to them and optimizes within one stream.

Loss recovery optimization. There are many previous research efforts in individually optimizing the retransmission mechanisms [21,62,68] or redundancy strategies [5,22,29,40,51,59,61]. Even for the joint optimization of redundancy and retransmission, we are not the first work to propose similar strategies. In wireless communications, there are already previous efforts in the joint optimization of redundancy and retransmission [12, 60]. There are also researches trying to combine retransmission and redundancy (e.g., WebRTC$_{2011}$ [71] and also [16, 37, 86]). However, as discussed in §2.4 and evaluated in §4, without optimizing redundancy over retransmissions, the *tail* performance is far from satisfactory for edge-based interactive streaming. Besides, there are also research efforts trying to adopt loss-resilient video codec [41, 80], which are unfortunately not deployable for services in the wild due to their hardware support. To the best of our knowledge, Hairpin is the first work to (i) jointly optimize the retransmission and redundancy towards the tail performance, and (ii) deploy in a real interactive streaming application in production. There are also researches to reduce packet losses by manipulating sending patterns [20, 38], which can work together with Hairpin.

Low-latency interactive streaming. Finally, as an emerging direction, low-latency interactive streaming also attracts much attention. At the transport layer, intensive efforts have also been devoted to the optimization of CCAs [26, 41, 48, 90], or the cross-layer optimization for interactive streaming to link layer [31]. As for the infrastructure, recent efforts propose to introduce edge computing for shorter latency [58, 77, 89], which are orthogonal to Hairpin. In contrast, Hairpin is inserted between the transport layer and the application layer, and is designed to optimize the redundancy of edge-based interactive streaming. In the evaluation and deployment of Hairpin (§4), we have already integrated Hairpin with some efforts above for a better user experience. Moreover, there are other research efforts [34, 84] that optimize other application metrics for interactive video streaming in image or video quality (e.g., SSIM [81] or PSNR [6]), which are orthogonal to the interaction delay (the delay for each frame) we focus on in this paper – they focus on the sharpness of the video but we focus on the interaction lag that users may have

6 Conclusion

We propose Hairpin, a packet loss recovery mechanism for edge-based interactive streaming to differentiate retransmissions and jointly optimize redundancy with retransmissions. Hairpin motivates the joint optimization with real-world measurements, and optimizes the redundancy and retransmissions with Markov decision process. Both trace-driven simulations and real-world deployments show that the joint optimization significantly reduces the DMR and BWC compared with state-of-the-art solutions.

This work does not raise any ethical issues.

Acknowledgements

We sincerely thank our shepherd Junchen Jiang, anonymous reviewers, and labmates in the Routing Group from Tsinghua University for their valuable feedback. This work is sponsored by the National Natural Science Foundation of China (No. 62221003 and 62372261) and the Tsinghua-Tencent Collaborative Grant. Bo Wang and Mingwei Xu are the corresponding authors.

References

[1] Pareto front - wikipedia. https://en.wikipedia.org/wiki/Pareto_front.

[2] 5g can make remote driving a reality, telefonica and ericsson demostrate at mwc. https://www.telefonica.com/en/web/press-office/-/5g-can-make-remote-driving-a-reality-telefonica-and-ericsson-demostrate-at-mwc, 2017.

[3] Cloud gaming (beta) with xbox game pass — xbox. https://www.xbox.com/en-US/xbox-game-pass/cloud-gaming, 2020.

[4] Critical services report: Video conferencing (uk) — blog. https://samknows.com/blog/critical-services-report-video-conferencing-uk, 2020.

[5] Issue 93006: Update to media_opt_util: - code review. https://webrtc-codereview.appspot.com/93006, 2020.

[6] Peak signal-to-noise ratio - wikipedia. https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio, 2020.

[7] Psa: Webrtc m88 release notes. https://groups.google.com/g/discuss-webrtc/c/A0FjOcTW2c0/m/UAv-veyPCAAJ, 2020.

[8] Stadia - one place for all the ways we play. https://stadia.google.com/, 2020.

[9] Your games. your devices. play anywhere — nvidia geforce now. https://www.nvidia.com/en-us/geforce-now/, 2020.

[10] Finite element method - wikipedia. https://en.wikipedia.org/wiki/Finite_element_method, 2021.

[11] Optimizing 5g for a new class of low-latency experiences [video]. https://www.qualcomm.com/news/onq/2021/07/20/optimizing-5g-new-class-low-latency-experiences, 2021.

[12] Ashfaq Ahmed, Arafat Al-Dweik, Youssef Iraqi, Husameldin Mukhtar, Muhammad Naeem, and Ekram Hossain. Hybrid automatic repeat request (harq) in wireless communications systems and standards: A contemporary survey. *IEEE Communications Surveys & Tutorials*, 2021.

[13] Amit. Huawei news — huawei launched cloud mobile phone. https://www.huaweiupdate.com/huawei-launched-cloud-mobile-phone/, 2020.

[14] ArsTechnica. Nvidia gtx 1080 review: The new performance king. https://arstechnica.com/gadgets/2016/05/nvidia-gtx-1080-review/4/, 2016.

[15] Venkat Arun and Hari Balakrishnan. Copa: Practical delay-based congestion control for the internet. In *Proc. USENIX NSDI*, 2018.

[16] Luca Baldantoni, Henrik Lundqvist, and Gunnar Karlsson. Adaptive end-to-end fec for improving tcp performance over wireless links. In *Proc. IEEE ICC*, 2004.

[17] Matthew Ball and Jacob Navok. Challenge #3: Enormous bandwidth costs and operational burdens — cloud gaming: Why it matters and the games it will create. https://www.matthewball.vc/all/cloudmiles, 2020.

[18] Asha Barbaschow. Alibaba unveils cloud 2.0, wuying cloud computer, and xiaomanlv logistics robot. https://www.zdnet.com/article/alibaba-unveils-cloud-2-0-wuying-cloud-computer-and-xiaomanlv-logistics-robot/, 2020.

[19] Apurv Bhartia, Bo Chen, Feng Wang, Derrick Pallas, Raluca Musaloiu-E, Ted Tsung-Te Lai, and Hao Ma. Measurement-based, practical techniques to improve 802.11 ac performance. In *Proc. ACM IMC*, 2017.

[20] Saad Biaz and Nitin H Vaidya. "de-randomizing" congestion losses to improve tcp performance over wired-wireless networks. *IEEE/ACM Transactions on Networking*, 2005.

[21] Ethan Blanton, Dr. Vern Paxson, and Mark Allman. TCP Congestion Control. IETF RFC 5681, 2009.

[22] J-C Bolot, Sacha Fosse-Parisis, and Don Towsley. Adaptive fec-based error control for internet telephony. In *Proc. IEEE INFOCOM*, 1999.

[23] brianhu. Google meet troubleshooting playbook - network and hardware troubleshooting. https://www.googlecloudcommunity.com/gc/Workspace-Product-Articles/Google-Meet-Troubleshooting-Playbook-Network-and-Hardware/ta-p/165810, 2021.

[24] James Bulman and Peter Garraghan. A cloud gaming framework for dynamic graphical rendering towards achieving distributed game engines. In *Proc. USENIX HotCloud*, 2020.

[25] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *ACM Queue*, 2016.

[26] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Congestion control for web real-time communication. *IEEE/ACM Transactions on Networking*, 2017.

[27] Marc Carrascosa and Boris Bellalta. Cloud-gaming: Analysis of google stadia traffic. *arXiv:2009.09786*, 2020.

[28] Hao Chen, Xu Zhang, Yiling Xu, Ju Ren, Jingtao Fan, Zhan Ma, and Wenjun Zhang. T-gaming: A cost-efficient cloud gaming system at scale. *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[29] Ke Chen, Han Wang, Shuwen Fang, Xiaotian Li, Minghao Ye, and H. Jonathan Chao. Rl-afec: Adaptive forward error correction for real-time video communication based on reinforcement learning. In *Proc. ACM MMSys*, 2022.

[30] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. Scheduling mix-flows in commodity datacenters with karuna. In *Proc. ACM SIGCOMM*, 2016.

[31] Wei Chen, Liangping Ma, and Chien-Chung Shen. Congestion-aware mac layer adaptation to improve video teleconferencing over wi-fi. In *Proc. ACM MMSys*, 2015.

[32] Sheng Cheng, Han Hu, Xinggong Zhang, and Zongming Guo. Rebuffering but not suffering: Exploring continuous-time quantitative qoe by user's exiting behaviors. In *Proc. IEEE INFOCOM*, 2023.

[33] Yuchung Cheng, Neal Cardwell, Nandita Dukkipati, and Priyaranjan Jha. The RACK-TLP Loss Detection Algorithm for TCP. IETF RFC 8985, 2021.

[34] Mallesham Dasari, Kumara Kahatapitiya, Samir R. Das, Aruna Balasubramanian, and Dimitris Samaras. Swift: Adaptive video streaming with layered neural codecs. In *Proc. USENIX NSDI*, 2022.

[35] Andrea Di Domenico, Gianluca Perna, Martino Trevisan, Luca Vassio, and Danilo Giordano. A network analysis on cloud gaming: Stadia, geforce now and psnow. *Network*, 2021.

[36] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proc. ACM EuroSys*, 2012.

[37] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing web latency: the virtue of gentle aggression. In *Proc. ACM SIGCOMM*, 2013.

[38] Tobias Flach, Pavlos Papageorge, Andreas Terzis, Luis Pedrosa, Yuchung Cheng, Tayeb Karim, Ethan Katz-Bassett, and Ramesh Govindan. An internet-wide analysis of traffic policing. In *Proc. ACM SIGCOMM*, 2016.

[39] Mary Jo Foley. Microsoft marches toward launching its 'cloud pc' service, possibly this summer. `https://www.zdnet.com/article/microsoft-marches-toward-launching-its-cloud-pc-service-possibly-this-summer/`, 2021.

[40] Silas L Fong, Ashish Khisti, Baochun Li, Wai-Tian Tan, Xiaoqing Zhu, and John Apostolopoulos. Optimal streaming codes for channels with burst and arbitrary erasures. *IEEE Transactions on Information Theory*, 2019.

[41] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *Proc. USENIX NSDI*, 2018.

[42] GFXBench. 3d graphics performance of google pixel c. `https://gfxbench.com/device.jsp?D=Google+Pixel+C`, 2017.

[43] Moinak Ghoshal, Pranab Dash, Zhaoning Kong, Qian Xu, Y.Charlie Hu, Dimitrios Koutsonikolas, and Yuanjie Li. Can 5g mmwave enable multi-user ar apps? In *Proc. PAM*, 2022.

[44] Yu Guan, Chengyuan Zheng, Xinggong Zhang, Zongming Guo, and Junchen Jiang. Pano: Optimizing 360 video streaming with a better understanding of quality perception. In *Proc. ACM SIGCOMM*, 2019.

[45] Osama Haq, Mamoon Raja, and Fahad R Dogar. Measuring and improving the reliability of wide-area cloud paths. In *Proc. WWW*, 2017.

[46] Stefan Holmer, Mikhal Shemer, and Marco Paniconi. Handling packet loss in webrtc. In *2013 IEEE International Conference on Image Processing*, 2013.

[47] Manish Jain and Constantinos Dovrolis. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with tcp throughput. In *Proc. ACM SIGCOMM*, 2002.

[48] Ingemar Johansson and Zaheduzzaman Sarker. Self-Clocked Rate Adaptation for Multimedia. IETF RFC 8298, 2017.

[49] Alan Jones, Peter Sevcik, and Rebecca Wetzel. Internet connection requirements for effective video conferencing to support work from home and elearning — netforecast. `https://www.netforecast.com/wp-content/uploads/NFR5137-Videoconferencing_Internet_Requirements.pdf`, 2021.

[50] Teemu Kämäräinen, Matti Siekkinen, Antti Ylä-Jääski, Wenxiao Zhang, and Pan Hui. A measurement study on achieving imperceptible latency in mobile cloud gaming. In *Proc. ACM MMSys*, 2017.

[51] Balázs Kreith, Varun Singh, and Jörg Ott. Fractal: Fec-based rate control for rtp. In *Proc. ACM Multimedia*, 2017.

[52] Tong Li, Kai Zheng, Ke Xu, Rahul Arvind Jadhav, Tao Xiong, Keith Winstein, and Kun Tan. Tack: Improving wireless transport performance by taming acknowledgments. In *Proc. ACM SIGCOMM*, 2020.

[53] Xiaofei Liao, Li Lin, Guang Tan, Hai Jin, Xiaobin Yang, Wei Zhang, and Bo Li. Liverender: A cloud gaming system based on compressed graphics streaming. *IEEE/ACM Transactions on Networking*, 2016.

[54] Ruilin Liu, Daehan Kwak, Srinivas Devarakonda, Kostas Bekris, and Liviu Iftode. Investigating remote driving over

the lte network. In *Proceedings of the 9th International Conference on Automotive User Interfaces and Interactive Vehicular Applications*, pages 264–269, 2017.

[55] Bill Marczak and John Scott-Railton. Move fast and roll your own crypto: A quick look at the confidentiality of zoom meetings - the citizen lab. https://citizenlab.ca/2020/04/move-fast-roll-your-own-crypto-a-quick-look-at-the-confidentiality-of-zoom-meetings/, 2020.

[56] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. Achieving Consistent Low Latency for Wireless Real Time Communications with the Shortest Control Loop. In *Proc. ACM SIGCOMM*, 2022.

[57] Zili Meng, Tingfeng Wang, Yixin Shen, Bo Wang, Mingwei Xu, Rui Han, Honghao Liu, Venkat Arun, Hongxin Hu, and Xue Wei. Enabling High Quality Real-Time Communications with Adaptive Frame-Rate. In *Proc. USENIX NSDI*, 2023.

[58] China Mobile and ZTE. Powered by sa: 5g mec-based cloud game innovation practice. GSMA 5G Case Studies (https://www.gsma.com/futurenetworks/wp-content/uploads/2020/03/Powered-by-SA-5G-MEC-Based-Cloud-Game-Innovation-Practice-.pdf), 2020.

[59] Marcin Nagy, Varun Singh, Jörg Ott, and Lars Eggert. Congestion control using fec for conversational multimedia communication. In *Proc. ACM MMSys*, 2014.

[60] Hoang Anh Ngo and Lajos Hanzo. Hybrid automatic-repeat-request systems for cooperative wireless communications. *IEEE Communications Surveys & Tutorials*, 16(1):25–45, 2013.

[61] Chinmay Padhye, Kenneth J Christensen, and Wilfrido Moreno. A new adaptive fec loss control algorithm for voice over ip applications. In *Proc. IEEE INFOCOM*, 2000.

[62] Colin Perkins, Orion Hodson, and Vicky Hardman. A survey of packet loss recovery techniques for streaming audio. *IEEE Network*, 1998.

[63] Feng Qian, Bo Han, Qingyang Xiao, and Vijay Gopalakrishnan. Flare: Practical viewport-adaptive 360-degree video streaming for mobile devices. In *Proc. ACM Mobicom*, 2018.

[64] Vincent Roca, Mathieu Cunche, Jerome Lacan, Amine Bouabdallah, and Kazuhisa Matsuzono. Simple Reed-Solomon Forward Error Correction (FEC) Scheme for FECFRAME. IETF RFC 6865, 2013.

[65] Vincent Roca, Jani Peltotalo, Jerome Lacan, and Sami Peltotalo. Reed-Solomon Forward Error Correction (FEC) Schemes. IETF RFC 5510, 2009.

[66] Carolyn Rowe, Diana Hanson, Chiffers Craig, David Coulter, Justin Gilmore, David Byrd, Ajayan Borys, Kelly Baker, Baard Hermansen, Serdar Soysal, et al. Microsoft teams call flows - microsoft teams — microsoft docs. https://docs.microsoft.com/en-us/microsoftteams/microsoft-teams-online-call-flows, 2021.

[67] Michael Rudow, Francis Y. Yan, Abhishek Kumar, Ganesh Ananthanarayanan, Martin Ellis, and K.V. Rashmi. Tambur: Efficient loss recovery for videoconferencing via streaming codes. In *Proc. USENIX NSDI*, 2023.

[68] Pasi Sarolahti, Markku Kojo, and Kimmo Raatikainen. F-rto: an enhanced recovery algorithm for tcp retransmission timeouts. *ACM SIGCOMM Computer Communication Review*, 2003.

[69] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, and Van Jacobson. RTP: A Transport Protocol for Real-Time Applications. IETF RFC 3550, 2003.

[70] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the h. 264/avc standard. *IEEE Transactions on circuits and systems for video technology*, 2007.

[71] Mikhal Shemer. Commit - video_coding robustness: Updating hybrid mode's settings — google git. https://webrtc.googlesource.com/src/+/ae7a0522c59d932d72f3d3377c38bebab7ab2b31%5E%21/, 2011.

[72] Hang Shi, Yong Cui, Feng Qian, and Yuming Hu. Dtp: Deadline-aware transport protocol. In *Proc. APNet*, 2019.

[73] Hang Shi, Lei Zhang, Xutong Zuo, Qian Wu, Hewu Li, and Yong Cui. Multipath deadline-aware transport proxy for space network. *IEEE Internet Computing*, 2021.

[74] Shu Shi, Cheng-Hsin Hsu, Klara Nahrstedt, and Roy Campbell. Using graphics rendering contexts to enhance the real-time video coding for mobile cloud gaming. In *Proc. ACM Multimedia*, 2011.

[75] Ivan Slivar, Lea Skorin-Kapov, and Mirko Suznjevic. Cloud gaming qoe models for deriving video encoding adaptation strategies. In *Proceedings of ACM International Conference on Multimedia Systems (MMSys)*, 2016.

[76] Kaarmukilan S.P. What is hairpin net shot in badminton? - quora. https://www.quora.com/What-is-hairpin-net-shot-in-badminton/answer/Kaarmukilan-S-P, 2020.

[77] Zhaowei Tan, Yuanjie Li, Qianru Li, Zhehui Zhang, Zhehan Li, and Songwu Lu. Supporting mobile vr in lte networks: How close are we? *Proc. ACM SIGMETRICS*, 2018.

[78] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter tcp (d2tcp). In *Proc. ACM SIGCOMM*, 2012.

[79] Martijn van Otterlo and Marco Wiering. *Reinforcement Learning and Markov Decision Processes*, pages 3–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[80] Tingfeng Wang, Zili Meng, Mingwei Xu, Rui Han, and Honghao Liu. Enabling high frame-rate uhd real-time communication with frame-skipping. In *Proc. ACM Workshop on Hot Topics in Video Analytics and Intelligent Edges*, 2021.

[81] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 2004.

[82] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *Proc. ACM SIGCOMM*, 2011.

[83] Raphael Wimmer, Andreas Schmid, and Florian Bockes. On the latency of usb-connected input devices. In *Proc. ACM CHI*, pages 1–12, 2019.

[84] Saman Zadtootaghaj, Steven Schmidt, Saeed Shafiee Sabet, Sebastian Möller, and Carsten Griwodz. Quality estimation models for gaming video streaming services using perceptual video quality dimensions. In *Proc. ACM MMSys*, 2020.

[85] Mo Zanaty, Varun Singh, Ali C. Begen, and Giridhar Mandyam. RTP Payload Format for Flexible Forward Error Correction (FEC). IETF RFC 8627, 2019.

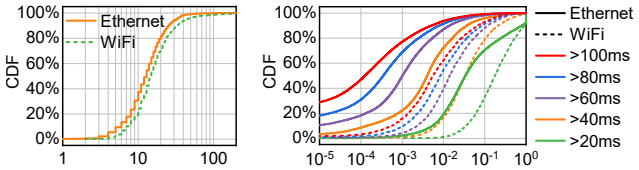
[86] Fan Zhai, Yiftach Eisenberg, Thrasyvoulos N Pappas, Randall Berry, and Aggelos K Katsaggelos. Rate-distortion optimized hybrid error control for real-time packetized video transmission. *IEEE Transactions on Image Processing*, 2006.

[87] Lei Zhang, Yong Cui, Junchen Pan, and Yong Jiang. Deadline-aware transmission control for real-time video streaming. In *Proc. IEEE ICNP*, 2021.

[88] Songyang Zhang. Soonyangzhang/webrtc-gcc-ns3: test google congestion control on ns3. `https://github.com/SoonyangZhang/webrtc-gcc-ns3`, 2020.

[89] Xu Zhang, Hao Chen, Yangchao Zhao, Zhan Ma, Yiling Xu, Haojun Huang, Hao Yin, and Dapeng Oliver Wu. Improving cloud gaming experience through mobile edge computing. *IEEE Wireless Communications*, 2019.

[90] Xiaoqing Zhu, Rong Pan, Michael A. Ramalho, and Sergio Mena de la Cruz. Network-Assisted Dynamic Adaptation (NADA): A Unified Congestion Control Scheme for Real-Time Media. IETF RFC 8698, 2020.



(a) Frame-level delay (b) Session-level delay ratio

Figure 18: Network delay distributions of the interactive streaming service of company T. Delay ratio is the ratio of frames with a delay of >20,>40,>60,>80 and >100 ms in each session. Note that the delay here is measured at the application layer (details in §4.2).

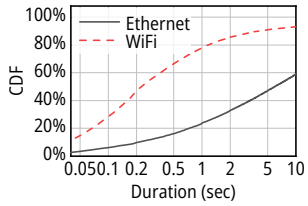


Figure 19: Distribution of network RTT maintenance duration in our interactive streaming service.

Appendices

A Measurements in Production

We present our measurement results on the cloud gaming service X in production to support some claims in the paper.

To investigate the effect of edge acceleration of interactive streaming in the wild, we conduct a measurement campaign on the cloud gaming service X. The measurements last for one week with thousands of sessions (containing heterogeneous users through Ethernet, WiFi with Windows and MacOS systems) and are presented in Fig. 18. As shown in Fig. 18(a), the majority of network delay collected at the granularity of video frame falls into 10-20ms for both Ethernet and WiFi. We also measure the flow-level delay ratio at different thresholds and present the results in Fig. 18(b). With the edge acceleration, the ratio of frames with longer than 100ms delay in most flows is less than 10^{-2}. Among them, Ethernet flows perform slightly better than WiFi flows. This validates the effectiveness of edge acceleration: the average network delay could be reduced to 10-20ms with a proper edge acceleration.

We further measure the fluctuation of RTT by the duration when RTT is roughly kept at the same level. We quantify it by calculating the transmission chance (i.e., layer L) for the RTT measured by each frame, and calculate the duration when the chance is kept the same. For example, given a deadline of 100ms in this paper, when the RTT measurements are [26ms, 18ms, 17ms, 22ms, 17ms, 19ms, 19ms], the transmission chances are [3, 5, 5, 4, 5, 5, 5]. In this case, the durations of each transmission chance are [1, 2, 1, 3], which are denoted as *RTT maintenance durations*. We present the distribution of RTT maintenance duration measured in our cloud gaming service in Fig. 19. The RTT maintenance duration of Ethernet is much longer than that of WiFi, indicating that Ethernet has a more stable end-to-end delay. Meanwhile, the median duration of both Ethernet and WiFi is above hundreds of milliseconds, which is

Notation	Explanation
Inputs:	
α	Network loss rate.
T	Remaining time till the deadline.
RTT	The network round-trip time.
Θ	The network bottleneck bandwidth.
F	The frame size of that frame.
Intermediate variables:	
L	Remaining transmission chance.
$l(n,r)$	The number of lost packets at the r-th layer with n data packets.
$k(n,r)$	The number of redundant packets at the r-th layer with n data packets.
DMR	Deadline miss rate.
BWC	Bandwidth cost.
Outputs:	
β_i	Redundancy rate at the i-th layer.
b_i	FEC block size at the i-th layer.

Table 2: Notations in §3 and Appendix B.

much higher than the feedback loop of Hairpin. This indicates that *RTT does not frequently change, and Hairpin is able to detect and react to the fluctuations of RTT*.

B Optimization Model

In this section, we present the notations used in the Markov chain in §3.3. We list all notations in Table 2. We further present the detailed designs here.

B.1 Optimization of the Redundancy Rate

We build an absorbing Markov chain to model the redundancy and calculate the deadline miss rate considering retransmission, as shown in Fig. 8. We first define the *state* in the Markov chain as (n,r), where n is the number of unacknowledged packets within the block, and r is the number of retransmission. For example, $(d_0,0)$ represents the initial transmission where all d_0 packets have not been received before (since it is the first time of transmission). $(3,2)$ denotes that there are still 3 data packets that need to be retransmitted for the second time.

For node (d,l), given redundancy rate β, its DMR follows:

$$DMR(d,l;\beta)=\sum_{d'=0}^{d} p((d,l)\to(d',l-1);\beta)\cdot DMR(d',l-1) \quad (6)$$

where $p((d,l)\to(d',l-1);\beta)$ is the transition probability from (d,l) to $(d',l-1)$ and could be calculated based on the current loss rate α and redundancy rate β. Similarly, the BWC could also be updated as:

$$BWC(d,l;\beta)=\beta\frac{d}{F}+\sum_{d'=0}^{d} p((d,l)\to(d',l-1);\beta)\cdot BWC(d',l-1) \quad (7)$$

where the latter term is the additional BWC introduced in this layer l.

We then calculate the transition probability between states in the Markov chain. For the transition between state (n_1,r) to $(n_2,r-1)$, we know that n_2 data packets are lost in the r-th transmission and need to be transmitted for the $(r+1)$ time. We first discuss the scenario of $n_2>0$. We denote the total number of packet losses (including data and redundancy) in the r-th transmission as $l(n_1,r)$. We denote the number of redundant packets in the r-th transmission as $k(n_1,r)$. Since the packet losses of all packets should not be less than the packet losses of data packets, we have $l(n_1,r)\geqslant n_2$. Meanwhile, since there are only $k(n_1,r)$ redundant packets in total, we have $l(n_1,r)\leqslant n_2+k(n_1,r)$. We also have $l(n_1,r)>k(n_1,r)$, otherwise the lost packets could be recovered with FEC. Therefore, the probability of n_2 data packet losses under the condition of $l(n_1,r-1)$ total packet losses follows the hypergeometric distribution:

$$\begin{aligned}&H(n_2;n_1+k(n_1,r-1),n_1,l(n_1,r))\\&\quad=\binom{n_1}{n_2}\binom{k(n_1,r)}{l(n_1,r)-n_2}\Big/\binom{n_1+k(n_1,r)}{l(n_1,r)}\end{aligned} \quad (8)$$

Thus, the transition probability from (n_1,r) to $(n_2,r-1)$ is:

$$\begin{aligned}&p((n_1,r)\to(n_2,r-1))=\\&\sum_{l(n_1,r)} H(n_2;n_1+k(n_1,r),n_1,l(n_1,r))\cdot P(l(n_1,r)\text{ losses})\end{aligned} \quad (9)$$

On the other hand, at the loss rate of α, losing $l(n_1,r)$ packets in all $n_1+k(n_1,r)$ packets follows the Binomial distribution:

$$\begin{aligned}&P(l(n_1,r)\text{ losses})=Bi(l(n_1,r);n_1+k(n_1,r),\alpha)=\\&\quad\binom{n_1+k(n_1,r)}{l(n_1,r)}\alpha^{l(n_1,r)}(1-\alpha)^{n_1+k(n_1,r)-l(n_1,r)}\end{aligned} \quad (10)$$

Therefore, by substituting Eq. 8 and 10 into Eq. 9, we can have the transition probability for $n_2>0$. Similarly, when state transits from (n_1,r) to $(0,r-1)$, then the number of lost packets in the r-th layer of Fig. 8 must be less than $k(n_1,r)$. Therefore, the transition probability satisfies:

$$p((n_1,r)\to(0,r-1))=\sum_{i=0}^{k(n_1,r)} Bi(i;n_1+k(n_1,r),\alpha) \quad (11)$$

B.2 Optimization of Block Size

In the following analysis, we are going to compare the utility of transmitting the whole frame for L chances, or splitting the frame into several blocks and some of them enjoying L+1 chances. With that, we assume that the dispersion is less than one RTT.

Therefore, when the block size is set to d, there are N_{L+1} blocks that could enjoy L+1 chances of transmission, and the remaining N_L blocks with L chances of transmission, where

$$\begin{aligned}N_{L+1}&=\left\lfloor\frac{DDL-(L+1)\cdot RTT}{d/\Theta}\right\rfloor\\N_L&=\left\lceil\frac{F}{d}\right\rceil-N_{L+1}\end{aligned} \quad (12)$$

Therefore, the on-time delivery of the frame requires the on-time delivery of each block. Since the deadline miss rate is equal to one minus the probability of on-time delivery, we have the frame

DMR (FDMR) given a certain block size d as:

$$\begin{aligned} 1-FDMR(d) &= (1-DMR(L+1,d))^{N_{L+1}} \cdot (1-DMR(L,d))^{N_L} \\ \Rightarrow FDMR(d) &= 1-(1-DMR(L+1,d))^{N_{L+1}} \cdot (1-DMR(L,d))^{N_L} \\ &= N_{L+1} \cdot DMR(L+1,d) + N_L \cdot DMR(L,d) \end{aligned} \tag{13}$$

where the last equation holds since $DMR(L, d) \ll 1$ and $(1-\alpha)^n = 1-n\alpha$ when $\alpha \ll 1$. As for the bandwidth cost, recalling Eq. 7, the number of extra packets of the frame is the sum of the number of extra packets for each block. Since the BWC of each block shares the same denominator (frame size S), the frame BWC is also the sum of BWC of each block:

$$FBWC(d) = N_{L+1} \cdot BWC(L+1,d) + N_L \cdot BWC(L,d) \tag{14}$$

Therefore, the optimal block size is:

$$d_{opt} = \underset{d}{\operatorname{argmax}}\ utility(FDMR(d), FBWC(d)) \tag{15}$$

In our implementation, we iterate the possible block size B from 1 to the frame size S, and store the optimal block size in each scenario in an offline lookup table. Since the $DMR(L,B)$ and $BWC(L,B)$ are accessible in the absorbing Markov chain constructed above, the construction of the table is time-efficient.

C Implementation Details

We are going to introduce the sending mechanism beneath Hairpin and the implementation of the redundancy optimization in Hairpin.

Acknowledgement aggregation. In wireless networks, researchers also propose to aggregate several acknowledgements at the client side to alleviate the uplink interference [52]. However, the delayed acknowledgement might also interfere with the measurements of RTT, delay the detection of packet losses and waste potential chances of retransmission. In our implementation, to eliminate the interference from acknowledgement mechanisms, we disable the aggregation of acknowledgements. The precise measurement of RTT in the scenario of aggregated acknowledgement could also be implemented with recent efforts such as TACK [52], which is out of our scope.

Note that this is different from the aggregation on wireless routers [19]. Such aggregations due to wireless channel competition should be reflected in our measurements of network RTT fluctuations in Fig. 4. In our simulation with online measurements and deployments in production, Hairpin behaves well even with the RTT fluctuations.

FEC codec. For the scenarios with a redundancy rate of $\leqslant$100%, we implement the FEC codec as RS-FEC, as suggested by many other related efforts [65]. We refer the readers to [65] for the details of RS-FEC. However, when implementing the redundancy rate of >100%, RS-FEC is not designed to reliably recover lost packets in all cases. For example, when there are 2 data packets and 4 FEC packets, RS-FEC cannot always recover 2 data packets when there are 4 packet losses due to the invertibility of the decoding matrix: it depends on whether two packets received at the client are linearly independent at the generation matrix.

Therefore, we implement a customized FEC codec. For example, for data packets a and b, when considering them as two numbers (with a length of up to 12kbits), we could calculate $a+b$, $a+2b$, $2a+b$, etc., and send them to the client. The only overhead is the additional bits that could overflow from the addition, which is much less than the data bits. Moreover, as shown in Fig. 25, in most cases the redundancy rate is less than 100%. Therefore, the overall decoding overhead is also acceptable. We leave the further adoption of advanced FEC codec when the redundancy rate is >100% as our future work.

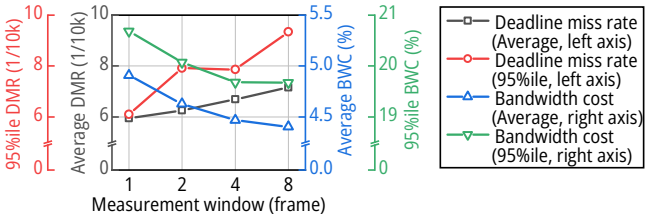


Figure 20: Sensitivity of the measurement window in §3.4.

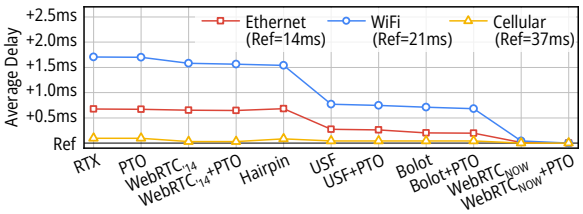


Figure 21: Average end-to-end delay of in the experiments in §4.3. We trim the lowest average delay in different traces for comparison.

D Supplementary Experiments

Measurement window. We also evaluate the performance of Hairpin by adjusting the measurement window of the network conditions that we discussed in §3.4. Since Hairpin optimizes the redundancy parameters based on real-time measurements of the network conditions, the size of the measurement window might affect the performance of Hairpin. We vary the measurement window from the last 1 to 8 frames and reconduct the experiments over WiFi traces. We measure the average and 95th percentile DMR and BWC, and present the results in Fig. 20. The DMR and BWC are quite robust: By varying the measurement window from 1 to 8, the average DMR and average BWC vary within 0.47%-0.49% and 6.94%-7.19%, which is subordinate to the improvements in §4.3 (Fig. 11(b)). In practice, operators can decide the measurement window based on the fluctuations of network conditions.

Per-frame latency of Hairpin. Besides, we also measure the average end-to-end delay for the successfully delivered frames in the experiments in §4.3 for Hairpin and different baselines. As shown in Fig. 21, the average end-to-end delay of Hairpin does increase compared to the baseline with the lowest average delay. However, the increase is only 0.1-1.5ms for all traces, which is negligible compared with the RTT (1%-7%), and considering

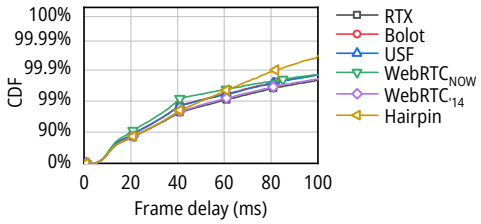
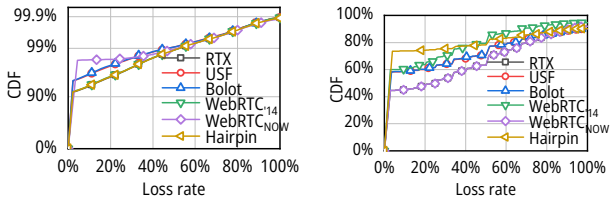


Figure 22: The distribution of the delivery time of each frame. Note that the y-axis is log-scaled.



(a) Initial transmission (b) The 3rd retransmission

Figure 23: Distribution of loss rates by frame in each round of transmission.

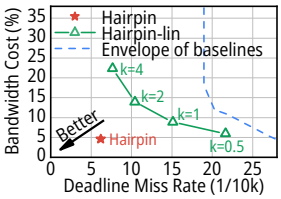


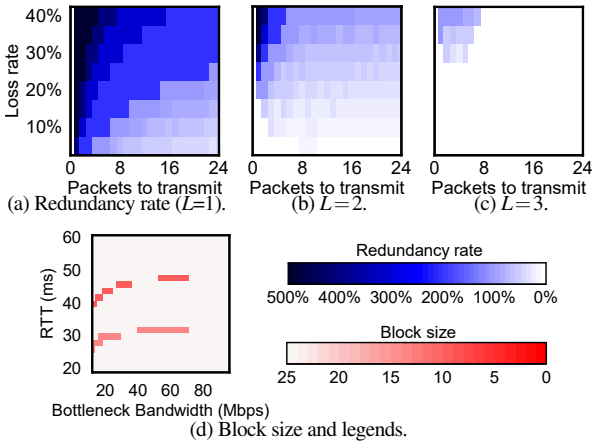
Figure 24: Heuristic-based Hairpin (Hairpin-lin). The envelope of baselines is from Figure 11(b).

the deadline effect we discussed in §2.1. Furthermore, operators could also adopt less aggressive mappings (e.g., increasing λ) to tradeoff between the tail delay and average delay.

We also present the distribution of the delay of each frame in Fig. 22. Similar to Fig. 21, the average (median) latency of frames of Hairpin is similar to other baselines. However, Hairpin could reduce the tail latency significantly. For example, Hairpin can reduce the 99.9th percentile frame latency to 80ms while all baselines of longer than 100ms. Looking at the vertical axis, Hairpin is also capable of reducing the ratio of higher than 100ms by more than a half, as shown in Fig. 11(b).

Loss rates in each round. We further present the distributions of loss rates of all frames in each round (specifically, initial transmission and the third retransmission) in Fig. 23. This expands the results in Fig. 15(b). We can tell from Fig. 23(a) that due to the conservative redundancy strategy of Hairpin, the loss rate of Hairpin is higher. However, when retransmission starts, Hairpin is able to maintain a low loss rate – which means a high success rate in delivering frames – compared to other baselines. This shows the strategy of Hairpin: conservatively adding FEC packets when deadline is far away, and aggressively adding FEC packets to retransmissions.

The improvements of using Markov chain. As we analyzed in §3.2, a strawman solution is good but not enough to fully utilize the design space of redundancy and retransmission. Thus, we also evaluate the heuristic baseline we present in §3.1 (denoted



(a) Redundancy rate (L=1). (b) $L=2$. (c) $L=3$.

(d) Block size and legends.

Figure 25: Optimization results by Hairpin. Fig. 25(a) to 25(c) present the redundancy rate with different transmission chances L.

as Hairpin-lin, with sweeping the coefficient k from 0.5 to 4, and present the results in Figure 24. As we can see, Hairpin-lin (green line) does improve the trade-off compared to existing baselines (dashed blue line). Yet, there is still a half gap between Hairpin-lin and the Markov chain-based Hairpin (the red star). Therefore, it is necessary to analytically formulate the problem with the Markov chain to further push the trade-off forward.

Understanding Hairpin's decisions. We further present the redundancy rate results of Hairpin to provide a deeper understanding of how Hairpin optimizes in different scenarios. For redundancy rate, since the optimization of the absorbing Markov chain (§3.2) relies on the remaining transmission chance L, loss rate, remaining data packets to transmit, and the frame size, we present the optimized redundancy rate over different parameters in Fig. 25(a) to 25(c). With more transmission chances, Hairpin would decrease the redundancy rate and rely on retransmissions for packet loss recovery. With fewer packets to retransmit, Hairpin also prefers a higher redundancy rate, as discussed in §3.2. Moreover, when the number of packets to transmit is small, the optimized redundancy rate is up to 500% in Fig. 25(a), demonstrating the effectiveness of a redundancy rate of $>100\%$.

As for the optimization of block size, as we also discussed in §3.2, the optimal block size is the frame size (24 packets) in many cases. Nevertheless, as we discussed, at the decision boundary of remaining transmission chance, smaller block sizes do enjoy a slightly better performance by having additional transmission chances. As shown in Fig. 25(d), although the optimal block size is the frame size in most cases, when the RTT is around 33ms and 50ms (the dividing point between 1, 2, and 3 transmission chances), the optimal block size might be smaller than the frame size. For example, compared to setting the block size to the frame size, the DMR with the optimized block size of Hairpin could be further reduced by 1.78× around the RTT of 50ms and bottleneck bandwidth of 60Mbps. We optimize the block size for the last mile performance improvement.

E Limitations

Delay components in interactive streaming. Hairpin could have maximum benefits when the end-to-end network delay dominates the total delay from the video encoder to the decoder in Fig. 10. This is generally true in interactive streaming services. Related measurement studies also demonstrate that the network delay is still one of the bottlenecks of edge-based interactive streaming [43, 57]. Therefore, we focus on the optimization of streams between edge servers and clients. Our deployments in the wild demonstrate that optimizing the network latency could significantly improve the user's experience (note that DMR is measured end-to-end). Hairpin can also work with the optimization of other delay components (e.g., encoding, decoding, etc.) to further improve the performance.

Deployment efforts for applications. Another concern of deploying Hairpin is that both the server and the client need modification to support the redundancy and retransmissions. There are previous efforts implementing the FEC mechanism over TCP [16, 37], which needs to modify the TCP protocol stack at the client and are not suitable for products at scale. For scenarios where TCP is compulsory for transport, the deployment of Hairpin may depend on the ability to modify the reception mechanism of TCP packets at the client. However, most interactive streaming applications adopt UDP to reduce the network delay [23, 35, 55, 66], including our service. In this case, Hairpin could be implemented within the application at the server and the client, which is practical for most applications.

Finding Adversarial Inputs for Heuristics using Multi-level Optimization

Pooria Namyar[†‡], Behnaz Arzani[†], Ryan Beckett[†], Santiago Segarra[†⋆],
Himanshu Raj[†], Umesh Krishnaswamy[†], Ramesh Govindan[‡], Srikanth Kandula[†]
[†]*Microsoft,* [‡]*University of Southern California,* [⋆]*Rice University*

Abstract– Production systems use heuristics because they are faster or scale better than their optimal counterparts. Yet, practitioners are often unaware of the performance gap between a heuristic and the optimum or between two heuristics in realistic scenarios. MetaOpt is a system that helps analyze these heuristics. Users specify the heuristic and the optimal (or another heuristic) as input, and MetaOpt encodes these efficiently for a solver to find performance gaps and their corresponding adversarial inputs. Its suite of built-in optimizations helps it scale to practical problem sizes. We used MetaOpt to analyze heuristics from three domains (traffic engineering, vector bin packing, and packet scheduling). We found a production traffic engineering heuristic can require 30% more capacity than the optimal in realistic cases. We modified the heuristic based on the patterns in the adversarial inputs MetaOpt discovered and reduced the performance gap by 12.5×. We examined adversarial inputs to a vector bin packing heuristic and proved a new lower bound on its performance.

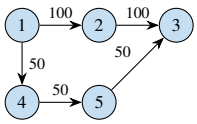
1 Introduction

Many solutions to network and systems problems are heuristic approximations to potentially intractable optimal algorithms [3, 20, 35, 38, 39, 53, 57, 58, 76]. These heuristics are often faster or scale better than their optimal counterparts. However, operators often do not fully understand how these heuristics will behave with new and untested inputs or how far from the optimal their results may drift in realistic use.

For example, Microsoft uses a heuristic, demand pinning (DP), on its wide-area network [46, 56]. It routes small demands (*i.e.,* demands $\leq$ a threshold) through their shortest path and uses a more computationally complex optimization to route the rest [38] (DP reduces the number of demands the optimization routes and completes faster). In Fig. 1, we show an example where DP allocates 40% less demand than the optimal routing. With this gap, Microsoft may have to either over-provision its network by 40%, delay 40%, or drop 40% of its customers' demands! Note that the 40% gap is a lower bound, and the worst-case gap can be higher.

We often do not understand the potential impact of such heuristics at scale: Does their gap depend on the problem size? Which inputs make them perform poorly? Are there realistic scenarios that they struggle with? We ask if we can develop techniques to *analyze heuristics* and answer such questions.

As a first step towards this goal, we have developed



Demand		DP (thresh = 50)		OPT	
src-dest	value	path	value	path	value
1-3	50	1-2-3	50	1-4-5-3	50
1-2	100	1-2	50	1-2	100
2-3	100	2-3	50	2-3	100
		Total DP	**150**	Total OPT	**250**

FIGURE 1: Suboptimal performance of DP. (left) Topology with unidirectional links. (right) A set of demands and their flow allocations using the DP heuristic and the optimal (H′) solution. DP first sends the demands at or below the threshold (50) over their shortest paths and then optimally routes the remaining demands.

MetaOpt, a system that allows users to *automatically discover the performance gap* between a heuristic H and any other function H′ for much larger problem sizes than in Fig. 1. MetaOpt also returns the *adversarial inputs* to these functions that cause large performance gaps. Users can use MetaOpt to (a) understand the performance gap of H relative to the optimal or to another heuristic; and (b) examine adversarial inputs to provide performance bounds on H or to modify H to improve its performance gap (§2).

In many problem domains, such as traffic engineering [38, 39, 46, 58], vector bin packing [35, 60], and packet scheduling [5, 64, 74], we can specify both H′ and H either as an optimization (with an objective and several constraints) or as a feasibility problem (with a set of constraints). Then, we can model the problem of finding large performance gaps between H and H′ in the language of optimization theory:

$$\underset{\text{s.t. input } \mathcal{I} \in \text{ConstrainedSet}}{\arg\max} \; \mathsf{H}'(\mathcal{I}) - \mathsf{H}(\mathcal{I}), \tag{1}$$

where H() and H′() take $\mathcal{I}$ as input and solve the corresponding algorithms. ConstrainedSet specifies a set of constraints that limit the set of values $\mathcal{I}$ can take.

In theory, we could throw this model at a solver [16, 36] and find performance gaps, but existing solvers do not support these optimizations. This model is an instance of a bi-level optimization [15] (with connections to Stackelberg games [23], see §6), and practitioners have to rewrite them into single-level optimizations before using a solver [15]. Rewriting a bi-level optimization into a single-level one by hand is tedious and can lead to poor performance if done incorrectly.

MetaOpt abstracts away this complexity and only asks users to specify H′ and H (§3). It also provides helper functions to make it easier to specify H′ and H. These functions are especially useful for constructs (*e.g.,* conditionals, randomization) that are harder to express as optimization constraints.

Under the hood, MetaOpt performs rewrites automatically and supports multiple solvers (Gurobi [36] and Zen [16]). We add three techniques to scale MetaOpt to large problems (*e.g.,* large topologies and demands for traffic engineering). First, since many rewrites can introduce non-linearities, we carefully select which part of the input to rewrite. Second, we introduce a new rewriting technique (Quantized Primal-Dual) that allows MetaOpt to trade-off between scale and optimality. Third, we design a new partitioning technique for graph-based problems to improve MetaOpts scalability further.

We show the versatility of MetaOpt[1] by using it to study several heuristics in traffic engineering [38, 39, 46, 58], vector bin packing [35, 60], and packet scheduling [5, 64, 74]. We have applied MetaOpt to (a) study performance gaps of these heuristics, (b) analyze adversarial inputs to prove properties, and (c) devise and evaluate new heuristics (see Table 1):

- DP can allocate 33% less demand compared to the optimal in large topologies. We analyzed adversarial inputs MetaOpt found and designed modified-DP which reduced this gap by an order of magnitude.
- We show for the first time that a two-dimensional vector bin packing heuristic FFDSum can require at least twice as many bins as the optimal *across all problem sizes*.
- A recently proposed programmable packet scheduler, SP-PIFO [64], an approximation of PIFO [64], can delay the highest priority packet by at least 3× relative to PIFO.

2 Heuristic Analysis at a Glance

Network and systems designers use heuristics when the optimum is too expensive to compute at relevant problem scales. We use examples of heuristics to motivate the types of analyses designers might wish to perform and describe how MetaOpt can aid these analyses.

2.1 Heuristics and their Importance

We describe heuristics from traffic engineering, cluster resource allocation, and switch packet scheduling.

Traffic engineering (TE). There are many techniques to scale TE solutions to large networks/demands (see §A.2 for details):

Demand Pinning (DP) [46, 56] is a heuristic that Microsoft uses in production. It pre-allocates flows along the shortest path for any node pair with demand smaller than a threshold T_d and uses the SWAN [38] optimizer on the rest. When many demands are small, this can result in substantial speedup.

Partitioned Optimization Problems (POP) [58] divides node pairs (and their demands) uniformly at random into partitions. It then assigns each partition an equal share of the edge capacities and solves the original problem (*e.g.,* the SWAN LP optimization [38]) once per partition. POP is faster than SWAN because it can solve each LP sub-problem much faster than the original [22] and can do so in parallel.

Vector bin packing (VBP). Production deployments use VBP to allocate resources in clusters efficiently [2, 35, 37, 53, 68]. One version of VBP takes a set of balls and bins with specific sizes and multiple dimensions (*e.g.,* memory, CPU, GPU [32, 48, 69]) and tries to pack the balls into the fewest number of bins. The optimal algorithm for this version is APX-hard [71]. Instead, many practitioners use a heuristic, first fit decreasing (FFD), which is greedy and iterative. At each step, FFD picks the unassigned ball with the *largest weight* and places it in the *first* bin that *fits* (has enough capacity). Prior works propose different ways to weigh the balls [35, 37, 53, 60, 66, 67, 72]. One variant, FFDSum, uses the sum across all dimensions as the weight of a ball.

Packet scheduling [5, 74]. Push-In-First-Out (PIFO [64]) queuing is a scheduling primitive that enables various packet scheduling algorithms for programmable switches. SP-PIFO [5] uses n priority FIFO queues to approximate PIFO and presents a heuristic we can implement at line rate. In SP-PIFO, each queue has a priority (*usually* equal to the priority of the last packet in it). A queue only admits a new packet if its priority is higher than the packet's priority. The algorithm scans queues from lowest to highest priority and places the new packet in the first queue that accepts it. SP-PIFO updates all queue priorities if no queue admits the packet.

Performance Analyses. Often, heuristic designers wish to answer questions such as:

- How far is my heuristic from the optimum?
- What inputs cause my heuristic's performance to degrade at practical problem instances?
- How can I redesign my heuristic to improve its performance?
- How can I compare the performance of two heuristics?
- Can I prove lower bounds on a heuristic's performance relative to another standard algorithm?

2.2 MetaOpt, a Heuristic Analyzer

MetaOpt is a heuristic analyzer that can help designers answer these questions. It finds (a) the *performance gap* between a given $\mathsf{H}()$ and an alternative $\mathsf{H}'()$ (where $\mathsf{H}'()$ can be the optimal solution) and (b) the adversarial inputs to $\mathsf{H}()$ that cause the performance gap. The performance of a heuristic measures its solution quality. Users decide what performance metric to use; for example, they can define the performance as the total flow that the heuristic admits in TE or the average packet delay in packet scheduling. The performance gap is the difference between the performance of $\mathsf{H}()$ and $\mathsf{H}'()$. MetaOpt can analyze a broad set of well-defined heuristics (see §3). We discuss its limitations in §5.

2.3 Using MetaOpt to Analyze Heuristics

Heuristic designers can use MetaOpt to answer the questions we presented in §2.1. They can use MetaOpt in two ways:

[1] Our code is available at https://github.com/microsoft/MetaOpt.

	MetaOpt finds performance gaps	MetaOpt helps prove properties	MetaOpt helps modify heuristics
TE	DP and POP can be 33.9% and 20% less efficient than optimal. MetaOpt finds realistic demands with strong locality that produce the same gap.	–	Modified-DP (which we designed with MetaOpt's help) has an order of magnitude lower gap than DP. We cannot improve the gap by running DP and POP in parallel.
VBP	MetaOpt finds tighter performance gaps under realistic constraints. It also finds the same examples which took theoreticians decades to find and prove a tight bound for 1d-FFD.	MetaOpt helped prove a tighter approximation ratio for 2-d FFDSum than previously known [60].	–
PIFO	MetaOpt finds SP-PIFO can delay the highest priority packet by 3×, and also finds inputs where AIFO incurs 6× more priority inversion than SP-PIFO.	MetaOpt helped prove a new bound on the weighted average delay of SP-PIFO.	Modified-SP-PIFO (which MetaOpt helped design) has 2.5× lower weighted average packet delay.

TABLE 1: MetaOpt (1) finds the performance gap between the heuristic and optimal; (2) helps prove various properties about the heuristic; and (3) helps modify them to improve their performance.

(a) to find performance gaps and (b) to prove properties or improve heuristics based on the adversarial inputs it finds. We next describe the results we obtained by applying MetaOpt to the heuristics in §2.1. We present more details and other results in §4. These use cases are not the only way operators can use MetaOpt, but they demonstrate its versatility (see §8).

Finding performance gaps. We show how MetaOpt helps find performance gaps in TE and packet scheduling.

Performance gaps in traffic engineering. We use MetaOpt to find the performance gaps for DP and POP, where $H'()$ is the optimal multi-commodity flow algorithm (§A.1). We measure the performance gap as the difference between the heuristic's and the optimal's total flow, normalized by the total network capacity. The performance gap is a lower bound on the *optimality gap*, the worst-case gap between the two.

We find DP and POP incur 33.9% and 20% relative performance gaps on a large topology (Cogentco, §4). This means there exists (and we can find) adversarial traffic demands that cause DP to use at least 33.9% more capacity than optimal. Network operators that use DP may need to over-provision the network by that much to satisfy this demand.

MetaOpt, by default, searches for adversarial inputs among all possible demands. We can constrain MetaOpt to search over *realistic* demands. These are sparse and exhibit strong locality, which means few node pairs that are close to each other exchange most of the traffic [3]. When we run MetaOpt with these constraints, the gap for POP and DP remains almost the same as before, but the discovered adversarial demands exhibit stronger locality and are sparser (*i.e.,* the adversarial demands that MetaOpt finds becomes more realistic).

Performance gaps in packet scheduling. We compare SP-PIFO to PIFO. We compute and compare the priority-weighted average packet delay (§4) between the two algorithms, which penalizes them if they increase the delay of high-priority packets. MetaOpt shows there exists an input packet sequence where SP-PIFO is 3× worse than PIFO.

We also use MetaOpt to compare SP-PIFO and AIFO [74] (two heuristics). AIFO emulates PIFO through a single FIFO queue and replaces $H()$ in this scenario. MetaOpt finds inputs for which AIFO incurs 6× more priority inversions than SP-PIFO. Such analyses can help designers weigh performance trade-offs against switch resource usage.

Proving properties of and improving heuristics. MetaOpt discovers performance gaps and the corresponding adversarial inputs. We show how to prove performance bounds for these heuristics or to improve them in several cases[2]. These require the user to *analyze the inputs further* to see if they share a common pattern (see §B.2 for an example).

A new bound for vector bin-packing. VBP heuristics try to minimize the number of bins they use. Theoreticians prove bounds on their approximation ratio: the worst-case ratio of the number of bins the heuristic uses compared to the optimal over any input. Recent work [60] showed 2-dimensional FFDSum asymptotically approaches an approximation ratio of 2 (where the optimal uses nearly infinite bins). We prove (§4) that the approximation ratio is always at least 2 — even when the optimal requires a finite number of bins!

A new bound and a better heuristic for packet scheduling. We analyzed adversarial inputs MetaOpt found for SP-PIFO and proved a lower bound on its priority-weighted average delay relative to PIFO. The bound is a function of the priority range and the number of packets.

Adversarial inputs to SP-PIFO trigger priority inversions, which means they cause SP-PIFO to enqueue high-priority packets behind low-priority ones. We tested a Modified-SP-PIFO that splits queues into groups; it assigns each group a priority range and runs SP-PIFO on each group independently. This modification reduces the possibility of packets with vastly different priorities interfering with each other and causing priority inversions. Modified-SP-PIFO reduces the performance gap of SP-PIFO by 2.5×. Users should weigh the trade-off between the improved performance gap, the

[2]One may have to sacrifice other metrics such as run-time to improve the gap. Users have to weigh this trade-off and decide if it is acceptable.

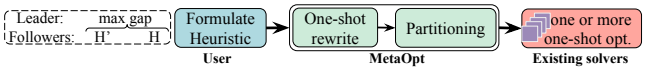


FIGURE 2: MetaOpt's workflow involves four steps; (1) User encodes H and H′ §3.2, (2) MetaOpt automatically applies rewrites to obtain a single-level optimization §3.3, (3) it partitions the problem into smaller subproblems to achieve scalability §3.5, (4) MetaOpt feeds the resulting optimizations into existing solvers [36, 55] and finds large performance gaps.

Optimal	Heur.	Formul.	Rewrite
OptMaxFlow	POP	Convex(Random) §A.3	KKT/PD
	DP	Convex(Conditional) §A.3	KKT/PD
VBP MILP	FFD	NonConv.(Greedy) §B.1	Feasibility
PIFO MILP	SP-PIFO	NonConv.(Priority) §C.1	Feasibility
	AIFO	NonConv.(Admission) §C.2	Feasibility

TABLE 2: Overview of the five heuristics we explored in this paper. We cover how to formulate the heuristics as optimizations in the appendix and discuss their rewrites as constraints in §3.3.

memory allocation, and the impact on ongoing traffic to decide if they should deploy Modified-SP-PIFO.

Improving traffic engineering heuristics. We found that DP performs poorly when small demands traverse long paths. We used MetaOpt to analyze a Modified-DP, which routes demands on their shortest paths if the shortest path is less than k hops and the demand is less than T_d. This simple change reduced the performance gap by an order of magnitude. Modified-DP presents a trade-off between the speed and the performance gap of the heuristic. Users can control this trade-off through k and T_d. MetaOpt can guide users in choosing these values by quantifying each choice's performance gap.

3 MetaOpt Design

Our goal is to build a *widely applicable* system that finds *large performance gaps* between H and H′ *quickly and at scale*.

3.1 MetaOpt Approach

This is a hard problem when H and H′ are arbitrary algorithms, but we observe that we can formulate many heuristics in networks and systems as:

Convex optimization problems in which the heuristic seeks to optimize an objective subject to a collection of constraints. DP and POP fall into this category: DP solves optimization for large demands, and POP solves optimization for each partition.

Feasibility problems in which the heuristic searches for a solution that satisfies a collection of constraints. FFDSum and SP-PIFO fall into this category: FFDSum packs balls into bins subject to weight and capacity constraints, and SP-PIFO places packets into the queues based on their priorities.

We can find the performance gap between any H′ and H through a bi-level or meta optimization (hence MetaOpt) as long as H falls in one of these two classes. We do not need convexity for H′ — we need it to be either an optimization or

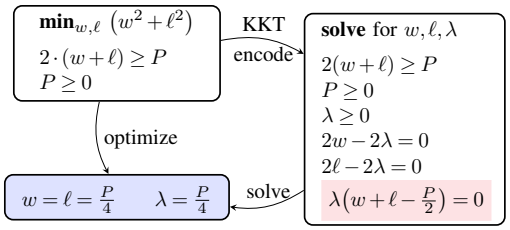


FIGURE 3: Rewrite using KKT in an example where we find a rectangle's optimal width w and length ℓ such that its perimeter is $\geq P$. The inner variables are w and ℓ. The right panel shows the feasibility problem using the KKT theorem. Equations with λ variables correspond to first-order derivatives of inequality constraints in the original problem. P is a variable of the outer optimization but is treated as a constant in the inner problem.

a feasibility problem. We model the problem as[3]:

$$\begin{aligned}
&\arg\max_{\mathcal{I}}\ \mathsf{H}'(\mathcal{I})-\mathsf{H}(\mathcal{I}) && \text{(leader problem)}\\
&\text{s.t.}\quad \mathcal{I}\in \mathsf{ConstrainedSet} && \text{(input constraints)}\\
&\mathsf{H}'(\mathcal{I})=\max_{\mathbf{f}'\in\mathcal{F}'}\mathsf{H}'_\mathsf{Objective}(\mathbf{f}',\mathcal{I}) && \text{(optimal)}\\
&\mathsf{H}(\mathcal{I})=\max_{\mathbf{f}\in\mathcal{F}}\mathsf{H}_\mathsf{Objective}(\mathbf{f},\mathcal{I}) && \text{(heuristic)}
\end{aligned}\qquad (2)$$

where the *leader* or *outer* optimization maximizes the difference between the two functions (*i.e.,* the performance gap) over a space of possible inputs $\mathcal{I}$. This leader problem is subject to *follower* or *inner* problems (H′ and H). We model the performance of H' on input $\mathcal{I}$ through H'_Objective. This function decides the values for the variables $\mathbf{f}'$, internally encodes problem constraints, and computes the overall performance of H′. H'_Objective treats the outer problem's variables, $\mathcal{I}$, as input and constant. We define H_Objective the same way.

Bi-level Optimization: A Brief Primer. Modern solvers do not directly support the style of bi-level optimizations we described in Equation 2 [36]. To solve these, users need to rewrite the bi-level optimization as a single-level optimization [15]. These rewrites convert an optimization problem into a set of feasibility constraints: if the inner problems are both optimizations, the rewrites will replace both H′() and H() with a set of feasibility constraints in the outer optimization. The resulting formulation is a single-level optimization that modern solvers [16, 36] can attempt to solve.

Fig. 3 shows an example of an inner problem and the corresponding rewrite. This somewhat contrived example finds a rectangle's optimal length and width subject to some constraints (the outer problem may want to optimize P). The rewrite uses the Karush–Kuhn–Tucker (KKT) theorem [22] and converts convex optimizations with at least one strictly feasible point [22] into feasibility problems. The theorem states any point that solves the new problem matches the solution of the original. We describe another technique in

[3] Note that we can transform minimization optimizations to maximization by negating their objective.

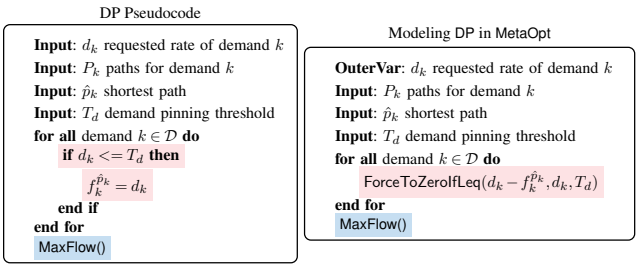


FIGURE 4: Modeling DP in MetaOpt. (see Table A.1 for notation)

§3.4, which exploits the Primal-Dual theorem [22]. Both of these rewriting techniques produce a single-level optimization equivalent to the bi-level formulation if the inner problems are convex [15]. The problem MetaOpt solves has properties that allow us to reduce the overhead of these rewrites (see §3.3).

3.2 MetaOpt: The User View

Inputs. MetaOpt could have asked the user to input the single-level formulation, which can be hard and error-prone. The rewritten single-level formulation can have an *order of magnitude* more constraints than the original bi-level formulation (§4) and is hard to optimize for practitioners who do not understand bi-level optimizations.

MetaOpt simply lets the user input H and H′ (Fig. 2). It then automatically produces a single-level formulation, optimizes the rewrites, feeds these into a solver of the user's choice (MetaOpt currently supports Gurobi [36] and Zen [16]), and produces performance gaps and adversarial inputs.

How to specify H or H′. It can be hard to describe H or H′ in the optimization language. When we modeled problems in MetaOpt, we observed certain constructs are common across many heuristics. We encode these as helper functions to make it easier for users less familiar with optimization theory to model their heuristics. The set of helper functions is not complete (there may be constraints for which we have not devised a corresponding helper function) but the interface is extensible and we can add new functions as needed.

For instance, a user who wants to compare the performance of DP and the optimal TE multi-commodity flow would have to specify the formulation of both of these algorithms. Standard textbooks describe the former [17], so we focus on the latter. DP involves a conditional (an `if` statement on the left of Fig. 4) where the outcome is determined based on the demands. These demand values are *variables* of the outer problem and unknown apriori, which means we need to determine the outcome of these conditionals as part of the optimization and model them as convex constraints.

We can use the big-M method [22] (§A.3) to convert this `if` statement into constraints optimization solvers support. MetaOpt provides a helper function, ForceToZeroIfLeq (see Fig. 4), to help users do this conversion. This level of indirection makes it easier for the user to specify DP and also

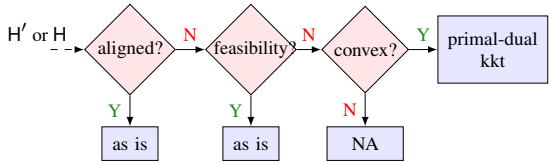


FIGURE 5: MetaOpt automatically converts the bi-level problem to a single-level optimization. It supports any follower, which either (1) is a convex optimization; (2) is a feasibility problem; or (3) has an objective that aligns with the outer problem.

gives MetaOpt the flexibility to optimize or change the formulations when needed. For example, the big-M method causes numerical instability in larger problems, and MetaOpt uses an alternate method to convert it to constraints (see §A.3).

Our helper functions (§D, Table A.8) codify common design patterns and help specify constraints across a diverse set of problems. We show how to use and combine them to model other heuristics, such as, FFD, which involves greedy decisions, and SP-PIFO, which involves dynamic priority updates. These helper functions encode succinct and readable models.

3.3 Automatic Rewrites

MetaOpt produces a bi-level optimization from H′ and H (Equation 2) and then automatically rewrites it. While the underlying theory behind rewrites is well-known [15], to our knowledge, there are no automated rewriters, and the process has required human intervention until now. We need to be careful when we automate rewrites. For example, one challenge is modeling non-linear constraints that involve the multiplication of variables. In the Primal-Dual rewrite (§3.4), the constraints in the dual depend on the type of optimization (maximization or minimization) and whether the corresponding primal variable is unconstrained, positive, or negative [22].

We have developed automatic rewrite techniques for KKT, Primal-Dual, and a new variant of the latter (*i.e.,* Quantized Primal-Dual §3.4). Users can choose which rewrite they use.

MetaOpt does not naïvely rewrite the bi-level formulation and only rewrites the inner problems if necessary. We call this technique *selective rewriting*. It avoids rewriting in two cases (Fig. 5): when the inner problem is a feasibility problem or when it is "aligned". In these cases, we can directly merge the inner problem's constraints into the outer problem and remove its objective if they have one.

An *aligned inner problem* is one where optimizing the outer problem's objective directly or indirectly optimizes the inner problem as well. We observe that the objective of MetaOpt is such that one of H′ or H is always aligned with the outer problem. The outer problem maximizes H′ and minimizes H to maximize the gap. This aligns with H′ if H′ is a maximization problem and with H when H is a minimization (both H and H′ solve the same problem, so they are either both maximizations or both minimizations).

For all other instances, we need to rewrite the inner problem. MetaOpt currently rewrites an inner problem only if it is an

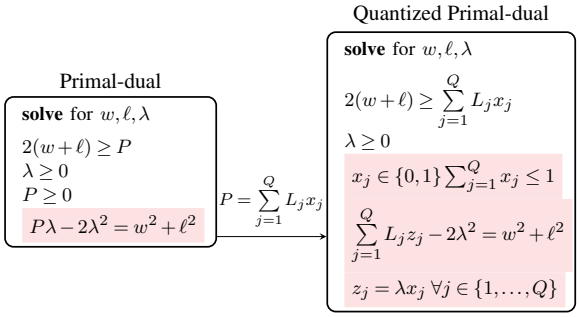


FIGURE 6: Left shows rewrite using primal-dual theorem. It has fewer constraints and different multiplicative terms ($P\lambda$ versus $\lambda(w+\ell-\frac{P}{2})$ in the KKT rewrite). On the right, we show how to quantize the parameters of the outer problem (P). The QPD rewrite no longer has any multiplicative terms since we can linearize the multiplication of binary (x_j) and continuous variables (λ). The quadratic terms (*e.g.,* w^2) is due to the quadratic objective of the original problem. L_js are constants.

unaligned convex optimization. KKT and Primal-Dual apply to these cases. MetaOpt also supports *unaligned non-convex* inner problems that can be written as a feasibility problem (*e.g.,*SP-PIFO and FFD). See §5 for other extensions.

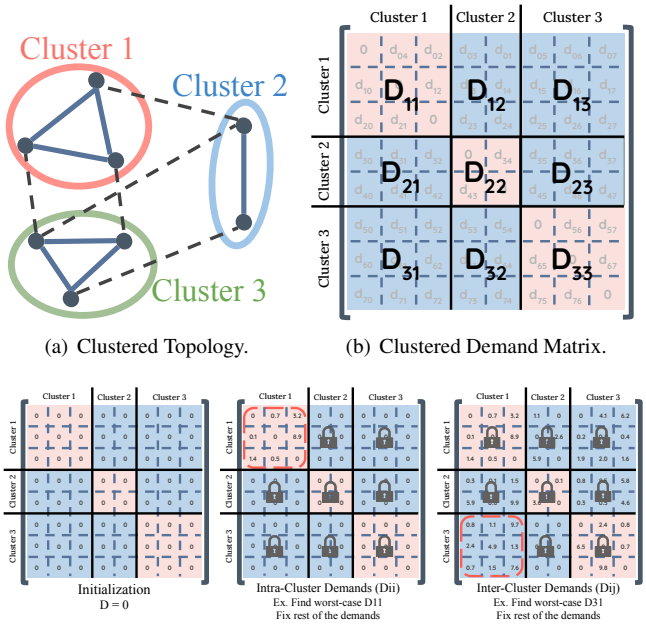
With all this, MetaOpt generates a single-level formulation that is equivalent to the bi-level optimization, preserves the theoretical properties, and scales well.

3.4 The Quantized Primal-Dual Rewrite

The KKT rewrite does not scale beyond small problems. It introduces multiplicative terms (pink highlighted constraint in Fig. 3) that commodity solvers support (special ordered sets in Gurobi [36] and disjunctions in Z3 [55]). However, these constraints with multiplicative terms slow down the solvers and dictate their latency.

A similar observation holds for the Primal-Dual rewrite [15]. It uses the strong duality theorem [22] to convert an optimization into a feasibility problem. According to this theorem, any feasible point of a convex problem is optimal iff the primal objective at that point is the same as the dual. Therefore, the Primal-Dual rewrite contains a constraint that ensures the primal and dual objectives are equal in addition to the primal and the dual constraints. Fig. 6 (left) shows the Primal-Dual rewrite for the optimization in Fig. 3. This rewrite can generally result in non-linear constraints that impact the scalability.

To scale, we have developed a technique called *Quantized Primal-Dual* (QPD) that converts the Primal-Dual rewrite into a simpler problem (see Fig. 6). In QPD, we replace the input P with $\sum_{j=1}^{Q} L_j x_j$ where L_js are constants we choose a priori and x_js are binary variables. We require $\sum_j x_j \leq 1$, which means the outer problem has to pick one of the $Q+1$ values $(0, L_1, \ldots, L_Q)$ for P. We only need to quantize the leader's variables that appear in the multiplicative terms of



(a) Clustered Topology. (b) Clustered Demand Matrix.

(c) MetaOpt's Clustering Method.

FIGURE 7: Partitioning in MetaOpt. We first find the demands that maximize the gap between H′ and H in each cluster. We then fix the demands within each cluster and, one by one, find the demands between pairs of clusters that increase the gap.

the Primal-Dual rewrite. The inner problem is still optimal under this rewrite, but we trade off the optimality of the outer problem for speed by quantizing the input space.

Two challenges remain: (a) determining the number of quanta and (b) picking the values (L_j). Using more quanta leads to more integer variables, slowing down the solver. Using fewer quanta results in lower-quality adversaries as we will limit the input to only a few pre-selected values.

Since QPD rewrites are much faster to solve than the other rewrites, we can sweep multiple quanta choices and pick the best. We use the exact KKT rewrite on smaller problems to find good candidates. We observe empirically that adversarial inputs occur at so-called extreme points. For example, the worst-case demands have either 0 or the maximum possible value in POP[4]. Although we do not have a formal proof, we conjecture that the intuition behind these observations is similar to the intuition behind the simplex theorem [18].

We evaluate how these rewrites impact the scalability and investigate the approximation gap of QPD in §4.

3.5 Partitioning to Scale MetaOpt

We have found it necessary to use more aggressive scaling techniques to analyze problem instances at practical scales, such as TE heuristics for realistic topologies and demands. One such technique is *partitioning*. We can partition any

[4]For DP, the worst-case demands take values of 0, the demand pinning threshold T_d, or the maximum possible demand.

Topology	#Nodes	#Edges	#Part.	DP	POP
Cogentco	197	486	10	33.9%	20.76%
Uninett2010	74	202	8	28.4%	20.15%
Abilene [65]	10	26	–	12.69%	17.31%
B4 [39]	12	38	–	13.16%	17.89%
SWAN [38]	8	24	–	2.29%	22.08%

TABLE 3: Details of the topologies used in §4.1 and discovered gap.

problem, but we show the key steps in Fig. 7 for the TE heuristics where the problem has an intrinsic graph structure.

First, we partition nodes in the underlying network graph into clusters and solve the rewritten single-level optimization on each cluster in parallel. In this step, we only consider the intra-cluster demands (the diagonal pink blocks in Fig. 7).

We then freeze the demands from the last step and find the demands between pairs of clusters that worsen the gap by solving the rewritten problem on each pair. This step iteratively fills the blue blocks of the demand matrix in Fig. 7.

We can parallelize the second step between cluster pairs with little overlap and produce an overall demand by adding the values MetaOpt discovers after invoking each optimization. This method speeds up MetaOpt because each individual optimization, whether per cluster or per cluster pair, is much smaller than the overall problem.

We empirically find this partitioning approach consistently discovers inputs with large performance gaps. This is because more than one adversarial input exists, and our partitioning method does not bias against them. For example, the adversarial inputs for DP follow a common pattern where demands between distant nodes are just below the threshold. For such inputs, the heuristic wastes the capacity of many links by routing the demands on their shortest paths. In contrast, the optimal routing allocates those link capacities to multiple demands between nearby nodes. Our partitioning method still allows MetaOpt to find many inputs with this pattern.

4 Evaluation

We apply MetaOpt to traffic engineering, vector bin packing, and packet scheduling heuristics to show its generality. MetaOpt helped us quantify and understand the performance gaps of heuristics, prove theoretical properties, and design heuristics with lower performance gaps. Table 1 summarizes our findings. We also show the importance of our optimizations in MetaOpt and quantify its speed and scalability.

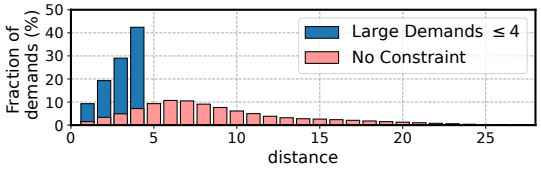
Implementation. Our prototype is in C# and uses Gurobi v9.5.2 [36]. We also have a port that uses Z3 [55]. To partition the graph, we adapt the previous code [3, 25] for spectral clustering [59] and FM partitioning [19, 24] and report results for different cluster numbers and clustering techniques.

4.1 Heuristics for WAN Traffic Engineering

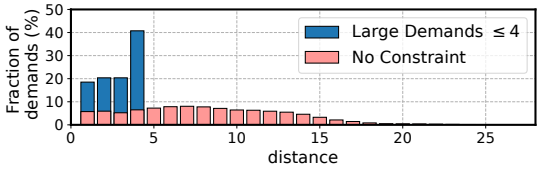
In this section, we (a) obtain performance gaps for DP and POP with respect to the optimal max-flow algorithm and (b) devise modified versions of these heuristics based on our analysis of their adversarial patterns.

Heu	Additional Constraints on $\mathcal{I}$	Density	Gap
DP	–	54.06%	33.9%
	locality (distance of large demands ≤ 4)	12.03%	33.4%
POP	–	16.14%	20.76%
	locality (distance of large demands ≤ 4)	4.74%	20.70%

(a) Impact of adding locality constraints on gap and density.



(b) Impact of adding locality constraints on DP



(c) Impact of adding locality constraints on POP

FIGURE 8: Using MetaOpt to find practical adversarial inputs on Cogentco. We can find more local and sparser adversarial inputs by constraining the input space.

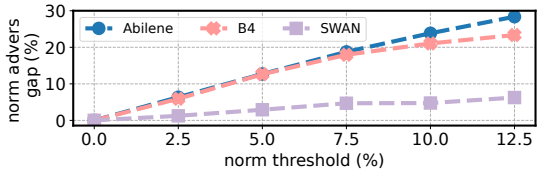
Experiment Setup. We use K-shortest paths [73] to find the paths between node pairs ($= 4$ if unspecified). We constrain the demands to be below a maximum value (half the average link capacity if unspecified) to ensure they are realistic and a single demand does not create a bottleneck. For DP, we vary the demand pinning threshold (=5% of average link capacity if unspecified). For POP, we vary the number of partitions (=2 if unspecified) and report the average gap over 5 random trials (see §A.3). We report runtimes on an AMD Opteron 2.4GHz CPU (6234) with 24 cores and 64GB of memory and use all available threads (unless mentioned otherwise). We timeout each optimization after 20 minutes.

Topologies. We use two large topologies from [1] and three public production topologies [38, 39, 65] (Table 3).

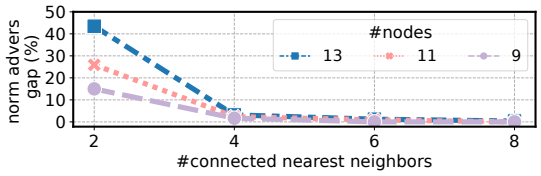
Metrics. We normalize the performance gap by the sum of the link capacities so we can compare across different scales.

Finding performance gaps. We compare DP and POP to the optimal max-flow on topologies that range from 8 to nearly 200 nodes (Table 3). We use the QPD rewrite (§3.4) and partitioning (§3.5) for most experiments but do not need the partitioning technique for small topologies (SWAN, B4, and Abilene). DP's performance gap ranges from 2% to over 33%. POP also exhibits a large performance gap (up to 22%).

These performance gaps are over any possible input demand. We can also use MetaOpt to obtain performance gaps on realistic inputs. Production demands are sparse and exhibit strong locality [3]. We can express these properties in MetaOpt through constraints on the input space ($\mathcal{I}$ in Equa-

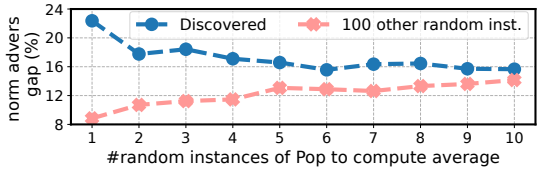


(a) Gap vs. the threshold value for DP.

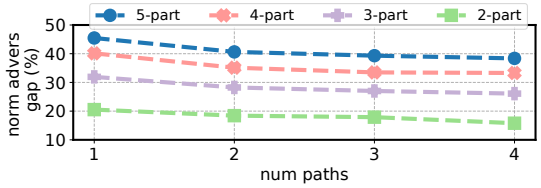


(b) Gap vs. connectivity.

FIGURE 9: DP's performance gap increases with the threshold and decreases with the connectivity.



(a) Gap vs. instances to approximate the expected value.

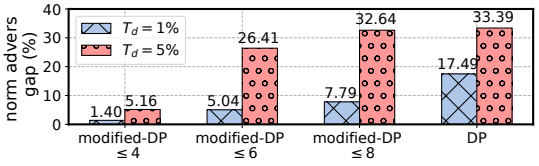


(b) Gap vs. #paths and #partitions for avg POP.

FIGURE 10: POP's performance gap when varying (a) #instances to approximate average and (b) #paths and #partitions.

Heuristic	Distance	Threshold wrt avg link cap
DP	–	0.1%
modified-DP	≤ 6	1% (10x)
	≤ 4	5% (50x)

(a) Maximum threshold such that discovered gap $\sim 5\%$.



(b) DP vs. modified-DP

FIGURE 11: We propose modified-DP based on insights from MetaOpt. It only pins small demands between near nodes, is more resilient, and pins more demands with the same gap.

tion 2). The gaps for DP and POP on those inputs are only slighly lower than in the unconstrained case, but we find adversarial demands that are sparser and more local (Fig. 8).

Designing heuristics with lower performance gap. We can use the adversarial inputs from MetaOpt to design new heuristics or explore whether we can combine heuristics to improve their gap. We first describe how we identified patterns in the adversarial inputs of DP and POP and then show how we used these patterns to improve these heuristics' performance gap.

Adversarial input patterns for DP. Intuitively, DP's performance gap increases as we increase its threshold since the heuristic forces more demands on their shortest path. Yet, the gap grows faster on some topologies even though they have roughly the same #nodes, #edges, and diameter.

We used synthetic topologies to study DP. To create each topology, we start with a ring graph and then connect each node to a varying number of its nearest neighbors. The results (Fig. 9(b)) indicate that the performance gap grows with the (average) shortest path length (fewer connections across nearest neighbors = longer shortest paths). Intuitively, if the shortest path lengths are longer on average, DP will use the capacity on more edges to route the small demands. This reduces the available capacity to route the rest of the demands.

Adversarial input patterns for POP. Since POP is a random heuristic, we search for inputs that maximize the expected gap. We approximate this expectation by an empirical average over n random partition samples. Then, we check whether the adversarial inputs can generalize by testing them on 100 other random instances. When MetaOpt uses a small number of samples to estimate the expected gap, the adversarial inputs overfit. We can improve its generalization by increasing the number of samples but at the cost of scalability. We find $n = 5$ permits scaling without overfitting (Fig. 10(a)).

POP's performance gap increases as we increase the number of paritions because each partition (1) gets a smaller fraction of each edge's capacity, and (2) has less information about the global state. We can reduce this gap by increasing the number of paths as it helps each partition better allocate the fragmented capacity.

Modified-DP. DP has a higher performance gap when nodes have a larger average distance. We use this insight to modify DP and design a heuristic with a lower gap. Modified-DP only pins demands that are (a) smaller than a threshold and (b) *between nodes less than k hops apart* (user specifies k). This heuristic routes small demands between distant nodes optimally, leaving more capacity for other demands. Its gap is $12.5\times$ smaller than DP for $T_d = 1\%$ and $k = 4$.

Increasing the distance threshold in modified-DP allows for better scalability by pinning more demands but at the cost of a higher performance gap. MetaOpt can help users adjust the parameter k based on their needs.

Modified-DP has another benefit. We can use a higher demand threshold (10 - $50\times$) and maintain the same gap as the original DP. We show this by using MetaOpt to compute the maximum threshold each method can admit while having a

max #balls	ball size granularity	FFD($\mathcal{I}_{\text{MetaOpt}}$)
20	0.01×	8
20	0.05×	7
14	0.01×	7

TABLE 4: MetaOpt finds slightly tighter bounds when constraining the number and size of balls. For OPT($\mathcal{I}$)=**6**, the tightest known theoretical bound for FFD [30] is **8**. This assumes the input can have unlimited balls of any size.

OPT($\mathcal{I}$)	MetaOpt		theoretical bound [60]	
	#balls	approx ratio	#balls	approx ratio
2	6	2.0	4	1.0
3	9	2.0	12	1.33
4	12	2.0	24	1.5
5	15	2.0	40	1.6

TABLE 5: MetaOpt finds adversarial examples with tighter approx. ratio for 2d-FFD than the best known theoretical bound [60].

gap $\leq 5\%$ (Fig. 11(a)). Operators can leverage this to pin more demands when small demands exhibit strong locality [3].

Meta-POP-DP. This meta-heuristic runs POP and DP in parallel and selects the best solution for each input. The two heuristics appear to have distinct adversarial inputs: DP underperforms when distant pairs have small demands, and POP when large demands that go through the same link end up in the same partition. We expected combining them would reduce the performance gap significantly compared to each one. But MetaOpt shows the new heuristic only improves the performance gap by 6% on the Cogentco topology. It finds inputs where small demands are between distant pairs (adversarial to DP) and large demands are between nearby nodes that end up in the same partition (adversarial to POP).

4.2 Heuristics for Vector Bin Packing

We use MetaOpt to (a) derive performance gaps that verify known results for FFD in one dimension and (b) prove a new property for FFD in 2 dimensions.

Finding performance gaps. Performance in FFD is measured by the number of bins required to fit a given number of balls. After decades of theoretical studies on 1d-FFD [14, 30, 43, 52], the work in [30] established the tight bound $\text{FFD}(\mathcal{I}) \leq \frac{11}{9}\text{H}'(\mathcal{I}) + \frac{6}{9}$ for any $\mathcal{I}$ (H′ is the optimal). To prove tightness, the authors craft a careful example where $\text{H}'(\mathcal{I}) = 6$ bins and $\text{FFD}(\mathcal{I}) = 8$. MetaOpt found the same example when we constrained its inputs to $\text{H}'(\mathcal{I}) = 6$ and proved FFD needs 8 bins in the worst-case.

[30] assumes (1) $\mathcal{I}$ can have an unlimited number of balls and (2) the balls in $\mathcal{I}$ can have any size (even $0.00001cm^3$!). However, when packing jobs (balls) in machines (bins) [68], we often know *a priori* an upper bound on the number of jobs or the quantization levels for resource requirements. We can incorporate such constraints and ensure MetaOpt finds practical performance gaps. As Table 4 shows, when we constrain the number of balls and the ball sizes, MetaOpt finds adversarial inputs that produce a tighter bound compared to [30].

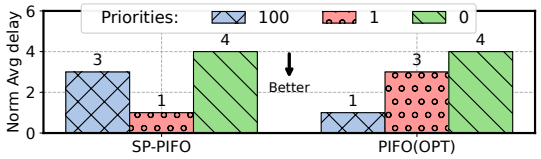


FIGURE 12: SP-PIFO can delay the highest priority packet (rank = 0) by 3×. We show the average delay of packets with the same priority normalized by the average delay of the highest-priority packets in PIFO. We assume packets have priorities between 0 - 100, and the queues can admit all the packets (similar to SP-PIFO). When 10K packets arrive at the same time, and the queues drain at 40 Gbps, the average delay for the highest priority packets in PIFO is $0.74ms$ (the performance gap in this figure is independent of the number of packets).

MetaOpt max objective	#priority inversions	
	SP-PIFO [5]	AIFO [74]
AIFO() − SP-PIFO()	6	37
SP-PIFO() − AIFO()	24	11

TABLE 6: Using MetaOpt to compare two heuristics. We show the number of priority inversions on an 18-packet trace from MetaOpt. The total queue size is 12 and SP-PIFO has 4 queues.

Proving properties. While multi-dimensional FFD is widely used in practice [35, 37, 53], its theoretical guarantees are less well understood. Recently, [60] crafted an example where 2-dimensional FFDSum uses α times more bins than the optimal. $\alpha \in [1, 2)$ and $\alpha \to 2$ as the optimal tends to infinity. In other words, α is strictly less than 2 for a finite size problem.

MetaOpt found adversarial inputs with $\alpha = 2$ *for every problem size we considered* (Table 5). For example, when the optimal uses 4 bins, MetaOpt finds an adversarial input with 12 balls causing FFDSum to use 8 bins. In constrast, [60] uses 24 balls and only achieves approximation ratio of 1.5.

We studied the adversarial inputs from MetaOpt and proved the following theorem, establishing an approximation ratio of at least 2 for FFDSum. §B.2 contains the detailed proof.

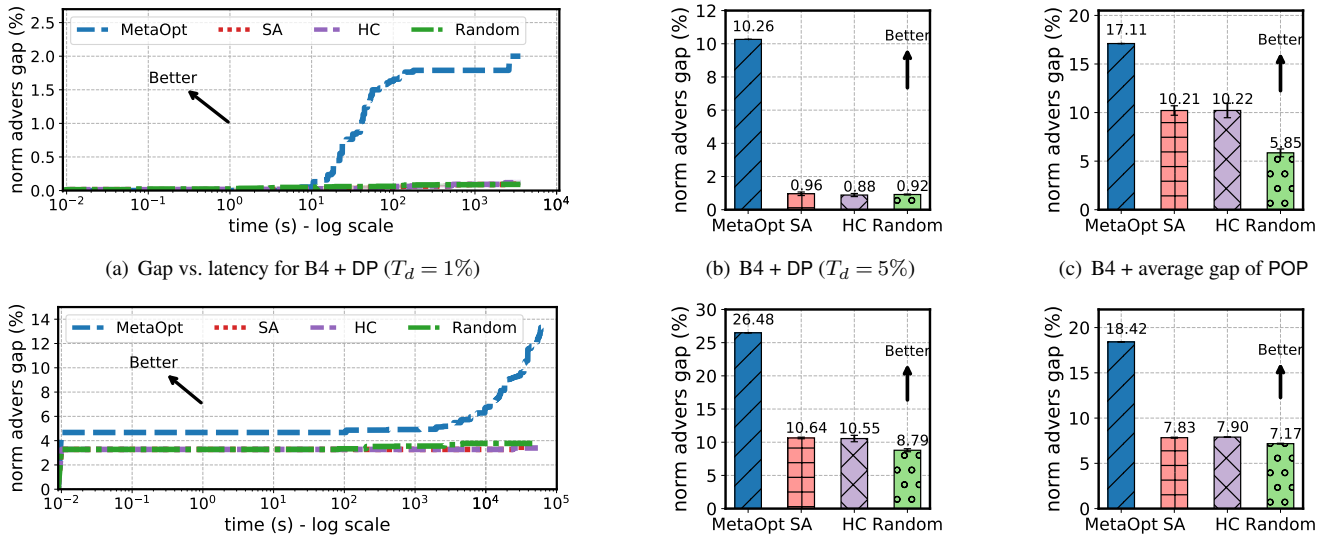
Theorem 1. *In 2-dimensional VBP, for any $k > 1$, there exists an input $\mathcal{I}$ with $OPT(\mathcal{I}) = k$ and $FFDSum(\mathcal{I}) \geq 2k$.*

4.3 Heuristics for Packet Scheduling

We use MetaOpt to compare the performance of SP-PIFO with both optimal (PIFO [64]) and another heuristic (AIFO [74]). Unlike SP-PIFO, AIFO uses a single queue but adds admission control based on packet priorities to approximate PIFO.

Finding performance gaps. We use the average delay of packets weighted by their priorities to compare PIFO and SP-PIFO. MetaOpt discovers packet traces where SP-PIFO fails to prioritize packets correctly and incurs 3× higher delays for high-priority packets than PIFO (Fig. 12).

We also use MetaOpt to compare SP-PIFO and AIFO. Unlike SP-PIFO, AIFO is designed for shallow-buffered switches, and its admission control can drop packets. For a fair compar-



(a) Gap vs. latency for B4 + DP ($T_d = 1\%$)

(b) B4 + DP ($T_d = 5\%$)

(c) B4 + average gap of POP

(d) Gap vs. latency for Cogentco + DP ($T_d = 1\%$)

(e) Cogentco + DP ($T_d = 5\%$)

(f) Cogentco + average gap of POP

FIGURE 13: MetaOpt is faster and finds larger gaps between OPT and POP or DP on Cogentco and B4. We report the gap relative to the total capacity and use only one thread to run each method for fair comparison (SA = Simulated Annealing, HC = Hill Climbing).

ison, we assume both heuristics use the same switch buffer size, and we split the buffer evenly across SP-PIFO queues. With limited buffers, these algorithms may drop packets, so we need to consider the impact of their respective drop rates when comparing their performance. We borrow a metric from SP-PIFO: we count k priority inversions when a packet is inserted in a queue after k lower priority packets (even if the queue is full and the packet would have been dropped). We found (Table 6):

AIFO sometimes outperforms SP-PIFO because (1) it has one large queue instead of n smaller ones – SP-PIFO drops many packets when faced with a burst of packets with the same priority since it assigns them to a single smaller queue; and (2) SP-PIFO lacks admission control – we can create an adversarial pattern where lower-priority packets arrive right before a group of high-priority ones to make SP-PIFO admit the lower-priority packets and drop the higher-priority ones.

But SP-PIFO also sometimes outperforms AIFO because (1) AIFO lacks a sorting mechanism, which can cause high-priority packets to get delayed behind lower-priority ones, and (2) AIFO depends on an estimate of the distribution based on the most recent window of packets. MetaOpt found traces in which a few packets with entirely different priorities compared to others can disrupt AIFO's distribution estimate!

Proving properties. MetaOpt shows that SP-PIFO's adversarial inputs exhibit significant priority inversions. We used these inputs to prove a lower bound on the worst-case performance gap, in terms of priority-weighted average delay, between SP-PIFO and PIFO (see §C.3):

Theorem 2. *For any number of packets $N \geq 1$, integer priorities between $0 - R_{max}$ and $q \geq 2$ queues, there exists a sequence of packets $\mathcal{I}$ where the difference between the weighted average packet delay that results from SP-PIFO is*

$$(R_{max} - 1)(N - 1 - p)p \quad \textit{where} \quad p = \lceil (N-1)/2 \rceil \qquad (3)$$

worse compared to PIFO.

Designing better heuristics. SP-PIFO uses the same set of queues to schedule packets with a wide range of priorities. We found it underperforms when the difference between packet priorities is large. Theorem 2 also confirms this, as the gap is proportional to R_{max}.

We evaluated a Modified-SP-PIFO, in which we limited the range of packet priorities that can compete with each other: we formed m queue groups where each group served a fixed priority range. SP-PIFO runs on queues within a group. This modified version can reduce the gap of SP-PIFO by $2.5\times$.

4.4 Evaluating MetaOpt

We show MetaOpt can find solutions faster than other baseline search methods and helps users describe H′ and H in a compact way. We also show how our various design choices help.

How fast MetaOpt discovers a performance gap. Alternatives to MetaOpt include random search, hill climbing [29], and simulated annealing [45]. Random search repeatedly picks new random inputs and returns the one with a maximum gap. Hill climbing (HC) and simulated annealing (SA) use information from past observed inputs to guide the search (more details in §E).

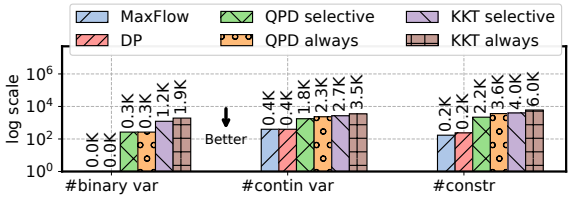


FIGURE 14: Users specify DP and OPT in MetaOpt. We show the complexity of these specifications and the rewrites in terms of the number of variables and constraints (see Fig. A.2 for POP).

MetaOpt finds 1.7× - 17× larger gaps than the next best (Fig. 13). The baselines fail since they ignore the heuristic's details and treat it as a black box. MetaOpt uses its knowledge of the heuristic and the topology to guide the search.

MetaOpt is the only method that consistently discovers substantially larger gaps over time, while other techniques get stuck in local optima and improve only slightly even after many hours (Fig. 13(a), 13(d)).

Input and rewrite complexity. Users specify H′ and H, and MetaOpt automatically applies selective rewrites (§3.3) to scale better. We evaluate how complex these specifications and rewrites are regarding the number of binary variables, continuous variables, and constraints (Fig. 14). In general, solvers perform better if these quantities are lower.

We use DP as an example to highlight three features in MetaOpt's design (Fig. 14). The three metrics show the user's inputs are more compact than the rewritten optimization; they have a fifth of the constraints and half the number of continuous variables. This quantifies how MetaOpt's automatic rewrites can reduce the user's burden. Selective rewrites are important: we can use them to reduce the number of constraints (2.2K vs. 3.6K for QPD) and continuous variables (1.8K vs. 2.3K for QPD) compared to when we always rewrite the bi-level optimization. We can produce more compact specifications (with fewer variables and constraints) through QPD compared to KKT, even with selective rewrites. This explains why it helps MetaOpt scale.

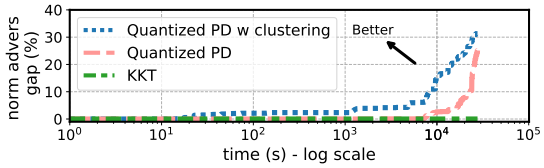
The impact of partitioning. MetaOpt partitions the problem to find larger gaps faster than both (non-partitioned) quantized primal-dual and KKT, even on medium-sized topologies (Fig. 15(a)). For larger topologies, KKT and primal-dual cannot find large gaps without partitioning.

As we increase the number of partitions, MetaOpt scales better and finds larger gaps until it eventually plateaus (10 for Cogentco in Fig. 15(b)). We can slightly improve the gap if we double the solver timeout.

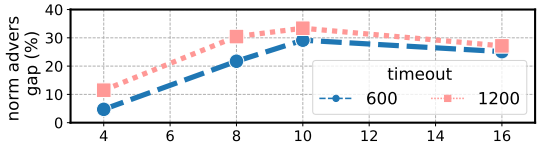
The inter-cluster step in partitioning is important, especially for heuristics that underperform when demands are between distant nodes (DP in Fig. 15(c)). The partitioning algorithm also impacts the discovered gap (Fig. 15(d)).

The impact of quantization. To quantify, we compare the relative difference between the gap from quantized primal-dual and KKT (which does not use quantization). We found quantization has little impact on solution quality: 4% for DP and 0 for POP on B4 (we can not scale KKT to larger topologies). For POP, we use two quantiles: 0 and the max demand. If a demand d in an uncongested partition falls in between these values, forcing the demand to zero cannot decrease the gap: the rewrite's throughput would drop by d, and the optimal throughput by some value between 0 and d. A similar argument applies to the congested case.

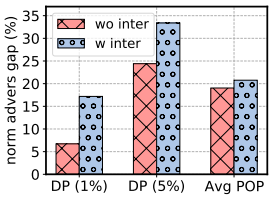
We use three quantiles for DP: 0, the threshold, and the max demand. For a high enough threshold, quantized primal-dual may avoid assigning the threshold value to demands between distant nodes to not violate capacity constraints, whereas KKT can assign any value, which causes the relative difference in solution quality.



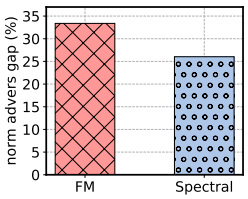
(a) KKT vs Quantized primal-dual vs w partitioning on Uninett2010



(b) Number of partitions and solver timeout on Cogentco



(c) Inter-cluster on Cogentco



(d) Graph partitioning on Cogentco

FIGURE 15: Partitioning helps MetaOpt find larger gaps faster.

5 Discussion

We have shown MetaOpt applies to various heuristics in multiple domains. We have developed a simple partitioning technique to scale MetaOpt to large problem instances (with partitioning, it focuses on finding large performance gaps instead of the worst-case). We defer the following to future work:

Application to a broader set of heuristics. MetaOpt only applies to heuristics, which we can model as convex or feasibility problems. A number of heuristics in systems and networking do not fit in either of these two categories (e.g., mixed integer *maximizations or minimizations* [76] that we cannot cast as a feasibility problem). Our approach in MetaOpt has roots in the theory of Stackelberg games and goes beyond bi-level

optimization. We discuss how the theory in this space can help extend MetaOpt's scope in §6.

Finding adversarial settings. The set up (*e.g.,* topology in DP) influences the performance gap as well. Users can model this aspect as part of MetaOpt and find adversarial problem settings. This may impact MetaOpt's scalability.

Ease of use. In MetaOpt, users have to write their heuristics in the optimization language, which requires expertise. MetaOpt provides a set of helper functions to simplify the process. However, there is room for improvement in enabling users to model their heuristics and extending MetaOpt to *explain why* a heuristic underperforms automatically.

6 Extending MetaOpt's scope

We observe the dynamics of the problem MetaOpt addresses resembles leader-follower games (Stackelberg equilibria [49]). In such games, a leader maximizes their payoff and controls the inputs to one or more followers. With this *fixed* input from the leader, the followers have to respond and optimize for their own payoff and decide outputs, which the leader does not control but influences their payoff.

These games apply to a variety of leader-follower combinations (*e.g.,* optimization-based and Bayesian) and there are various techniques to find the equilibrium in such games [21, 44, 70, 75]. Hence, in theory, we can use these techniques to analyze the performance gaps of a broader class of heuristics than MetaOpt currently supports, as long as we have a leader-follower combination where we know how to compute the equilibrium. This is future work.

7 Related Work

To the best of our knowledge, no prior work finds *provable* adversarial inputs for heuristics that approximate optimal problems. Our techniques (e.g., big-M, convex rewrites, and generally translating the problem to one that is amenable to off-the-shelf solvers) are not per-se novel [8, 26, 31], and prior work in networking have used some of these theories [6, 11, 12, 42]. However, no other work has combined them in this way. We also extend them to randomized, conditional, and sequential non-convex heuristics. Without our changes, we could not apply existing solvers directly or quickly find large gaps.

Our qualitative results – the optimality gap and hard examples for POP, DP, FFD, SP-PIFO, and AIFO – are novel. We also find and prove tighter bounds for the optimality gap of FFD and SP-PIFO.

Our work is different from most prior techniques. Traditional algorithmic worst- or average-case analyses [27, 54] are specific to an individual heuristic and must be applied case by case. We cannot do such analyses for some heuristics as they only find loose bounds or do not account for realistic input constraints (none exist for DP, POP, or [63, 76]).

Verification methods seek inputs that violate an *statically-specified safety or correctness* invariant on a given function [41]. In contrast, we look for inputs that maximize the *performance* gap.

Model checking approaches based on SMT solvers [4, 7, 9, 10, 34] can search for adversarial inputs that result in performance gaps greater than a fixed bound when users can encode both the optimal and heuristic as pure feasibility problems. However, these approaches cannot handle bi-level optimization, where the optimal or heuristic must be formulated as optimizations (*e.g.,* traffic engineering).

Local search algorithms [29, 45] apply to any (potentially black-box) heuristic or optimal algorithm. However, the flip side of such generality is that they are slow on large input spaces, get stuck in local optima, and fail to find practical inputs because they ignore the inner workings of the heuristic.

Recent work finds malicious inputs to learned techniques [33, 51]. However, none of these find provably large gaps or even consider the optimal algorithm. Other broadly related work include [13, 28, 50, 61].

Our partitioning approach is different from [3, 58]. NCFlow and POP need to return a *feasible solution* given an input (*e.g.,* one that respects capacity and path constraints). As a result, they have to combine the solutions from all partitions to ensure their feasibility. This makes them complex when they have to enforce global constraints. For example, POP [58] sacrifices quality and partitions demands separately to ensure the sub-problems enforce rigorous constraints. MetaOpt does not need such constraints because it generates an input for the problem, not the solution. Notice a certain 'coming full circle' aspect here: we use a similar (but not the same) partitioning to analyze the optimality gap of POP quickly.

This paper is an extended version of [56]. Compared to this workshop paper, we changed the methodology to improve generality and scalability, added helper functions for ease of use, added support for heuristics from VBP and packet scheduling, and did a more extensive evaluation.

8 Conclusion

MetaOpt is a heuristic analyzer for heuristics that can be posed as an optimization or a feasibility problem. It can be used to find performance gaps at scale, prove lower bounds on worst-case gaps, and devise improvements to heuristics. At its core, MetaOpt solves a bi-level optimization problem. To do this efficiently, it selectively rewrites heuristic specifications as a single-level optimization, and incorporates several scaling techniques. Future work can include using it to evaluate and improve other heuristics, increasing its expressivity (§6), and identifying infeasible inputs instead of adversarial ones.

Acknowledgments. We thank our shepherd, Marco Chiesa, and the anonymous reviewers for their insightful comments. We also thank Siva Kakarla, Rodrigo Fonseca, Jeff Mogul, Jay Lorch, and Daniel Berger for their helpful feedback. This material is based upon work supported in part by the U.S. National Science Foundation under grant No. CNS-1901523.

References

[1] Internet Topology Zoo. http://www.topology-zoo.org/.

[2] Yarn resource allocation of multiple resource-types. https://bit.ly/3YMDL2Z.

[3] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. Contracting wide-area network topologies to solve flow problems quickly. In *NSDI*, 2021.

[4] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. Automating network heuristic design and analysis. In *HotNets*, 2022.

[5] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. {SP-PIFO}: Approximating {Push-In}{First-Out} behaviors using {Strict-Priority} queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 59–76, 2020.

[6] David Applegate and Edith Cohen. Making intra-domain routing robust to changing and uncertain traffic demands: Understanding fundamental tradeoffs. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 313–324, 2003.

[7] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. Formal methods for network performance analysis. In *NSDI*, 2023.

[8] Bryan Arguello, Richard L. Chen, William E. Hart, John D. Siirola, and Jean-Paul Watson. Modeling bilevel program in pyomo. https://www.osti.gov/servlets/purl/1526125.

[9] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. Starvation in end-to-end congestion control. In *SIGCOMM*, 2022.

[10] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. Toward formally verifying congestion control behavior. In *SIGCOMM*, 2021.

[11] Behnaz Arzani, Alexander Gurney, Bo Li, Xianglong Han, Roch Guerin, and Boon Thau Loo. Fixroute: A unified logic and numerical tool for provably safe internet traffic engineering. *arXiv preprint arXiv:1511.08791*, 2015.

[12] Behnaz Arzani, Nicholas Iodice, Steven Hwang, Prahalad Venkataramanan, Roch Geurin, and Boon Thau Loo. Sunstar: A cost-effective multi-server solution for reliable video delivery. *arXiv preprint arXiv:1812.00109*, 2018.

[13] Nirav Atre, Hugo Sadok, Erica Chiang, Weina Wang, and Justine Sherry. Surgeprotector: Mitigating temporal algorithmic complexity attacks using adversarial scheduling. In *SIGCOMM*, 2022.

[14] Brenda S Baker. A new proof for the first-fit decreasing bin-packing algorithm. *Journal of Algorithms*, 1985.

[15] Y. Beck and M. Schmidt. A Gentle and Incomplete Introduction to Bilevel Optimization. https://optimization-online.org/?p=17182, July 2023.

[16] Ryan Beckett and Ratul Mahajan. A general framework for compositional network modeling. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 8–15, 2020.

[17] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Englewood Cliffs, 1992.

[18] Dimitris Bertsimas and John N Tsitsiklis. *Introduction to linear optimization*, volume 6. Athena Scientific Belmont, MA, 1997.

[19] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks, 2008.

[20] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. TEAVAR: striking the right utilization-availability balance in WAN traffic engineering. In *SIGCOMM*, 2019.

[21] Branislav Bosansky and Jiri Cermak. Sequence-form algorithm for computing stackelberg equilibria in extensive-form games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.

[22] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[23] Wu chang Feng, Francis Chang, Wu chi Feng, and Jonathan Walpole. Provisioning on-line games: A traffic analysis of a busy counter-strike server.

[24] A. Clauset, M.E.J. Newman, and C. Moore. Finding community structure in very large networks. *Phys. Rev.*, 2004.

[25] Aaron Clauset. Fast Modularity Community Structure Inference Algorithm. https://bit.ly/3aAVGQH.

[26] Benoît Colson, Patrice Marcotte, and Gilles Savard. Bilevel programming: A survey. *4OR*, 2005.

[27] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[28] Anthony Corso, Robert Moss, Mark Koren, Ritchie Lee, and Mykel Kochenderfer. A survey of algorithms for black-box safety validation of cyber-physical systems. *Journal of AI Research*, 2021.

[29] L. Davis. Bit-climbing, representational bias, and test suit design. *Proc. Intl. Conf. Genetic Algorithm*, pages 18–23, 1991.

[30] György Dósa. The tight bound of first fit decreasing bin-packing algorithm is $ffd(i) \leq 11/9opt(i) + 6/9$. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies: First International Symposium, ESCAPE*. Springer, 2007.

[31] Pablo Garcia-Herreros, Lei Zhang, Pratik Misra, Erdem Arslan, Sanjay Mehta, and Ignacio E Grossmann. Mixed-integer bilevel optimization for capacity planning with rational markets. *Computers & Chemical Engineering*, 86:33–47, 2016.

[32] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[33] Tomer Gilad, Nathan H. Jay, Michael Shnaiderman, Brighten Godfrey, and Michael Schapira. Robustifying network protocols with adversarial examples. In *HotNets*. ACM, 2019.

[34] Saksham Goel, Benjamin Mikek, Jehad Aly, Venkat Arun, Ahmed Saeed, and Aditya Akella. Quantitative verification of scheduling heuristics. *arXiv preprint arXiv:2301.04205*, 2023.

[35] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[36] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022.

[37] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: Vm allocation service at scale. In *OSDI*, 2020.

[38] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, 2013.

[39] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, and Min Zhu. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, 2013.

[40] Virajith Jalaparti, Ivan Bliznets, Srikanth Kandula, Brendan Lucier, and Ishai Menache. Dynamic pricing and traffic engineering for timely inter-datacenter transfers. In *SIGCOMM*, 2016.

[41] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41:21:1–21:54, 2009.

[42] Chuan Jiang, Sanjay Rao, and Mohit Tawarmalani. Pcf: provably resilient flexible routing. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 139–153, 2020.

[43] David S Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[44] Jan Karwowski, Jacek Mańdziuk, and Adam Żychowski. Sequential stackelberg games with bounded rationality. *Applied Soft Computing*, 132:109846, 2023.

[45] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[46] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. Decentralized cloud wide-area network traffic engineering with BLASTSHIELD. In *NSDI*, 2022.

[47] Alok Kumar et al. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *SIGCOMM*, 2015.

[48] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms, 2022.

[49] Tao Li and Suresh P Sethi. A review of dynamic stackelberg game models. *Discrete & Continuous Dynamical Systems-B*, 22(1):125, 2017.

[50] Zinan Lin, Hao Liang, Giulia Fanti, and Vyas Sekar. Raregan: Generating samples for rare classes. *arXiv preprint arXiv:2203.10674*, 2022.

[51] Roland Meier, Thomas Holterbach, Stephan Keck, Matthias Stähli, Vincent Lenders, Ankit Singla, and Laurent Vanbever. (self) driving under the influence: Intoxicating adversarial network inputs. In *HotNets*. ACM, 2019.

[52] Yue Minyi. A simple proof of the inequality $ffd(l) \leq 11/9opt(l) + 1$, $\forall l$, for the ffd bin-packing algorithm. *Acta Mathematicae Applicatae Sinca*, 1991.

[53] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking beyond GPUs for DNN scheduling on Multi-Tenant clusters. In *OSDI*, 2022.

[54] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge university press, 1995.

[55] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[56] Pooria Namyar, Behnaz Arzani, Ryan Beckett, Santiago Segarra, Himanshu Raj, and Srikanth Kandula. Minding the gap between fast heuristics and their optimal counterparts. In *HotNets*, 2022.

[57] Pooria Namyar, Behnaz Arzani, Srikanth Kandula, Santiago Segarra, Daniel Crankshaw, Umesh Krishnaswamy, Ramesh Govindan, and Himanshu Raj. Solving Max-Min Fair Resource Allocations Quickly on Large Graphs. In *NSDI*, 2024.

[58] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving large-scale granular resource allocation problems efficiently with POP. In *SOSP*, 2021.

[59] Andrew Y Ng, Michael I Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *NIPS*, 2002.

[60] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. January 2011.

[61] Pedro Reviriego and Daniel Ting. Breaking cuckoo hash: Black box attacks. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[62] Thomas Sauerwald. Sorting networks. https://www.cl.cam.ac.uk/teaching/1415/AdvAlgo/advalg.pdf.

[63] Rachee Singh, Nikolaj Bjorner, Sharon Shoham, Yawei Yin, John Arnold, and Jamie Gaudette. Cost-effective capacity provisioning in wide area networks with Shoofly. In *SIGCOMM*, 2021.

[64] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 44–57, New York, NY, USA, 2016. Association for Computing Machinery.

[65] Stanford University IT. Abilene core topology, 2015.

[66] Mark Stillwell, David Schanzenbach, Frédéric Vivien, and Henri Casanova. Resource allocation algorithms for virtualized service hosting platforms. *Journal of Parallel and Distributed Computing*, 2010.

[67] Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici. A scalable application placement controller for enterprise data centers. In *WWW*, 2007.

[68] Abhishek Verma, Madhukar Korupolu, and John Wilkes. Evaluating job packing in warehouse-scale computing. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, 2014.

[69] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *EuroSys*, 2015.

[70] Jiali Wang, He Chen, Rujun Jiang, Xudong Li, and Zihao Li. Fast algorithms for stackelberg prediction game with least squares loss. In *International Conference on Machine Learning*, pages 10708–10716. PMLR, 2021.

[71] Gerhard J. Woeginger. There is no asymptotic ptas for two-dimensional vector packing. *Inf. Process. Lett.*, 1998.

[72] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, 2007.

[73] Jin Y. Yen. Finding the K Shortest Loopless Paths in a Network. *Management Science*, 17(11):712–716, 1971.

[74] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 179–193, New York, NY, USA, 2021. Association for Computing Machinery.

[75] Yunxiao Zhang and Pasquale Malacaria. Bayesian stackelberg games for cyber-security decision support. *Decision Support Systems*, 148:113599, 2021.

[76] Zhizhen Zhong, Manya Ghobadi, Alaa Khaddaj, Jonathan Leach, Yiting Xia, and Ying Zhang. Arrow: Restoration-aware traffic engineering. In *SIGCOMM*, 2021.

Term	Meaning
$\mathcal{V}, \mathcal{E}, \mathcal{D}, \mathcal{P}$	Sets of nodes, edges, demands, and paths
N, M, K	Number of nodes, edges, and demands, i.e., $N = \lvert\mathcal{V}\rvert, M = \lvert\mathcal{E}\rvert, K = \lvert\mathcal{D}\rvert$
c_e, p	c_e: capacity of edge $e \in \mathcal{E}$ path p: set of connected edges
(s_k, t_k, d_k)	The kth element in $\mathcal{D}$ has source and target nodes ($s_k, t_k \in \mathcal{V}$) and a non-negative volume (d_k)
$\mathbf{f}, f_k^p$	$\mathbf{f}$: flow assignment vector with elements f_k f_k^p: flow for demand k on path p

TABLE A.1: Multi-commodity flow problems' notation.

A Details of Traffic Engineering

Table A.1 summarizes our notation.

A.1 Multi-commodity flow problem

The optimal form of WAN-TE typically involves solving a multi-commodity flow problem. Given a set of nodes, capacitated edges, demands, and pre-chosen paths per demand, a flow allocation is feasible if it satisfies demand and capacity constraints. The goal is to find a feasible flow to optimize a given objective (e.g., total flow [3], max-min fairness [38,39,57], or utility curves [47]). We define the feasible flow over a pre-configured set of paths as (see Table A.1)

$$\begin{aligned}
&\mathsf{FeasibleFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq \{\mathbf{f} \mid && && && (4)\\
&f_k = \sum_{p\in\mathcal{P}_k} f_k^p, && \forall k \in \mathcal{D} && \text{(flow for demand } k\text{)}\\
&f_k \le d_k, && \forall k \in \mathcal{D} && \text{(flow below volume)}\\
&\sum_{k,p\mid p\in\mathcal{P}_k, e\in p} f_k^p \le c_e, && \forall e \in \mathcal{E} && \text{(flow below capacity)}\\
&f_k^p \ge 0 && \forall p \in \mathcal{P}, k \in \mathcal{D} && \text{(non-negative flow)}\}
\end{aligned}$$

Among all the feasible flows, the optimal solution seeks to maximize the total flow across the network:

$$\begin{aligned}
\mathsf{OptMaxFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq &\arg\max_{\mathbf{f}} \sum_{k\in\mathcal{D}} f_k && (5)\\
&\text{s.t.} \quad \mathbf{f} \in \mathsf{FeasibleFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}).
\end{aligned}$$

A.2 More details on DP and POP heuristics

Demand Pinning (DP) [46]. First, it routes all demands at or below a predefined threshold T_d through their shortest path. It then jointly routes the rest of the demands optimally over multiple paths:

$$\mathsf{DemandPinning}(\mathcal{D},\mathcal{P}) \triangleq \{\mathbf{f} \mid f_k^p = \begin{cases} d_k & \text{if } p \text{ is shortest path in } P_k \\ 0 & \text{otherwise} \end{cases}, \forall k \in \mathcal{D} : d_k \le T_d\}, \quad (6)$$

We can pose DP as an optimization with constraints that route demands below the threshold on the shortest paths:

$$\begin{aligned}
\mathsf{DemPinMaxFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq &\arg\max_{\mathbf{f}} \sum_{k\in\mathcal{D}} f_k && (7)\\
&\text{s.t.} \quad \mathbf{f} \in \mathsf{FeasibleFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P})\\
&\quad\ \ \mathbf{f} \in \mathsf{DemandPinning}(\mathcal{D},\mathcal{P})
\end{aligned}$$

Partitioned Optimization Problems (POP). [58] POP divides node pairs (and their demands) uniformly at random into partitions, assigns each partition an even share of edge capacities, and solves the original problem (*e.g.,* the SWAN optimization [38]) in parallel, once per partition.

$$\mathsf{POPMaxFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq \bigcup_{\text{part. c}} \mathsf{OptMaxFlow}(\mathcal{V},\mathcal{E}_c,\mathcal{D}_c,\mathcal{P}), \quad (8)$$

where $\cup$ is the vector union, the per-partition demands $\mathcal{D}_c$ are disjoint subsets of the actual demands drawn uniformly at random, and the per-partition edge list $\mathcal{E}_c$ matches the original edges but with proportionally smaller capacities.

A.3 Formulation of DP and POP

Demand Pinning formulation for quantized demands. DP has conditional or *if* clauses: if a demand is smaller than the threshold T_d, then route it over its shortest path; otherwise, use the optimal algorithm to route it.

We describe the shortest path for demand k using $\widehat{p}_k$ and write the *if* clause as:

$$\widehat{f_k^{p_k}} \ge \sum_{q=1}^{Q} \mathbb{1}\left[L_q \le T_d\right] L_q x_q^k, \ \forall k \in \mathcal{D}. \quad (9)$$

The $\{0, L_1, \ldots, L_Q\}$ are the quantas and x_q^ks are binary variables that pick which quanta is active for demand k. The term $\mathbb{1}[L_q \le T_d]$ is constant at runtime and only shows which terms exist in the sum.

Note that $d_k = \sum_q L_q x_q^k$ by the definition of quantization. Thus, if the demand d_k is smaller than the threshold T_d, Equation 9 will ensure that the allocation on the shortest path is equal to d_k.[5]

Demand Pinning big-M formulation. We can also encode DP using the standard big-M approach from optimization theory. This formulation does not require quantized demands and is useful for the KKT rewrite. However, big-M can cause numerical instability at scale.

[5]When $d_k \le T_d$, Equation 9 effectively becomes $\widehat{f_k^{p_k}} \ge d_k$. In the converse case ($d_k > T_d$), Equation 9 becomes $\widehat{f_k^{p_k}} \ge 0$ (a no-op).

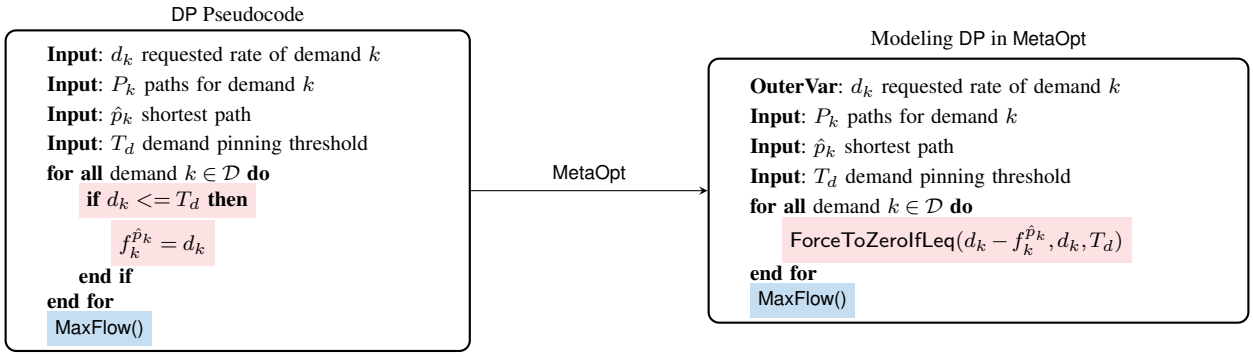


FIGURE A.1: The pseudocode for DP and how users can model it in MetaOpt using the helper functions.

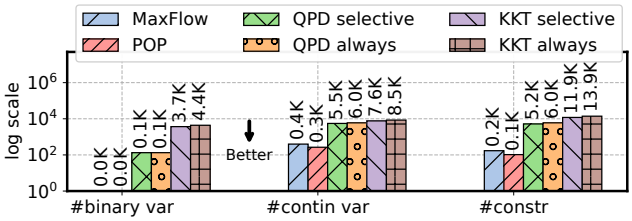


FIGURE A.2: The complexity of user's input for POP and the subsequent rewrites in terms of the size of the optimization.

$$\sum_{p \in P_k,\, p \neq \widehat{p_k}} f_k^p \leq \max\left(M(d_k - T_d), 0\right),\ \forall k \in \mathcal{D},$$

$$d_k - f_k^{\widehat{p_k}} \leq \max\left(M(d_k - T_d), 0\right),\ \forall k \in \mathcal{D},$$

where M is a large pre-specified constant. Notice that whenever demand d_k is below the threshold T_d, the constraints allocate zero flow on all but the (default) shortest path — DP routes the *full demand* on the default path in such cases. We can use standard optimization theory to convert the max in these constraints into a set of linear constraints [22] (this requires us modifying the objective but does not impact the final solution's quality).

Partitioned Optimization Problems Formulation. POP is convex as it is the union of solutions to disjoint linear optimizations (Equation 8). It is hard to encode POP as it uses random partitions, which makes POP ($\mathcal{I}$) a random variable in the leader problem, but MetaOpt needs a deterministic representation of the heuristic. We can consider a specific instance of each random variable (*e.g.,* one random assignment of demands to partitions). However, it will overfit to that instance and not reflect POP's true behavior.

We use the *expected value* or *tail percentile* of the gap from multiple random trials. To compute the average, we replace H in Equation 2 with its expected value and approximate the expectation through empirical averages over a few randomly generated partitions (see §4.1). To find the tail, we use a sorting network [40, 62] to compute the desired percentile across multiple random trials.

In addition, we encode an advanced version of POP in §A.4 that splits large demands across multiple partitions instead of assigning each demand to one partition.

A.4 POP Client Splitting

In §2, we introduce the (basic) POP heuristic [58], which incorporates *resource splitting* for our WAN TE problem, and in §A.3, we present POP as a convex optimization. The work in [58] also specifies an extended full-fleshed version of POP that incorporates "*client splitting*". We next show how to express this variant as a convex optimization problem.

We can think of POP client splitting as an operation that takes in a set of demands $\mathcal{D}$ and returns a modified set $\mathcal{D}_{\text{cs}} = \text{ClientSplit}(\mathcal{D})$ that can then be input into POP as in (8). The function ClientSplit() generates several duplicates of the existing demands and reduces their volume in proportion. It performs several operations where it replaces $(s_k, t_k, d_k) \in \mathcal{D}$ with two elements of the form $(s_k, t_k, d_k/2)$. It iterates and repeats this operation until it terminates (see [58]).

We encode a version of client splitting where we split an element in $\mathcal{D}$ if its demand value d_k is larger than or equal to a threshold d_{th}, and we keep splitting it until either we get to a predefined number of maximum splits (of the original demand[6]) or the split demand is lower than d_{th}.

Without loss of generality, we describe this idea for a single demand d_1: we can replicate this process for all demands in $\mathcal{D}$. Take for example the scenario where we split the demand at most twice (which creates at most 4 virtual clients). We a-priori encode all the flow variables for all possible splits of d_1: we use seven variables of the form $f_{i,j}$ instead of the single f_i where f_{1_1} is the flow if we do not split the client; $f_{1,2}$ and $f_{1,3}$ are the flows if we split the client once and $f_{1,4}$ through $f_{1,7}$ are the flows if we split the client twice. These

[6]Notice that [58] pre-specifies a total aggregated number of splits across all clients whereas we set the a maximum for per-client splits. This slight modification facilitates the convex representation of the heuristic.

Term	Meaning
i, j, d	indexes for ball, bin and dimension
$\mathbf{Y}_i, \mathbf{C}_j$	Multi-dim vectors of ball and bin sizes
W_i	Weight of ball i
α_{ij}	$= 1$ if ball i is assigned to bin j and 0 otherwise
$\mathbf{x}_{ij}$	Vector of resources allocated to ball i in bin j
f_{ij}	$= 1$ if ball i can fit in bin j and 0 otherwise

TABLE A.2: Our notation to formulate FFD as a feasibility problem. We use bold to indicate multi-dimensional vectors and capitalize variables, which are typically constants for FFD, but can be variables of the outer problem in MetaOpt.

flows have to satisfy:

$$0 \le f_{1,1} \le d_1,$$
$$0 \le f_{1,i} \le \frac{d_1}{2}, \text{ for } i \in \{2,3\}$$
$$0 \le f_{1,i} \le \frac{d_1}{4}, \text{ for } i \in \{4,5,6,7\}.$$

The variable $f_{1,1}$ should be zero unless $d_1 < d_{\text{th}}$ (when we do not split clients), which we can achieve using big-M constraints (§A.3):

$$f_{1,1} \le \max(M(d_{\text{th}} - d_1), 0).$$

We want $f_{1,2}$ and $f_{1,3}$ to be exactly zero unless $d_1 \ge d_{\text{th}}$ and $d_1/2 < d_{\text{th}}$, which we can achieve by doing

$$f_{1,i} \le \max(M(d_1 - d_{\text{th}} + \epsilon), 0), \quad \text{for } i \in \{2,3\},$$
$$f_{1,i} \le \max(M(d_{\text{th}} - d_1/2), 0), \quad \text{for } i \in \{2,3\},$$

where we added the small pre-specified $\epsilon > 0$ to allow for the case where $d_1 = d_{\text{th}}$. Lastly, we want $f_{1,4}$ through $f_{1,7}$ to be exactly zero unless $d_1 \ge 2d_{\text{th}}$. We encode this as:

$$f_{1,i} \le \max(M(d_1 - 2d_{\text{th}} + \epsilon), 0), \quad \text{for } i \in \{4,5,6,7\}.$$

We can replicate this procedure for all d_k and encode POP with client splitting as a convex optimization problem. Once this is done, the techniques in §3.4 apply.

B Details of Vector Bin Packing

B.1 Formulation of FFD (First-Fit-Decreasing)

We formulate the first-fit-decreasing heuristic as a feasibility problem (a set of constraints and no objective). Such a formulation allows the bi-level optimization in MetaOpt to become a single-level optimization without a rewrite (§3.3). To our knowledge, this formulation of FFD is novel.

Modeling FFD using MetaOpt's helper functions. Fig. A.3 (right) shows how users can easily model FFD without having to go through the mathematical details. It also shows the mapping between the helper functions and the pseudocode using different colors.

Details of how we model FFD as an optimization. Table A.2 lists our notation. The model uses binary variables. It is not a scalable method to solve FFD in practical systems, and we propose it only as an effective method to find adversarial inputs for the FFD heuristic.

	Bin index $j \longrightarrow$					
Fit f_{ij}	0	0	1	0	1	1
RHS of Equation 11	0	$\frac{1}{2}$	1	$\frac{2}{4}$	$\frac{4}{5}$	$\frac{4}{6}$
α_{ij}	0	0	1	0	0	0

TABLE A.3: Illustrating how we model *first-fit* in Equation 11.

Modeling *decreasing* ball weights: Recall that FFD places the unassigned ball with the largest weight in each iteration. We can use a sorting network [62] to ensure we pick balls in decreasing order. Instead, we propose a simpler alternative:

We observe the ball-weighting functions are a fixed function of the ball size. Let $\mathbf{Y}$ be a multi-dimension vector that captures the size of the i'th ball on each dimension. Then, the weight of the i'th ball, W_i, is $\sum_d Y_i^d$ in FFDSum [66], $\prod_d Y_i^d$ in FFDProd [72] and Y_i^1/Y_i^2 in FFDDiv [67] respectively.[7]

We constrain the input space (ConstrainedSet in Equation 2) to ensure that we assign balls in decreasing order of their weight if we assign them based on their index:

$$W_i \ge W_{i+1} \ \forall \text{item } i \tag{10}$$

Modeling *first-fit*: FFD assigns each ball to the first bin that has enough capacity. Let bins be ordered by their index and α_{ij} be a binary variable with value of 1 iff bin j is the first bin (i.e., the one with the smallest index) that has enough capacity for ball i. We model the first-fit constraint as:

$$\alpha_{ij} \le \frac{f_{ij} + \sum_{\text{bin } k<j}(1 - f_{ik})}{j} \quad \forall \text{item} i, \forall \text{bin } j \tag{11}$$

$$\sum_{\text{bin } j} \alpha_{ij} = 1 \quad \forall \text{item } i \tag{12}$$

Table A.3 shows an example that illustrates this constraint in action. It is easy to prove that the right-hand-side of Eqn. 11 is 1 for the first bin where the ball can fit (i.e., smallest index in set of bins $\{j \| \text{ fit } f_{ij} = 1\}$) and less than 1 for all other bins. The second constraint is necessary to ensure that $\alpha_{ij} = 1$ for exactly the first-fitting bin for each ball.

Modeling resource allocation and capacity constraints: We first ensure our allocation is consistent with the ball assignment: we allocate sufficient resources only from the assigned bin (not any other bin). We can do this simply by: $\mathbf{x}_{ij} \triangleq \mathbf{Y}_i \alpha_{ij}$. Here, the resource assigned to a ball i at bin j ($\mathbf{x}_{ij}$) is simply the product of the ball size vector $\mathbf{Y}$ with the assignment indicator variable α. But $\mathbf{Y}$ is a variable of the outer (or leader) problem, so such equations are non-linear. To linearize, we use a technique similar to what is colloquially known as big-M in optimization literature. Let Z be an

[7] The FFDDiv function applies only to two dimensions.

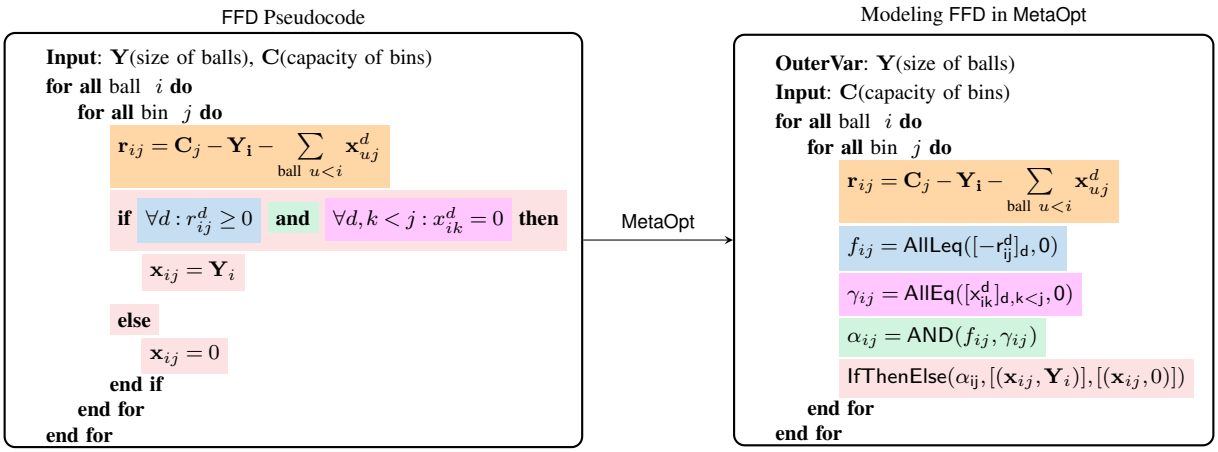


FIGURE A.3: The pseudocode for FFD and how users can model it in MetaOpt using the helper functions.

appropriately large postive constant, then:

$$x^d_{ij} \leq Z\alpha_{ij} \quad \forall \text{item } i \; \forall \text{bin } j \; \forall \text{dim } d \tag{13}$$

$$\sum_{\text{bin } j} x^d_{ij} = Y^d_i \quad \forall \text{item } i \; \forall \text{dim } d \tag{14}$$

We define the residual capacity of bin j after placing ball i in it as (remember we allocate ball i only after we place all the balls with a lower index):

$$\mathbf{r}_{ij} \triangleq \mathbf{C}_j - \mathbf{Y}_i - \sum_{\text{balls } u<i} \mathbf{x}_{uj} \quad \forall \text{item } i \; \forall \text{bin } j \tag{15}$$

The sum on the right captures how much resources we have already allocated to other balls from this bin. We next ensure f_{ij} is 1 iff the bin j has adequate resources to fit ball i. Let M be some appropriately large positive constant:

$$\min_d r^d_{ij} \leq Mf_{ij} \leq M + \min_d r^d_{ij}, \quad \forall \text{item } i \; \forall \text{bin } j \; \forall \text{dim } d \tag{16}$$

Here, if ball i fits in bin j, the residual capacity r^d_{ij} should be greater than 0 across all dimensions d. Therefore, Equation 16 clamps Mf between a positive number and M plus that positive number (remember r_{ij} is the remaining capacity of bin j *after* placing ball i in it).[8] Since f is a binary variable, the only feasible assignment in this case is 1. Conversely, if the ball does not fit in a bin, the residual capacity r^d_{ij} is below 0 on at least one dimension d and the constraint in Eqn. 16 clamps Mf to be between a negative number and M plus that negative number which forces f to be 0. In practice we set the value of M to be larger than the largest single-dimension bin capacity (*i.e.*, $\max_{j,d} C^d_j$).

[8] Corner case: when the residual capacity is precisely 0 on all dimensions, we want f to still be 1, but these constraints will allow f to be 0 as well. This is a rare case, but it can occur in practice. We can solve this corner case in a few different ways, including adding a small value ϵ to the left-most term in Equation 16.

Unique solution for FFD: The Equations 11–16 uniquely specify a solution to the iterative first-fit-decreasing heuristic. These constraints are linear even if the ball and bin sizes are variables in the outer problem. This is key because MetaOpt can apply without having to rewrite the heuristic follower.

Counting the number of bins: In this case, MetaOpt seeks inputs that cause the heuristic to use more bins than the optimal. To find such adversarial inputs, the outer (leader) problem needs the number of bins used by FFD:

$$\text{Num. bins used by FFD} \triangleq \sum_{\text{bin } j} \max_{\text{ball } i} \alpha_{ij}. \tag{17}$$

The term simply counts bins that have at least one ball. This is linear (max has a linear rewrite) and does not give rise to any additional concerns.

B.2 Proof of Theorem 1

Our goal is to show that for any $k > 1$, an input $\mathcal{I}$ exists where $\mathsf{FFDSum}(\mathcal{I})$ needs at least $2k$ bins while $\mathsf{OPT}(\mathcal{I})$ only needs k. Since we are proving a lower bound on the approximation ratio of FFDSum, it suffices to show an example for each k. We do this by the following; for every value of $k > 1$, we can find m and p such that $k = 2m + 3p$ and $p \in \{0, 1\}$. Then, we create an example consisting of $6m + 9p$ balls where $\mathsf{FFDSum}(\mathcal{I}) = 2\mathsf{OPT}(\mathcal{I}) = 2k$. We show the constructed example in Table A.4 along with the allocation from OPT and FFDSum.

C Details of Packet Scheduling Heuristics

We describe how we model SP-PIFO [5] and AIFO [74] as feasibility problems. These formulations are to the best of our knowledge novel. Table A.5 lists our general notations, and Table A.6 and Table A.7 show our specific notations for SP-PIFO and AIFO respectively.

Definition (Ranks and Priorities). Packet scheduling papers [5, 74] use both ranks and priorities: a packet with a

Ball ID	Ball Size	Wei ght	Num	Bin ID (OPT)	Bin ID (FFD)
1	[0.92, 0.00]	0.92	m	B1	B1
2	[0.91, 0.01]	0.92	m	B2	B2
3	[0.48, 0.20]	0.68	$\times p$	C1	C1
4	[0.68, 0.00]	0.68		C2	C2
5	[0.52, 0.12]	0.64		C3	C1
6	[0.32, 0.32]	0.64		C3	C2
7	[0.19, 0.45]	0.64		C2	C3
8	[0.42, 0.22]	0.64		C1	C3
9	[0.10, 0.54]	0.64		C1	C4
10	[0.10, 0.54]	0.64		C2	C5
11	[0.10, 0.53]	0.63		C3	C6
12	[0.06, 0.48]	0.54	m	B2	B1
13	[0.07, 0.47]	0.54	m	B1	B2
14	[0.01, 0.53]	0.54	m	B1	B3
15	[0.03, 0.51]	0.54	m	B2	B4

TABLE A.4: Constructed example to prove the approximation ratio of 2d-FFDSum is always lower bounded by 2 for any value of $\mathsf{OPT}(\mathcal{I}) > 1$.

higher rank has a lower priority and vice-versa. If a packet has rank R_p, and R_{max} is the maximum possible rank, we can compute the the packet's priority by $R_{max} - R_p$. This ensures that all the packets with a rank lower than R_p have higher priority values, and all those with a rank higher than R_p have lower priority values.

C.1 Formulation of SP-PIFO

SP-PIFO approximates PIFO [64] using n strict priority FIFO queues. It keeps a packet rank for each queue (*i.e.,* queue rank) that shows the lower bound on packet ranks the queue admits. For each packet, it starts from the lowest priority queue until it finds the first queue that can admit the packet (packet rank $\geq$ queue rank). If a queue admits the packet, SP-PIFO adds the packet to the queue and updates the queue rank to the recently admitted packet's rank (*i.e.,* push up). If none of the queues admit the packet (packet rank $<$ highest-priority queue rank), it reduces the rank of all the queues such that the highest-priority queue can admit the packet (*i.e.,* push down). Fig. A.4 (left) shows the pseudocode for SP-PIFO.

Modeling SP-PIFO using MetaOpt's helper functions. Fig. A.4 (right) shows how users can easily model SP-PIFO without having to go through the mathematical details. It also shows the mapping between the helper functions and the pseudocode using different colors.

Modeling push down. SP-PIFO reduces the rank of all the queues if none of the queues can admit the packet. This happens when the rank of the highest priority queue is higher than the packet rank (R_p). We model this as:

$$\hat{l}_q^p = l_q^{p-1} + max(0, l_N^{p-1} - R_p) \tag{18}$$

This constraint keeps the queue ranks the same if the packet

Term	Meaning
P, p	Number of packets and index for packet
R_{max}, R_p	Maximum rank and Rank of packet p
w_p	Weight of packet p
d_{pj}	$= 1$ if packet p dequeued after j, o.w. $= 0$

TABLE A.5: Our notation for formulating packet scheduling heuristics as feasibility problems. We capitalize variables which are typically constants for heuristic but can be variables of the outer problem in MetaOpt.

Term	Meaning
N, q	number of queues, and index for queue
$\mathbf{l}^{p-1}$	vector of queue ranks when deciding for packet p
$\hat{\mathbf{l}}^p$	vector of queue ranks after push down for packet p
x_{pq}	$= 1$ if packet p is in queue q, o.w. =0

TABLE A.6: Additional Notations for SP-PIFO.

rank R_p is greater than the highest priority queue l_N^{p-1}. Otherwise, it applies push down and reduces the rank of all the queues so that the highest priority queue can admit the packet (after the update, the rank of highest priority queue N is the same as packet rank R_p).

Deciding on the proper queue. Recall SP-PIFO adds a packet to the queue with the lowest priority among the ones that can admit the packet; that is the q that admits the packet ($R_p \geq \hat{l}_q^p$) but the one lower priority queue $q-1$ does not admit the packet ($R_p < \hat{l}_{q-1}^p$). We model this as following:

$$M x_{pq} \leq M + R_p - \hat{l}_q^p \qquad \forall \text{packet}\, p\, \forall \text{queue}\, q \tag{19}$$

$$M x_{pq} \leq M + \hat{l}_{q-1}^p - R_p - \epsilon \qquad \forall \text{packet}\, p\, \forall \text{queue}\, q \tag{20}$$

$$\sum_q x_{pq} = 1 \qquad \forall \text{packet}\, p \tag{21}$$

where M is a large constant ($\geq R_{max}$) and ϵ is a small constant (< 1). The first constraint ensures a queue with rank greater than the rank of the packet does not admit the packet (if $R_p < \hat{l}_q^p$, the constraint forces x_{pq} to 0). The second constraint ensures a queue does not admit the packet if a lower priority queue admits the packet (if $R_p \geq \hat{l}_{q-1}^p$, the constraint forces x_{pq} to). The last constraint forces the optimization to place the packet in one of the queues.

Modeling push up. Recall SP-PIFO updates the rank of the queue to the most recently admitted packet's rank. We model this as:

$$l_q^p = \hat{l}_q^p + x_{pq}(R_p - \hat{l}_q^p)\ \forall \text{packet}\, p\ \forall \text{queue}\, q \tag{22}$$

This constraint only updates the rank of queue q to R_p if the packet is placed in the queue ($x_{pq} = 1$). We can linearize this constraint [22].

Unique solution for SP-PIFO. We can combine these con-

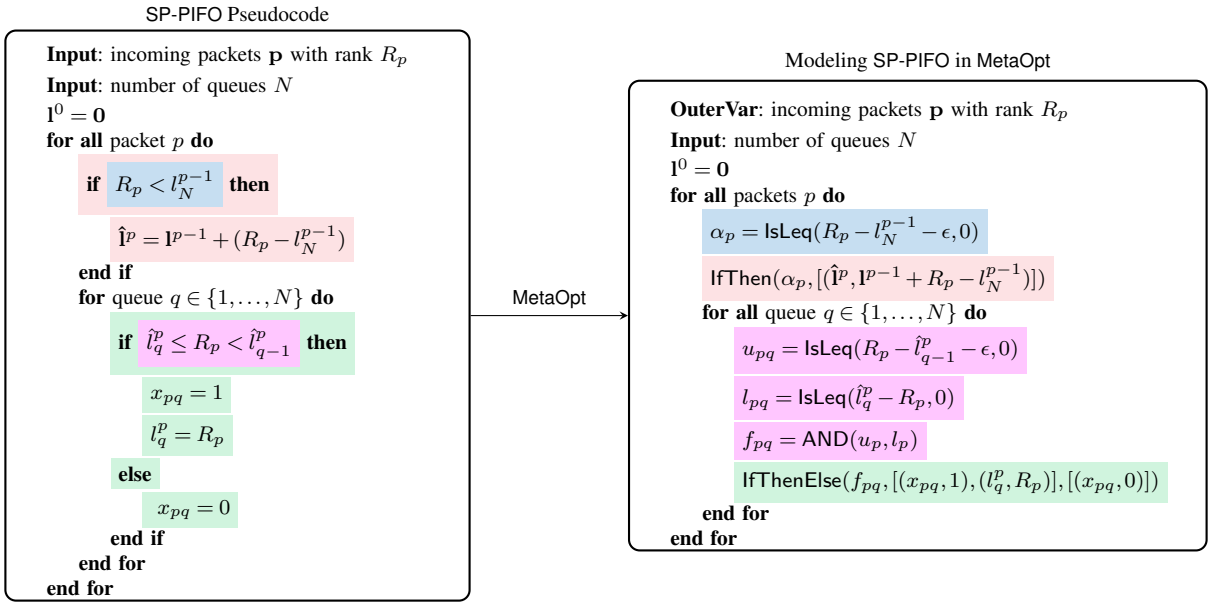


FIGURE A.4: The pseudocode for SP-PIFO and how users can model it in MetaOpt using the helper functions.

straints to uniquely specify SP-PIFO's decisions on a sequence of incoming packets. All these constraints are linear or linearizable using standard techniques even though packet ranks are variables in the outer problem.

Computing weighted average delay. We measure the gap in terms of the average delay of forwarding packets weighted by their priority. To measure delay of a packet, we count how many packets SP-PIFO decides to dequeue before it. Let d_{pj} indicate whether packet p is dequeued after packet j. We model the weighted average delay as:

$$\text{Weighted avg delay} = \frac{1}{P} \sum_{\text{pkt } p, j \neq p} (R_{max} - R_p) d_{pj} \tag{23}$$

Next, we define d_{pj}. We first assign weights to the packets such that the weights respect the order in which the packets should be dequeued (a packet p has a higher weight than j if it should be dequeued before j). We assign weights w_p as:

$$w_p = -p + \sum_{\text{queue } q} qP x_{pq} \tag{24}$$

This weighting guarantees that (1) a packet from a higher priority queue always has a higher weight than a packet from a lower priority queue, and (2) among the packets in the same priority queue, the one arrived earlier has a higher weight. Packet p is dequeued after packet j if the weight of packet j is higher.

$$w_j - w_p \leq M d_{pj} \leq M + w_j - w_p \;\; \forall \text{packets} p, j \tag{25}$$

Note that weights are unique.

C.2 Formulation of AIFO

AIFO [74] is an admission control on top of a FIFO queue that tries to approximate the same set of packets a PIFO queue would admit and is specifically designed for shallow buffers. AIFO keeps a window of recently seen packet ranks and computes the relative rank of the new packet with respect to this window. Then, it compares this quantile estimate with the fraction of available space in the queue (multiplied by some constant burst factor). If the quantile is lower or equal, AIFO admits the packet. Otherwise, it drops the packet.

Finding quantile estimate. Recall that AIFO computes how many packets in its recent window have lower ranks than an incomming packet p. We model this as:

$$R_p - R_j \leq M g_{pj} \leq M + R_p - R_j - \epsilon \tag{26}$$
$$\forall \text{packet} p \forall \text{packet} j: \; p - K \leq j \leq p - 1$$

$$g_p = \sum_{\text{pkt } j: p-K \leq j \leq p} g_{pj} \tag{27}$$

where M is a large constant ($M \geq R_{max}$) and ϵ is a small constant ($\epsilon < 1$). Observe that the first constraint compares the current packet with the last K packets (the ones in the window). For every pair, if $R_p > R_j$, the left constraint in Equation 26 forces $M g_{pj}$ to be positive and consequently $g_{pj} = 1$ (packet j in the window has a lower rank). If $R_p \leq R_j$, the right constraint forces g_{pj} to be 0. Equation 27 keeps track of the number of packets in the window with rank less than the rank of packet p. For packets $p < K$, we add some additional variables that represent the rank of packets arrived

Term	Meaning
B	burst factor
C	queue size in the number of packets
a_p	1 if packet p is admitted, $=0$ if dropped
K	number of samples in the window to estimate the quantile
g_p	number of packet ranks in the window smaller than rank of packet p
g_{pj}	1 if rank of packet j in the window is smaller than rank of packet p

TABLE A.7: Additional Notations for AIFO.

and departed before this sequence.

Deciding to admit or drop. Recall that AIFO admits the packet if the quantile estimate of the current packet rank is less than some factor of the available capacity of the queue. We model this as:

$$\hat{c}_p = B\frac{C - \sum_{\text{pkt } i<p} a_i}{C} \quad \forall \text{packet } p \tag{28}$$

$$\hat{c}_p - g_p + \epsilon \leq Z a_p \leq Z + \hat{c}_p - g_p \quad \forall \text{packet } p \tag{29}$$

where Z is a large constant ($\geq$ maximum of window size and queue size). The first constraint computes the fraction of available capacity multipled by a constant (burst factor in [74]) and the second constraint ensures $a_p = 0$ (*i.e.,* we drop the packet) if the quantile estimate is higher than the available capacity metric $\hat{c}_p$ and $a_p = 1$ otherwise.

Computing the final ordering of packet is similar to SP-PIFO. These constraints in combination find AIFO's unique solution. All the constraints are also linear.

C.3 Proof of Theorem 2

We use the same approach as our proof for FFDSum. We show a constructive example that matches the gap. In our example, first p packets with the lowest rank (=0) arrive, then 1 packet with the highest possible rank (=R_{max}), and finally, p^* packets with the second highest possible rank (=$R_{max}-1$).

Given this sequence of packets; SP-PIFO first adds all the p packets with the lowest rank to the lowest priority queue and then, updates the queue rank to 0 (*i.e.,* push up). Then, it adds the highest rank packet to the lowest priority queue and updates the queue rank to R_{max}. Lowest priority queue can not admit the packets with rank=$R_{max}-1$ anymore because the condition to admit a packet is that the queue rank should be lower than the packet rank. So, all the p^* packets are enqueued in a higher priority queue. As a result, all these p^* packets are going to be forwarded before the p packets with highest priority (Fig. A.5 shows this using an example).

We can compute the weighted sum of packet delays for SP-PIFO and PIFO as:

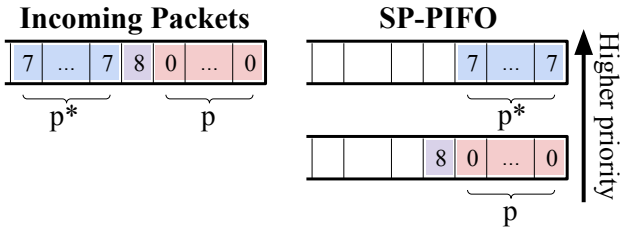


FIGURE A.5: Example of the input trace for SP-PIFO when $R_{max} = 8$. In this case, SP-PIFO dequeues all the packets with the lowest rank (highest priority) after all the second lowest priority packets ($r = R_{max} - 1$). Packets arrived earlier are on the right side of the queue.

$$\text{Wdelay}_{\text{PIFO}} = \frac{R_{max}p(p-1)}{2} + pp^* + \frac{p^*(p^*-1)}{2} \tag{30}$$

$$\text{Wdelay}_{\text{SP-PIFO}} = \frac{p^*(p^*-1)}{2} + R_{max}pp^* + \frac{R_{max}p(p-1)}{2} \tag{31}$$

We can compute the difference in the weighted sum of delays as:

$$\begin{aligned}\text{Wdelay}_{\text{SP-PIFO}} - \text{Wdelay}_{\text{PIFO}} &= (R_{max}-1)pp^* \\ &= (R_{max}-1)(N-1-p)p\end{aligned} \tag{32}$$

Note that $p + p^* = N - 1$. We can derive Theorem 2 by finding the maximum of Equation 32.

D List of MetaOpt Helper Function

Table A.8 lists the helper functions in MetaOpt. MetaOpt internally and automatically translates these into constraints. For specific use cases, please refer to Fig. A.1 for DP, Fig. A.3 for FFD, and Fig. A.4 for SP-PIFO.

E Black-box search methods

We next describe our baselines in more detail. We compared MetaOpt to these baselines in §4.

Random search. This strawman solution picks random inputs, computes the gap, and returns the input that resulted in the maximum gap after .

Hill climbing is a simple local search algorithm. It first randomly chooses an arbitrary input $\mathbf{d}_0$ and then generates its neighbors ($\mathbf{d}_{aux}$): it adds to $\mathbf{d}_0$ a value, which it draws from a zero-mean σ^2-variance Gaussian distribution. If this neighboring input increases the gap the hill climber moves to it. Otherwise it draws another neighbor.

The hill climber repeats these steps until it fails to make progress and terminates. This happens when it fails to increase the gap for K consecutive iterations. The hill climber outputs its current solution as a local maximum once it terminates (Algorithm 1).

Helper Function	Description
$\mathsf{IfThen}(\mathsf{b}, [(\mathsf{x_i}, \mathsf{F_i}())])$	if binary variable $b = 1$ then $x_i = F_i()$ for all i.
$\mathsf{IfThenElse}(\mathsf{b}, [\mathsf{x_i}, \mathsf{F_i}()], [(\mathsf{y_j}, \mathsf{G_j}())])$	if binary variable $b = 1$ then $x_i = F_i()$ for all i, otherwise $y_j = G_j()$ for all j.
$\mathsf{b} = \mathsf{AllLeq}([\mathsf{x_i}], \mathsf{A})$	$b = 1$ if all x_is are $\leq$ a constant A, otherwise $b = 0$.
$\mathsf{b} = \mathsf{IsLeq}(\mathsf{x}, \mathsf{y})$	$b = 1$ if $x \leq y$, otherwise $b = 0$.
$\mathsf{b} = \mathsf{AllEq}([\mathsf{x_i}], \mathsf{A})$	$b = 1$ if all x_is are $=$ a constant A, otherwise $b = 0$.
$\mathsf{b} = \mathsf{AND}([\mathsf{u_i}])$	$b = 1$ if all u_is are $= 1$, otherwise $b = 0$.
$\mathsf{b} = \mathsf{OR}([\mathsf{u_i}])$	$b = 1$ if at least one $u_i = 1$, otherwise $b = 0$.
$\mathsf{y} = \mathsf{Multiplication}(\mathsf{u}, \mathsf{x})$	Linearizes multiplication of a binary variable u and a continuous variable x. (Internally, we choose a simpler encoding if x is non-negative)
$\mathsf{y} = \mathsf{MAX}([\mathsf{x_i}], \mathsf{A})$	$y =$ maximum of x_is and a constant A.
$\mathsf{y} = \mathsf{MIN}([\mathsf{x_i}], \mathsf{A})$	$y =$ minimum of x_is and a constant A.
$[\mathsf{b_i}] = \mathsf{FindLargestValue}([\mathsf{x_i}], [\mathsf{u_i}])$	$b_i = 1$ if x_i is the largest among the group of variables x_j with corresponding $u_j = 1$, otherwise $b_i = 0$. At least one $b_i = 1$.
$[\mathsf{b_i}] = \mathsf{FindSmallestValue}([\mathsf{x_i}], [\mathsf{u_i}])$	$b_i = 1$ if x_i is the smallest among the group of variables x_j with corresponding $u_j = 1$, otherwise $b_i = 0$. At least one $b_i = 1$.
$\mathsf{r} = \mathsf{Rank}(\mathsf{y}, [\mathsf{x_i}])$	$r =$ rank of variable y among the group of variables $[x_i]$ (quantile).
$\mathsf{ForceToZeroIfLeq}(\mathsf{v}, \mathsf{x}, \mathsf{y})$	Forces $v = 0$ if $x \leq y$ (users can model this with IfThen, but this one is customized and faster). Internally, we choose a simpler encoding if v is binary.

TABLE A.8: MetaOpt's helper functions. (b and u are binary variables, and x and y are continuous variables)

Algorithm 1 Hill climbing

Input: $\mathbf{d}_0, \sigma^2, K$
$\mathbf{d} \leftarrow \mathbf{d}_0,\ k \leftarrow 0$
while $k < K$ **do**
  $\mathbf{d}_{\text{aux}} \leftarrow \max(\mathbf{d} + \mathbf{z}, \mathbf{0})$ where $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I})$
  if $\text{gap}(\mathbf{d}_{\text{aux}}) > \text{gap}(\mathbf{d})$ **then** $\mathbf{d} \leftarrow \mathbf{d}_{\text{aux}},\ k \leftarrow -1$ **end if**
  $k \leftarrow k + 1$
end while
Output: $\mathbf{d}$

We re-run the hill climber M_{hc} times with different initial inputs and return the solution that produces the maximum gap to minimize the impact of the starting point.

Simulated annealing refines hill-climbing and seeks to avoid getting trapped in a local maxima [45]. The difference between the two algorithms is simulated annealing may still (with some probability) move to a neighboring input even if that input does not improve the gap.

Simulated annealing gradually decreases the probability of moving to inputs that do not change the gap: it defines a temperature term, t_p, which it decreases every K_p iterations to $t_{p+1} = \gamma t_p$. Here, $0 < \gamma < 1$ which ensures $t_p \to 0$. If $\text{gap}(\mathbf{d}_{\text{aux}}) \leq \text{gap}(\mathbf{d})$, we have $\mathbf{d} \leftarrow \mathbf{d}_{\text{aux}}$ with probability $\exp(\frac{\text{gap}(\mathbf{d}_{\text{aux}}) - \text{gap}(\mathbf{d})}{t_p})$. We repeat the process M_{sa} times and return the best solution.

Hill climbing vs simulated annealing. Hill climbing has less parameters and is better suited for smooth optimizations where there are few local-optima. But simulated annealing is better suited for intricate non-convex optimizations with many local-optima because its exploration phase, although slower, allows it to avoid local optima and works better in the long run.

Both of these algorithms have a number of hyperparameters: we run grid-search to find the ones that produce the highest gap.

Towards provably performant congestion control

Anup Agarwal[†], *Venkat Arun*[⋆], *Devdeep Ray*[†], *Ruben Martins*[†], *Srinivasan Seshan*[†]
[†]*Carnegie Mellon University*, [⋆]*University of Texas at Austin*

Abstract

We seek to ease the design of congestion control algorithms (CCAs) that *provably* perform well under diverse network scenarios including, cellular links, policers, token bucket filters, operating system jitter, etc. Guaranteeing performance under such conditions is hard as it requires considering combinatorial possibilities of CCA and network interactions. We build a framework that allows us to reason about CCAs. It describes (1) the *necessary* actions that any performant CCA must take, and (2) a provably *sufficient* amount of information for CCAs to consider when deciding their sending rate. Combining this framework with techniques in formal methods, we synthesize CCAs that provably perform across a diverse set of network conditions. Our methodology also led us to discover and prove fundamental impossibility results.

1 Introduction

End-to-end congestion control algorithms (CCAs) for the Internet must operate on varied network paths, each with its unique combination of physical links (e.g., wired, cellular, low-latency data center, satellite [3, 14, 15, 31, 57, 61, 63]) and processing elements (e.g., load balancers, schedulers, NICs, switches, routers). On such paths, CCAs balance often conflicting objectives like utilization, delay, packet losses, convergence time, fairness, and flow-completion time.

A robust general-purpose CCA that performs well across diverse network scenarios has remained elusive. All existing general-purpose CCAs (e.g., Cubic, BBR, PCC, Copa) perform poorly in some practical scenario [6, 7, 17, 20, 60] (§2).

We report on our attempt at an ambitious task: "to design CCAs that can *provably* meet performance objectives on a broad set of network paths". Unsurprisingly, this was hard. Our initial optimism came from recent work, CCAC [7] that uses computational methods to search for network behaviors that break a given CCA. We thought we could simply iterate over CCA design by asking CCAC if it violates a performance property, and if so, fix the CCA and repeat the process.

This was not as straightforward. While CCAC describes a scenario where a given CCA breaks, *it does not tell us how to fix the CCA*, i.e., (Q1) how should it change its sending rate choices, and in doing so (Q2) what signals/statistics might it consider. CCAC also does not tell us (Q3) if the performance property is even achievable by some CCA.

To answer these questions, we examined the CCAs developed over the last few decades. They consider numerous statistics to infer congestion (e.g., derivatives, integrals, exponentially weighted moving averages of loss/delay signals). There is no consensus on what statistics a CCA should maintain. On closer observation, we find that the statistics maintained by CCAs (implicitly) describe latent properties of the path that the CCA is running on. For instance, `ssthresh` in Reno/New Reno [33, 34] estimates a lower bound on the *BDP* of the network; BBR [15] explicitly maintains estimates of bandwidth and propagation delay.

We formalize this intuition by defining *belief set* as the set of paths (described using parameters like link rate, propagation delay, and buffer size) that the CCA believes it could be running on and the possible instantaneous state(s) of each path (e.g., packets in queue/on the wire). Given a model of the network (e.g., CCAC [7]), we give a canonical way to compute the belief set as the exhaustive set of paths and states that can explain the history of observations made by the CCA.

The belief set (or beliefs) gives us a way to model *all* CCAs and formally reason about them. First, we show that it is *necessary* for any performant CCA to shrink the size of the belief set, i.e., reduce uncertainty in the possible paths it could be running on, e.g., probe to check if the link rate could be higher. Second, we formally prove that the belief set is a *sufficient* set of statistics for a performant CCA to consider, i.e., if a CCA can ensure a certain performance property, then a "belief-based CCA" can also ensure it, where a belief-based CCA is one whose sending rate is a pure function of the belief set.

Beliefs help answer the above questions (Q1-3) and enable two key results. First, by combining beliefs with CCAC, we built a tool, CCmatic. It uses program synthesis techniques to systematically solve the *search problem*: "find a CCA in a *search space* that ensures given *performance properties* over all specified *network paths or scenarios*". Sufficiency of beliefs allows us to define an exhaustive and tractable search space, necessity of beliefs allows us to define performance properties, and techniques from CCAC allow us to define the network paths. CCmatic synthesized CCAs that guarantee performance across all paths described by models like CCAC, where existing CCAs struggle to even guarantee 1% utilization [7]. Despite being designed for theoretical "worst-case" links, the synthesized CCAs outperform or match existing CCAs on empirical links that resemble "average-case" networks. By design, the synthesized CCAs are short, modular, human-interpretable, and come with proven performance guarantees.

Second, experimenting with CCmatic, it sometimes reported

that no CCA in the search space could meet the performance property, hinting that perhaps our performance property cannot be achieved. Despite the sufficiency of the belief set, this is not a definitive proof because CCmatic only explores a subset of belief-based CCAs due to computational limits. Nevertheless, using the sufficiency and necessity properties of the belief set, we prove a previously unknown fundamental tradeoff between loss and convergence time on shallow buffered networks. Intuitively, the combination of short buffers and jitter creates uncertainty in delay measurements, forcing CCAs to rely on loss-based signals. If CCAs probe for bandwidth aggressively, they converge faster but risk losses. If they probe conservatively, they mitigate losses but converge slowly. We quantify this relationship and synthesize CCAs that achieve different points on the Pareto frontier. Note, while this tradeoff may seem intuitive, formalizing it requires careful treatment (§6.2).

Our contributions are: (1) the belief framework to reason about congestion control (§4), (2) CCmatic, a tool to synthesize CCAs (§5), (3) the CCAs synthesized by CCmatic, proofs about their performance properties, and their empirical evaluation (§6.1, §6.3, §6.4), and (4) the impossibility theorems (§6.2). Note, to provide provable performance guarantees, we make several assumptions which we hope to relax in future work (§2, §8). In particular, we do not formally study fairness between multiple flows.

2 Motivation

We motivate beliefs using an example and outline our goals.

Belief set example. Consider a hypothetical CCA that knows it is running on a simple link with constant round-trip propagation delay $R_m = 100$ ms, an infinite buffer $\beta = \infty$, and a constant, but unknown, bandwidth C MBps. Initially, the CCA could be running on *any* path on the Internet, i.e., C could be any non-zero value (e.g., 100 MBps). I.e., the CCA believes $C \in (0, \infty)$ MBps. Say the CCA has been sending at rate $\lambda = 10$ MBps, and observes that all packets are ACKed 100 ms after transmission (i.e., $RTT = R_m$). Such RTTs could be produced by any path with $C \geq 10$ MBps. Now, say the CCA increases λ to 15 MBps. If RTTs increase or losses happen, the CCA can conclude that $C \leq 15$ MBps. Combining this with CCA's past observations, we can update the belief set to $C \in [10, 15]$ MBps. Otherwise, if RTT remains at 100 ms, then $C \in [15, \infty)$ MBps.

We can compute beliefs for *any* CCA given its past observations on a network model. The belief set serves as a useful tool to reason about the performance of *any* CCA. For instance, if the CCA above believes that $C \in [15, \infty)$ MBps, then it needs to keep increasing its rate until it obtains an upper bound on C by deliberately causing losses or increasing RTTs. Otherwise, without an upper bound, the CCA risks arbitrarily low utilization because the actual link rate could be arbitrarily large, e.g. 1500 MBps. I.e.,

LEMMA 2.1. *To avoid arbitrarily low utilization, a CCA needs to* shrink *the set of possible paths (the belief set) it could be running on by obtaining an upper bound on C.*

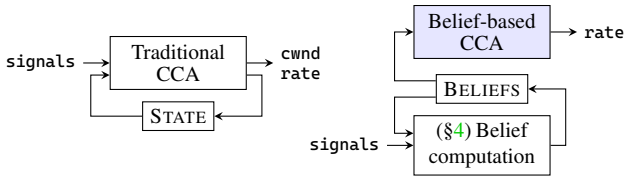


Figure 1: Beliefs standardize the state in congestion control.

In addition to it being necessary to shrink beliefs, we show in §4 that the belief set is the only information a CCA needs in order to decide sending rate. This is because the belief set is the only information a CCA needs to estimate the performance impact of its actions. This vastly simplifies CCA design (Fig. 1).

Traditionally, CCAs decide (Q1) what state (statistics) to maintain from input signals, and (Q2) sending rate. For instance, to answer Q1, CCA designers often consider "what does packet loss tell us about the state of the network?", "when can bandwidth be measured?", "what length of interval should be considered to sample bandwidth?" [15, 16].

Belief computation is uniquely determined by a network model and exhaustively derives all possible information about the network's path/state directly from the time series of CCA's sending and acknowledgment sequence numbers.[1] We no longer need to make ad-hoc decisions to answer Q1. This effectively decouples Q1 and Q2 and standardizes the state a CCA has to maintain. With beliefs, the CCA only decides the computation in the shaded box that maps beliefs to sending rate.

Goals and non-goals. We want CCAs that *provably* achieve bounds on utilization, delays, losses, and time to converge to variations in link rate, under the following scenarios:

S1. Non-congestive delays or jitter. Delays can occur due to reasons unrelated to congestion [6, 29], e.g., delayed ACKs, ACK aggregation, OS scheduling, delays at the MAC and physical layers. This hinders CCAs from bounding end-to-end delays while ensuring high utilization. Traditional loss based CCAs [31, 33, 34, 52] fill up queues until they experience a loss and cannot bound delays. Delay based CCAs [8, 12, 15, 21, 46, 56, 57] use variation in measured round-trip times (RTTs) to estimate congestion. Jitter can cause these CCAs to mis-estimate congestion and send at a rate as much as $10\times$ away from the correct rate [7, 21, 46, 60]. For instance, jitter can trick BBR [15] and Copa [8] into achieving near-zero utilization [7]. Recent learning-based CCAs (§7) also do not explicitly consider jitter.

S2. Shallow buffers with jitter. Paths with shallow buffers are common on the internet [24, 27, 28], because they prevent buffer-bloat and help manage cost/area/power of routers [4, 23]. On such paths, it is challenging to maintain utilization while avoiding excessive losses. Traditional loss-based CCAs, including Cubic [31], get poor utilization in single flow cases ([55], §6.4). ACK-clocked CCAs, despite pacing, send bursts of packets due to jitter (e.g., ACKs-aggregation [29]) which

[1] RTTs, ACK rates, and losses can be derived from sequence numbers.

risks excessive losses [7]. BBRv1 [15] is paced, but uses aggressive probes that incur O(*BDP*) losses periodically [17]. BBRv2 [27] and BBRv3 [28] incur lower loss on average but still incur O(*BDP*) losses in some cases (Appendix H).

In this paper, we only focus on the single-flow case with an infinite backlog of data to transmit when designing provably performant CCAs for S1 and S2. Designing provably fair CCAs robust to jitter is a hard problem [6] that we do not address. Our current formal framework gives no guarantees or predictions on the outcome of multi-flow experiments. Nevertheless, we empirically evaluate the fairness of our synthesized CCAs under simple network conditions (Appendix H). We find that some of our CCAs are fair while others are not. We believe that addressing S1 and S2 in the single-flow case is a key step towards addressing the multi-flow case. S1 and S2 are important real-world scenarios that have been the focus of recent changes to BBR [27, 28]. There are several situations where a flow is alone in its bottleneck queue [13] where our insights immediately apply (e.g., cellular networks). Our loss-convergence tradeoff (§6.2) also applies with multiple flows and may guide how buffers should be sized for networks with jitter.

Formally addressing S1 and S2, even in the single-flow case, was challenging. As described above, none of the existing CCAs achieve our goals. Beliefs and automated reasoning allowed us to systematically explore the design space of CCAs, unveiling previously unknown tradeoffs and novel CCA mechanisms. We discovered and quantitatively proved a fundamental tradeoff between loss and convergence time on networks with shallow buffers. We synthesized CCAs that are on this Pareto frontier. Existing CCAs, unaware of the tradeoff, make sub-optimal tradeoffs or only explore a subset of useful points on the Pareto frontier. We also discovered new ways to use sending rate decisions to augment the information obtained from *RTT*s and ACK rates. This leads to better estimates of the network parameters (bottleneck bandwidth/buffer) and state (queuing or extent of congestion) (§6.2.1). We illustrate here with two examples.

Example 1. CCAs use the gap between instantaneous and minimum *RTT*s to estimate queueing delay. Non-queueing delays can create *RTT*s larger than the minimum *RTT* and cause a CCA to erroneously infer that a queue is built up. However, if the CCA has been sending at a low rate, then we know there is no queue buildup even if there is inflation in *RTT*s. In such cases sending rate choices provide a better estimate about queueing than *RTT*s alone.

Example 2. Say a CCA sent a burst of packets to probe for available bandwidth and observed that the probe did not incur any packet loss. Then we can conclude that either the buffer or the bandwidth is large enough to have accommodated the burst. I.e., both the buffer and bandwidth cannot be small as that would have incurred a loss. We cannot make such a conclusion by relying on ACK rate and *RTT* measurements alone. Due to jitter, a burst may not lead to an immediate increase in ACK rate ([7], §4) which traditionally would have allowed us to conclude bandwidth is large. Likewise, an inflation in *RTT*s could be due to jitter, and we cannot assume that the bottleneck buffer is large enough to accommodate the inflation in *RTT*s.

CCmatic automatically synthesizes CCAs that use such insights to make non-trivial decisions about when/how to probe/drain. E.g., it realized that draining is necessary not only to maintain low delay, but also to restrict losses when probing for bandwidth on paths with jitter and shallow buffers (§6.2.1).

Our work addresses an important and challenging set of scenarios, and establishes a formal methodology for further exploration. In the future, we hope to extend our methodology to formally explore fairness between multiple flows, and robustness to application-limited flows and non-congestive losses (§8).

C – bottleneck link rate [r]	$\theta(t)$ – instantaneous inflight [b]
R_m – round trip prop. delay [t]	$q(t)$ – inst. bottleneck queue [b]
β – bottleneck buffer size [b]	$qdel(t)$ – inst. queueing delay [t]
D – max per-packet jitter [t]	$RTT(t)$ – inst. round trip time [t]
MSS – maximum segment size [b]	$S(t)$ – cumulative service [b]
$BDP = C \cdot R_m$ [b]	$A(t)$ – cumulative arrivals [b]
β_s – buffer in seconds (β/C) [t]	$L(t)$ – cumulative loss [b]
T – time steps [unitless]	$[.]_L(t)$, $[.]_U(t)$ – inst. lower and upper bounds on parameter or state, e.g., $C_L(t), C_U(t)$
$\lambda(t)$ – inst. sending rate [r]	

Table 1: Glossary of symbols. The square brackets show the units: bytes [b], rate [r], and time [t]. inst. = instantaneous. Inflight is bytes that are unacknowledged and not inferred as lost.

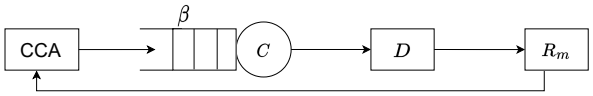


Figure 2: CBR-delay network model.

3 Network models

We use CCAC [7] to succinctly express and efficiently explore the scenarios in §2. CCAC uses a single bottleneck abstraction to summarize the cumulative effects multiple elements on a network path. It uses bounded model checking [19] to provide a trace of CCA execution under various network behaviors.

Our investigations revealed that CCAC expresses behaviors that are perhaps too adversarial for any CCA to handle (§6.1, §6.2). So, we explore two other network models that are *weaker*, i.e., they are less challenging from the CCA's point of view as they capture strictly fewer behaviors. We briefly describe these models and use notation from Table 1. Note, if a CCA works on a stronger model then it also works on a weaker model, and an impossibility result for a weaker model holds for a stronger model.

CBR-delay. This is motivated from [6]. It abstracts the network as a constant bit rate (CBR) box followed by a non-deterministic delay (or jitter) box, and a propagation delay of R_m seconds (shown in Fig. 2). The CBR box has a constant (over time), but arbitrary bottleneck bandwidth of C bytes/second, and a buffer of size β bytes. It expresses the queueing

or *congestive delays* (and losses) at the bottleneck queue in a network path. The delay box can add up to D seconds of delay *non-deterministically*. **Note, non-determinism is different from randomness or stochasticity (e.g., uniform random delays).** Non-determinism allows the network to *arbitrarily* inject bursts and provably express the cumulative effect of various sources of jitter [6, 7, 10, 42]. Non-determinism (as opposed to randomness) can express non-congestive delays that may have causal effects or correlations.

CCAC [7]. It includes all behaviors captured by CBR-delay. Additionally, CCAC can non-deterministically accept a burst of packets without building up a queue, effectively hiding congestive delays/losses even when the CCA is sending above the link rate. In contrast, CBR-delay can inject non-congestive delays but not non-deterministically hide congestive delays. This is a crucial and previously unknown distinction that changes the tradeoffs that CCAs must make (§6.1, §6.2).

Ideal link. It cannot add any jitter to packets. It is simply a FIFO queue, with a constant (but arbitrary) bandwidth and propagation delay. Several theoretical analysis [9, 18, 45, 49, 62] have used similar modeling. We study this to compare CCAs designed for the ideal link vs. stronger models.

Variations in link rate over time. Like CCAC, all the models assume a fixed link rate over time. We use CCAC's approach to express variations in link rates. CCAC and CBR-delay express short-term link rate variations using jitter. All the models express long-term link rate variations by arbitrarily choosing initial conditions. E.g., a trace that begins with a high congestion window (*cwnd*) relative to C emulates a scenario where link rate decreased. Alternatively, a large initial queue buildup, and low *cwnd* can emulate a case where the link rate decreased and the CCA backed off, but the queue has not drained. One can stitch such traces together to explore longer executions with potentially multiple link rate variations (§5.2).

Formal definition. Mathematically, we view a network model as a relation that relates $\langle$`path`,`state`,`CCA_action`$\rangle$ to $\langle$`next_state`,`CCA_feedback`$\rangle$. Note, due to non-determinism, the relation may map a CCA action on a given path and state to multiple feasible next states and feedbacks. The network model also defines a set of initial states, and the domains of path, state, action, and feedback.

For example, in the CBR-delay model, a path is described by the parameters: link rate, propagation delay, amount of jitter, and buffer size, e.g., $\langle C,R_m,D,\beta\rangle$; and state by: bytes in the bottleneck queue (q) and bytes in flight (θ), e.g., $\langle q,\theta\rangle$. The model's relation is defined by constraints, such that feasible solutions to the constraints are the tuples in the relation.

4 Belief framework

DEFINITION 4.1. *A **belief set** (or beliefs) for a given network model is the set of paths (and their latest states) that could have produced (according to the network model) the historical sequence of CCA's observations. It is the CCA's* belief *about the paths and states of the network it is running on.*

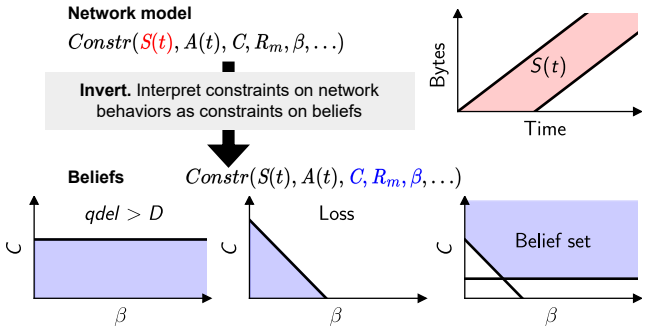


Figure 3: Inverting the network model to compute beliefs. Depending on the CCA's observations, we may get different bounds on the belief set. The bottom three plots illustrate the constraints on the belief set we get depending on whether the CCA observed $qdel > D$, loss, or neither (for CBR-delay).

For example, for the CBR-delay model, the belief set is a set of tuples of the form $\langle C,R_m,D,\beta,q,\theta\rangle$. Such a tuple is in the belief set *if and only if* it can explain (according to the network model), the observations of the CCA thus far. We use the term observations to collectively refer to CCA's actions and feedback from the network.

Computing beliefs. We can "invert" the network model's relation to compute beliefs (Fig. 3). Specifically, the constraints of the model describe the feasible ways in which the network's state and feedback can evolve (e.g., how it services (delays) packets, drops packets, and builds queues), given the CCA's sending behavior, the network path, and the network's initial state. I.e., the constraints describe feasible combinations of $\langle S(t),L(t),q(t),\theta(t)\rangle$, $A(t)$, $\langle C,R_m,D,\beta\rangle$, and $\langle q(0),\theta(0)\rangle$. If we fix the observations, i.e., $\langle A(t),S(t),L(t)\rangle$, then the constraints describe the feasible paths and states that could have produced CCA's observations. This is the belief set. Each constraint of the network model gives us a constraint on the belief set. This is similar to conversion of a partially observable markov decision process (POMDP) into a belief MDP [37]. We perform the inversion §5.1 and §6.2.

As a quick example, for the CCAC model, in an interval of length T, the network can serve at a rate C and inject a burst of D seconds, i.e., $S(T)-S(0) \leq CT+CD$, where $S(t)$ is cumulative bytes served (delivered) until time t. If we invert this constraint on S to a constraint on C, we get $C \geq \frac{S(T)-S(0)}{T+D}$.

Shrinking beliefs is necessary. CCAs need to shrink the size of the belief set, i.e., infer the possible parameters and states of the network they are running on. We illustrate this using a family of lemmas (Lemma 2.1, Lemma 4.1) that show the dimensions along which beliefs need to shrink to ensure different performance properties.

LEMMA 4.1. *To ensure an upper bound on queueing delay (qdel), a CCA needs to shrink the set of possible propagation*

delays the network could have.

Consider a CCA that aims to ensure $qdel \leq 10$ ms on an ideal link. Say it has set $cwnd = 10$ MB, and observes that all RTTs are 100 ms, yielding an average throughput of $cwnd/RTT = 100$ MBps. Such RTTs can be explained by $C = 100$ MBps, and any $R_m \in [0,100]$ ms, and $qdel = 100 - R_m$, i.e., $qdel \in [0,100]$ ms. Now, say the CCA decreases $cwnd$ to 5MB. If RTTs are still 100ms, i.e., RTTs do not decrease with $cwnd$, the CCA can conclude that $R_m = RTT = 100$ ms and $qdel = 0$ ms. Otherwise, if RTTs drop to 50 ms, i.e., the average throughput is still 100 MBps, R_m and $qdel$ still remain in $[0,100]$ ms.

Until a CCA decreases $cwnd$ to the point that RTTs stop decreasing, it cannot obtain a lower bound on R_m, consequently it cannot ensure an upper bound on $qdel$.

Beliefs are sufficient. Beliefs allow a CCA to compute possible next state(s) and feedback(s) (and consequently potential future performance) for different sending rate choices. Due to this, a CCA does not need to look at any information other than the belief set when deciding its actions.

THEOREM 4.1. *If there exists a* deterministic CCA *that ensures a performance property on a network model, then, there exists a* belief-based *CCA that ensures the performance property on the model.* Where, a deterministic CCA is a CCA whose actions are a function of the entire history of past observations (i.e., its actions and feedback) and a belief-based CCA is a CCA whose actions are a function of the belief set computed using the network model over the history of past observations.

For this theorem, we assume that the performance property is specified as a boolean valued function over a belief set and CCA's action on that set. We show in Appendix A how the properties we use can be expressed in this form.

We use this theorem to (1) synthesize CCAs as function of the belief set, and (2) to prove impossibility results, i.e., if there is no action that can ensure a performance property over all tuples in a valid belief set, then the performance property is not achievable. We use such arguments in §6.2.

Appendix A gives a formal proof by constructing a belief-based CCA using the deterministic CCA. Here we give the intuition. We view congestion control as a 2-player (CCA vs. network) zero-sum game. The network tries to prevent the CCA from achieving its performance property. The CCA chooses its sending rate and the network delivers, delays, or drops packets. The only "rule" is that the network's actions must correspond to some path in the network model.

The belief set exhaustively summarizes the history of the game, making it *memoryless* (similar to the board state in chess). Beliefs serve as a "board" by meeting the two requirements: (R1) we can determine the set of feasible moves for the players from the board allowing us to enforce the game rules (Lemma A.1), and (R2) we can update the board by applying the moves (Lemma A.2). As a result, future progressions of the game (and any optimal strategies) depend only on the board (beliefs) irrespective of the history that led to the board.

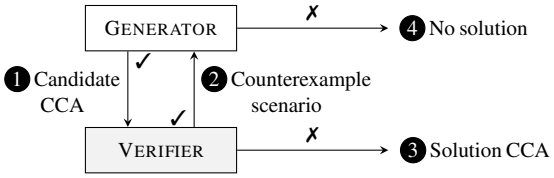


Figure 4: CEGIS loop (adapted from [1]).

5 CCmatic: Synthesizing CCAs

We use Counter-Example Guided Inductive Synthesis (CEGIS) [51] (a program synthesis technique) to synthesize (search for) CCAs given a specification (i.e., a network model and a performance property).

CEGIS iteratively generates a candidate CCA from a search space ❶, and finds a counterexample scenario (network path, initial state, and non-deterministic choices) that breaks the performance property for the candidate CCA ❷ (Fig. 4). The search space is pruned using the counterexamples (see below), and eventually the loop terminates if either (1) the verifier cannot find a counterexample (and thus, the CCA achieves the performance property) ❸, or (2) the entire search space has been pruned (no CCA in the search space can achieve the performance property) ❹.

We implement the generator and verifier using the constraint solver Z3 [47], by encoding the search inputs (i.e., search space, network models, and performance properties) into SMT (Satisfiability Modulo Theories) constraints in the theory of linear real arithmetic (LRA) [38]. In the generator, we only search for CCAs that pass unit checks (e.g. do not add bytes with seconds). For each counterexample, we add constraints to prune all CCAs that make the same sending rate choices as the candidate CCA on the counterexample. We explored encodings that prune more CCAs, these did not yield significant reduction in search time, and we omit their details.

For encoding into SMT, we use beliefs to define the search space (§5.1) and a transition system abstraction to systematically define performance properties (§5.2). For the network models, we adopt the encoding proposed by CCAC [7]. It discretizes time and uses Network Calculus [41] style formulas to constrain how the network serves packets. Note, the synthesis happens offline. One can directly implement the synthesized CCAs in network stacks like the Linux kernel and QUIC [40]. For completeness, we provide details on the implementation of the verifier and generator in Appendix C.

5.1 Belief-based CCA Template

In CEGIS, the search space is often specified using a template or grammar. The template has placeholders (or holes) that the generator fills to synthesize a concrete CCA. It describes the **inputs** (e.g., loss/delays signals) that the CCA takes and the mathematical/logical **operators** it can use to compute its outputs (i.e., rate and state). Due to Theorem 4.1, the templates do not need to describe state computations. They can just take beliefs as inputs and produce rate as the output

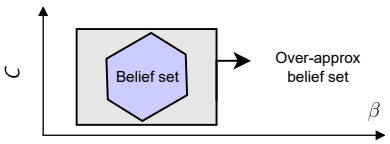


Figure 5: Over-approximating beliefs.

to fill the shaded box in Fig. 1.

Inputs. We face two challenges with beliefs. First, the belief set may be a complicated object in the ⟨`path`,`state`⟩ space. For easier encoding, we *over-approximate* it using closed-form expressions (see below). Second, our network models do not directly model variations in link rate (§3). When the link rate varies, it can go outside the belief set. To address this, we recompute beliefs using a recent history of observations (§5.1.2).

Over-approximating beliefs. We construct closed-form expressions representing bounds on the network parameters and states. For example, we use C_L and C_U to represent lower and upper bounds on the C values in the belief set, and pass these bounds as input to the CCA. By default, we pass upper and lower bounds on C (C_U, C_L), and $qdel$ ($qdel_U$, $qdel_L$) as a proxy for q. We describe how we compute them in §5.1.1. We assume the CCA knows R_m, and design CCAs for $D = R_m$ (i.e., jitter can be as large as R_m, e.g., due to WiFi ACK aggregation [29]). This enables efficient synthesis by discretizing time in units of R_m (as in CCAC [7]). In reality, a CCA does not know R_m. Nevertheless, we show in Appendix B that if we use a CCA designed for R_m equal to the minimum RTT seen thus far, we can guarantee performance because the synthesized CCAs are inherently robust to uncertainty in RTTs caused by jitter.

While the closed-form expressions may over-approximate beliefs, i.e., include extra paths that cannot produce the CCA's observations (Fig. 5), they never remove paths that can produce its observations. We use the verifier's (i.e., CCAC's) assistance to both validate correctness of the closed-form expressions and to derive them (§5.1.1).

Over-approximation (as opposed to under-approximation) does not break soundness. The CCA believes it could be running on extra paths, and needs to take actions that do not violate performance on the extra paths. However, over-approximation does break completeness, i.e., Theorem 4.1 no longer applies.[2] It may happen that we summarize two different histories using the same approximate beliefs even though the actual belief sets are different. A belief-based CCA is allowed to take different actions on the histories, but a CCA in the template is forced to take the same action. As a result, CCmatic may output "no solution in template" even though a belief-based CCA works. When this happens, we explore weaker network models where we can add additional beliefs (e.g., β_L, q_U) or tighten existing ones (§6.2.1). Despite approximations, CCmatic synthesizes novel CCAs (§6.1).

Operators. Some CCAs use nonlinear operators like cube root [31] or division [8]. Non-linearities slow down SMT solvers [58]. Instead, we search for piece-wise linear functions to map belief bounds to sending rates, represented as (nested) if-else statements. The generator synthesizes the conditionals and expressions in these statements as linear combinations of the belief bounds. While CCmatic only searches for CCAs in the template, we are able to generalize CCmatic's insights to arbitrary CCAs, e.g., impossibility results in §6.2.

[2] Note, Theorem 4.1 also does not apply because we only explore a subset of belief-based CCAs using CCmatic.

Listing 1: Belief-based CCA template

```
cond_i = ?C_U + ?C_L + ?qdel_U + ?qdel_L +
                ?MSS/R_m + ?R_m > 0
expr_j = ?C_U + ?C_L + ?MSS/R_m
if (cond_1):  rate = expr_1;
elif (cond_2):  ...
else:  rate = expr_n;
rate = max(rate, MSS/R_m)  # Ensure +ve rate.
```

Summary. Our templates take the form in Listing 1. ? denotes holes to be chosen by the generator. Different templates have different number (and nesting) of the "`if`" conditions. To keep the search tractable, we restrict the domain of the holes to be a small finite set, e.g., $\{-3,-5/2,-2...,3\}$.

5.1.1 Computing belief bounds

Queueing delay. Traditionally CCAs estimate queueing delay as $RTT - R_m$ [8]. However, this does not account for non-congestive delays. In the CCAC and CBR-delay models, $RTT(t) = R_m + qdel(t) +$ `jitter(t)`, where $0 \leq$ `jitter(t)` $\leq D$. Consequently, $qdel(t) \in [qdel_L(t), qdel_U(t)]$ where,

$$qdel_U(t) = RTT(t) - R_m$$
$$qdel_L(t) = \max(0, RTT(t) - R_m - D) \quad (5.1)$$

Link rate. BBR [15] estimates link rate as the measured ACK rate. Jitter can create transient variations in ACK rate and mislead this estimate. Using the verifier (CCAC), we computationally derive the set of link rates that can explain an average ACK rate of r (i.e., the CCA received rT bytes in an interval $[t_1, t_2]$ of length $t_2 - t_1 = T$ seconds). We also derive this set analytically in Appendix B.

We query CCAC for feasible values of r after fixing $\lambda(t) = \lambda$ (for all t). We repeat for different values of λ to obtain the bounds: $rT \in [C(T-D), C(T+D)]$ if $\lambda \geq C$, and $rT \in [\lambda(T-D), C(T+D)]$ otherwise. A CCA can be sure that $\lambda \geq C$ in two cases, (1) $RTT > R_m + D$ over the *entire* interval (or $qdel_L > 0$) (indicating non-zero queueing), or (2) there are losses in the interval (we verify this, see below). By inverting the bounds on r, we get bounds on C (algebraic steps in Appendix B):

$$C_L(t) = \max_{0 \leq t_1 \leq t_2 \leq t} \frac{r \cdot T}{T + D} \quad C_U(t) = \min_{0 \leq t_1 \leq t_2 \leq t} \frac{r \cdot T}{T - D} \quad (5.2)$$

Note, C_U is only computed over the intervals where $qdel_L > 0$ or loss > 0. We checked these calculations by asking CCAC if there is a trace and CCA (i.e., CCAC is free to choose $\lambda(t)$) for which $C \notin [C_L, C_U]$. CCAC returned UNSAT, confirming that no traces violate our calculation.

5.1.2 Handling stale beliefs

Our network models do not explicitly model variations in link rate. Such variations can make the beliefs **inconsistent (or stale)**, i.e., the network can take actions outside the belief set, leading the CCA to make bad decisions. For example, the link rate may decrease below C_L making the beliefs stale. The CCA may still transmit at rate C_L thinking that $C \geq C_L$, but cause losses due to the reduced link rate.

We "time out" beliefs periodically (e.g., every $10R_m$), and also when they become **empty (or invalid)**, e.g., $C_U < C_L$. On a timeout, we re-compute beliefs using the history of observations since the last timeout. When beliefs become inconsistent by a large margin, they become invalid quickly. However, if they are slightly inconsistent, they may remain valid. The periodic (speculative) timeouts helps make beliefs consistent in the second case. For example, say C decreases below C_L, i.e., $C = C_L - \varepsilon$ for some $\varepsilon > 0$. The beliefs become empty when $C_U < C_L$. $C_U = \frac{r \cdot T}{T - D}$, where r could be as large as $\frac{C \cdot (T+D)}{T}$. For $C_U < C_L$, we need $\frac{C(T+D)}{T-D} < C_L = C + \varepsilon$. This can take arbitrarily long time for arbitrarily small ε.

We retain our performance guarantees as long as *any* of the following holds: (1) the network parameters change infrequently so that they become consistent on the periodic timeouts, (2) parameters change by large margin, so that beliefs become invalid and timeout, or (3) parameters change within the belief set (i.e., remain consistent). We may violate our guarantees if the parameters *frequently* (e.g., every R_m) change to values just outside the belief set. However, since the beliefs are slightly off (e.g., 5%), our guarantees will also only be slightly off. Note, in general, frequent changes in network parameters is a hard problem for end-to-end congestion control due to feedback delay. Our CCAs react to changing network parameters at similar timescales as existing end-to-end CCAs (Appendix F, Appendix H).

Note, the periodic timeouts can interfere with a CCA's probes to estimate the belief set. We add constraints to prevent unnecessary timeouts. E.g., a CCA might be draining the queue and may not observe any delays/losses. If we recompute beliefs only using the recent history, there may be no upper bound on C, and the CCA would need to re-probe C_U from scratch. To prevent this, we only time out beliefs when the size of belief set is small (e.g., $C_U \leq 1.1C_L$), and we put bounds on how much the belief set can expand, e.g., $C_U(\texttt{now}) \leq 2C_U(\texttt{last_timeout})$. These restrictions along with the timeout period affect how quickly CCAs tracks changes in link rate. Appendix F (Lemma F.2) studies this theoretically and Appendix H (Fig. 19 and Fig. 18) studies this empirically.

5.2 Transition system based properties

The verifier uses bounded model checking to explore short snapshots of a CCA's execution under the network model. On such snapshots, metrics like long-term average utilization may be violated (e.g., CCA may take time to ramp up sending rate when the link rate increases). We use CCAC's approach to prove lemmas over the snapshots and stitch them using mathematical induction (on time) to prove properties about arbitrarily long executions. To do this stitching, we need to define what "progress" CCAs need to make in a transient period (e.g., CCA ramps up sending rate until it meets a utilization objective and then maintains utilization). CCACs approach to define progress is ad hoc and CCA dependent. This becomes unwieldy as the number of objectives and CCAs increase. To systematically state lemmas, we use a transition system abstraction. We use it to build (1) proofs about the performance of CCAs, and (2) invariants used for synthesis.

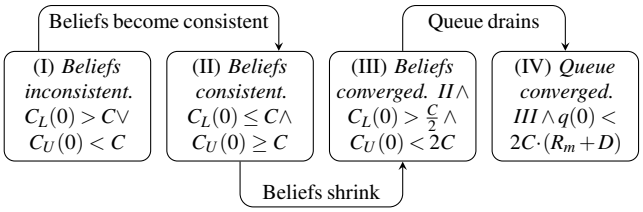


Figure 6: An example transition system. The formulas in the boxes (e.g., "$III \wedge q(0) < 2C \cdot (R_m + D)$") define the states.

Transition system. Fig. 6 shows an example transition system. The states are represented as symbolic boolean formulas (e.g., "beliefs are consistent", or $C_L(0) \leq C \leq C_U(0)$). For each state, users specify (1) transitions made in time 0 to T (e.g., "beliefs shrink": $C_L(T) > 1.5C_L(0)$) and (2) objectives during the period (e.g., "delay is at most D seconds": $\wedge_{t \in [0,T]} qdel(t) \leq D$). Note, users only declare "what" properties the CCA should ensure, CCmatic figures out "how".

A belief-based CCA under our network models typically makes the following transitions. Whenever the link rate changes, the beliefs can become inconsistent (stale). Eventually, the beliefs become consistent (due to §5.1.2), then the beliefs shrink (as this is necessary from §4), and finally, the CCA reaches the steady state (e.g., state IV) where it meets its steady-state objectives. Users can express both steady-state and transient objectives. For instance, high loss may be acceptable during slow start (e.g. in state I-III), but not for subsequent bandwidth probes (e.g., in state IV).

Proofs and encoding. We build a *lemma* for each state (e.g., State I $\implies$ (State I objectives $\wedge$ State I transitions)) and take the conjunction (logical and) of these lemmas. This conjunction serves as the *proof of performance* for the CCA, as it *exhaustively* describes the states that the CCA visits and performance it ensures in each state. It is exhaustive because the disjunction of the transition system states is a tautology, i.e., covers all possible states. Appendix F gives the encoding of the lemmas and shows how they work together in a proof.

The transitions happen eventually, i.e., may occur over multiple steps. Due to this, the encoding of lemmas needs to ensure that transition progress adds up over stitched executions. For instance, we encode "beliefs shrink" as "at least one of $C - C_L$ and $C_U - C$ decreases" *and* "neither

increases". The second literal, "neither increases", is required. Without it, both C_L and C_U can increase in one execution, and decrease in the subsequent execution and the progress does not add up. Such a CCA can meet the "at least one" criterion without ever transitioning to state III.

Synthesis invariant. We do not directly use the proof (i.e., the conjunction of lemmas) for synthesis. Instead, we build *under-specified* invariants that are *necessary* for a proof but not sufficient. We under-specify for two reasons. First, the lemmas contain constants that depend on the CCA, which is not known at the time of synthesis (e.g., $\mathbf{2}C$, $\frac{C}{\mathbf{2}}$, in Fig. 6). Second, before synthesis, we do not even know if there exists a CCA that can meet the lemmas. Under-specification allows us to synthesize reasonable-looking CCAs, which we then process *post synthesis* (see below).

To under-specify, we drop literals (inequalities) with unknown constants. Notice, that such literals define states III and IV (Fig. 6). Due to dropping, we cannot distinguish between states II-IV. We coalesce states II-IV into one, and allow the CCA to take any transition that is valid for states II-IV (effectively taking disjunction of the transitions). We similarly combine state III and IV objectives into `steady_state_obj`, and retain state II objectives as `transient_obj`. The result is Eq. 5.3 below. We use it for synthesizing CCAs with different choices of objectives (§6.1).

$$\begin{aligned}&\text{beliefs inconsistent} \implies \\ &\quad (\text{State I objectives} \wedge \text{beliefs become consistent}) \\ &\wedge \text{beliefs consistent} \implies (\texttt{transient_obj} \wedge (\text{beliefs shrink} \\ &\quad \vee \text{large queue drains} \vee \texttt{steady_state_obj}))\end{aligned} \tag{5.3}$$

Users are free to choose the degree of under-specification. They may drop literals (as we do), set loose bounds for the constants (e.g., $C_U(0) < 10C$ for state III), or even synthesize CCAs that meet specific constants.

Post synthesis. The synthesis invariant is not a sufficient proof. Hence, after synthesis, we build proof lemmas for the solution CCAs. To determine the constants in the lemmas, we use binary search to identify the region of values for which the lemmas hold. We describe this process in Appendix F. If there is no value of constants for which a particular lemma holds, we tweak the CCA, invariant, and/or the lemma(s). For example, when trying to build proof lemmas for a particular synthesized CCA (`cc_probe_slow` in §6.1), we found that the queue needs to be drained before beliefs can converge, i.e., "queue converged" in state IV (Fig. 6) needs to happen before "beliefs converged" in state III. So we reorder and redefine the states in the transition system and lemmas to reflect this and build a proof of performance for `cc_probe_slow`.

6 Results

We present four types of results. (§6.1) CCAs synthesized by CCmatic for various environments and objectives combinations. (§6.2) Fundamental tradeoffs inspired by negative outputs of CCmatic. (§6.3) Proofs that the synthesized CCAs ensure their performance objectives. (§6.4) Empirical evaluation of the synthesized CCAs to validate our mathematical modeling and proofs of performance.

6.1 Synthesis queries

A query describes the search inputs: (1) search space, (2) network model, (3) performance properties. Using CCmatic is an iterative process. One may realize that the performance properties are infeasible, the network model is too adversarial, or the approximations in the template are limiting. As we ran queries, our understanding improved, and we built new queries that better reflected our requirements (Table 2).

Objectives. We require CCAs to ensure a lower bound on the utilization, and upper bounds on the amount/frequency of losses and bytes in flight (to bound packet latencies). We add all these objectives to `steady_state_obj` in Eq. 5.3. Our primary focus is on exploring asymptotic bounds, e.g., are losses $O(C)$, $O(\log(C))$, etc. So we specify asymptotic bounds with loose constants. For example, we query CCAs that ensure utilization $\geq 50\%$, and inflight $\leq 5 \cdot C \cdot (R_m + D)$. The loose constants allow under-specifying the synthesis invariant (§5.2). Later, when building proofs, we identify the best constants the synthesized CCAs can achieve (Appendix F). Note, the inflight bound has to be at least $CR_m + CD$. Because, to provide a utilization lower bound, a CCA needs to fill the wire (CR_m bytes) and build D seconds or CD bytes of queueing (Theorem 2 of [6]).

We classify loss amount into *small* and *large*. Small means that loss is at most a few packets independent of C, or $O(1)$. Otherwise, loss is large, or $\omega(1)$, it increases with C. The verifier (SMT solver) explores traces numerically and there is no direct way to encode $\omega(1)$. As a workaround, we classify more than $3MSS$ loss as large. For frequency, we query CCAs that cause at most one small loss every other R_m, and never incur large losses. By default, we put these constraints in `steady_state_obj`, later we also put them in `transient_obj`, e.g., G4.

Environments. We sought CCAs that can handle paths with arbitrary buffer sizes and adversarial delay jitter as modeled by CCAC and CBR-delay (§2, §3). As stepping stones, we designed CCAs for restricted buffer sizes, viz. (1) infinite ($\beta = \infty$), (2) small fixed ($\beta = \frac{1}{2}CD$), and (3) large ($\beta \geq 3 \cdot C \cdot (R_m + D)$). These values formed interesting thresholds during our experimentation. We represent arbitrary buffer as $\beta \geq 0$, i.e., the verifier is free to choose the buffer size.

Templates and solutions. We synthesize *all* CCAs that satisfy a query. When CEGIS finds a solution, we prune it from the search space and let the loop continue until it cannot find more solutions. For each solution, we also prune CCAs that have same coefficients for the belief bounds but different coefficients for the constants (i.e., MSS/R_m, R_m) to avoid enumerating similar CCAs.

When CCmatic does not produce a solution, we increase (1) the program size (number of "`if`" expressions), and (2) the

	Environment		Objectives	Template		Trace Length (# R_m)	# Solutions	Time (secs)	# Itr (CEGIS)
	Network model (§3)	β	Constrain loss (`transient_obj`)	# Expr ("if")	Search size (# CCAs)				
G1	CCAC	Infinite	N/A	2	3×10^4	4	0	16	40
	CCAC	Infinite	N/A	2	3×10^4	5	4 CCA1	48	58
	CCAC	Infinite	N/A	2	3×10^4	11	6 CCA1 CCA3	7328	64
G2	CCAC	Small fixed	No	2	3×10^4	7	6 CCA2 CCA3	4942	49
G3	CCAC	Arbitrary	No	2	3×10^4	11	2 CCA3	7699	52
G4	CCAC	Small fixed	Yes	3	10^8	11	0	5067	79
	CCAC	Arbitrary	Yes	3	10^8	11	0	6851	76
G5	Ideal	Infinite	N/A	2	3×10^4	3	48	26	86
	Ideal	Small fixed	Yes	2	3×10^4	3	194	51	229
	Ideal	Arbitrary	Yes	2	3×10^4	3	23	21	67
G6	CBR-delay	Arbitrary	Yes	2	2×10^6	7	20 CCA4	19331	513
G7	CCAC	Large	No	2	3×10^4	11	6 CCA1 CCA3	9812	79
	CCAC	Large	Yes	2	3×10^4	11	4 CCA1	6884	45
	CCAC	Arbitrary	Yes only if $\beta \geq 3C(R_m+D)$	4	3×10^{11}	9	6 CCA5	46582	691

Table 2: Summary of queries. The "# solutions" column also lists CCAs that are representative of the solutions produced. Some entries in "constrain loss" column are not applicable (N/A) as (congestive) loss is not possible when buffer is infinite.

Listing 2: Synthesized CCAs ($\alpha_R = MSS/R_m$)

G1 CCA1 `cc_qdel`
```
if (qdel_L > 0):
    rate = 1/2C_L
else:
    rate = 2C_L
```

G2 CCA2 `cc_probe_fast`
```
if (C_U < 2C_L):
    rate = C_L
else:
    rate = 2C_L
```

G3 CCA3 `cc_probe_drain`
```
if (C_U > 2C_L − 2α_R):
    rate = 2C_L + α_R
else:
    rate = C_L − α_R
```

G6 CCA4 `cc_probe_slow`
```
if (q_U > MSS):
    rate = C_{L,λ} − q_U/R_m
else:
    rate = 2C_{L,λ} + α_R
```

G7 CCA5 `cc_probe_qdel`
```
if (C_U < 3/2C_L):
    if (qdel_L > R_m):
        rate = α_R
    else:
        rate = C_L
else:
    if (C_U > 2C_L):
        rate = 2C_L + α_R
    else:
        rate = C_L
```

length of the trace that the verifier considers (to give CCAs more time to show the progress required by our invariant). E.g., in G1, we get more solutions as we increase the trace length. For other queries we only show queries with the largest templates and longest traces. We also add new or tighter beliefs to the templates when we explore weaker models, e.g., G6.

Synthesized CCAs. Listing 2 shows the synthesized CCAs. We group similar queries together. G1, G2, and G3 explore CCAC with infinite, small and arbitrary buffer respectively. CCmatic synthesized CCA1 (`cc_qdel`), CCA2 (`cc_probe_fast`), and CCA3 (`cc_probe_drain`). CCmatic automatically figured out non-trivial insights about the network. E.g., `cc_qdel` simultaneously guarantees bounds on utilization and inflight. It sends above C_L until $qdel_L > 0$. $qdel_L > 0$ (Eq. 5.1) can only happen if queue is non-zero which can only happen if the link is utilized. Likewise, draining whenever $qdel_L > 0$ ensures inflight is bounded.

`cc_probe_fast` and `cc_probe_drain` "probe" when C_L and C_U are far. They send above C_L resulting in either increasing C_L (due to increase in ACK rate), or decreasing C_U (due to increase in *qdel* or losses). `cc_probe_drain` additionally drains queues by sending below C_L. `cc_probe_fast` does not drain as it was synthesized for a small buffer which trivially bound inflight.

We empirically evaluated `cc_probe_drain` and found that it incurs periodic large loss (experimental setup described in §6.4). This was surprising as we specifically queried for a CCA that avoids large losses in steady state. We realized this happened because of under-specifying the synthesis invariant. Due to the disjunction, "beliefs shrink $\vee\cdots\vee$ `steady_state_obj`", the CCA is allowed to cause large losses if this allows shrinking beliefs. As a result, on the periodic belief timeouts (§5.1.2), `cc_probe_drain` caused losses when re-probing to re-estimate beliefs (similar to BBR's 8 cycle probes).

G4 We updated our query to ensure that when beliefs are consistent, CCA's probes (increasing sending rate) should not incur large losses.[3] I.e., we "AND" the synthesis invariant with the formula: "beliefs consistent $\implies$ (sending rate increases $\implies$ no large loss)". This is equivalent to updating `transient_obj` in Eq. 5.3. CCmatic did not produce any solution after this modification.

G5 To dig deeper, we investigated weaker network models (§3). We set $D = 0$ to emulate an ideal link. CCmatic synthesized a CCA that sends at rate "$C_L + MSS/R_m$", allowing it to probe for bandwidth while risking at most constant losses. However, this does not work with CCAC. Since CCAC can delay packets, this probe may not lead to an immediate increase in ACK rate. For CCAC, a CCA needs to build D seconds of queueing to disambiguate effects of utilization (congestion) from non-congestive delays (see §6.2). Sending at $C_L + MSS/R_m$ takes $\Omega(C_L)$ time to build a queue of D seconds, and the same CCA cannot show progress (shrink

[3] Large losses may still happen when the link rate decreases. No CCA can avoid this due to feedback delay.

beliefs) in a short fixed-length trace.

G6 We ran synthesis for CBR-delay using the same beliefs as CCAC. CCmatic could not synthesize any CCA that could avoid large loss. This led us to discover and prove a fundamental tradeoff between loss and convergence time (§6.2). The proof led us to a tighter beliefs for CBR delay. Using these, CCmatic synthesized CCA4 (`cc_probe_slow`). `cc_probe_slow` meets the loss-convergence tradeoff implying that it is tight. It risks $O(1)$ packet loss and takes $O(C)$ time to converge. For the synthesis, we added belief bounds on link rate ($C_{L,\lambda}$), buffer size, and bytes in queue (q_U). We explain these bounds and working of `cc_probe_slow` in §6.2.1.

G7 There is no loss-convergence tradeoff when buffers are large. We want a CCA that converges fast without incurring large loss. Additionally, on shallow buffers, we want large loss to occur only when needed i.e., for probing (shrinking beliefs). CCAs already synthesized do not fit this bill. CCA1 (`cc_qdel`) avoids large losses when the buffer is large, but causes large losses on short buffers even when it is not shrinking beliefs. CCA3 (`cc_probe_drain`) avoids large loss when it is not shrinking beliefs, but causes large losses when probing even on large buffers. CCmatic synthesized CCA5 (`cc_probe_qdel`). It gets the best of `cc_qdel` and `cc_probe_drain`.

6.2 Loss vs. convergence tradeoff

In G4 and G6, CCmatic had failed to produce CCAs that risk at most constant loss on CBR-delay and CCAC. All human designed CCAs that we could think of also failed. On investigating the counterexample traces for the human and machine designed CCAs, we suspected that it is impossible to avoid large loss events. On trying to prove this, we discovered a tradeoff between amount of loss and *convergence time*, i.e., time it takes for a CCA to ramp up its sending rate to the link rate. **This tradeoff applies whenever the link rate increases, and the CCA needs to ramp up (including slow start).**

THEOREM 6.1. *For an end-to-end deterministic CCA running on a CBR-delay network with parameters $\langle C, R_m, D, 0 < \beta \leq CD \rangle$, to avoid getting arbitrarily low utilization, the CCA must either (1) cause $\omega(1)$ packet loss, i.e., losses that increases with C, or (2) take $\Omega(C(R_m + \beta_s))$ time to converge to the link rate. Where, $\beta_s = \beta/C$, i.e., buffer size in seconds.*

In general, for a CCA to ramp from C_0 to C while risking $O(f(C))$ loss, the convergence time is $\Omega(F^{-1}(C)(R_m + \beta_s))$, where F^{-1} is the inverse of F, and function F is defined as:

$$F(0) = C_0 \quad \text{and,} \quad F(k) = F(k-1) + f(F(k-1))/\beta_s$$

If C is not in the domain of F^{-1}, we evaluate F^{-1} at the smallest value greater than C in the domain of F^{-1}. The function $F(k)$ tracks the maximum rate a CCA can ramp up to in k *RTT*s under the loss allowance f. Correspondingly, $F^{-1}(C)$ gives the minimum number of *RTT*s needed to ramp up to rate C under loss allowance f.

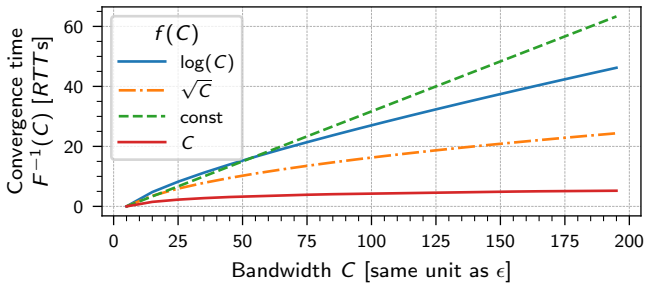


Figure 7: Each curve shows $F^{-1}(C)$ corresponding to $f(C)$ in the legend. I.e., under loss allowance f, the time F^{-1}, to converge from some small (positive) rate ε to C. Time from C_0 to C is given by "time from ε to C" − "time from ε to C_0" or $F^{-1}(C) - F^{-1}(C_0)$.

Fig. 7 shows the convergence time F^{-1} for different choices of loss allowances f. For example, if the CCA is willing to risk O(C) losses, i.e., $f(C) = C$, then the convergence time is $\Omega(\log(C/C_0))$. If $f(C) =$ `const` (independent of C), then the convergence time is $\Omega(C - C_0)$. For other functions f, the convergence time may not have a nice closed form expression.

While this tradeoff may seem intuitive, it only holds when there is complicated jitter. On ideal links, a CCA can ramp up in $O(1)$ time while risking $O(1)$ packet losses, using packet trains [39]. The inter-arrival times for a packet train is the inverse of the bottleneck rate (i.e., MSS/C) and reveals C. Such techniques may even work for links with iid (independent and identically distributed) jitter. However, in practice, links do exhibit complicated jitter patterns and break such bandwidth estimates [22].

Below, we provide intuition and outline the proof for Theorem 6.1. We also prove a similar loss-convergence tradeoff for ideal links when packet trains are not allowed and $\beta = 0$. This proof is similar to that of Theorem 6.1, we omit it for brevity.

Intuition & counterexample trace. The tradeoff is valid only on shallow buffers (i.e., $\beta \leq CD$). This renders *RTT* measurements meaningless, forcing the CCAs to rely on losses . On larger buffers, faster convergence is attainable by relating how *RTT* varies with varying sending rate. When $\beta \leq CD$, the queueing delays are at most D seconds. The delay box can choose jitter such that *RTT*s are at most $R_m + D$. Such *RTT*s could be due to queueing delays (resulting from varying sending rates), or due to jitter in the delay box, and there is no way for the CCA to distinguish between the two.

We investigated a CCA that additively increases its sending rate every *RTT* until it sees a loss, to see why it cannot avoid large loss. The verifier gave us a trace where the CCA keeps blindly increasing its sending rate even when it is already above the link rate. Eventually the queue builds up and $O(BDP)$ loss happens. The CCA needs to resort to blind increases because it does not get any feedback until it causes loss (as queueing delay measurements are meaningless). If these blind increases are aggressive, this results in larger losses with faster convergence and vice versa. `cc_probe_slow` drains queues along with

additive increments to meet the bound in Theorem 6.1 (§6.2.1).

Proof outline. (Full proof in Appendix D) We first show that due to $\beta \leq CD$, a CCA must cause loss to avoid arbitrarily low utilization (**Step 1.**). Then we compute a *tight* lower bound belief ($C_{L,\lambda}$) for C, for CBR-delay (belief computations in §5.1.1 were for CCAC) (**Step 2.**). The CCA could be running on any link with $C \geq C_{L,\lambda}$, and it needs to ensure it does not cause loss on any of these links. This allows us to compute the amount of loss a CCA risks any time it probes for bandwidth (**Step 3.**). If we restrict this risk of loss to a constant independent of C, it gives us a constraint on how quickly the CCA can ramp up, giving us a lower bound on the convergence time (**Step 4.**).

6.2.1 `cc_probe_slow` shows Theorem 6.1 is tight

We describe beliefs for CBR-delay, how `cc_probe_slow` works, and why `cc_probe_slow` does not work for CCAC.

Bandwidth and buffer beliefs. For CBR-delay, to obtain an upper bound on C, a CCA needs to cause loss or build more than D seconds of queueing (from **Step 1.** of Theorem 6.1). We compute the set of paths $\langle C, \beta \rangle$ that can produce CCA's observations until time t^*, such that the CCA has not observed $qdel > D$ or loss until t^*. This means that until $t^* - RTT(t^*)$, the enqueued bytes never exceeded the dequeued bytes by more than D seconds (CD bytes) or buffer size. I.e., $\forall t_1, t_2$, such that, $0 \leq t_1 \leq t_2 \leq t^* - RTT(t^*)$:

$$\int_{t_1}^{t_2} \lambda(s)ds - C\cdot(t_2 - t_1) \leq CD \tag{6.1a}$$

$$\text{and, } \int_{t_1}^{t_2} \lambda(s)ds - C\cdot(t_2 - t_1) \leq \beta \tag{6.1b}$$

From (6.1a), we define $C_{L,\lambda}$ as:

$$C_{L,\lambda}(t^*) = \max_{0 \leq t_1 \leq t_2 \leq t^* - RTT(t^*)} \frac{\int_{t_1}^{t_2} \lambda(s)ds}{t_2 - t_1 + D} \tag{6.2}$$

We evaluate (6.1b) over the interval $[t_1^*, t_2^*]$ (length $T^* = t_2^* - t_1^*$)[4] corresponding to the tightest bound on $C_{L,\lambda}$:

$$C_{L,\lambda}(t^*)\cdot(t_2^* - t_1^* + D) - C\cdot(t_2^* - t_1^*) \leq \beta$$

$$\text{or, } \quad C\cdot T^* + \beta \geq C_{L,\lambda}(t^*)\cdot(T^* + D) \tag{6.3}$$

From (6.2) and (6.3), we get the belief set illustrated in Fig. 8: $\{\langle C, \beta \rangle \mid C \geq C_{L,\lambda}(t^*) \wedge C\cdot T^* + \beta \geq C_{L,\lambda}(t^*)\cdot(T^* + D)\}$. **$C_{L,\lambda}$ tells two things, (1) the link rate is at least $C_{L,\lambda}$, and (2) if the link rate is $C_{L,\lambda}$, then the buffer is at least $C_{L,\lambda}D$ bytes.**

In the SMT encoding, we only evaluate Eq. 6.2 over intervals of length $T^* = R_m$. This is because the measurement interval T^* influences CCA's probing behavior (see below and Appendix E). The initial conditions (including initial beliefs) are chosen by the verifier (§3). As a result, verifier's initial choice of T^* influences the CCAs probing which should be in full control of the CCA. Fixing $T^* = R_m$ solves this issue.

Queue beliefs. For ease of discussion we define "probe interval" as an interval that leads to increase in $C_{L,\lambda}$ using Eq. 6.2. We show in Appendix E that probe intervals needs to be start with a drained queue, otherwise, the CCA risks losing all packets in the queue at the beginning of the probe interval. For the CCA to gauge the queue state, we add q_U as an upper bound on the bottleneck queue bytes. We compute it as: (bytes already in queue) + (enqueued bytes) − (dequeued bytes on smallest C). I.e., $q_U(t + \delta t) = \max(0, q_U(t) + (\lambda(t) - C_{L,\lambda}(t))\delta t)$.

[4] Other intervals may produce a tighter Eq. 6.3 but we over-approximate.

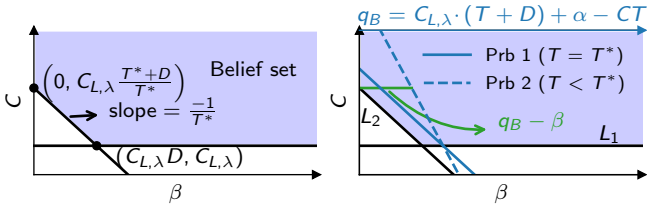


Figure 8: Belief set for CBR-delay (left). Queue buildup as a function of C (right). L_1, L_2 lines are the belief constraints.

In summary, the template now includes inputs as $C_{L,\lambda}$, $T^* = R_m$, and q_U. For brevity, we omit details on timeouts for these beliefs.

How `cc_probe_slow` works. We discuss a generalized version called `cc_probe_slow_k`. It drains queues until $q_U \leq \alpha$ (according to Listing 2). Then probes by sending $C_{L,\lambda}\cdot(T^* + D) + \alpha$ bytes paced over T^* time. α is a constant and $T^* = kR_m$ is the interval length over which the CCA measures $C_{L,\lambda}$. This probe ensures (1) progress (beliefs shrink), i.e., causes losses, $qdel > D$, or increases $C_{L,\lambda}$; while ensuring (2) loss $\leq 2\alpha$.

Progress. The probe builds queues. On the network with rate C, the queue build up is (enqueued bytes) − (dequeued bytes) $= C_{L,\lambda}\cdot(T^* + D) + \alpha - C\cdot T^*$ ("Prb 1" illustrates this in Fig. 8). On the network with rate $C_{L,\lambda}$, the queue build up is $C_{L,\lambda}D + \alpha$ (i.e., more than D seconds). If the CCA does not observe $qdel > D$, then it knows that C is higher than $C_{L,\lambda}$, and the line L_1 (Fig. 8) moves up. I.e., $C_{L,\lambda}$ becomes $\frac{C_{L,\lambda}\cdot(T^*+D)+\alpha}{T^*+D} = C_{L,\lambda} + \frac{\alpha}{T^*+D} > C_{L,\lambda}$. Likewise, if the CCA does not observe loss, then L_2 shifts right. If the CCA does observe $qdel > D$ or loss, then it knows that it sent above C and obtains an (implicit) upper bound on C, ensuring a lower bound on utilization.

Loss. Loss due to the probe is $\Delta L =$ (bytes already in queue) + (queue buildup) − (buffer) $= q + q_B - \beta$. In Fig. 8, the horizontal distance between a point (β, C) in the belief set and the line Prb 1 shows $q_B - \beta$. This distance, i.e., "$C_{L,\lambda}\cdot(T^* + D) + \alpha - CT^* - \beta$" is at most α (from Eq. 6.3). Bytes already in the queue is also at most α as probe only happens when ($q \leq$) $q_U \leq \alpha$. Hence, $\Delta L \leq 2\alpha$, i.e., constant independent of C.

Also notice, if we probe over a shorter interval $T < T^*$, then there are networks (β, C) for which $q_B - \beta$ is larger (Prb 2 in Fig. 8) and loss on those networks is $O(C)$. To avoid large loss, the probing interval T needs to be at least as long as the past measurement interval T^* (Appendix E formally proves this).

`cc_probe_slow` steady-state behavior. In steady state, `cc_probe_slow` follows a "Probe, Drain$_1$, Drain$_2$" cycle. Table 3 shows the result of this cycle on two paths in the belief set. These lie on the corners of belief set on the line L_2 (Fig. 8).

Path	Metrics	Probe	Drain$_1$	Drain$_2$
		Duration		
		T^*	$D=R_m$	$D=R_m$
	Sent by CCA	$C_{L,\lambda}\cdot(T^*+D)+\alpha$	α	$C_{L,\lambda}R_m-2\alpha$
	q_U	$C_{L,\lambda}D+\alpha$	2α	0
P_1	Serviced$_1$	$C_{L,\lambda}T^*$	$C_{L,\lambda}D$	$C_{L,\lambda}R_m$
	Queue$_1$	$C_{L,\lambda}D$	2α	0
	Loss$_1$	α	0	0
	Util$_1$	100%	100%	100%
P_2	Serviced$_2$	$C_{L,\lambda}\cdot(T^*+D)$	α	$C_{L,\lambda}R_m-2\alpha$
	Queue$_1$	0	0	0
	Loss$_2$	α	0	0
	Util$_2$	100%	$\approx$0%	$\frac{T^*}{T^*+D}\cdot$100%

Table 3: Steady-state behavior of `cc_probe_slow` on two extreme paths in the belief set: $P_1=\langle C=C_{L,\lambda},\beta=C_{L,\lambda}D\rangle$ and $P_1=\langle C=C_{L,\lambda}\frac{T^*+D}{T^*},\beta=0\rangle$. "Serviced" shows bytes serviced by the CBR box, and q_U is computed at the end of the probe/drain duration. Note, for drain$_1$, the CCA sends packets due to line 7 of the template (Listing 1), even though `rate`$=C_{L,\lambda}-q_U/R_m<0$.

Notice the two paths are very different, but CCA's observations are exactly the same. The loss is the same, and the delay box can ensure *RTT*s are the same (as $qdel\leq D$). In fact all paths on the line joining P_1 and P_2 can produce CCA's observations, and form the steady-state belief set. Utilization is lowest on P_2 and highest on P_1. The bandwidths of P_1 and P_2 differ by a factor of $\frac{T^*}{T^*+D}$. To bound inflight on P_1, CCA's average sending rate cannot be more than $C_{L,\lambda}$, as a result, the average utilization on P_2 cannot be more than $\frac{T^*}{T^*+D}$. The only way to increase utilization is to reduce uncertainty (size of belief set) by increasing T^*. This also increases convergence time (e.g. if the link rate increases when the CCA is in steady state).

If we replace α with $f(C_{L,\lambda})$, where $f(.)$ is the loss allowance (§6.2). Then the resulting family of CCAs (parameterized by $f(.)$ and T^*) allows us to tune the tradeoff between loss vs. convergence time vs. utilization. We can even adapt these over time (Appendix H).

Discussion. The probe works as CBR-delay ensures that if a probe did not cause loss in the past then repeating the probe will not cause a loss. This is not true for CCAC due to its *non-deterministic* token bucket filter (TBF). CCAC can arbitrarily decide how many tokens to keep in the bucket. A past probe may not have lost packets as the token bucket was full. However, on repeating the probe, CCAC can choose to keep the token bucket empty, and drop $O($`bucket`$)$ packets. Due to this, we conjecture that either large losses cannot be avoided for CCAC or will require asymptotically longer convergence time (e.g., quadratic instead of linear in *BDP*).

6.3 Proofs of performance

Ensuring the under-specified synthesis invariant is not a sufficient proof (§5.2). We summarize the lemmas that serve as the proof of performance for CCA1 (`cc_qdel`) and CCA4 (`cc_probe_slow`). These represent CCAs designed for deep and shallow buffers respectively. We give the full list and encoding of lemmas in Appendix F. Note, we describe theoretical worst-case bounds. Empirical performance is better (§6.4).

`cc_qdel`. On a CCAC link with parameters $\langle C,R_m,D=R_m,\beta\geq 3C\cdot(R_m+D)\rangle$, `cc_qdel` ensures that beliefs become consistent exponentially fast, they converge exponentially fast, and it drains queue at a rate proportional to C.[5] After beliefs are consistent, converged, and the queue is drained, the CCA is in steady state and remains in steady state. Note, this is despite the periodic beliefs timeouts (§5.1.2), i.e., the beliefs remain close to each other even after timing out. In steady state (state IV in §5.2), `cc_qdel` gets at least 89% utilization, keeps $RTT\leq 4.4(R_m+D)$ seconds, and loses at most 3 packets in any R_m duration. Additionally, `cc_qdel` never incurs large loss events when probing for bandwidth as long as the beliefs are consistent, i.e., in states II, III, IV.

`cc_probe_slow`. On the CBR-delay model with parameters $\langle C,R_m,D,\beta\rangle$, `cc_probe_slow` ensures that beliefs ($C_{L,\lambda}$, q_U) become consistent exponentially fast, and $C_{L,\lambda}$ converges additively. I.e., when the link rate decreases, `cc_probe_slow` ramps down exponentially fast. When the link rate increases, `cc_probe_slow` ramps up additively. In steady state, `cc_probe_slow` ensures at least 30% utilization, keeps $RTT\leq 1.5(R_m+D)$. It ensures that it loses at most 2 packets in any R_m duration whenever beliefs are consistent and the link rate has not decreased. Note, in §6.2.1, we showed steady-state utilization $\geq\frac{T^*}{T^*+D}100\%=50\%$ (for $T^*=R_m=D$). The proved worst-case utilization is lower because $C_{L,\lambda}$ may reduce on timeouts.

6.4 Empirical evaluation

Our goal with empirical evaluation is to validate our mathematical modeling and proofs of performance. Further evaluation is warranted before deployment.

Implementation. We implement CCA1 (`cc_qdel`) and CCA4 (`cc_probe_slow_k`, see §6.2.1 for "`_k`") over UDP using [54]. For `cc_probe_slow_k`, we run `cc_qdel` until a large loss event (resembling TCP slow start), and set $\alpha=5MSS$ instead of $1MSS$ to account for false-negatives in loss detection. As a result, probes may lose $2\alpha=10MSS$ bytes.

We initially implemented the CCAs in the Linux kernel, but found bugs in kernel's pacing implementation and the `cong_control` API to be insufficient (Appendix G). We compare against Cubic, BBRv1 (Linux kernel v5.4.0), BBRv2 [27], BBRv3 [28], and Copa [8, 54].

Scenarios and metrics. We use iperf to generate traffic and mahimahi [48] to emulate scenarios with jitter and shallow buffers (§2) with the aim of validating our performance proofs. We measure utilization, delay, loss, and convergence time metrics under a variety of parameter $\langle C,R_m,D,\beta\rangle$ choices. Each tuple constitutes a different "run". We emulate jitter by injecting

[5] A CCA cannot drain the queue faster than this. Even if it stops sending packets, the queue will only drain by $C\cdot t$ bytes in time t.

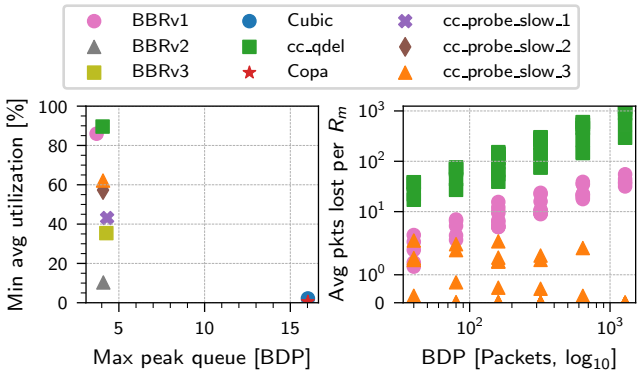


Figure 9: Summary of utilization, queue, and loss. Right subplot shows runs with $\beta \leq BDP$. Additionally, for visibility, we only show three CCAs, and use symlog scale on y-axis.

up to $D = R_m$ seconds of *uniformly random* delay, while ensuring FIFO service and constant average link rate (Appendix H).

Results. Fig. 9 summarizes utilization, queue (proxy for delay), and losses across runs (e.g., each run gives us an average utilization, and we compute the minimum across runs). Appendix H shows the metrics for each run and also studies convergence time and fairness. The synthesized CCAs are within their proven performance bounds and achieve tradeoffs between loss, convergence, utilization, and delay that prior CCAs cannot achieve.

`cc_probe_slow` meets the proven lower bound on utilization, upper bound on *RTT*s, and incurs at most constant loss (independent of BDP) across buffer sizes, bandwidths and propagation delays. On the same networks, Copa starves as it is not robust to jitter [7]. BBRv1 is able to ensure utilization despite random jitter, but incurs excessive losses (that increase with the *BDP*) on shallow buffers. Cubic gets low utilization when buffers are short ($\beta \ll BDP$) and also bloats queues when buffers are large ($\beta \gg BDP$). `cc_qdel`'s performance is similar to BBRv1, but with provable guarantees on utilization even with worst-case jitter. Unlike BBRv1, the v2 and v3 variants do not incur high losses on average but incur $O(BDP)$ losses to converge exponentially fast when the link rate increases (Appendix H). They also get lower utilization than the v1 variant. `cc_probe_slow_k` gets higher utilization at the cost of higher convergence time (Appendix H) with increasing `k`.

7 Related work

Our work extends [2], which introduced the concept of CCA synthesis using CEGIS. We propose the belief framework to make such synthesis far more practical and powerful. Related work not covered in §2 can be classified into:

Automatic CCA design. Past works have explored online learning [20], reinforcement learning [36, 43, 53], model predictive control (MPC) [32, 35], and (partially observable) markov decision process (POMDP) formulations [56]. CCAs produced by these works are not human-interpretable or are not explicitly designed for adversarial network behaviors. CCmatic CCAs are modular, human-interpretable, and provably robust under adversarial network behaviors.

Reasoning about CCAs. Past works use different network models: (1) deterministic (e.g. fluid model) [18, 62], (2) stochastic [30], and (3) non-deterministic [6]. Some also limit their scope to a small class of CCAs [9, 18, 62]. Deterministic models are easier to reason about but may not accurately reflect real world behaviors. With stochastic models, it is hard to deduce probability distributions that characterize real networks. Non-deterministic models do not require a distribution but may be too adversarial. Beliefs can be computed regardless of the modeling choice, and facilitate reasoning about complicated network models across all possible CCAs. Beliefs and CCmatic add to the emerging toolkit for performance reasoning.

8 Discussion, limitations, and future work

We built the belief framework allowing us to both (1) build novel CCAs and (2) prove tradeoffs between objectives. Using this, we built CCmatic to automatically synthesize CCAs for different environment/objective combinations, alleviating humans from figuring out complex details like when/how long to probe/drain. CCmatic also gives insights when objectives are infeasible. Due to formal methods and program synthesis, CCmatic CCAs are human-interpretable and provably performant.

While our work makes significant progress, it has several limitations that, if addressed, would bring us closer to "solving congestion control". First, we focus on single-flow scenarios. Designing provably fair CCAs will likely require a contract between CCAs allowing them to disambiguate effects of jitter from the actions of other flows. For instance, (1) flows could agree on a mapping between delay (or delay variation) and their sending rates [6], or (2) flows could agree on how much they can increase or decrease their sending rate in a single *RTT* ([18], Appendix H). Second, the CCAs synthesized for adversarial noise perform reasonably on ideal links. This may not hold as we explore other scenarios/objectives, and we may require extensions for average-case analysis. Third, we only explore a subset of belief-based CCAs due to computational limits. We hope to use techniques like robust adversarial reinforcement learning [50] and minimax [44] to improve design space exploration, given our 2-player game formulation in Theorem 4.1.

We believe our methodology (i.e., inverting environment models to build beliefs) is applicable to other domains where environment models exist. E.g., adaptive bitrate (ABR) algorithms and scheduling. ABR shares similar environments as CCAs, and recent work has built models for scheduling [5, 26].

Acknowledgments

We would like to thank anonymous reviewers and our shepherd Philip Brighten Godfrey for feedback that helped improve our paper. This work was supported in part by NSF grants CNS-2212102, and CNS-2212390.

References

[1] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. "Counterexample Guided Inductive Synthesis Modulo Theories". In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 270–288.

[2] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. "Automating Network Heuristic Design and Analysis". In: *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. HotNets '22. Austin, Texas: Association for Computing Machinery, 2022, pp. 8–16.

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. "Data Center TCP (DCTCP)". In: *Proceedings of the ACM SIGCOMM 2010 Conference*. SIGCOMM '10. New Delhi, India: Association for Computing Machinery, 2010, pp. 63–74.

[4] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. "Sizing Router Buffers". In: *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '04. Portland, Oregon, USA: Association for Computing Machinery, 2004, pp. 281–292.

[5] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. "Formal Methods for Network Performance Analysis". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 645–661.

[6] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. "Starvation in End-to-End Congestion Control". In: *Proceedings of the 2022 ACM SIGCOMM 2022 Conference*. SIGCOMM '22. Amsterdam, Netherlands: Association for Computing Machinery, 2022.

[7] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. "Toward Formally Verifying Congestion Control Behavior". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 1–16.

[8] Venkat Arun and Hari Balakrishnan. "Copa: Practical Delay-Based Congestion Control for the Internet". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 329–342.

[9] D. Bansal and H. Balakrishnan. "Binomial congestion control algorithms". In: *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*. Vol. 2. 2001, 631–640 vol.2.

[10] Anne Bouillard, Marc Boyer, and Euriell Le Corronc. *Deterministic Network Calculus: From Theory to Practical Implementation*. John Wiley & Sons, 2018.

[11] Patricia Bouyer, Stéphane Le Roux, Youssouf Oualhadj, Mickael Randour, and Pierre Vandenhove. "Games where you can play optimally with arena-independent finite memory". In: *Logical Methods in Computer Science* 18 (2022).

[12] L.S. Brakmo and L.L. Peterson. "TCP Vegas: end to end congestion avoidance on a global Internet". In: *IEEE Journal on Selected Areas in Communications* 13.8 (1995), pp. 1465–1480.

[13] Lloyd Brown, Yash Kothari, Akshay Narayan, Arvind Krishnamurthy, Aurojit Panda, Justine Sherry, and Scott Shenker. "How I Learned to Stop Worrying About CCA Contention". In: *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. HotNets '23. Cambridge, MA, USA: Association for Computing Machinery, 2023, pp. 229–237.

[14] Carlo Caini and Rosario Firrincieli. "TCP Hybla: A TCP Enhancement for Heterogeneous Networks". In: *Int. J. Satell. Commun. Netw.* 22.5 (Sept. 2004), pp. 547–566.

[15] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. "BBR: Congestion-Based Congestion Control". In: *ACM Queue* 14, September-October (2016), pp. 20–53.

[16] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, Ian Swett, Jana Iyengar, Victor Vasiliev, Priyaranjan Jha, Yousuk Seung, and Van Jacobson. *BBR Congestion Control Work at Google IETF 101 Update*. [Online; accessed 4. Mar. 2024]. Slides 4, 17. URL: https://datatracker.ietf.org/meeting/101/materials/slides-101-iccrg-an-update-on-bbr-work-at-google-00.

[17] Neal Cardwell, Yuchung Cheng, Soheil Hassas Yeganeh, Ian Swett, and Van Jacobson. *BBR Congestion Control*. [Online; accessed 26. Jun. 2023]. Mar. 2022. URL: https://datatracker.ietf.org/doc/html/draft-cardwell-iccrg-bbr-congestion-control-02.

[18] Dah-Ming Chiu and Raj Jain. "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks". In: *Computer Networks and ISDN Systems* 17.1 (1989), pp. 1–14.

[19] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. "Bounded Model Checking Using Satisfiability Solving". In: *Formal Methods in System Design* 19.1 (July 2001), pp. 7–34.

[20] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. "PCC: Re-architecting congestion control for consistent high performance". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 2015, pp. 395–408.

[21] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. "PCC Vivace: Online-Learning Congestion Control". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 2018, pp. 343–356.

[22] Constantinos Dovrolis, Parameswaran Ramanathan, and David Moore. "What do packet dispersion techniques measure?" In: *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*. Vol. 2. IEEE. 2001, pp. 905–914.

[23] M. Enachescu, Y. Ganjali, A. Goel, N. McKeown, and T. Roughgarden. "Routers with Very Small Buffers". In: *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. 2006, pp. 1–11.

[24] Tobias Flach, Pavlos Papageorge, Andreas Terzis, Luis Pedrosa, Yuchung Cheng, Tayeb Karim, Ethan Katz-Bassett, and Ramesh Govindan. "An Internet-Wide Analysis of Traffic Policing". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 468–482.

[25] Hugo Gimbert and Wiesław Zielonka. "Games Where You Can Play Optimally without Any Memory". In: *CONCUR 2005 - Concurrency Theory*. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 428–442.

[26] Saksham Goel, Benjamin Mikek, Jehad Aly, Venkat Arun, Ahmed Saeed, and Aditya Akella. *Quantitative Verification of Scheduling Heuristics*. 2023. arXiv: 2301.04205 [cs.LO].

[27] Google. *BBRv2*. [Online; accessed 1. Aug. 2023]. URL: https://github.com/google/bbr/tree/v2alpha.

[28] Google. *BBRv3*. [Online; accessed 14. Sep. 2023]. URL: https://github.com/google/bbr/tree/v3.

[29] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. "ABC: A Simple Explicit Congestion Controller for Wireless Networks". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 353–372.

[30] Prateesh Goyal, Mohammad Alizadeh, and Thomas E. Anderson. *Optimal Congestion Control for Time-varying Wireless Links*. 2022. arXiv: 2202.04321 [cs.NI].

[31] Sangtae Ha, Injong Rhee, and Lisong Xu. "CUBIC: A New TCP-Friendly High-Speed TCP Variant". In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 64–74.

[32] Cunwu Han, Dehui Sun, Lei Liu, and Song Bi. "A new robust model predictive congestion control". In: *Proceeding of the 11th World Congress on Intelligent Control and Automation*. IEEE. 2014, pp. 4189–4193.

[33] Tom Henderson, Sally Floyd, Andrei Gurtov, and Yoshifumi Nishida. *RFC 6582: The NewReno Modification to TCP's Fast Recovery Algorithm*. [Online; accessed 15. Aug. 2023]. Apr. 2012. URL: https://datatracker.ietf.org/doc/html/rfc6582.

[34] Janey C. Hoe. "Improving the Start-up Behavior of a Congestion Control Scheme for TCP". In: *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '96. Palo Alto, California, USA: Association for Computing Machinery, 1996, pp. 270–280.

[35] Amjad J Humaid, Hamid M Hasan, and Firas A Raheem. "Development of model predictive controller for congestion control problem". In: *feedback* 2 (2014), p. 3.

[36] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. "A deep reinforcement learning perspective on internet congestion control". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 3050–3059.

[37] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. "Planning and acting in partially observable stochastic domains". In: *Artificial intelligence* 101.1-2 (1998), pp. 99–134.

[38] Samuel Kolb, Stefano Teso, Andrea Passerini, and Luc De Raedt. "Learning SMT(LRA) Constraints Using SMT Solvers". In: *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. IJCAI'18. Stockholm, Sweden: AAAI Press, July 2018, pp. 2333–2340.

[39] K. Lai and M. Baker. "Measuring bandwidth". In: *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*. Vol. 1. 1999, 235–245 vol.1.

[40] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik

Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. "The QUIC Transport Protocol: Design and Internet-Scale Deployment". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 183–196.

[41] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer, 2001.

[42] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Vol. 2050. Springer Science & Business Media, 2001.

[43] Wei Li, Fan Zhou, Kaushik Roy Chowdhury, and Waleed Meleis. "QTCP: Adaptive congestion control with reinforcement learning". In: *IEEE Transactions on Network Science and Engineering* 6.3 (2018), pp. 445–458.

[44] G.F. Luger. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Pearson Addison-Wesley, 2009.

[45] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. "The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm". In: *SIGCOMM Comput. Commun. Rev.* 27.3 (July 1997), pp. 67–82.

[46] Tong Meng, Neta Rozen Schiff, P Brighten Godfrey, and Michael Schapira. "PCC proteus: Scavenger transport and beyond". In: *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 2020, pp. 615–631.

[47] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[48] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. "Mahimahi: Accurate Record-and-Replay for HTTP". In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, July 2015, pp. 417–429.

[49] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. "Modeling TCP Throughput: A Simple Model and Its Empirical Validation". In: *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '98. Vancouver, British Columbia, Canada: Association for Computing Machinery, 1998, pp. 303–314.

[50] Lerrel Pinto, James Davidson, Rahul Sukthankar, and Abhinav Gupta. "Robust Adversarial Reinforcement Learning". In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, Aug. 2017, pp. 2817–2826.

[51] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. "Sketching Concurrent Data Structures". In: *SIGPLAN Not.* 43.6 (June 2008), pp. 136–148.

[52] K. Tan, J. Song, Q. Zhang, and M. Sridharan. "A Compound TCP Approach for High-Speed and Long Distance Networks". In: *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. 2006, pp. 1–12.

[53] Chen Tessler, Yuval Shpigelman, Gal Dalal, Amit Mandelbaum, Doron Haritan Kazakov, Benjamin Fuhrer, Gal Chechik, and Shie Mannor. "Reinforcement Learning for Datacenter Congestion Control". In: *SIGMETRICS Perform. Eval. Rev.* 49.2 (Jan. 2022), pp. 43–46.

[54] venkatarun95. *genericCC*. [Online; accessed 9. Jun. 2023]. URL: https://github.com/venkatarun95/genericCC.

[55] Curtis Villamizar and Cheng Song. "High performance TCP in ANSNET". In: *ACM SIGCOMM Computer Communication Review* 24.5 (1994), pp. 45–60.

[56] Keith Winstein and Hari Balakrishnan. "TCP Ex Machina: Computer-Generated Congestion Control". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. Hong Kong, China: Association for Computing Machinery, 2013, pp. 123–134.

[57] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. "Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 459–471.

[58] Dongpeng Xu, Binbin Liu, Weijie Feng, Jiang Ming, Qilong Zheng, Jing Li, and Qiaoyan Yu. "Boosting SMT solver performance on mixed-bitwise-arithmetic expressions". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 651–664.

[59] Lisong Xu, K. Harfoush, and Injong Rhee. "Binary increase congestion control (BIC) for fast long-distance

networks". In: *IEEE INFOCOM 2004*. Vol. 4. 2004, 2514–2524 vol.4.

[60] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. "Pantheon: the training ground for Internet congestion-control research". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 731–743.

[61] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. "Adaptive Congestion Control for Unpredictable Cellular Networks". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: Association for Computing Machinery, 2015, pp. 509–522.

[62] Doron Zarchy, Radhika Mittal, Michael Schapira, and Scott Shenker. "Axiomatizing Congestion Control". In: *Proc. ACM Meas. Anal. Comput. Syst.* 3.2 (2019), 33:1–33:33.

[63] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. "Congestion Control for Large-Scale RDMA Deployments". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: Association for Computing Machinery, 2015, pp. 523–536.

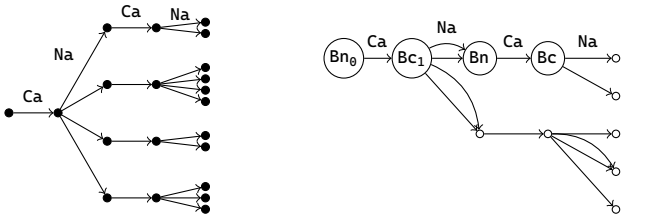
A Beliefs are sufficient

LEMMA A.1. *The set of feasible actions that the network is allowed to take in the future can be determined by the belief set.*

Proof. Recall, the only rule in our game is that the sequence of network's actions should be allowed by some path in the network model. As a result, a network action is feasible if and only if the sequence of past actions combined with the network action is allowed by some path in the network model. We will show that the network can compute the set of all feasible actions using the belief set.

We argue that the set of feasible actions is the set of actions that are allowed by some $\langle$path,state$\rangle$ tuple in the belief set. First, if an action $\mathcal{A}$ is allowed on path $\mathcal{P}$, and state $\mathcal{S}$ from the belief set. Then all the past network actions are allowed by path $\mathcal{P}$ (as $\mathcal{P}$ is in the belief set, and a path is in the belief set if all past actions can be explained by it). So the sequence of actions, including past and $\mathcal{A}$, is consistent with the path $\mathcal{P}$, hence $\mathcal{A}$ is a feasible action. Second, if an action $\mathcal{A}$ is not allowed on any $\langle$path,state$\rangle$ tuple in the belief set, then it cannot be a feasible action as the sequence of past actions and $\mathcal{A}$ cannot be explained by any single path in the network model.

□



(a) Game tree $\mathcal{T}$. **(b)** Condensed graph $\mathcal{G}$.

Figure 10: Constructing condensed graph from game tree.

LEMMA A.2. *Belief set for a future time can be computed by the belief set at the current time and the CCA's observations between the current and future times.*

Proof. We compute the future belief set as follows. Enumerate all tuples in the current belief set. Filter out the tuples that cannot produce the trace of CCA's observations between the current and future times. Filtering can be done by replaying the CCA's sending rate choices on the tuple, and checking if the tuple has feasible network actions that produce the exact observations of the CCA. The set of remaining tuples is same as the future belief set (as if it were computed using the entire history of CCA's observations). This is because a tuple is in the remainder set iff it can produce both (1) all CCA's observations till now (as it is in current belief set) and from now to the future time (as it was not filtered out). □

THEOREM. *If there exists a deterministic CCA that ensures a given performance property, on a given network model, then, there exists a* belief-based *CCA that ensures the performance property on the network model. Where, the beliefs are derived from the given network model, and the performance property is defined as a boolean valued function of the belief set and the action taken by the CCA on that belief set.*

Proof. We construct a belief-based CCA ($\mathcal{A}_\mathcal{B}$) by inspecting the executions of the given deterministic CCA ($\mathcal{A}_\mathcal{D}$), and show that $\mathcal{A}_\mathcal{B}$ ensures the given performance property ($\mathcal{P}_f$).

Preliminaries. An execution (or trace) is a sequence of CCA actions ($\texttt{Ca}_\texttt{i}$) and network actions ($\texttt{Na}_\texttt{i}$), e.g., $\langle \texttt{Ca}_1, \texttt{Na}_1, \texttt{Ca}_2, \texttt{Na}_2, \ldots \rangle$. We say that an execution is valid if it conforms to the rules of the game, i.e., the sequence of network actions correspond to some path in the network model.

We annotate each step of the execution by the beliefs computed over the entire history until that step, e.g., the belief set is $\texttt{Bc}_\texttt{i}$ after $\texttt{Ca}_\texttt{i}$, and $\texttt{Bn}_\texttt{i}$ after $\texttt{Na}_\texttt{i}$. At the start of all traces, the belief set is $\texttt{Bn}_0$. The annotation looks like: $\langle |_{\texttt{Bn}_0} \texttt{Ca}_1 |_{\texttt{Bc}_1} \texttt{Na}_1 |_{\texttt{Bn}_1} \texttt{Ca}_2 |_{\texttt{Bc}_2} \texttt{Na}_2 |_{\texttt{Bn}_2} \ldots \rangle$.

The performance property is defined as a function $\mathcal{P}_f$ that maps $\langle$Belief, CCA action$\rangle$ (e.g., $\langle \texttt{Bn}, \texttt{Ca} \rangle$) to `True` or `False`. While in §5.2 and §6.3, we do not directly state the performance properties this way, we show that they can be defined this way at the end of the proof.

Construction. The high level summary of the construction is that $\mathcal{A}_\mathcal{B}$ can arbitrarily pick one CCA action (sending rate choice) whenever $\mathcal{A}_\mathcal{D}$ takes different actions on the same belief set. As a result, $\mathcal{A}_\mathcal{B}$ is a pure function of belief set (it does not take different actions on the same belief). First, we will show that $\mathcal{A}_\mathcal{B}$ function is well-defined i.e., it assigns a rate choice to each belief Bn that it can witness in an execution. We will show this is because $\mathcal{A}_\mathcal{B}$ only ever reaches a belief set Bn if $\mathcal{A}_\mathcal{D}$ reached it. We will use Lemma A.2 and Lemma A.1 to show this. Second, we will show that $\mathcal{A}_\mathcal{B}$ satisfies the performance property because it only takes an action Ca on Bn if $\mathcal{A}_\mathcal{D}$ took it and so $\mathcal{P}_f(\langle \mathtt{Bn}, \mathtt{Ca} \rangle)$ evaluates to true for $\mathcal{A}_\mathcal{B}$ as it evaluated true for $\mathcal{A}_\mathcal{D}$.

We consider the set of all executions of $\mathcal{A}_\mathcal{D}$ on the given network model. We visualize them in the form of a (directed) game tree $\mathcal{T}$ (Fig. 10a). The nodes in $\mathcal{T}$ are placeholders. A unique node includes the entire history of the CCA and network actions until the node. If it is the turn of the CCA to play, the node has an outgoing edge labelled with the CCA action Ca. If it is the network's turn to play, the node has outgoing edges corresponding to all feasible network actions, each labelled with a network action Na. We label each node in the tree with the belief set computed over the history of actions until the node. We get two kinds of labels, (1) beliefs after CCA action (Bc), and (2) beliefs after network action (Bn).

From the tree, we construct a condensed graph $\mathcal{G}$ (Fig. 10b) as follows. Within each label type, we merge the nodes that have the same belief value (i.e., if two nodes have labels $\mathtt{Bn}^I$, $\mathtt{Bn}^{II}$, with $\mathtt{Bn}^I = \mathtt{Bn}^{II}$, we coalesce the nodes). Likewise for $\mathtt{Bc}^I = \mathtt{Bc}^{II}$. Note, we do not merge nodes with different label types even if they have the same belief value. Then for each node with a Bn label type (i.e., CCA's turn to play), we just keep one outgoing edge (arbitrarily) resembling the CCA action that $\mathcal{A}_\mathcal{B}$ takes on that belief value.

Note, in our construction of $\mathcal{G}$, we never remove edges corresponding to network actions, and all edges (both CCA and network actions) have the same destination node label in $\mathcal{T}$ and $\mathcal{G}$.

$\mathcal{A}_\mathcal{B}$ is the desired belief-based CCA. With the above construction, we will argue that (1) $\mathcal{A}_\mathcal{B}$ is well-defined, i.e., it assigns an action (outgoing edge) to each belief set that it can reach (ever witness in any execution), and (2) all actions of $\mathcal{A}_\mathcal{B}$ satisfy the performance property.

$\mathcal{A}_\mathcal{B}$ is well-defined. $\mathcal{A}_\mathcal{B}$ assigns a rate choice (CCA action) to each belief set that it can witness under the network model. We prove by contradiction. Say there is a valid execution e produced by $\mathcal{A}_\mathcal{B}$ that reaches a belief set $\mathtt{Bn}_\mathtt{i}$ for which $\mathcal{A}_\mathcal{B}$ never assigns an action, and $\mathtt{Bn}_\mathtt{i}$ is the first such belief in the execution. $\mathcal{A}_\mathcal{B}$ does not define an action for $\mathtt{Bn}_\mathtt{i}$ in two cases, (1) $\mathtt{Bn}_\mathtt{i}$ is not in $\mathcal{G}$, and (2) $\mathtt{Bn}_\mathtt{i}$ is in $\mathcal{G}$ but does not have an outgoing edge.

Case 1. We know $\mathtt{Bn}_\mathtt{i} \neq \mathtt{Bn}_\mathtt{0}$ because $\mathtt{Bn}_\mathtt{0}$ is in $\mathcal{G}$ (it is the root node of $\mathcal{T}$, and we do not remove any nodes when constructing $\mathcal{G}$). As a result, e looks like $\langle \cdots |_{\mathtt{Bn}_{\mathtt{i-1}}} \mathtt{Ca}_\mathtt{i} |_{\mathtt{Bc}_\mathtt{i}} \mathtt{Na}_\mathtt{i} |_{\mathtt{Bn}_\mathtt{i}} \rangle$, where, $\mathtt{Bn}_{\mathtt{i-1}}$ is in $\mathcal{G}$.

Since $\mathcal{A}_\mathcal{B}$ produced e, $\mathcal{A}_\mathcal{B}$ took action $\mathtt{Ca}_\mathtt{i}$ on the belief $\mathtt{Bn}_{\mathtt{i-1}}$, i.e., $\mathtt{Ca}_\mathtt{i}$ is the outgoing edge for node $\mathtt{Bn}_{\mathtt{i-1}}$ in $\mathcal{G}$. $\mathcal{A}_\mathcal{B}$ only takes this action, if $\mathcal{A}_\mathcal{D}$ took this action on a node with label $\mathtt{Bn}_{\mathtt{i-1}}$. We argue that in $\mathcal{T}$, this action leads to a node with label $\mathtt{Bc}_\mathtt{i}$. This comes from Lemma A.2, $\mathcal{A}_\mathcal{D}$ could have arrived from any history at the belief $\mathtt{Bn}_{\mathtt{i-1}}$, but taking the action $\mathtt{Ca}_\mathtt{i}$ on $\mathtt{Bn}_{\mathtt{i-1}}$ updates the belief to $\mathtt{Bc}_\mathtt{i}$ no matter the history. Thus, from our construction, $\mathtt{Bc}_\mathtt{i}$ also exists in $\mathcal{G}$ and is the destination node on the edge $\mathtt{Ca}_\mathtt{i}$ (as we do not remove nodes in the construction and the destination node labels are same in $\mathcal{G}$ and $\mathcal{T}$ for each edge).

Now, we argue that in $\mathcal{T}$, $\mathtt{Na}_\mathtt{i}$ is an outgoing edge for the node with label $\mathtt{Bc}_\mathtt{i}$. This is because according to e, $\mathtt{Na}_\mathtt{i}$ is a feasible network action on the belief $\mathtt{Bc}_\mathtt{i}$, as a result $\mathtt{Na}_\mathtt{i}$ is a feasible action on the node with label $\mathtt{Bc}_\mathtt{i}$ independent of history in $\mathcal{T}$ that led to this node (from Lemma A.1). Since the game tree describes *all* executions of $\mathcal{A}_\mathcal{D}$, each node with label type Bc has an edge for each feasible network action, so $\mathcal{T}$ has an edge with action $\mathtt{Na}_\mathtt{i}$ on the node with label $\mathtt{Bc}_\mathtt{i}$. Again from Lemma A.2, $\mathtt{Na}_\mathtt{i}$ takes belief from $\mathtt{Bc}_\mathtt{i}$ to $\mathtt{Bn}_\mathtt{i}$ in $\mathcal{T}$.

Since $\mathtt{Bc}_\mathtt{i}$, $\mathtt{Na}_\mathtt{i}$ and $\mathtt{Bn}_\mathtt{i}$ exist in $\mathcal{T}$, they also exist in $\mathcal{G}$. And in $\mathcal{G}$, $\mathtt{Na}_\mathtt{i}$ points from $\mathtt{Bc}_\mathtt{i}$ to $\mathtt{Bn}_\mathtt{i}$, and $\mathtt{Ca}_\mathtt{i}$ points from $\mathtt{Bn}_{\mathtt{i-1}}$ to $\mathtt{Bc}_\mathtt{i}$. This is because we never remove any edges corresponding to network actions, nor do we remove any nodes, and the destination nodes have same labels in $\mathcal{T}$ and $\mathcal{G}$. Since $\mathtt{Bn}_\mathtt{i}$ exists in $\mathcal{G}$, we arrive at a contradiction.

Case 2. $\mathtt{Bn}_\mathtt{i}$ exists in $\mathcal{G}$ only if it exists in $\mathcal{T}$. All nodes with label type Bn have a CCA action in $\mathcal{T}$. So $\mathtt{Bn}_\mathtt{i}$ has an action in $\mathcal{T}$. In constructing $\mathcal{G}$, we never remove all edges for a node with label type Bn, so we will keep at least one outgoing edge (action) for $\mathtt{Bn}_\mathtt{i}$. This is a contradiction.

$\mathcal{A}_\mathcal{B}$ satisfies $\mathcal{P}_f$. Recall, $\mathcal{P}_f$ is defined as a function of $\langle \mathtt{Bn}, \mathtt{Ca} \rangle$. From our construction, $\mathcal{A}_\mathcal{B}$ takes an action Ca on Bn only if $\mathcal{G}$ has edge Ca on Bn. This happens only if $\mathcal{T}$ has such and edge which happens only if $\mathcal{A}_\mathcal{D}$ takes Ca on Bn. Since, $\mathcal{A}_\mathcal{D}$ satisfies the performance property, $\mathcal{P}_f(\langle \mathtt{Bn}, \mathtt{Ca} \rangle) = \mathtt{True}$. Hence, all of $\mathcal{A}_\mathcal{B}$'s actions satisfy the performance property. □

Performance properties in §5.2 and §6.3 can be expressed as a boolean valued function of belief set and CCA's action on that belief set. At a high level, our performance properties dictate that the CCA (1) ensures some bounds on metrics like loss, delays, and utilization, and (2) makes progress. We show how both these can be expressed as a function of belief and CCA action.

We can compute delays, losses, and utilization for each tuple in the belief. The tuple tells us the starting queue, link rate, and buffer; and the CCA action tells us the sending rate. We can use this to calculate if the sending rate choice inflates delays or causes losses (e.g., using $\frac{dq}{dt} = (\lambda(t) - C)^+$, where $(x)^+ = \max(0, x)$). Likewise, we can compute if the link is utilized during the action using starting queue, link rate and the sending rate choice. If the CCA violates the delay, loss, utilization bounds on any tuple then the property evaluates to

`False` on the ⟨belief,CCA action⟩ pair, otherwise the property evaluates to `True`.

For progress, there are two types, (1) shrinking beliefs, and (2) stabilizing queue. For the first, we can check if an action can lead to shrinking beliefs by emulating all valid network actions. The CCA makes progress only if the beliefs shrink no matter the network action. For the second, the queue state is present in the belief tuple, and we can compute how the rate choice and link rate in the tuple will change the queue; and determine whether the queue will stabilize.

Note, our proof does not hold for arbitrary safety properties defined as boolean valued function over the entire execution. For instance, an arbitrary safety property might require that every time a CCA sends at rate 10 Mbps, in the next *RTT* it must send at 20 Mbps. Since we change the specific decisions that $\mathcal{A}_{\mathcal{B}}$ makes compared to $\mathcal{A}_{\mathcal{D}}$, such a safety property can be violated.

Also, note that we cannot express our performance properties solely as a function of the belief set. For example, the belief set does not tell us if the CCA will cause loss on an action, and we need to know the CCA's action on a belief set to evaluate loss. This is different from chess where the winning condition is just a function of the board state. Thus, Theorem 4.1 is non-obvious [11, 25].

B Computing beliefs

B.1 Propagation delay and jitter

We assume that the CCA knows R_m and D. Due to discretization of time in the SMT encoding, all quantities with units of time are integer multiples of the discretization interval. The time for synthesis and verification scales with the number of intervals considered (§6.1). To cover as many *RTT*s in as few intervals as possible, we set $R_m = D = 1$ interval. As a result the constant value 1 reveals R_m and D in the templates.

This is a non-issue as jitter inherently creates uncertainty in *RTT*s. CBR-delay with parameters $\langle R_m, D\rangle$ can emulate parameters $\langle R_m', D'\rangle$ as long as $R_m' \geq R_m$ and $R_m' + D' \leq R_m + D$. Now, say the actual network parameters are $\langle {R_m}^a, D^a\rangle$. From the first *RTT* measurement we set $R_m = RTT \in [{R_m}^a, {R_m}^a + D^a]$ (as there are no other flows, so no queuing). At this point, we run the CCA designed for $\langle R_m, D = R_m\rangle$ and get the performance guarantees for $\langle R_m, D\rangle$ as long as $D^a \leq R_m + D - {R_m}^a$. If some future RTT' is $< R_m$, we update $R_m = RTT'$ and repeat our argument, this time with better performance guarantees as R_m is closer to ${R_m}^a$. E.g., if we guarantee $q \leq R_m$ for $\langle R_m, D\rangle$, then on the actual network we guarantee $q \leq R_m \leq {R_m}^a + D^a$ and our guarantee improves as ${R_m}^a - R_m$ decreases. Same reasoning holds for CCAC.

In summary, we run the CCA designed for $\langle R_m = minRTT, D = minRTT\rangle$ and get the guarantee we promised for the network $\langle minRTT, minRTT\rangle$, when we are actually running on the network $\langle {R_m}^a, D^a\rangle$. We get these guarantees as long as $D^a \leq 2minRTT$, which is true if the real network can be captured by the CBR-delay model with parameters $D^a \leq R^a$.

Listing 3: CCAC constraints on S

$$S(t) \leq T_A(t) = S_U(t) \quad \text{(B.1)}$$
$$S(t) \leq A(t) - L(t) \quad \text{(B.2)}$$
$$S(t) \geq T_A(t-D) = S_L(t) \quad \text{where,} \quad \text{(B.3)}$$
$$T_A(t) = C \cdot t - W(t) \quad \text{(B.4)}$$
$$q(t) \geq 0 \implies W'(t) = 0 \quad \text{(B.5)}$$
$$q(t) = A(t) - L(t) - T_A(t) \quad \text{(B.6)}$$

$T_A(t)$ – cumulative tokens (in units of bytes) admitted
$q(t)$ – inst. bytes in the bottleneck queue
$W(t)$ – cumulative tokens (in units of bytes) wasted
$L(t)$ – cumulative bytes lost

B.2 Bandwidth

We show the analytical derivation of the link rate belief bounds (C_L and C_U) for the CCAC model. Before that we briefly give relevant background on CCAC (for detailed background please see [7]).

Background on CCAC. CCAC models the network as a generalized token bucket filter. It puts constraints (Listing 3) on how the network serves packets (i.e., $S(t)$, service curve, or cumulative bytes serviced by time t), based on how packets arrive into the network (i.e., $A(t)$, cumulative bytes arrived by time t).

Tokens arrive at rate C bytes/secs, i.e., $C \cdot t$ tokens arrive by time t. On a token arrival, the network decides whether to admit it or waste it (Eq. B.4). The network uses admitted tokens to send packets, so the packets sent are upper bounded by the number of admitted tokens (Eq. B.1). Also, the network cannot send more bytes than arrived but not lost (Eq. B.2).

To emulate non-congestive jitter, the network can choose to delay sending packets even when tokens are available. To ensure jitter is bounded, all tokens must be used within D seconds, and the network should only admit a token if it knows it will be used within D seconds.[6] This puts a lower bound on service (Eq. B.3).

Additionally, to prevent the network from wasting all the tokens, tokens cannot be wasted when there are packets waiting for tokens, i.e., there are packets that have arrived (but not lost) and do not have corresponding admitted tokens (Eq. B.5 and Eq. B.6). These packets are put into a queue, and they build congestive delays. Other packets that have corresponding tokens but have not been delivered as considered as facing non-congestive delays.

Analytical derivation. CCAC visualizes its constraints using graphs like Fig. 11. From such graphs, we can determine the minimum and maximum average ACK rate (r) that a CCA

[6] Due to non-determinism, the network can look into the future to make these decisions.

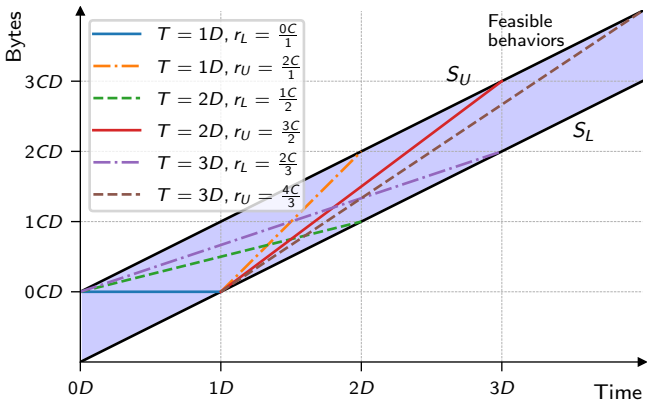


Figure 11: Computing ACK rate range based on C. ACK rate is the slope of the lines. Over different time intervals (T), the lines show feasible service curves with the maximum (r_U) and minimum (r_L) slopes. Service curves need to be non-decreasing and lie in the shaded region.

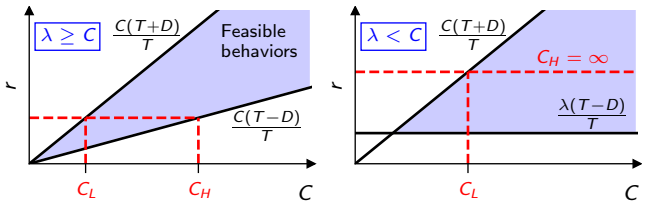


Figure 12: Inverting bounds on observations to get bounds on network parameters. For observed ACK rate, draw horizontal line (red dashed line) corresponding to the observed ACK rate r, the points intersecting with the feasible region give the range of link rates $[C_L, C_U]$ that could have produced r.

can get for different λ and C choices. Specifically, consider the case when $\lambda \geq C$. In this case, tokens cannot be wasted ($W' = 0$). As a result S_L and S_U have slope C, and the network has to ensure that at all times S is between the S_L and S_U lines. We look at different feasible choices of S, and find the maximum and minimum slope of S (ACK rate) over time intervals of different lengths. Note, we are interested in the "average" ACK rate over the intervals, so we look at the slope of the line joining the start and end points of the interval.

Fig. 11 shows different feasible S curves along with minimum and maximum slopes in different intervals. For example, over an interval of length D, $r \in [0, C]$. Similarly, $r \in [C/2, 3C/2]$ for $2D$ long intervals, $r \in [2C/3, 4C/3]$ for $3D$ long intervals. For kD long intervals, $r \in [\frac{k-1}{k}C, \frac{k+1}{k}C]$. Following a similar exercise, we find that for intervals with length T, $r \in [\frac{C(T-D)}{T}, \frac{C(T+D)}{T}]$. I.e., S follows an ideal link with rate C with an additional burst or delay of CD bytes.

We repeat this for the case when $\lambda < C$. Here, we consider two extreme cases: (C1) S_L, S_U have slope λ, and (C2) S_L, S_U have slope C. The first case corresponds to the network wasting tokens because packets arrive slower than tokens, and the second corresponds to a large queue build up at time $t = 0$ preventing the network from wasting tokens. When we inspect the graphs (not shown, similar to Fig. 11), we find that over intervals of length T, $r \in [\frac{\lambda(T-D)}{T}, \frac{\lambda(T+D)}{T}]$ in case C1, and $r \in [\frac{C(T-D)}{T}, \frac{C(T+D)}{T}]$ in case C2. When we look at other cases between C1 and C2, i.e., S_L and S_U take slopes in the range $[\lambda, C]$, we get more feasible values of r. Taking the union of all the r ranges, we find that in any T long interval, whenever $\lambda < C$, $r \in [\frac{\lambda(T-D)}{T}, \frac{C(T+D)}{T}]$.

We invert the bounds on r to get bounds on C. Fig. 12 illustrates this. Specifically, when $\lambda > C$, we have, $\forall T$:

$$\frac{C \cdot (T-D)}{T} \leq r \leq \frac{C \cdot (T+D)}{T}$$

On rearranging, we get:

$$\frac{r \cdot T}{T+D} \leq C \leq \frac{r \cdot T}{T-D} \tag{B.7}$$

Likewise, when $\lambda < C$, we have, $\forall T$:

$$\frac{\lambda \cdot (T-D)}{T} \leq r \leq \frac{C \cdot (T+D)}{T}$$

On rearranging, we get:

$$\frac{r \cdot T}{T+D} \leq C \tag{B.8}$$

From Eq. B.7 and Eq. B.8, we get:

$$C \geq \max_T \frac{r \cdot T}{T+D} = C_U \quad \text{and} \quad C \leq \min_T \frac{r \cdot T}{T-D} = C_L$$

Where C_U can only be computed over intervals where $\lambda > C$.

The CCA can compute C_L and C_U as defined. r can be measured directly by the CCA. The CCA can also infer if $\lambda(t) > C$ at all time steps t in an interval if $(qdel_L(t) > 0) \vee (L'(t) > 0)$ for all time steps t in the interval. We checked that this condition holds by querying CCAC if $C > C_U$ can happen when we compute C_U over intervals where $(qdel_L(t) > 0) \vee (L'(t) > 0)$ is true. CCAC returned UNSAT implying that C has to be $\leq C_U$.

C Synthesis details

For completeness, we describe the workings of the generator and verifier in CEGIS (§5). For ease of understanding, we interpret the search (or program synthesis) problem as a $\exists\forall$ formula [1]. For instance, "does there *exist* a belief-based CCA, such that *forall* traces captured by the network model, the CCA ensures the desired performance properties". More formally, the formula is:

$$\exists\, \boxed{?}\, \forall \texttt{trace}\, (\texttt{CCA} \wedge \texttt{Network}) \implies \texttt{Performance} \tag{C.1}$$

$\boxed{?}$ and `trace` represent vector of variables. $\boxed{?}$ are the holes in the CCA template (Listing 1). Assigning value to the holes produces a concrete CCA. A `trace` (timeseries) describes the execution of the CCA under the network model. It is specified using the dimensions of the network model's relation (§3), e.g., path, feedback, CCA actions. In our case, the `trace` includes variables like C, β, R_m, D, $rate(t)$, $A(t)$, $S(t)$, $L(t)$, $T_A(t)$, $W(t)$ (Table 1, Listing 3).

The `CCA`, `Network`, and `Performance` are boolean valued SMT formulas (in the theory of linear real arithmetic (LRA) [38]) over the $\boxed{?}$ and `trace` variables. `CCA` ensures that the congestion control decisions are made according to the

CCA. It encodes that the $rate(t)$ variables are assigned using the values of ? and `trace` variables according to the CCA template. We produce this encoding by symbolically executing the CCA template. `Network` encodes what assignments to the `trace` variables result in executions that are deemed feasible according to the network model. These look like Listing 3. We also encode belief computations (§5.1.1) and timeouts (§5.1.2) in `Network`, these constraints merely populate the belief bounds and don't affect the actions that the network takes. `Performance` encodes the desired objectives using the transition system. We use the synthesis invariants (Eq. 5.3) to specify `Performance`.

The formula can be read as "does there exist an assignment to the holes in the template such that for all traces, if the packets are sent according to the CCA and the service/delay/loss decisions are made according to the network model, then the trace satisfies the desired performance properties".

Verifier operation. For a given CCA (i.e., value of ?), the verifier produces a counterexample `trace` by solving the formula:

$$? = \texttt{value of } ? \wedge \texttt{CCA} \wedge \texttt{Network} \wedge \neg \texttt{Performance} \quad \text{(C.2)}$$

This is a quantifier-free formula (i.e., no $\exists$ or $\forall$ quantifiers). The verifier uses an SMT solver (e.g., Z3 [47]) to solve this formula. This is a formula on the `trace` variables (as the ? variables have been substituted by fixed values). The assignment to the `trace` variables describes a trace where the given CCA violates the performance properties on the network model. If the formula is unsatisfiable, then there is no counterexample that breaks the CCA.

Generator operation. Given a set of counterexample traces (say $\mathbb{X}$), the generator solves the following formula to propose a new candidate CCA:

$$\bigwedge_{\texttt{trace} \in \mathbb{X}} (\texttt{CCA} \wedge \texttt{Network}) \implies \texttt{Performance} \quad \text{(C.3)}$$

This is a formula on the ? variables (as the `trace` variables have been substituted by concrete counterexamples from the set $\mathbb{X}$). The assignments to the ? variables that satisfy the formula are those on which either the `trace` is no longer feasible according to the CCA/network model, or it satisfies the performance properties.

Note, in this formulation, only `CCA` depends on the holes, the `Network` and `Performance` only depend on the `trace` variables. On each `trace` in $\mathbb{X}$, `Network` evaluates to `True` and `Performance` evaluates to `False` (as the `trace` was generated by the verifier by satisfying "$\cdots \wedge$ `Network` $\wedge$ `Performance`"). As a result, Eq. C.3 simplifies to:

$$\bigwedge_{\texttt{trace} \in \mathbb{X}} \neg \texttt{CCA} \quad \text{(C.4)}$$

Effectively, the new candidate should not produce the exact same trace of rate choices as made by the prior candidate CCAs. So all the CCAs that have the same buggy control flow exploited by the counterexample trace are pruned from the search space.

D Loss vs. convergence tradeoffs

Tradeoff theorem and proof. For ease of understanding, we show the proof for a CCA trying to ramp up its rate from 0 to the link rate C, while trying to avoid large loss events. Later we generalize this to CCA trying to ramp up from arbitrary C_0 to C, while trying to risk at most $O(f(C))$ losses for some function $f(.)$.

THEOREM. *For an end-to-end deterministic CCA running on a CBR-delay network with parameters $\langle C, R_m, D, 0 < \beta \leq CD \rangle$, to avoid getting arbitrarily low utilization, the CCA must either (1) cause $\omega(1)$ packet loss, i.e., losses that increases with C, or (2) take $\Omega(C(R_m + \beta/C))$ time to converge to the link rate.*

Proof. We will first show that under the parameters of the proof, i.e., $\beta \leq CD$, the CCA must cause loss to avoid arbitrarily utilizing the link (**Step 1.**). Then we compute a tight lower bound belief for C, under the CBR-delay link (our prior belief computations were for CCAC) (**Step 2.**). This allows us to compute the amount of loss the CCA risks any time it tries to probe for bandwidth (**Step 3.**). If we restrict this risk of loss to a constant independent of C, it gives us a constraint on how quickly the CCA can ramp up, giving us a lower bound on the convergence time (**Step 4.**).

For the proof, we assume the CCA knows R_m, D, and β_s. β_s is the seconds of queueing that the buffer can tolerate (i.e., β/C). Knowledge of these quantities only makes the proof stronger. If CCAs need to respect the tradeoff with the knowledge, then they also need to respect it without the knowledge. Note, `cc_probe_slow` meets the theorem bounds without knowing β_s.

Step 1. Since $\beta_s \leq D$, the CCA must cause loss to avoid arbitrarily low utilization. This immediately follows from Theorem 2 of [6], i.e., to avoid arbitrarily low utilization, the CCA must cause more than D queueing delay. If the buffer size is $\leq D$ seconds, the CCA will have to cause loss. For completeness, we repeat the proof in Appendix D.1.

Step 2. Until the time that loss happens, we compute the set of paths (link rates) that the CCA could be running on given its observations. If the CCA has not seen a loss event by time t^*, then the CCA never over-flowed the buffer until time $t^* - RTT(t^*)$. I.e., in any time interval before $t^* - RTT(t^*)$[7], the net bytes enqueued and the net bytes dequeued differ by at most β. I.e., $\forall t_1, t_2$, such that, $0 \leq t_1 \leq t_2 \leq t^* - RTT(t^*)$:

$$\int_{t_1}^{t_2} \lambda(s) ds - C \cdot (t_2 - t_1) \leq \beta$$

Substituting $\beta = C\beta_s$ and rearranging, we get, $\forall t_1, t_2$. $0 \leq t_1 \leq t_2 \leq t^* - RTT(t^*)$:

$$C \geq \frac{\int_{t_1}^{t_2} \lambda(s) ds}{t_2 - t_1 + \beta_s}$$

[7] Note, the CCA only knows what happened in the network one RTT ago. At time t, the CCA has no information about what may have happened in the network during the time interval $(t - RTT(t), t]$.

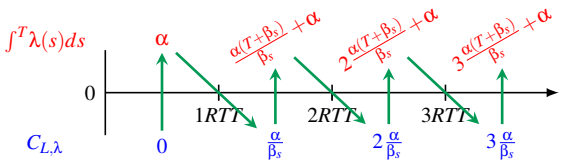


Figure 13: $C_{L,\lambda}$ puts constraints on $\int^T \lambda(s)ds$, and $\int^T \lambda(s)ds$ puts constraints on $C_{L,\lambda}$. Over the intervals, the red and blue values show the maximum value of $\int^T \lambda(s)ds$ and $C_{L,\lambda}$ respectively. The green arrows show the constraint dependencies. I.e., $C_{L,\lambda}$ in $[kRTT,(k+1)RTT)$ puts constraints on $\int^T \lambda(s)ds$ in $[kRTT,(k+1)RTT)$ and $\int^T \lambda(s)ds$ in $[kRTT,(k+1)RTT)$ puts constraints on $C_{L,\lambda}$ in $[(k+1)RTT,(k+2)RTT)$.

Based on this we define $C_{L,\lambda}$ as:

$$C_{L,\lambda}(t^*) = \max_{0\le t_1\le t_2\le t^*-RTT(t^*)} \frac{\int_{t_1}^{t_2}\lambda(s)ds}{t_2-t_1+\beta_s} \qquad \text{(D.1)}$$

The set of paths that can produce the trace up to t^* is $\{\langle C=C^*, \beta = C^*\beta_s\rangle | C^* \ge C_{L,\lambda}(t)\}$. The CCA could be running on any of these paths, and it needs to ensure its performance properties no matter which of these paths it is running on.

Step 3. On the path $\langle C^*, C^*\beta_s\rangle$, loss happens whenever the CCA over-flows the buffer, i.e., if the enqueued and dequeued bytes differ by more than the buffer size in some time interval. I.e., loss happens if for some interval $[t_1,t_2]$,

$$\int_{t_1}^{t_2}\lambda(s)ds > C^*\cdot(t_2-t_1)+C^*\cdot\beta_s$$

Specifically, on the path $\langle C_{L,\lambda}(t), C_{L,\lambda}(t)\beta_s\rangle$, i.e., the smallest link rate that can justify CCA's observations, the amount of loss is: $\Delta L(t_0,t) = \int_{t_0}^{t}\lambda(s)ds - \left(C_{L,\lambda}(t)\cdot(t-t_0)+C_{L,\lambda}(t)\cdot\beta_s\right)$

Step 4. Convergence can only happen after the CCA causes loss (until loss the CCA could be running on arbitrarily large link rate from **Step 1.**). It needs to do this while ensuring that the amount of loss it risks is bounded, i.e., for all intervals $0\le t_1\le t_2, \Delta L(t_1,t_2)\le\alpha$ for some constant α independent of C. We compute a lower bound on the time to loss under this constraint.

Loss happens only if buffer overflows, i.e., in some interval $[t_1^*,t_2^*]$ $\int_{t_1^*}^{t_2^*}\lambda(s)ds > C\cdot(t_2^*-t_1^*)+C\cdot\beta_s$. We also know that CCA needs to ensure $\Delta L(t_1^*,t_2^*)\le\alpha$. From this, loss happens only if $C_{L,\lambda}(t_2^*)\ge C-\alpha/\beta_s$. See Appendix D.1 for details.

We compute time for $C_{L,\lambda}$ to increase to $C-\alpha/\beta_s$. Initially $C_{L,\lambda}(0)=0$. Since $C_{L,\lambda}$ can only be computed after the first RTT (from the definition of $C_{L,\lambda}$), $C_{L,\lambda}$ is 0 over the entire interval $[0,RTT)$. This puts constraints on sending rate choices in $[0,RTT)$. Specifically, substituting $C_{L,\lambda}=0$ in $\Delta L\le\alpha$, we get, $\int_{t_1}^{t_2}\lambda(s)ds\le\alpha$ for all intervals $[t_1,t_2]$ inside $[0,RTT)$. Thus, if we compute $C_{L,\lambda}(t)$ at any time $t\in[RTT,2RTT)$, we get $C_{L,\lambda}(t)\le\frac{\alpha}{\beta_s}$ (because on intervals $0\le t_1\le t_2<2RTT-RTT$, i.e., $t_2<RTT$, the numerator in Eq. D.1, "$\int_{t_1}^{t_2}\lambda(s)ds$", is at most α).

We can similarly obtain that in interval $[kRTT,(k+1)RTT)$, $C_{L,\lambda}$ is at most $k\cdot(\alpha/\beta_s)$ (illustrated in Fig. 13). For $C_{L,\lambda}$ to ramp up to $C-\alpha/\beta_s$, we need $k\ge C\cdot(\beta_s/\alpha)-1$. Hence, the convergence time is at least k RTTs, or "$C\cdot(\beta_s/\alpha)-1$ RTTs". Before loss, each RTT can be as large as $R_m+\beta_s$ (either due to queueing delay or jitter), making the convergence time $=(C\cdot(\beta_s/\alpha)-1)\cdot(R_m+D)=\Omega(C(R_m+\beta_s))$.

□

Generalizing Theorem 6.1. If the CCA wants to ramp up from C_0 to C while ensuring its risks at most $f(C)$ loss, then the convergence time is $\Omega(F^{-1}(C)(R_m+\beta_s))$, where F^{-1} is the inverse of F and F is defined by the recursion:

$$F(0)=C_0 \quad \text{and,} \quad F(k)=F(k-1)+f(F(k-1))/\beta_s$$

If C is not in the domain of F^{-1}, we evaluate F^{-1} at the smallest value greater than C in the domain of F^{-1}.

We obtain this result by replacing $C_{L,\lambda}(0)=C_0$ in **Step 4.**, and ensuring that risk of loss $\Delta L\le f(C_{L,\lambda})$. The function $F(k)$ tracks the maximum possible value of $C_{L,\lambda}$ at any time in the interval $[kRTT,(k+1)RTT)$.

D.1 Proof details

We fill in the details skipped in the proof.

Step 1. We will prove below that the CCA must cause $RTT>R_m+D$ or loss to avoid arbitrarily low utilization. Since (1) the buffer is not big enough to build D seconds of delay (because $\beta_s<D$), and (2) the delay box in CBR-delay can avoid adding any jitter, the end-to-end delays can always be $\le R_m+\beta_s\le R_m+D$, forcing the CCA to cause losses (as it can no longer cause $RTT>R_m+D$).

Proof. We prove by contradiction, i.e., if the CCA does not cause $RTT > R_m + D$ or loss, then we can construct an execution where the CCA gets arbitrarily low utilization. Say the CCA produces an infinite execution with $RTT\le R_m+D$ without ever causing loss on a CBR-delay link (say link I) with bandwidth C_I and β_s seconds of buffering. This exact RTT sequence can be produced by another CBR-delay link (say link II) with rate $C_{II}\gg C_I$ (see construction below). Since the CCA is deterministic, and it gets same sequence of RTTs as feedback on both links, it will make the same sending rate choices on both the links.[8] However, the CCA's average sending rate is $\approx C_I$ (as it does not build large queues on link I given that its RTTs are at most R_m+D). Such sending rate gets arbitrarily low utilization on link II for arbitrarily large C_{II}.

Construction. Link II can produce same RTT sequence as link I, by choosing $\texttt{jitter}_{II}(t)=RTT_I(t)-R_m-qdel_{II}(t)$ (we use subscripts I and II to refer to quantities on link I and II respectively). With this choice $RTT_{II}(t)=R_m+qdel_{II}(t)+$

[8] Note RTTs capture all the information that a CCA can obtain from feedback. Metrics like loss, ACK-rate, etc. can be derived from packet send events and RTT sequence [6].

$\texttt{jitter}_{II}(t) = RTT_I(t)$. We just need to show that this is a feasible choice for $\texttt{jitter}_{II}$, i.e., $0 \le \texttt{jitter}_{II}(t) \le D$.

$\texttt{jitter}_{II}(t) \le D$.

We know $RTT_I(t) \le R_m + D$, or $RTT_I(t) - R_m \le D$,

thus $\texttt{jitter}_{II}(t) = RTT_I(t) - R_m - qdel_{II}(t)$
$\le D - qdel_{II}(t) \le D$
(as $qdel_{II}(t) \ge 0$ by definition of $qdel$)

$\texttt{jitter}_{II}(t) \ge 0$.

Since $C_{II} \gg C_I$, $qdel_{II}(t) \le qdel_I(t)$
(increasing C decreases congestive queueing delays)

As a result, $RTT_I(t) = R_m + qdel_I(t) + \texttt{jitter}_I(t)$
$\ge R_m + qdel_{II}(t) + \texttt{jitter}_I(t)$

Or, $RTT_I(t) - R_m - qdel_{II}(t) \ge \texttt{jitter}_I(t) \ge 0$

Thus, $\texttt{jitter}_{II}(t) = RTT_I(t) - R_m - qdel_{II}(t) \ge 0$

Step 4. Loss only happens when $C_{L,\lambda} \ge C - \alpha/\beta_s$. For loss to happen, the buffer needs to overflow, i.e., in some interval $[t_1^*, t_2^*]$

$$\int_{t_1^*}^{t_2^*} \lambda(s)ds > C \cdot (t_2^* - t_1^*) + C \cdot \beta_s \tag{D.2}$$

And, we also know the CCA need to ensure its risk of loss is bounded, i.e., $\Delta L(t_1^*, t_2^*) \le \alpha$, i.e.,

$$\int_{t_1^*}^{t_2^*} \lambda(s)ds - \left(C_{L,\lambda}(t_2^*) \cdot (t_2^* - t_1^*) + C_{L,\lambda}(t_2^*) \cdot \beta_s\right) \le \alpha$$

$$\text{or, } \int_{t_1^*}^{t_2^*} \lambda(s)ds \le C_{L,\lambda}(t_2^*) \cdot (t_2^* - t_1^* + \beta_s) + \alpha \tag{D.3}$$

We can only meet both these constraints (i.e., Eq. D.2 and Eq. D.3) when:

$$C_{L,\lambda}(t_2^*) \cdot (t_2^* - t_1^* + \beta_s) + \alpha > C \cdot (t_2^* - t_1^*) + C \cdot \beta_s$$
$$\text{or, } C_{L,\lambda}(t_2^*) > C - \alpha/(t_2^* - t_1^* + \beta_s)$$
$$\text{or, } C_{L,\lambda}(t_2^*) > C - \alpha/\beta_s$$

E Synthesizing `cc_probe_slow`

We describe properties of probe intervals used in §6.2.1 to inform belief computation and encoding. Recall, probe intervals are intervals that lead to increase in $C_{L,\lambda}$ through the equation:

$$C_{L,\lambda}(t^*) = \max_{0 \le t_1 \le t_2 \le t^* - RTT(t^*)} \frac{\int_{t_1}^{t_2} \lambda(s)ds}{t_2 - t_1 + D}$$

Measurement intervals influence probing intervals, or future probe intervals cannot be shorter than past probe intervals. Say in the past $C_{L,\lambda}^p$ was computed over an interval of length T^p. In the future, $C_{L,\lambda}$ increases to $C_{L,\lambda}^f = C_{L,\lambda}^p + \varepsilon$, and was computed over an interval $T^f < T^p$. We will show that this action risks losing more than constant loss, i.e., loss can increase with C.

The amount of loss the future probe risks is (from difference in net enqueued bytes, dequeued bytes, and buffer size):

$$\Delta L = \int^{T^f} \lambda(s)ds - \left(C \cdot T^f + \beta\right) \tag{E.1}$$

We make the following substitutions in Eq. E.1:

(1) $\int^{T^f} \lambda(s)ds = C_{L,\lambda}^f \cdot (T^f + D)$ (from the definition of $C_{L,\lambda}$).

(2) $C_{L,\lambda}^f = C_{L,\lambda}^p + \varepsilon$

(3) From the past probe, we know $\int^{T^p} \lambda(s)ds \le C \cdot T^p + \beta$ (as loss did not happen, Eq. 6.3), and $\int^{T^p} \lambda(s)ds = C_{L,\lambda}^p \cdot (T^p + D)$ (from the definition of $C_{L,\lambda}$). From these two inequalities, we get $C_{L,\lambda}^p \cdot (T^p + D) \le C \cdot T^p + \beta$.
We substitute $C \cdot T^p + \beta = C_{L,\lambda}^p \cdot (T^p + D)$, or $C_{L,\lambda}^p = \frac{C \cdot T^p + \beta}{T^p + D}$, to evaluate how much loss would happen on this network.

On making the substitutions, and algebraic simplifications, we get:

$$\Delta L = \frac{(CD - \beta) \cdot (T^p - T^f)}{T^p + D} + \varepsilon \cdot (T^f + D) \tag{E.2}$$

For $T^f < T^p$, ΔL can be an increasing function of C as long as $\beta < CD$, e.g., $\beta = C\beta_s$ for $\beta_s < D$. Hence, loss is not bounded by a constant independent of C.

A probe interval must start with drained queue. Say $[t_1, t_2]$ is a probing interval, i.e., it leads to an increase in $C_{L,\lambda}$, and it does not start with a drained bottleneck queue; then we will show that the probe interval risks losing all the packets in the bottleneck queue. As a result, to ensure losses are bounded, a CCA needs to ensure that the bottleneck queue is bounded at the beginning of the probe interval.

Say the $[t_1, t_2]$ probe interval causes an increase in $C_{L,\lambda}$ at time t, i.e., $0 \le t_1 \le t_2 \le t - RTT(t)$, and,

$$C_{L,\lambda}(t) = \frac{\int_{t_1}^{t_2} \lambda(s)ds}{t_2 - t_1 + D} > C_{L,\lambda}(t - \varepsilon) \tag{E.3}$$

for some small $\varepsilon > 0$. If this is not true, then $[t_1, t_2]$ is not a probing interval. By definition, $C_{L,\lambda}$ can only increase over time, so $C_{L,\lambda}(t - \varepsilon) \ge C_{L,\lambda}(t_2)$. Using this and Eq. E.3, we get, $\frac{\int_{t_1}^{t_2} \lambda(s)ds}{t_2 - t_1 + D} > C_{L,\lambda}(t_2)$, or,

$$\int_{t_1}^{t_2} \lambda(s)ds - C_{L,\lambda}(t_2) \cdot (t_2 - t_1 + D) > 0 \tag{E.4}$$

The risk of loss during the probe is: ΔL = (bytes already in queue) + (enqueued bytes) − (dequeued bytes+buffer). We evaluate this equation on the path $\langle C = C_{L,\lambda}(t_2), \beta = C_{L,\lambda}(t_2) D \rangle$. It can produce observations till time t_2 as it is in the belief set (derived in §6.2.1), assuming until t_2, the CCA has not witnessed $RTT > R_m + D$, or losses. Plugging this path into ΔL, we get:

$$\Delta L = q(t_1) + \int_{t_1}^{t_2} \lambda(s)ds - \left(C_{L,\lambda}(t_2) \cdot (t_2 - t_1) + C_{L,\lambda}(t_2) \cdot D\right) \tag{E.5}$$

From Eq. E.5 and Eq. E.4, we get $\Delta L > q(t_1)$.

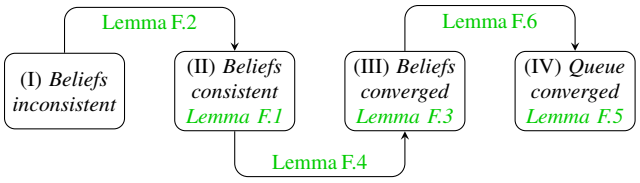


Figure 14: State transition lifetime of `cc_qdel`. Note, a link rate change can cause a transition between any two states (these "all to all" transitions are not shown).

F Proofs of performance

The synthesized CCAs ensure the synthesis invariant, but that is not sufficient to meet the performance objectives due to under-specification (§5.2). We build proofs, consisting of lemmas, that describe the states the CCA visits, transitions it makes, and the objectives it ensures. In the interest of space, we only discuss lemmas for `cc_qdel`. Lemmas for `cc_probe_slow` are similar but use $C_{L,\lambda}$, and q_U belief bounds instead of C_L and C_U.

Using the verifier, we checked that the lemmas are true for `cc_qdel` running on the CCAC network model with parameters $\langle C, R_m, D = R_m, \beta \geq 3C\cdot(R_m + D)\rangle$. We give a summary of how the lemmas work together, describe how we built them, and then discuss each lemma.

Summary. Fig. 14 describes how the lemmas are related to each other. The lemmas are true over any $10R_m$ seconds trace of the CCA's execution. We stitch together lemmas to reason about performance over arbitrarily long time horizons.

Whenever the link rate changes significantly, C_L and C_U beliefs may become inconsistent. The CCA ensures that these beliefs become consistent exponentially fast (Lemma F.2), they converge to a small range exponentially fast (Lemma F.4), and finally, the CCA reaches steady state (Lemma F.6), i.e., the bottleneck queue reduces additively at rate proportional to C.[9] The lemmas ensure that progress always happens in the same direction. Specifically, the beliefs cannot become inconsistent on their own, i.e., without link rate varying (Lemma F.1), the beliefs cannot diverge post convergence (Lemma F.3), the bottleneck queue cannot become unbounded after it converges (Lemma F.5). In other words, once the CCA reaches steady state (IV), it stays there. In the steady state (IV), `cc_qdel` gets at least 89% utilization, keeps RTTs to less than $4.4(R_m+D)$ seconds, and loses at most 3 packets in any R_m duration (Lemma F.7). Additionally, Lemma F.8 ensures that `cc_qdel` never incurs large loss events when probing for bandwidth as long as the beliefs are consistent, i.e., in states II, III, IV.

Building lemmas using binary search. The lemmas include non-trivial constants, we obtain all these by using binary search. For instance, to compute utilization in steady state, we ask the verifier if `cc_qdel` violates Lemma F.5 for different choices of utilization. When we ask this query for utilization = 100% the verifier gives a counterexample showing that `cc_qdel` can get lower than 100% utilization. However, when we repeat for 50%, the verifier says UNSAT, implying on (worst-case) all executions that start in steady state, `cc_qdel` gets at least 50% utilization. We repeat and find that `cc_qdel` ensures 89% utilization in steady state.

[9] A CCA cannot drain the queue faster than this. Even if the CCA stops sending packets, and the bytes are serviced at rate C, then the queue will only drain by $C\cdot t$ bytes in time t.

LEMMA F.1. *Initial beliefs are consistent* $\implies$ *final beliefs are consistent. Where, beliefs are consistent* $\equiv C_L(t) \leq C \leq C_U(t)$, *and we evaluate initial beliefs at* $t = 0$ *and final beliefs at* $t = T = 10R_m$.

This lemma verifies that our belief computations are correct. I.e., any bandwidth C that can produce the observations in the trace from $t = 0$ to $t = T$, and also produce observations before $t = 0$ (i.e., C is in the initial belief set or "initial beliefs were consistent") should be in the final belief set (or "final beliefs should be consistent"). This lemma is true for any CCA.

LEMMA F.2. *($\neg$ Initial beliefs are consistent)* $\implies$ *(final beliefs move towards consistency* $\vee$ *final beliefs are consistent* $\vee$ *steady state objectives). Where,*

Final beliefs move towards consistency $\equiv$
$(C_L$ *inconsistent* $\implies$ *C beliefs decrease*$)\wedge$
$(C_U$ *inconsistent* $\implies$ *C beliefs increase*$)$

C_L *inconsist.* $\equiv C_L(0) > C$, $\quad C_U$ *inconsist.* $\equiv C_U(0) < C$

C beliefs dec. $\equiv$ *At least one decreases* $\wedge$ *None increases*

At least one decreases $\equiv$
$C_L(T)\cdot 1.1 < C_L(0) \vee C_U(T)\cdot 1.1 < C_U(0)$

None increases $\equiv C_L(T) \leq C_L(0) \wedge C_U(T) \leq C_U(0)$

C beliefs inc. is defined symmetric to C beliefs dec.

Steady state objectives $\equiv$ (*Utilization lower bounded*
Inflight upper bounded $\wedge$ *No large loss events*)
$\equiv \frac{S(T)-S(0)}{C(T-D)} \geq 83\% \wedge$
$\forall t.\theta(t) \leq 4.4C\cdot(R_m+D) \wedge \forall t.L(t)-L(t-1) \leq 3MSS$

In each trace, the beliefs move towards consistency by a multiplicative factor, i.e., *exponentially fast*. The $1.1\times$ factor is a worst-case movement over all possible $10R_m$ long executions. In traces where C_L, C_U are far from C, beliefs move quicker. Beliefs start moving slower (but at least $1.1\times$) only when they are very close to C. So in practice the amount that the beliefs move varies over time and the beliefs become consistent much quicker than if they only improved by only $1.1\times$ every $10R_m$ seconds.

To encode "final beliefs move towards consistency", we require that (i) at least one of the C beliefs move in the correct direction *and* (ii) neither of them move in the wrong direction. The second clause (i.e., ii) is needed to ensure that the progress made by the CCA in consecutive traces adds up. Without this, it can happen that C_L increases and C_U decreases in the first trace, then C_U increases and C_L decreases, and so on. In each

trace at least one of C_L or C_U is moving in the right direction (satisfying clause i), but their progress is not adding up.

LEMMA F.3. *(Initial beliefs are consistent $\wedge$ initial beliefs are converged) $\implies$ (final beliefs are consistent $\wedge$ final beliefs are converged). Where, beliefs are converged* $\equiv C_L(t) \geq \frac{27C}{40} \wedge C_U(t) \leq 3C$.

When the beliefs are in the range $[\frac{27C}{40}, 3C]$, they do not go outside this range unless the link rate varies. This is despite periodic belief timeouts (§5.1.2). The beliefs are only timed out when they are well within this range, so that they stay within the range even after the timeout if the link rate hasn't changed.

Recall, we found the constants $27/40$ and 3 by binary search, i.e., the tightest range for which Lemma F.3 is true.

LEMMA F.4. *(Initial beliefs are consistent $\wedge \neg$ initial beliefs are converged) $\implies$ (final beliefs are consistent $\wedge$ (final beliefs shrink $\vee$ final beliefs are converged $\vee$ steady state objectives)). Where,*

Final beliefs shrink $\equiv$ *At least one improves* $\wedge$ *None degrade*

At least 1 imp. $\equiv C_L(T) > 1.7C_L(0) \vee C_U(T) < 1.7C_U(0)$

None degrade $\equiv C_L(T) \geq C_L(0) \wedge C_U(T) \leq C_U(0)$

The beliefs shrink by a multiplicative factor i.e., *exponentially fast*. Similar to Lemma F.2, we need the "at least one improves" *and* "none degrades" pattern to ensure that the progress made by the CCA adds up.

LEMMA F.5. *(Initial beliefs are consistent $\wedge$ initial beliefs are converged $\wedge$ initial bottleneck queue is bounded) $\implies$ (final beliefs are consistent $\wedge$ final beliefs are converged $\wedge$ final bottleneck queue is bounded). Where, bottleneck queue is bounded* $\equiv q(t) \leq 3.3C \cdot (R_m + D)$.

LEMMA F.6. *(Initial beliefs are consistent $\wedge$ initial beliefs are converged $\wedge \neg$ initial bottleneck queue is bounded) $\implies$ (final beliefs are consistent $\wedge$ final beliefs are converged $\wedge$ (final bottleneck queue is bounded $\vee$ bottleneck queue reduces)). Where, bottleneck queue reduces* $\equiv q(T) < q(0) - CR_m/2$.

To drain the queue, the CCA takes time that is linear in the queue size (i.e., it decreases queue *additively*, proportional to the *BDP* in each trace). Note, no CCA can multiplicatively reduce the number of bytes in the bottleneck queue. Even if the CCA stops sending packets, and the bytes are serviced at rate C, then the queue will only drain by $C \cdot t$ bytes in time t, this is independent of the number of bytes in the queue (i.e., not a multiple of the queue size).

LEMMA F.7. *(Initial beliefs are consistent $\wedge$ initial beliefs are converged $\wedge$ initial bottleneck queue is bounded) $\implies$ steady state objectives.*

LEMMA F.8. *(Initial beliefs are consistent) $\implies$ (rate increases $\implies$ no large loss events). Where, rate increases* $\equiv rate(T) > rate(0)$.

Once the C beliefs are consistent, when `cc_qdel` is probing (or ramping up, i.e., "rate increases"), it never incurs any large loss events losses. Large losses may happen if the link rate decreases (there is no way to avoid this). Note, this is only true on networks with large buffers, i.e., $\beta \geq 3C \cdot (R_m + D)$.

G Implementation issues in the Linux kernel

We originally implemented the synthesized CCAs in the Linux kernel. We uncovered bugs in the kernel's pacing implementation and also found the `cong_ctrl` API to be insufficient to implement the synthesized CCAs.

Pacing bug. To implement pacing, the Linux kernel pre-computes the "time to send the next packet" as the inverse of the pacing rate, i.e., inter-send-time. If the CCA's sending rate changes before the pre-computed time, then the kernel implements the wrong sending rate until the time to send the next packet. This leads to a discrepancy in the number of packets actually sent vs. the number of packets that the CCA wanted to send in a time interval.

In general, it is hard to implement pacing correctly. Pacing constrains two things, (1) a lower bound on the inter-send-time between packets and (2) number of packets sent in a time interval. Practically, a pacing implementation cannot meet both these constraints. Due to delays in CPU scheduling and interrupt handling, an instruction's execution may be delayed. These delays can cause a pacing implementation to miss a sending opportunity. When this happens, pacing implementation can either temporarily increase the sending rate to correct for the delayed opportunity (there by violating the lower bound on inter-send-time) or process the delayed sending opportunity as is (there by not sending any bytes corresponding to the delay and violating the constraint on total packets sent).

In fact, the pacing implementation of the genericCC [54] (the framework that we used to implement the synthesized CCAs in userspace over UDP) was also incorrect. We modified it so that it faithfully honors the lower bound on inter-send-time to ensure constant loss. Doing so creates a minor discrepancy between the packets sent over an interval vs. packets the CCA expected to send. Specifically, we maintain `last_sent_time` and compute `inter_send_time` whenever the CCA changes its sending rate. We launch a thread that polls (busy waits) to check if the sender is allowed to send (i.e., `current_time` is $\geq$ `last_sent_time` + `inter_send_time`).

The busy waiting can be avoided by setting two interrupts: (1) sleeping for time = `inter_send_time`, and (2) whenever ACKs are received. Before either of these two interrupts hit, no packets are sent or received and so there is no reason for CCA to change its rate, so it is okay to sleep until these interrupts as the `inter_send_time` does not become stale until these interrupts. Note, setting interrupts increases the delays caused in scheduling threads. This can increase the discrepancy between the actual packets sent vs. the packets the CCA expected to send. The pacing implementation may choose to transiently increase the sending rate to correct for

sending opportunities missed due to scheduling delays.

API. The Linux kernel also does not provide a direct API to change the sending rate after packets are sent. The kernel only gives a callback on ACKs. Since ACKs can be delayed due to non-congestive delays, the CCA ends up setting a potentially stale sending rate for $O(D)$ time. This can be worked around by setting up timer interrupts or instrumenting a callback on packet send events. The genericCC API provides callbacks on both packet send and receive events.

H Supplementary empirical evaluation

We empirically evaluate the synthesized CCAs across a variety of $\langle C, R_m, D, \beta \rangle$ parameters. Specifically, we explore combinations of $C \in \{24, 48, 96\}$ Mbps, $R_m \in \{20, 40, 80\}$ ms, $D = R_m$, and $\beta \in \{1/16, 1/8, 1/4, 1/2, 1, 2, 4, 8, 16\}$ *BDP*. Each tuple is a 60 seconds long "run". We discard the first 20 seconds of each run (to study steady-state behavior) and compute metrics over the remainder.

To emulate jitter, we inject up to R_m seconds of *random* ACK aggregation with a fixed average link rate. We sample `agg_delay` $\in [0, D = R_m)$ uniformly randomly. We serve a batch of $C \cdot$ `agg_delay` bytes after waiting for `agg_delay` seconds, ensuring an average bandwidth of C. We sample `agg_delay` after every batch.

Fig. 15 and Fig. 16 show the utilization, delay, and loss metrics for each run. The synthesized CCAs are withing their proof bounds in each run. In some of the runs, `cc_probe_slow`'s utilization is less than 50% (expected steady-state utilization in §6.2.1), this is because in those runs, it has not reached steady-state at the end of the run.

Note, BBRv2 and BBRv3 are not robust against adversarial jitter. On short buffers, they do not cause loss in steady-state. This is needed to ensure a lower bound on utilization on CBR-delay (**Step 1.** of Theorem 6.1), because otherwise their observations could be explained by an arbitrary large link rate. We believe this is why they get lower utilization when buffer is small in Fig. 15.

Convergence time. We study convergence time in Fig. 18 and Fig. 19. We show a run with $R_m = 80$ ms. For increasing link rate, we double the link rate every 20 seconds starting at 24 Mbps. For decreasing link rate, we halve the link rate every 20 seconds starting at 96 Mbps. In both cases we set the buffer size as the *BDP* when link (wire) rate is 96 Mbps. I.e., $\beta = BDP = 96$ Mbps $\cdot 80$ ms ≈ 624 packets (each packet sends 1538 bytes on the wire in our setup).

`cc_probe_slow` meets its convergence time bounds. It converges additively when link rate increases and exponentially fast when link rate decreases (due to belief timeouts).

BBRv2 and BBRv3 have low loss on average, but incur large loss events, i.e., $O(BDP)$ whenever link rate increases, and they start probing exponentially fast. This happens at the 20 second and 40 second marks when the link rate increases (not shown).

To synthesize a CCA that follows the tradeoff choice made by BBRv2 and BBRv3, we would need to remove the under-specification in our synthesis invariant. We want to be able to cause large losses when we are in state II (i.e., the link rate changed, and we want to converge), but do not want to cause large losses in state IV (when the link rate has not changed, but we just want to check if it may have increased after a belief timeout). Because of under-specification we cannot distinguish between these scenarios during synthesis. We leave this for future work.

We can manually design such a CCA using the version of `cc_probe_slow` parametrized by $f(.)$ and T^* (§6.2.1). We can adapt the loss allowance $f(.)$ depending on if the CCA is in steady-state (causing periodic small losses), or if the CCA needs to converge (is not causing losses). This can also be done in a fashion similar to BIC [59] and Cubic [31], i.e., loss allowance adapts depending on how much the link rate changed. We leave this exploration for future work.

Fairness. Currently, our formal theoretical framework does not provide any guarantees in multi-flow scenarios, nor does it provide any predictions for the outcomes of multi-flow experiments. Nevertheless, we explore the fairness properties of the synthesized CCAs empirically.

Fig. 17 shows a run with $C = 96$ Mbps, $R_m = 80$ ms, $\beta = 1/2$ BDP on an ideal link. We run 4 flows that share the same bottleneck. Each flow is started 30 seconds after the previous flow and lasts for $4 \times 30 = 120$ seconds. Results are qualitatively similar for other parameter and network model combinations. For each run (i.e., $\langle C, R_m, D, \beta \rangle$ combination mentioned at the beginning of the section), we compute the Jain's fairness index (JFI) for the average rates of the 4 flows between time 100 to 110 seconds (i.e., when all 4 flows are running). When $\beta \leq 2BDP$, the JFI for `cc_probe_slow` across the parameter combinations is 0.94 ± 0.06 (i.e., mean $\pm$ stddev) without jitter and 0.86 ± 0.11 with jitter. When $\beta > 2BDP$, `cc_probe_slow` effectively runs `cc_qdel` as there is no large loss event (§6.4). `cc_qdel` is unfair and gets JFI of 0.58 ± 0.24 without jitter and 0.43 ± 0.18 with jitter.

Even though `cc_probe_slow` was designed for single-flow scenarios, it is able to converge to a fair share of the available bandwidth. We believe this is because, to track changes in link rate, `cc_probe_slow` increases its effective rate ($C_{L,\lambda}$) additively and reduces its effective rate ($C_{L,\lambda}$) multiplicatively allowing it to reach a fair allocation similar to AIMD [18].

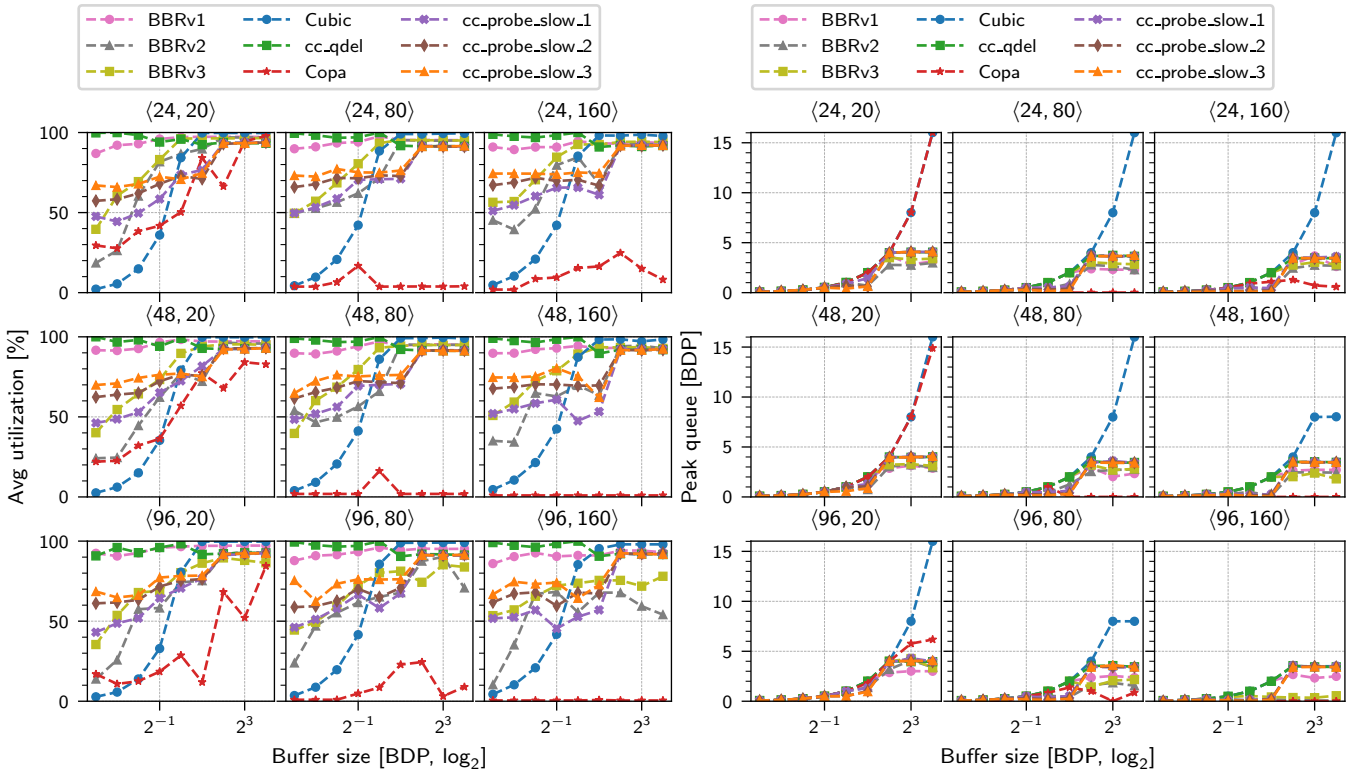


Figure 15: Average utilization and maximum queue use with varying bandwidth, propagation delay and buffer sizes for a link with random ACK aggregation. The tuple at the top of each subplot shows $\langle C$ Mbps,R_m ms$\rangle$.

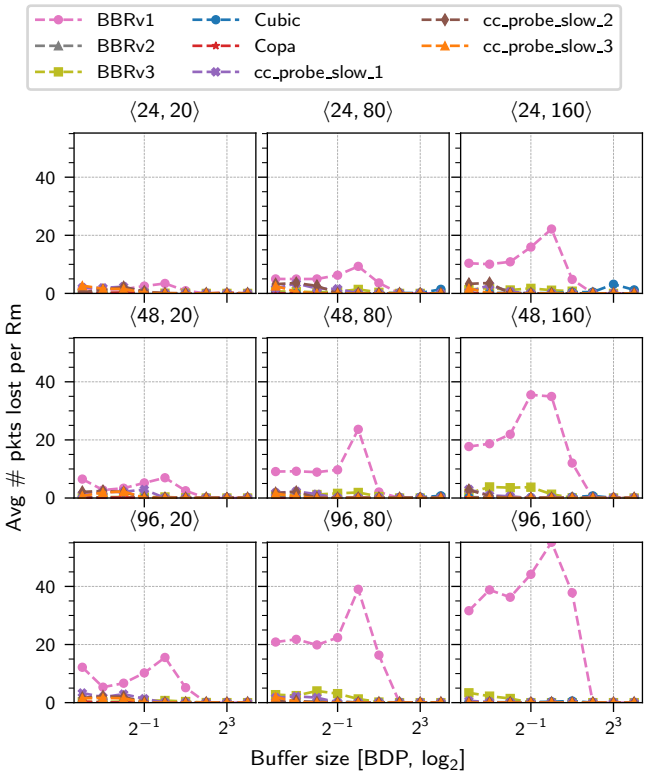


Figure 16: Packet loss with varying bandwidth, propagation delay and buffer sizes for a link with random ACK aggregation. The tuple at the top of each subplot shows $\langle C$ Mbps,R_m ms$\rangle$. We compute number of packets lost in every R_m long interval and take average over all the intervals. We omit `cc_qdel`, it causes $O(BDP)$ losses with a higher constant factor than BBRv1 (Fig. 9), and ends up skewing the graphs.

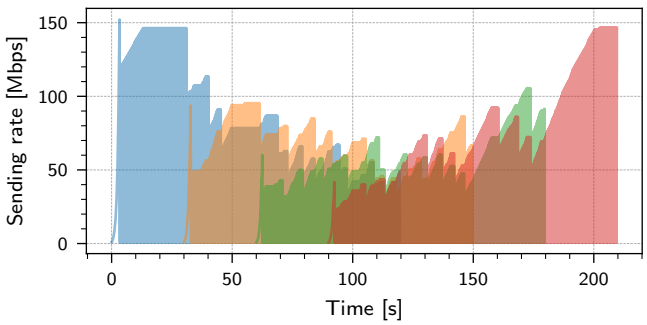


Figure 17: With multiple flows, `cc_probe_slow` is able to fairly share the available bandwidth. Every 30 seconds, we start a new flow for a total of 4 flows shown by the different colors. We use translucency to show the sending rates when the graphs overlap.

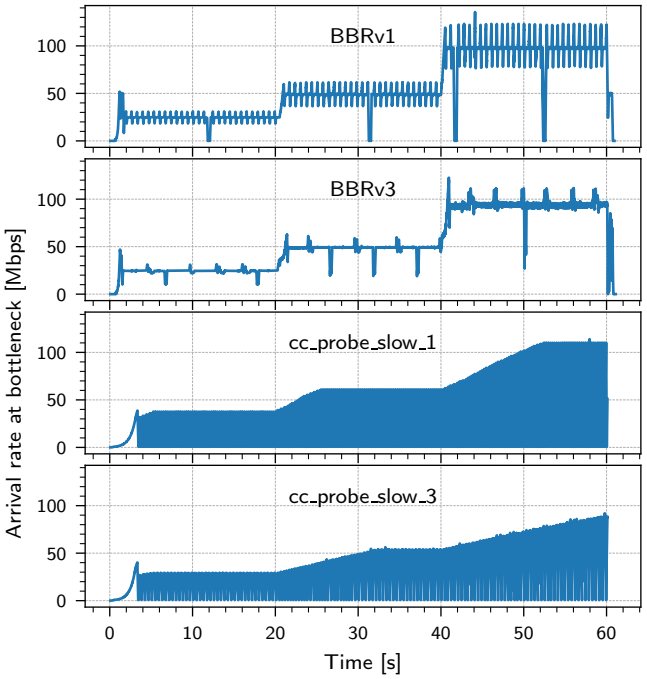


Figure 18: Convergence time for increasing C. `cc_probe_slow_k` converges additively when the link rate increases. Increasing `k` increases utilization by a multiplicative factor (Fig. 15) at the cost of increasing convergence time by a multiplicative factor.

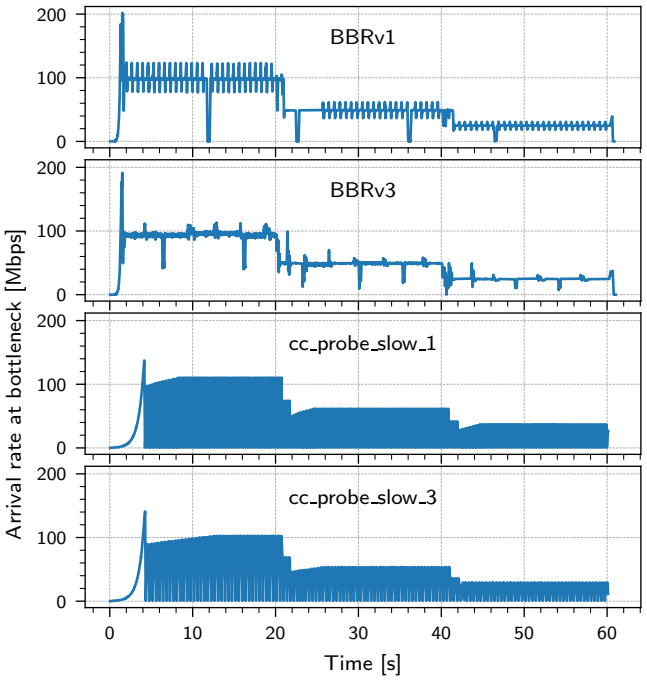


Figure 19: Convergence time for decreasing C. `cc_probe_slow_k` converges exponentially fast when the link rate decreases.

EPVerifier: Accelerating Update Storms Verification with Edge-Predicate

Chenyang Zhao[§ *], Yuebin Guo[§ *], Jingyu Wang[§ †], Qi Qi[§], Zirui Zhuang[§],
Haifeng Sun[§], Lingqi Guo[§], Yuming Xie[⋆], Jianxin Liao[§ †]
[§] *State Key Laboratory of Networking and Switching Technology,*
Beijing University of Posts and Telecommunications [⋆]*Huawei Technologies Co., Ltd*

Abstract

Data plane verification is designed to automatically verify network correctness by directly analyzing the data plane. Recent data plane verifiers have achieved sub-millisecond verification for per rule update by partitioning packets into equivalence classes (ECs). A large number of data plane updates can be generated in a short interval, known as *update storms*, due to network events such as end-to-end establishments, disruption or recovery. When it comes to update storms, however, the verification speed of current EC-based methods is often slowed down by the maintenance of their EC-based network model (EC-model).

This paper presents EPVerifier, a fast, partitioned data plane verification for update storms to further accelerate update storms verification. EPVerifier uses a novel edge-predicate-based (EP-based) local modeling approach to avoid drastic oscillations of the EC-model caused by changes in the set of equivalence classes. In addition, with local EPs, EPVerifier can achieve a partition of verification tasks by switches that EC-based methods cannot to get better parallel performance. We implement EPVerifier as an easy-to-use tool, allowing users to quickly get the appropriate verification results at any moment by providing necessary input. Both dataset trace-driven simulations and deployments in the wild show that EPVerifier achieves robustly fast update storm verification and superior parallel performance and these advantages expand with the data plane's complexity and storm size growth. The verification time of EPVerifier for an update storm of size 1M is around 10s on average, a 2-10× improvement over the state-of-the-art.

1 Introduction

Network faults such as forwarding loops, black holes, and reachability issues caused by misconfigurations, hardware or software problems can result in significant economic losses and social impact [12, 16, 21]. Manual troubleshooting is inefficient and prone to introduce new errors due to oversight. Therefore, *network verification* is proposed to automatically detect network correctness and avoid error-prone manual analysis by inferring all possible network behaviors based on the network's control plane configurations [5, 10, 13, 14, 25, 26, 32, 34] or data plane forwarding state [6, 15, 17–20, 23, 33, 37], which are known as *control plane verification* and *data plane verification*, respectively.

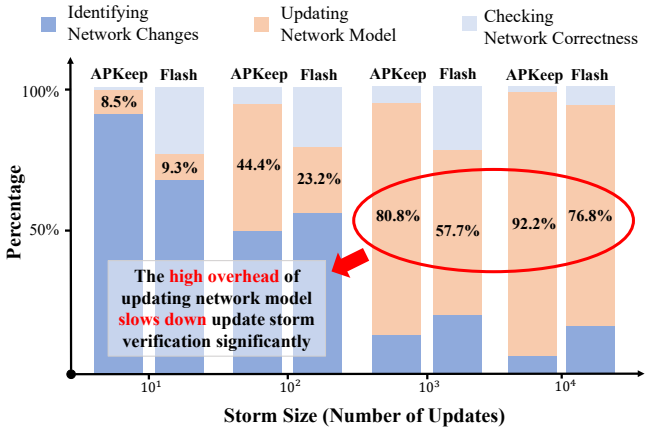


Figure 1: For the state-of-the-art equivalence-class-based data plane verifiers, the proportion of the Updating Network Model to the overall verification time explodes as the storm size (the number of updates in the update storm) expands.

In particular, this paper focuses on *data plane verification*, which directly checks the data plane to detect whether the network violates invariants (loop and blackhole) or other user-defined specifications (*e.g.,* reachability). A typical data plane verification process usually consists of three steps: 1) *Identifying network changes*, which identifies hop-by-hop forwarding behavior changes when updates come by analyzing rule dependency. 2) *Updating network model*, which updates the network model that is consistent with the data plane forwarding behavior 3) *Checking network correctness*, which checks the correctness of the data plane on a forwarding graph

*Chenyang Zhao and Yuebin Guo contributed equally to this work.
†Jingyu Wang and Jianxin Liao are the corresponding authors.

extracted from the network model. To be useful in practice, such data plane verifications need to be completed as rapidly as possible. Otherwise, the verification results may be invalid due to inconsistencies between the network state captured before verification and the actual network state that the updates are delivered into.

On the one hand, recent tools like APKeep [37] achieve sub-millisecond verification of each update when data plane updates arrive relatively slowly. Still, on the other hand, a large number of data plane updates can be arrived in a short time frame due to network events such as end-to-end establishments, disruption or recovery, etc., which is called *update storms* [15]. In this case, although Flash [15] gets substantial speed up on update storm verification by avoiding redundant computations and applying a series of engineering optimizations, updating the network model still slows down the verification speed. For example, as shown in Figure 1, in a synthetic wide-area backbone network containing 87 routers and 2308 links, the time consumption of updating network model as a percentage of the overall verification time increases with the size of the update storm and stays at a high level ($> 75\%$) regardless of whether the optimization is applied or not. This is because their network model is based on global equivalence classes (ECs) and therefore cannot avoid *network model oscillations*, *i.e.,* redundantly modify the part of the network model where no traffic change occurs. Besides, Flash divides the header space into multiple subspaces and distributes the verification of these subspaces to a cluster of verifiers in parallel. This simple and straightforward partition can be applied directly to all data plane verification systems. However, this kind of partition is prone to bottlenecks in practice because the updates that arrive are most likely *not evenly* distributed across subspaces.

To address the above problems and further accelerate update storms verification, we present EPVerifier, a fast, partitioned data plane verifier for update storms. Instead of partitioning packets into a set of ECs and maintaining an EC-based network model (EC-model) to represent the data plane forwarding state, EPVerifier maintains network packet flow with edge predicates(EPs) to form an EP-based network model, which we call *EP-model*. Each EP represents the local packet flow of one specific unidirectional edge. Although the time consumption of checking network correctness increases compared to the EC-based methods, such cost is worth it for update storms. This is because checking network correctness consumes much less time than updating the network model when there are a large number of updates, and the EP-model fundamentally avoids modifying the part of the network model where no packet flow changes, thus greatly speeding up the network model updating. For parallelism, EPVerifier can divide verification tasks in a switch-based way that EC-based methods cannot. This is because EP has a local nature that EC does not. Data plane updates that affect only the forwarding behavior of a single switch will also only affect edge predicates on that switch. However, using ECs to represent network traffic makes the same EC appear on different switches, meaning that traffic changes on one switch may also affect other switches. This local nature allows the EPVerifier to divide switches into clusters, which we call *regions*. We divide the regions after the update storms arrive to ensure that updates are *evenly* distributed among regions. We fully implement EPVerifier as an easy-to-use tool, users can quickly get the appropriate verification results at any moment by providing necessary input. The evaluation results on both datasets simulations and deployments in the wild show that EPVerifier achieves robustly fast update storm verification and superior parallel performance and these advantages expand with the data plane's complexity and storm size growth. Compared to the state-of-the-art, EPVerifier is up to more than $10\times$ faster.

Contributions. In summary, our main contributions are:

- We propose a novel network modeling approach based on edge predicates (EPs), which can quickly load a large number of data plane updates as they arrive. This approach fills the gap for fast update storm verification.
- A novel verification approach that can divide verification tasks by switches to achieve fast update storms verification in parallel.
- EPVerifier, an implementation of our approaches, evaluated on both datasets trace-driven simulations and deployments in the wild. EPVerifier achieves substantial gains compared to the state-of-the-art, and such advantages expand with the complexity of the data plane and the growth of the update storm size.

Roadmap. The problem definition for updating storm verification (§ 2) and the architecture and workflow of EPVerifier (§ 3) are given first, followed with examples to illustrate the difference between our approach and the state-of-the-art (§ 4). Then, the design details (§ 5) are presented and the experimental results (§ 6) are shown. After disscussing future (§ 7) and related work (§ 8), the conclusion is drawn (§ 9).

2 Problem Definition

Network model. On a data plane consists of N devices, the forwarding behavior of each device i is controlled by its routing table T_i, where each item $T_i[k]$ is a 3-tuple (m,a,p). Here m denotes the packets matched by this item, a denotes where the packets matched by this item will be forwarded to, and p denotes the priority of a routing table item. Since the m of different items in the same routing table may overlap, the packets governed by each item are determined by both m and p. When a packet arrives, device i iterates through each item on T_i in decending order of priority to get the highest priority item $T_i[k]$ that matches the packet and forwards the packet to $T_i[k].a$. The data plane forwarding behavior then can be modeled using an edge-labeled directed graph $G(V,E,l)$, where the nodes V represent devices, and edges E represent simplex

links. The function $l : E \rightarrow Label$ assigns to each edge a label that represents the packet flow on this edge. Note that different network models have different labels, for example, for the EC-model, the label is a set of equivalence classes, while for the EP-model it is an edge predicate.

Identifying network changes. Consider an update storm S consisting of K data plane updates, where each update is denoted as an insertion or deletion of a 4-tuple FIB rule $r(match, from, to, priority)$. And the installation process of these updates in the data plane is the insertion or deletion of the item $(match, to, priority)$ from the routing table T_{from}. However, the change of packet flow due to the entry insertions or deletions is not straightforward because of the priority, *i.e.*, we cannot directly modify the edge label based on $r.match$. Therefore, to specify the set of packets that a rule actually affects, we need a new field *hit*, introduced by [37], to represent the set of packets that a rule actually affects in the network. The *hit* is defined as follows:

$$r.hit = r.match \wedge \neg(\vee_{r'.prio>r.prio} r'.match) \quad (1)$$

Here r' denotes other FIB rules that share the routing table with r. With *hit* field, the network change of S is defined as follows:

$$S.change = (r_1.hit, ..., r_K.hit) \quad (2)$$

Updating network model. The update process of the network model G is actually the modification process of the edge label l. For the insertion of rule r, there are two parts of the edge label that need to be modified: 1) $r.hit$ need to be merged into the label on edge$(r.from, r.to)$ and 2) deleted from the label on edge$(r.from, *)$, here $*$ means any value that is not $r.to$. The similar goes for rule deletion.

Checking network correctness. After updating the network model, the verifier uses graph algorithms to check the correctness of the network on a forwarding graph $G'(V, E, l')$ extracted from the network model. Similar to G, the forwarding graph G' is also an edge-labeled directed graph, but l' is only a subset of l that affected by update storm S. Specifically, l' satisfies the following constraint:

$$l'[(v, v')] = l[(v, v')] \wedge (\vee_{i=1}^{i=K} r_i.hit) \quad \forall (v, v') \in E \quad (3)$$

Verification partition. There are two natural ways to decompose complex verification tasks into smaller, parallel computations: *subspace-based partition* and *switch-based partition*, as described in turn. First, the subspace-based partition achieves parallelism by dividing the forwarding state of the overall network into multiple subspaces. Namely, subspace-based partitions build multiple network models, each representing the forwarding behavior in a subspace, just like network slices. Second, the purpose of the switch-based partition is to divide the network model into different clusters of switches, which we call *regions*, and manage the forwarding state of different regions in parallel. In other words, switch-based partition only constructs one network model but maintains different parts simultaneously to keep the overall network model consistent with the data plane forwarding behavior.

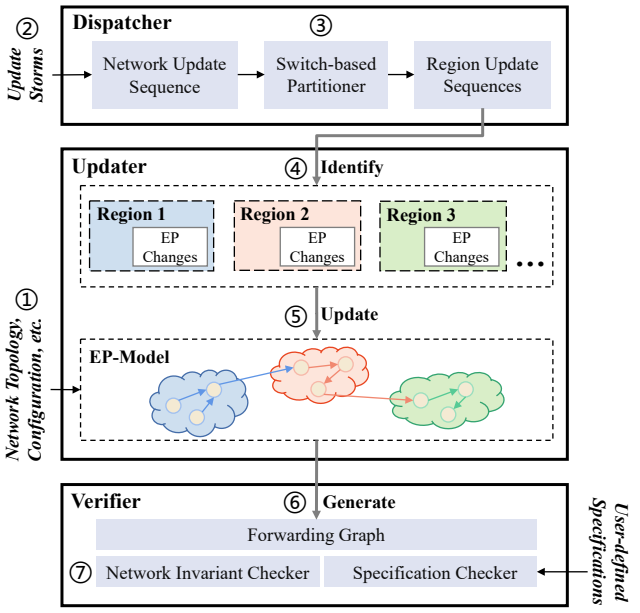


Figure 2: The architecture and workflow of EPVerifier, which requires three inputs: 1) the data plane topology and configuration provided by the controller for building the EP-model 2) Controller-generated update storms that will be delivered into the data plane 3) the user-defined specifications (reachability, waypointing, *etc.*)

3 Architecture and Workflow

As shown in Figure 2, the EPVerifier is composed of the following three parts:

The Dispatcher aims to guarantee the data plane updates in the update storms evenly across the different regions. When an update storm arrives, it divides the EP-model into multiple regions based on the distribution of data plane updates on each switch, ensuring that the number of updates in each region is as equal as possible.

The Updater corresponds to the identifying network change and updating network model of the data plane verification process. More specifically, it identifies the EP changes in each region in parallel and applies these changes to the EP-model.

The Verifier is responsible for checking network correctness. It first extracts the forwarding graph from the EP-model, and then applies the corresponding algorithm on this graph to verify the network invariants or user-defined specifications.

Workflow. A typical workflow of using the EPVerifier is as illustrated in Figure 2. First, the controller * needs to pro-

*Note that the EPVerifier can run in any environment that can provide data plane updates. For simplicity, we use "controller" to refer to the assisted system that provides data plane updates to it.

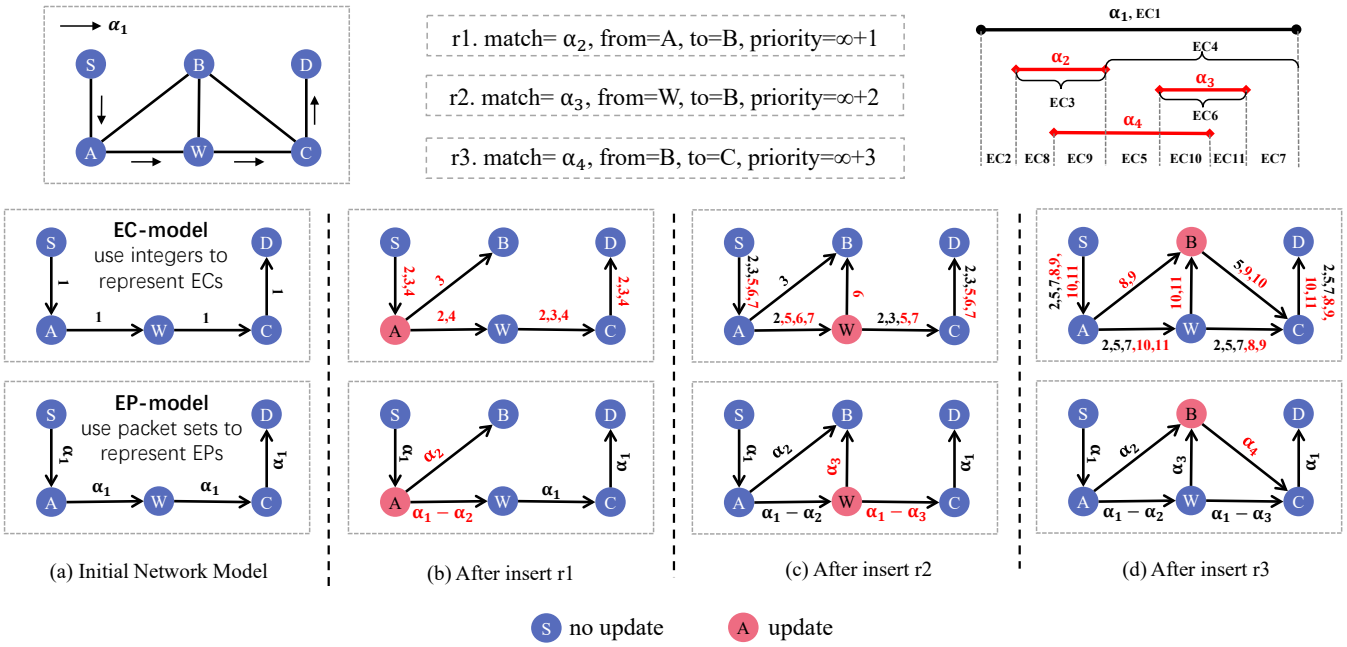


Figure 3: EPVerifier uses edge predicates to avoid the network model oscillations.

vide the data plane topology and configuration information to EPVerifier for initializing an EP-based network model in the bootstrap stage (①). Then, EPVerifier converts each update storm obtained from the controller into a network update sequence (②), where each item is a data plane update representing the insertion or deletion of a 4-tuple FIB rule (See § 5.2 for more details). The Dispatcher then divides the EP-model into multiple regions according to the distribution of updates (③). After that, the Updater identifies the EP changes in each region in multiple threads (④) and updates the EP-model to maintain its consistency with data plane forwarding behavior in parallel (⑤). Finally, the Verifier extracts the forwarding graph from the updated EP-model (⑥) and uses it to verify different specifications (⑦), note that here EPVerifier only verifies network invariants (loop and black-hole) by default, but users also have the option to provide additional user-define specfications such as reachability, waypointing, etc. (See § 5.3 for more details.)

4 Example

In this section, we use examples to illustrate 1) the difference between EP-model and EC-model (§ 4.1) and 2) the difference between switch-based partition for EP-model and subspace-based partition for EC-model (§ 4.2).

4.1 Network Model

As shown in Figure 3, our example is based on a small network of six switches. We use a directed edge-labeled graph to represent the forwarding state of the data plane, and the label on each edge indicates the packet flow on that edge. For simplicity, let us assume that an update storm consisting of three rules arrives at some point. The details of the three rules $r1$, $r2$ and $r3$ are shown in the top-middle of the Figure 3. The three rules are delivered to switches A, W and B and the set of matched packets are $\alpha_2, \alpha_3, \alpha_4$, respectively. For priority, $r1$, $r2$ and $r3$ have increasing priority and are higher than all existing rules in the network.

The initial state of the data plane is shown in the top-left corner of the Figure 3, where the packets matched by α_1 from S are forwarded to D via A, W and C. We show how α_1, α_2, α_3 and α_4 overlap with each other by parallel lines in the top-right corner of the Figure 3.

We now discuss how the EC-based method maintains EC-model. Since each equivalence class(EC) is a set of packets that experience the same forwarding actions throughout the network [20], thus, in our example, the network initially contains only one equivalence class, EC1, which denotes the identical group of packets represented by α_1. However, EC1 is continuously split into the final 7 ECs after inserting all three rules: First, with the insertion of $r2$, the action of all packets matched by α_2 on switch A changes from W to B because the priority of $r1$ is greater than all the old rules on switch A. This results in splitting the EC1 into EC2, EC3 and EC4 in the EC-model. After this, with the insertion of rule $r2$, EC4 is split into EC5, EC6 and EC7. Finally, after the insertion of rule $r3$, EC3 and EC6 are split into EC8, EC9 and EC10, EC11 respectively.

Network model oscillations. Note that although each rule insertion only affects the packet flow on 1-2 edges, each EC

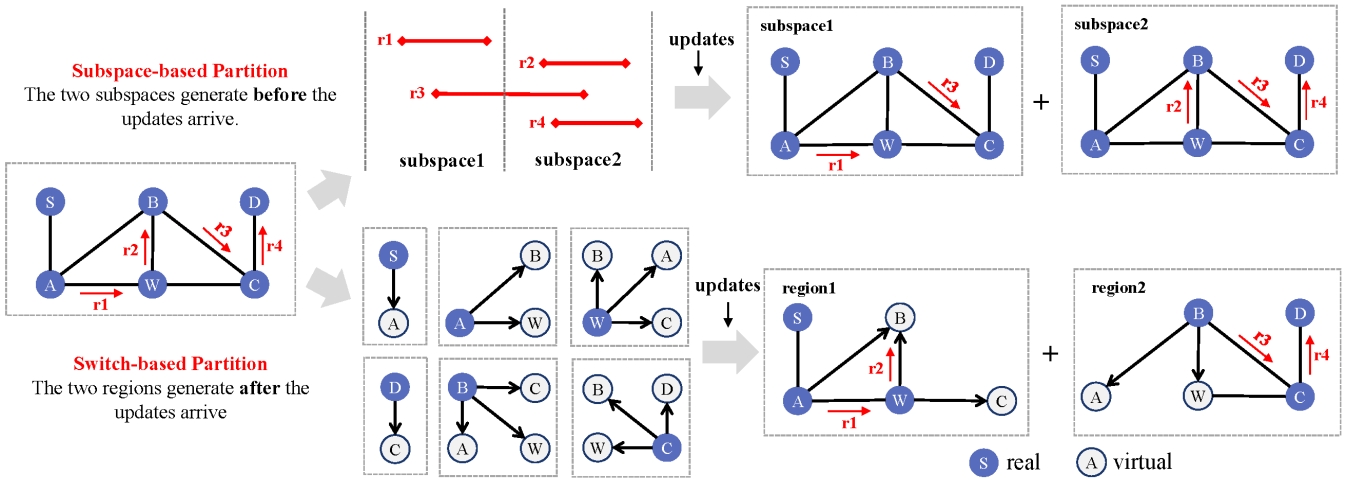


Figure 4: Two natural ways for the data plane verification partition.

split caused by rule insertions results in a large number of modifications to the EC model (red label in the forwarding graph). This is due to the fact that the equivalence classes of the entire network appear on a large number of edges in the EC-Model, *i.e.*, the global nature of ECs. Although the EC-based method can be improved in practice to avoid problems such as the explosion of the number of equivalence classes [37], redundant computation [15], the dramatic oscillation of the EC-model caused by changes in the set of equivalence classes are fundamentally unavoidable. Thus, we conclude that the global nature of EC fundamentally limits the speed of updating the network model and thus slows down the update storms verification.

We then illustrate how EPVerifier avoids the above oscillations of the network model and accelerates the verification time of update storms. Unlike the EC-model that divides packets into a set of ECs and uses ECs to label each directed edge in the network forwarding graph, the EP-model of EPVerifier represents packet flow in the network by maintaining edge predicates (EPs). An edge predicate is defined as follows.

Definition (Edge Predicate): An edge predicate(EP) is a 3-tuple$(from, to, P)$, indicating that any packet $p \in P$ will be forwarded by device $from$ to device to.

As the definition describes, each directed edge $(from, to)$ in the EP-model corresponds to each EP individually, determining that each EP in the network model will only appear once, *i.e.*, the local nature of EP. When a rule update comes, such local EPs allow EPVerifier to keep the network model consistent with the data plane by only modifying those EPs whose bound edges have packet flow changes. For example, as shown in Figure 3, we use the packet flow on each edge to represent the EPs in the network model. In this case, when rule $r1$ arrives, EP-model only needs to modify the EP on edge (A, W) and (A, B) because there are packet flow changes on them due to the installation of rule $r1$.

In summary, for the network model, the EC-based methods cannot avoid the oscillations in the network model caused by changes in the EC set due to the global nature of EC, which slows down the verification time of update storms. To further accelerate and achieve fast verification of update storms, EPVerifier uses EPs to represent packet flow in the network instead of global ECs to ensure that only the part of the network model with packet flow changes is modified.

4.2 Partition

Suppose we want to divide the overall verification task into two smaller tasks for parallel computation. As described in § 2, there are two natural ways to achieve parallelism: *subspace-based partition* and *switch-based partition*. We now illustrate the difference between them.

The subspace-based partition achieves parallelism by slicing the forwarding state of the overall network into multiple subspaces. Specifically, as shown in the left side of the Figure 4, The subspace-based partition creates two network models, each accepting the updates from subspace1 and subspace2, respectively. Ideally, when the updates are evenly distributed among different subspaces, this simple and straightforward partition can significantly reduce the number of updates a single network model needs to handle. However, since the subspaces and the corresponding network models are built *before* the arrival of update storms, the subspace-based partition will easily encounter the following limitations in practice: 1) updates are biased towards one subspace, *e.g.*, as shown in Figure 4, $r1, r2, r3$ and $r4$ are generally biased towards subspace2. 2) Some updates may affect multiple subspaces at the same time, *e.g.*, $r3$ affects both subspace1 and subspace2.

Unlike subspace-based partition, the purpose of the switch-based partition is to divide the forwarding graph into different clusters of switches, which we call *regions*, and manage the forwarding state of different regions in parallel. The key

Algorithm 1: Divide update storm R into N region update sequences

```
1 for i ∈ Range(len(V)) do
2     v ←the unvisited switch that has the maximum
        number of updates;
3     r ← the region sequence index that has the
        minimum number of updates;
4     add all updates on v to the sequence S_r;
5 return S;
```

Algorithm 2: GetHit(r)

```
1 r.hit ← r.match;
2 for r' ∈ r.from.rules do
3     if r'.prio > r.prio && r'.hit ∧ r.hit ≠ False then
4         r.hit ← r.hit ∧ ¬r'.hit;
5     if r.hit = BDD.False then break;
```

difference between switch-based partition and subspace-based partition is that the regions are divided *after* the update storms arrive. Therefore, switch-based partitioning can modify the partition of regions according to the update storms to make the number of updates in different regions as even as possible. For example, as shown in the right side of Figure 4, the switch-based partition first divides the network model in terms of switches. Then it combines the switches into region1 and region2 according to the rules distribution to ensure that the four rules are evenly distributed among the two regions.

While switch-based partition performance is better than subspace-based partition, EC-based methods are unable to apply this partition. This is because the edge label of the EC-model is a set of equivalence classes, which re-couples the various regions that were intended to be isolated from each other, and the synchronization costs associated with this coupling are unacceptable. Hence the state-of-the-art EC-based data plane verifier [15] only divides the network into multiple subspaces to squeeze out the parallel potential of the EC-model. In contrast, after employing edge predicates in place of equivalence classes, we make a key observation that EP's inherently local properties make it highly conducive for switch-based partition. This is because each update can affect only the EP of those edges emanating from the switch where the update is being installed, so EPVerifier can easily divide the EP-model into isolated regions and maintain them in parallel for fast verification of update storms.

5 Design Detail

In this section, we discuss the design detail of EPVerifier. We first show how to divide the network model into multiple regions by switches (§ 5.1). And then illustrate the EP-model updating process (§ 5.2), followed with an introduction on how to use the EP-model to check the correctness of the data plane (§ 5.3).

5.1 Switch-based Partition

Representation of the packet set P. Considering a packet with a match field of h bits, we can think of this packet as a boolean formula consisting of h boolean variables. For example, an IP match field of 128.0.0.0/16 can be represented as $x_1 \wedge \overline{x_2} \wedge \cdots \wedge \neg x_{16}$. Therefore, the EP, which represents a set of packets that can traverse through an edge, can also be represented as a Boolean formula. We adopt the methods of [33], use binary decision diagram (BDD [9]) to encode the packet matching field. This enables us to take advantage of the efficient logical operations provided by BDD, such as conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$), in order to compute and update EPs.

EPs that belong to different switches. The prerequisite for efficient operations between different BDDs is to share the same boolean variables, so existing BDD libraries usually use BDD managers to organize BDDs that share the same boolean variables, and only BDDs that belong to the same manager can perform logical operations with each other. However, in such cases, BDDs that belong to the same BDD manager incur significant synchronization costs in multi-threaded situations. Therefore, to accelerate update storms verification, EPVerifier employs distinct BDD managers for EPs in different switches to circumvent synchronization costs. With distinct switch EPs, EPVerifier can then freely combine the switches in the network into multiple regions.

According to the above description, for a certain update storm R, in addition to the set of edges E and the set of points V that represent the topological information of the data plane, the EPVerifier uses three variables to maintain the EP model with N regions: S, B and M, as described in turn. First, $S_1 - S_N$ indicates the update sequences of each region, where N indicates the maximum number of regions in the network. Note that N should be smaller than the number of CPU cores to avoid frequent context switching. Second, B is an array of BDD managers. For each $v \in V$, $B[v]$ is the BDD manager of switch v. Namely, All EPs on switch v are encoded by $B[v]$. Finally, M is a hash table, which takes the network model edge as the key. For each edge $e(v,v')$, $M[e]$ is a BDD encoded by $B[v]$, indicating the set of packets that can pass on e. Note that each $M[e]$ is initialized to $BDD.False$ at the beginning to indicate that there is no traffic on e.

The algorithm 1 shows a sample program for dividing the update storm R into N region update sequences. It uses a greedy algorithm to generate N update sequences as evenly as possible. Specifically, the algorithm picks the switch with the maximum number of updates in the update storm (Line 2) and adds all the updates to the sequence with the fewest updates (Line 3-4). Finally, after all the switches have been

Algorithm 3: UpdateEP(S_i)

```
1  for (r, isInsert) ∈ S_i do
2      if isInsert then RuleInsert(r, i) ;
3      else RuleRemove(r, i) ;
4  Function RuleInsert(r, i):
5      GetHit(r);
6      M[(r.from, r.to)] ← M[(r.from, r.to)] ∨ r.hit;
7      for r' ∈ r.from.rules do
           // sorted by decreasing priorities
8          if r'.prio ≤ r.prio && r'.hit ∧ r.hit ≠
            BDD.False then
9              if r'.to ≠ r.to then
10                 M[(r.from, r'.to)] ←
                    M[(r.from, r'.to)] ∧ ¬(r'.hit ∧ r.hit);
11             r'.hit ← r'.hit ∧ ¬r.hit;
12     r.from.rules ← r.from.rules ∨ r;
13 Function RuleRemove(r, i):
14     M[(r.from, r.to)] ← M[(r.from, r.to)] ∧ ¬r.hit;
15     r.from.rules ← r.from.rules \ r;
16     for r' ∈ r.from.rules do
           // sorted by decreasing priorities
17         if r'.prio ≤ r.prio && r'.match ∧ r.hit ≠
            BDD.False then
18             if r'.to ≠ r.to then
19                 M[(r.from, r'.to)] ←
                    M[(r.from, r'.to)] ∨ (r'.match ∧ r.hit);
20             r.hit ← r.hit ∧ ¬r'.hit;
21             r'.hit ← r'.hit ∨ (r.hit ∧ ¬r'.hit);
22         if r.hit = BDD.False then break;
```

visited, the final set of region update sequences S is returned (Line 5).

5.2 Updating EP-model

As described in § 2, rules sharing a routing table may have overlapping match fields, so the set of packets actually affected by a rule's insert or remove is determined by both match and priority, *i.e.*, the hit field.

As shown in Algorithm 2, we calculate the hit field of each arriving rule update to obtain the impact of this update on the forwarding behavior of the data plane. And this procedure illustrates that a rule update may directly or indirectly affect two types of EPs: 1) the EPs on edge$(r.from, r.to)$, which are directly affected by rule r, 2) and the EPs on other edges' $from = r.from$ which are indirectly affected by rule r, note that these EPs are directly affected by other rules that share the routing table with the rule r.

Each region rule sequence S_i is an array of 2-tuple$(r, isInsert)$ items. Each item indicates an insertion ($isInsert = True$) or deletion ($isInsert = False$) of a rule r in the network model. Each rule r is specified as a 4-tuple $(from, to, match, priority)$, where $match$ means the set of packets matched by this rule, $from$ and to means that this rule will be installed to device $from$ and its action is forwarded to device to.

Algorithm 3 summarizes the update process of EPs in the region i. For all rules that have an impact on region i, the EPVerifier calls different methods for rule insertion and deletion separately (Line1-3). For the insertion of rule r (Line 4-12), EPVerifier first calculates the hit field of r according to the definition of hit (Line 5) and updates the EPs directly affected by r (Line 6). After that, the algorithm iterates through all the rules r' that share a routing table with r in descending order of priority (Line 8) and updates the EPs indirectly affected by r (Line 8-10), and maintains $r'.hit$ (Line 11). Finally, EPVerifier adds rule r to the rule set $rules$ of $r.from$ to complete the insertion of rule r (Line 12). For the deletion of rule r (Line 13-22), since the hit field is already computed during rule insertion, the EPs on edge$(r.from, r.to)$ can be updated directly at this point using $r.hit$ (Line 14) and the rule r deleted from $r.from.rules$ (Line 15). Subsequently, similar to rule insertion, EPVerifier updates the EPs directly affected by those rules with lower priority than r (Line 16-22). Besides, we find that a rule r affects or is affected by another rule r' when their match fields consisting of h bits overlap. So we can use Trie [22] instead of BDD conjunction to speed up the rule update installation. Such optimization is omitted from Algorithm 3.

5.3 Verification

EP transfer algorithm. As described in § 5.1, EPVerifier employs distinct BDD managers for EPs in different switches to circumvent synchronization costs. However, BDDs that belong to different BDD managers cannot operate logical operations with each other. This is unacceptable for checking network correctness, which requires the operation of EPs on different switches. Therefore EPVerifier needs to use the EP transfer algorithm to migrate the packet flow changes in different regions to the same BDD manager and generate a forwarding graph for checking network correctness.

The migration process of EP is the process of migrating BDDs from a regional BDD manager to a network BDD manager. Specifically, EPVerifier uses two steps to complete the migration of BDDs: 1) In the region BDD manager, a BDD is transformed into a set of formulas it represents. For instance, a BDD representation of $(x_1 \wedge \overline{x_2} \wedge \overline{x_3}) \vee (\overline{x_1} \wedge \overline{x_2} \wedge x_3)$ would yield a formula set: $[x_1 \wedge \overline{x_2} \wedge \overline{x_3}, \overline{x_1} \wedge \overline{x_2} \wedge x_3]$. 2) In the Network BDD Manager, the formulas are re-converted into binary decision diagrams (BDDs). While in the worst-case rule, updates may affect all formulas in the formula set of EPs and thus cause excessive EP transfer overhead, our experiments (§ 6.4) show that most rule updates have little impact on EP

Algorithm 4: TransferEP(e, old, i)

```
1  delta ← old \ e;
2  for formula ∈ delta.formulas do
3      M'[e.edge] ←
         M'[e.edge] \ B0.encoding(formula);
4      affected ← affected ∨ B0.encoding(formula);
5  delta ← e \ old;
6  for formula ∈ delta.formulas do
7      M'[e.edge] ←
         M'[e.edge] ∨ B0.encoding(formula);
8      affected ← affected ∨ B0.encoding(formula);
```

formulas, *i.e.* only 1-2 atomic formulas change. Therefore, EPVerifier performs incremental migration of changed EPs rather than starting from scratch to avoid redundant calculations. In other words, in addition to maintaining a hash table M that stores network traffic managed by different BDD managers, EPVerifier also maintains an M' that stores network traffic encoded by a same BDD manager. Meanwhile, the process of maintaining M' is incremental, *i.e.*, after the updating of M, the EP transfer algorithm is used to migrate the changes in M into M'. We use B_0 to represent the BDD manager of M'. The algorithm 4 summarizes the incremental migration of an EP change $old \rightarrow e$ from M to the M'. Note that update storms may have both rule insertions and deletions, so the algorithm 4 migrates the decreases (Line 1-4) and increases (Line 5-8) of e into M' in two parts and records the set of packets $affected$ by this update storm for verification.

Algorithm 5: CheckInvariants(G)

```
1  for s ∈ V do
2      Traverse(n, affected, {});
3  Function Traverse(s, p, history):
4      if p = B0.False then return;
5      if s ∈ history then Alert('loop');
6      for (s, s') ∈ E do
7          if M'[(s, s')] ≠ B0.False then
8              p ← p \ M'[(s, s')];
9              history ∨ s';
10             Traverse(s', M'[(s, s')] ∧ p, history);
11             history \ s';
12             M'[(s, s')] ← M'[(s, s')] \ (M'[(s, s')] ∧ p);
13     if p ≠ B0.False then Alert('blackhole');
```

Network invariants. With the forwarding graph $G(V,E,M')$, operators can check network invariants, including loop and blackhole, by traversing G. The algorithm 5 summarizes the process by which EPVerifier verifies network invariants using the network forwarding graph. The algorithm starts the depth-first search (DFS) traversal from each node $n \in V$(Line1). In addition to the s representing the current node, the DFS traversal process carries two parameters: p and $history$, as described in turn. First, p represents the packets carried during the current traversal, encoded by BDD. Second, $history$ stores the nodes that have been visited. p is initialized to the set of packets $affected$ by this update storm at the beginning of the traversal, while $history$ is initialized to the empty set (Line2). The traversal stops when p is empty (Line 4). When the traversal reaches a previously traversed node, a forwarding loop exists in the network (Line 5). Otherwise, the algorithm starts with the neighbor node s' of s in the G for DFS (Line 6). For every s', the algorithm updates history and carries $M'(s,s') \wedge p$ for DFS (Line 8-11), and after the traversal of s', the algorithm avoids repeated traversals by updating M' (Line 12). Finally, when all neighboring nodes are traversed, a blackhole exists if packets remain in s (Line 13).

Algorithm 6: CheckOtherSpecification(G, l)

```
1  M'[l] ← B0.False;
2  Reach ← {};
3  Traverse(source, match, waypoints, dst);
4  for node ∈ (waypoints ∨ dst) do
5      if Reach[node] ≠ match then Alert('Failure');
6  Function Traverse(source, p, waypoints, dst):
7      if match = B0.False then return;
8      if source = dst then
9          Reach[source] ← Reach[source] ∨ match;
10         return;
11     if source ∈ waypoints then
         Reach[source] ← Reach[source] ∨ match;
12     for (source, next) ∈ E do
13         p ← p \ M'[(source, next)];
14         Traverse(next, M'[(source, next)] ∧
             p, waypoints, dst);
```

Other specifications. In addition to network invariants, Operators may need to check other specifications, such as whether certain packets can be forwarded from device a through device b to device c or whether network invariants are violated when a link is broken, etc. We now explore how EPVerifier verifies these specifications. Here, we define a specification as a 4-tuple$(match, source, waypoints, dst)$, *i.e.*, all packets matched by a match from device source must be forwarded in the network via $waypoints$ must be forwarded to device dst. For such a policy, we carry the packets representing all the matched packets from the source and traverse G to simulate the packet forwarding process in the network. This policy is not violated only when all packets can pass through $waypoints$ to dst intact. In the case of a change in network state, such as a broken link, we modify the network forwarding graph before traversal.

Table 1: Dataset Information.

Network	Node	Links	FIB rules	Storm Size
Airtel1	68	220	6.89×10^4	10K
Airtel2	68	260	9.84×10^4	10K
Berkeley	23	252	1.28×10^7	1M
INET	315	40770	2.49×10^8	1M
RF1755	87	2308	3.37×10^7	1M
RF3257	161	9432	7.45×10^7	1M
RF6461	138	8140	7.50×10^7	1M

The algorithm 6 summarizes how EPVerifier uses the network forwarding graph $G(V,E,M')$ to verify that when the link l fails, all packets from *source* that are matched by *match* can are forwarded to device *dst* via *waypoints*. The algorithm first sets the EP on l to False to simulate a link failure (Line 1), and then initializes the variable *Reach* (Line 2), which records the packets that can be reached on each *waypoint*. Unlike the verification invariant, the algorithm directly carries the set of packets *match* of interest to the user for DFS traversal (Line 3). During the subsequent traversal (Line 6-14), the algorithm keeps track of the packet forwarding process by maintaining the *Reach* variable. After the traversal is completed, the specification is verified by checking whether all packets in *match* reach *waypoints* and *dst* (Line 4-5).

6 Evaluation

We fully implement EPVerifier as an easy-to-use tool and test it exhaustively with trace-driven simulations and deployments in the wild.

6.1 Setup

Implementation. We implemented EPVerifier as an easy-to-use tool in ∼4000 lines of Java code. In particular, EPVerifier exposes a ***init()*** function that accepts data plane topology and configuration information for initializing the EP-model and a ***verify()*** function that accepts update storms with user-defined specifications (if any) for verifying data plane correctness. For the parallelism part, we use the standard asynchronous thread of OpenJDK. For BDD operations, we use JDD, a BDD library for Java [30].

Dataset. Table 1 summarizes the information about the dataset we use. They all come from the open source dataset [1,17]. The Airtel1 and Airtel2 are generated using the ONOS SDN-IP application [3,11]. And the remaining five datasets are synthetic datasets generated using the mechanism in [36]. Specifically, [17] collects IP prefixes from real-world BGP updates collected by the Route Views project [4] and computes the shortest paths in the network topology [27].

Methods to compare. We compare EPVerifier with two state-of-the-art data plane verifiers, APKeep [37] and Flash [15], because their evaluation identifies that they achieve the most generic and fastest data plane verification. 1) **APKeep***: Since we did not find an open-source implementation of APKeep, we implement APkeep ourselves following the pseudocode in [37], referred to as **APKeep***. 2) **Flash**: we use its open-source implementation in Java [2]. Apart from the above two methods, we also consider **EPVerifier***, standing for EP-model with subspace-based partition instead of switch-based partition to evaluate the performance gap between the subspace-based partition and our switch-based partition.

Simulation objective. We use the above dataset traces to discuss the following questions:

- what is the overall performance gap between EPVerifier and the state-of-the-art data plane verifiers? (§ 6.2)
- what is the performance gap between the EP-based network model and the EC-based network model? (§ 6.3)
- what is the performance gap between switch-based partition and subspace-based partition? (§ 6.4)

Simulation testbed. We run all our simulation experiments on a Ubuntu 20.04(x64) LTS with a 3.4 GHz Intel 8-core CPU and 32 GB of RAM. The OpenJDK v19.0.2 is installed for Java support.

Deployment. Apart from simulations, We deploy EPVerifier and Flash onto a network control plane of a commercial data center backbone network for testing the real-world performance of EPVerifier (§ 6.5). The placement and operations are similar to those described in § 3, operators provide information about the update storms they are interested in, as well as a snapshot of the network before the update storms are issued, to the verifiers, which then parse these information and check whether the delivery of each update storm violates network invariants or user-defined specifications.

6.2 Overall Performance

We now evaluate the overall performance of EPVerifier in dealing with update storms. For network correctness, we verify the presence of forwarding loops in the network because the verification of forwarding loops requires completely traversing the entire forwarding graph while other specifications only need to traverse part of it. Since we use an 8-core CPU, we divide the network into eight subspaces or eight regions to make full use of the eight threads. As described in Table 1, for each dataset, we randomly select 10K or 1M rule insertions to form a update storm and deliver it to EPVerifier. After this update storm is installed and the loop is verified, we record the memory consumption and install another update storm, which is composed of 10K or 1M rule deletions in the same order.

The overall time, memory consumption and the number of BDD operations for the above process under different scenarios and datasets are summarized in Table 2.

In terms of time consumption, we can see that EPVerifier achieves significant performance improvements on most datasets, especially on the last four datasets where the network topology is dense and prone to update storms; for example,

Table 2: Time, memory cost and BDD operations of different methods

Network	Time cost (s)				Memory Usage (MB)				BDD Operations			
	APKeep* (speedup)	Flash (speedup)	EPVerifier* (speedup)	EPVerifier (speedup)	APKeep*	Flash	EPVerifier*	EPVerifier	APKeep*	Flash	EPVerifier*	EPVerifier
Airtel1	0.44 (4.9×)	0.10 (1.1×)	0.03 (0.3×)	**0.09** (1×)	63.76	64.15	65.67	407.41	1×10^6	3×10^5	8×10^4	5×10^5
Airtel2	0.23 (2.6×)	0.08 (0.9×)	0.03 (0.3×)	**0.09** (1×)	66.37	63.78	65.37	399.48	1×10^6	3×10^5	1×10^5	6×10^4
Berkeley	2262.15 (157×)	34.29 (2.4×)	41.57 (2.9×)	**14.42** (1×)	1434.57	1312.56	1866.29	2829.26	7×10^9	1×10^7	3×10^6	4×10^6
INET	14458.69 (557×)	800.65 (31×)	38.60 (1.5×)	**25.98**(1×)	8178.28	6124.18	2149.38	5253.45	2×10^{10}	3×10^9	7×10^7	7×10^7
RF1755	10196.17 (856×)	100.08 (8.4×)	48.03 (4×)	**11.91** (1×)	2924.07	2419.23	2018.21	3364.27	2×10^{10}	2×10^8	4×10^6	5×10^6
RF3257	10473.87 (785×)	208.95 (15.7×)	37.05 (2.8×)	**13.34** (1×)	4947.75	3876.56	2191.80	3987.75	2×10^{10}	7×10^8	1×10^7	1×10^7
RF6461	8736.91 (668×)	157.57 (12×)	34.51 (2.6×)	**13.07** (1×)	4502.41	3567.80	2135.69	3987.23	2×10^{10}	6×10^8	1×10^7	1×10^7

EPVerifier is 557× and 31× faster than APKeep and Flash respectively on the INET dataset, and at least 10x faster on the last three datasets. The overhead of the network model oscillations (§ 4.1) depend mainly on the number of edges in the network model, so for the Airtel1 and Airtel2 with fewer edges and small storm sizes, the acceleration of update storm verification by EPVerifier is insignificant because the advantage gained by EPVerifier in the updating network model phase is overshadowed by the subsequent EP transfer (§ 5.3). In the meantime, the EPVerifier* using subspace-based partition does not require EP transfer and therefore performs best in these two datasets.

In terms of memory usage, EPVerifier uses more memory when the network is small because it uses a separate BDD manager on each switch and each BDD manager needs to allocate a certain size of cache when it is initialized. However, this memory consumption does not explode as the complexity of the network rises because the total number of updates as the source of BDDs is the same regardless of the number of BDD managers. It can be seen that in the last four dense networks, EPVerifier's memory consumption compares to that of Flash and APKeep.

Finally, the number of BDD operations is good evidence of the time performance improvement of EPVerifier. In all datasets, the number of BDD operations of EPVerifier* and EPVerifier is at least one magnitude less than that of Flash and APKeep.

In summary, we can conclude that EPVerifier achieves robustly fast update storm verification. In dense networks that are prone to update storms, EPverifier is more than 10× faster than state-of-the-art.

6.3 Performance of EP-model

We now consider the performance gap between the equivalence-class-based network model (EC-model) and our edge-predicate-based network model (EP-model). For all datasets, we randomly select updates at different sizes (starting from 1) to form update storms and use Flash and EPVerifier to verify the forwarding loop, respectively. We do not apply any partition to these verifiers to focus on the effects of the network model.

The result is shown in Figure 5. We find that the advantage of EP-model is not obvious when the storm size is small. This is because when the storm size is small, the additional BDD operations brought about by EP transfer and traversing the forwarding graph are not negligible compared to the improvement brought about by the model update. However, as the storm size increases, the EP-model advantage starts to emerge and this advantage increases with the size of the update storm. Notice that the acceleration effect of EP-model is not the same for different datasets. This is because the number of edges is different in each dataset. When the number of edges is higher, the number of labels that need to be modified for the network oscillation (§ 4.1) of EC-model is also higher. Therefore, the improvement brought by EP-model is more obvious at this time. For example, for the INET dataset with tens of thousands of edges, EP-model is much faster than EC-model even when the storm size is very small. Thus, we can conclude that the advantage of EP-model expands with the complexity of the data plane and the growth of storm size.

6.4 Performance of Switch-based Partition

Compared to subspace-based partition. We evaluate the performance gap between our switch-based partition and subspace-based partition. Specifically, we divide the network into two subspaces and manually place updates into the two subspaces to achieve a different distribution of updates in the subspaces. We then compare the performance of verifying the update storm composed of these updates under the two partitioning approaches with EP-model. The result is shown in Figure 6. The x-axis is the ratio of the variance of the number of updates in the two subspaces to the total number of updates, which indicates the average distribution degree of updates in the subspaces. Obviously, when the value of x is between 0 and 1, when x is 0, it means that the updates are evenly distributed in two subspaces, and when it is 1, it means that all updates are in one subspaces. The y-axis is the ratio of the verification time of the two partition methods. Thus, the switch-based partition performs better when y is greater than 1. We can see that the verification time of the subspace-based partition rises as the variance of the rules in the two subspaces increases. For the last five datasets in Table 1 with complex topology and large storm sizes, the switch-based partition still outperforms the subspace-based partition even if the updates are evenly distributed among the two subspaces. This is because the subspace-based partition uses only two BDD managers, one for each subspace, while the switch-based partition assigns a BDD manager to each switch.

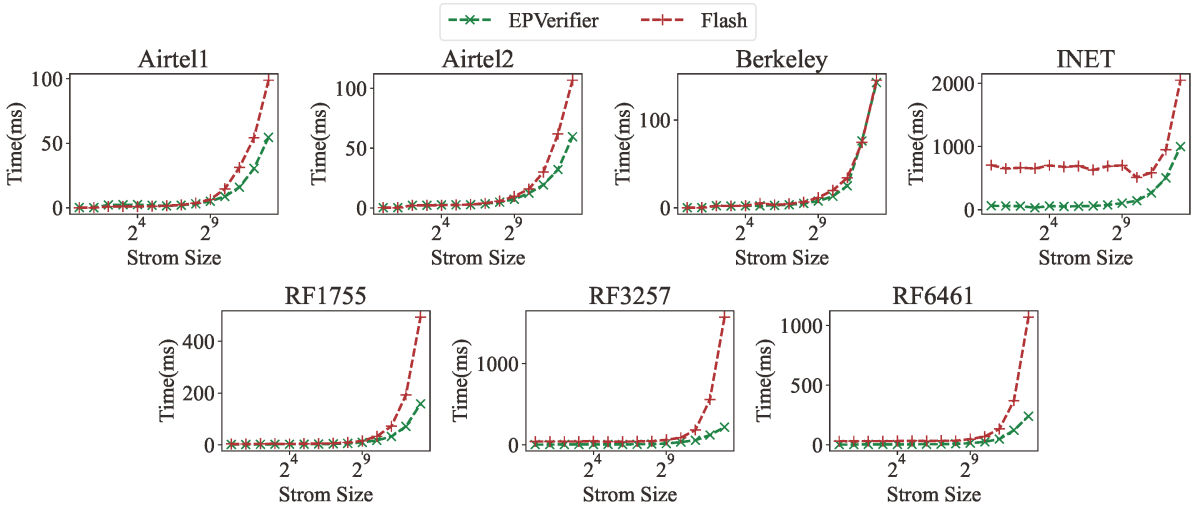


Figure 5: Verification time for different storm sizes

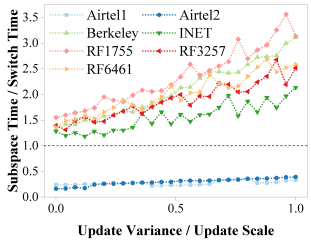


Figure 6: Effects of updates distribution on subspace-based partition verification time.

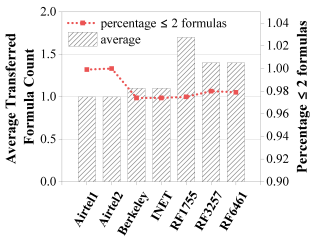


Figure 7: Each update only affects a few formulas.

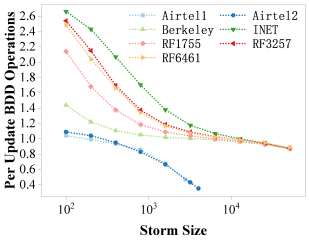


Figure 8: Per update BDD operations decrease due to the possible overlap of formulas affected by each update.

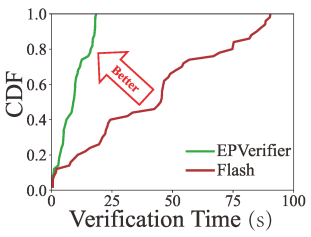


Figure 9: The update storms verification time in real-world network.

Too many BDDs in the BDD manager will slow down the BDD operations. In addition, for Airtel1 and Airtel2 datasets with simple topology and small storm size, the performance of the subspace-based partition is better because the number of BDDs in the model is limited in this case and the switch-based partition needs to go through a redundant EP transfer process.

Cost of EP transfer. Since switch-based partitions have to suffer additional EP transfer progress due to the limitation of BDD (§ 5.3), we now discuss the overhead of it. First, the complexity of the EP transfer algorithm depends mainly on the number of formulas that need to be transferred to the forwarding graph. Therefore, for each dataset, we select the device with the highest number of updates and apply these updates one by one to the EP-model and transfer them to the forwarding graph. We record the number of formulas affected by each update. Figure 7 shows the results. We can see that for all datasets, most of the updates (more than 97%) affect only one or two formulas, and the average number of transferred formulas does not exceed 2. Second, the main time of the EP transfer process is spent on the BDD operations. We generate update storms of different sizes and calculate the per update BDD operations during the EP transfer. As shown in Figure 8, for all datasets, with storm size increases, per update BDD operations tend to decrease due to the possible overlap of formulas affected by each update. Then, we can say that the overhead of EP transfer is acceptable, i.e., it does not explode with the complexity of the data plane and the storm size.

6.5 Deployment in the Wild

We evaluate the performance of EPVerifier by deploying it and Flash onto network control plane of a commercial data center backbone network with hundreds of switches and thousands of links. We verify 50 update storms with EPVerifier and Flash respectively and statistics the verification time of the both. The results are shown in Figure 9. The x-axis is the verification time and the y-axis is the Cumulative Distribution Function (CDF). Similar to the simulation results, EPVerifier still greatly accelerates update storm verification in real deployments.

7 Discussion and Future Work

The update storms. The concept of update storms is introduced by Flash [15]. However, a similar phenomenon of network update aggregation is prevalent in networks because the source of network updates, a change in the user's intent, generally affects a certain number of devices (virtual or physical), resulting in a large number of network updates. For example, in modern multi-tenant data centers, fast programming interfaces in the forwarding plane may result in thousands of updates per second [18].

Packet transformations. In addition to normal forwarding rules, there may be rules in the real network that can modify packet information, such as NAT. For these transformation rules, we can extend our design to handle them just like the fine-grained PPM network model of APKeep [37]. More specifically, we can model each transformation rule as a node in the network model to simulate the packet rewrite triggered by the corresponding rule. We leave a full design and implementation to future work.

Distributed verification. Most data plane verification tools use a centralized architecture where the server collects all data plane information and verifies it. While this architecture is inherently non-scalable, EPVerifier's switch-based partition is very suitable for implementing distributed, on-device data plane verification. A preliminary idea is that we can maintain the individual EP information of each switch and verify it centrally. Detailed design is left as one of our future works.

8 Related Work

Data plane verification. There is a long line of research on data plane verification. The early data plane verifiers [6, 7, 19, 23] use different formal methods to analyze the data plane snapshot. While useful, these verification tools are too poor to detect network errors in a timely manner and cannot handle real-time changes in the data plane to ensure that the results are correct in real-time. To overcome the above limitations, [17, 18, 20, 33] use novel algorithms and achieve per rule update real-time verification. Veriflow [20] is the first data plane verification tool that implements real-time verification by dividing packets into equivalence classes to split the verification task of the entire network into multiple independent verification tasks for the equivalence classes. Concurrent with Veriflow, Netplumber [18] implements incremental data plane verification based on Header Space Analysis (HSA). It is worth noting that Netplumber's HSA-based scheme for modeling network traffic information as a plumbing graph has similar localization properties as EP-model, but when the traffic in the network becomes progressively more complex, the plumber graph's node-by-node approach to the traffic limits its speed of verification, as demonstrated by experiments by [37]. Although these real-time data plane verifiers can achieve millisecond or even sub-millisecond verification speed, problems such as the exploding number of equivalence classes, excessive redundant computations, and inability to handle multiple match field still limit them. To further extend data plane verification, APKeep [37] solves the equivalence class explosion problem by merging equivalence classes, but when there are plenty of rule updates, many redundant computations in APKeep's PPM model still slow down the verification speed. Flash [15] avoids the redundant computations in the equivalence class model through a series of improvements and engineering optimizations. Inspired by [36], Flash divides the verification task according to subspace to speed up the verification of update storms. However, Flash still cannot solve the network oscillation problem caused by the equivalence class itself.

Control plane verification analyzes control plane information, network topology, and environmental context to verify the forwarding behavior of all packets and network intent under the data plane, which is generated by combining this information [5, 10, 13, 14, 25, 26, 32, 34]. They are complementary to EPVerifier and can use EPVerifier to accelerate the verification of the generated data planes.

Stateful network verification. Compared to data plane verification, the progress of stateful network verification is at a lower level and can only accomplish offline verification tasks. [28, 29] model stateful middleware by adding state information to packet headers based on symbolic execution techniques, and propose a modeling language SEFL to speed up the symbolic execution process. [24] proposes a middleware abstraction modeling method based on an SMT solver for verifying reachability and improves the verification speed by reducing the verification network size. [31] solves the verification problem of traffic isolation property by abstracting the processing order of packets. [8] further abstracts the state change of middleware based on [31] to achieve complexity optimization. [35] is implemented based on a symbolic model detection technique, which abstracts the network into individual packet forwarding models and achieves high scalability with a customized symbolic model detection algorithm.

9 Conclusion

This paper presented EPVerifier, a fast, partitioned data plane verification for update storms. EPVerifier achieves robustly fast update storm verification compared to the state-of-the-art. To achieve this, EPVerifier introduces an EP-based network model to avoid the network model oscillation and a switch-based Partitioning scheme to partition the update storm verification task into smaller parts for parallel computation. Both dataset trace-driven simulations and the deployments in the wild show that EPVerifier achieves substantial gains compared with the state-of-the-art. We believe the design of EP-model and switch-based partition can help data plane verification scale to large, complex data planes, which are prone to generate update storms.

Acknowledgements. We thank the anonymous NSDI review-

ers and our shepherd Brighten Godfrey for their valuable feedback. We thank Bokun Li, Jinming Wu and Ruilong Ma for their invaluable feedback on an early draft of this paper. This work is supported in part by the National Natural Science Foundation of China under Grants (62101064, 62171057, 62201072, 62071067), in part by the Ministry of Education and China Mobile Joint Fund (MCM20200202), Beijing University of Posts and Telecommunications-China Mobile Research Institute Joint Innovation Center.

References

[1] Delta-net. https://github.com/delta-net/datasets.

[2] Flash artifact for sigcomm22. https://github.com/snlab/flash.

[3] The onos project. https://opennetworking.org/onos/.

[4] Route views. http://www.routeviews.org/.

[5] ABHASHKUMAR, A., GEMBER-JACOBSON, A., AND AKELLA, A. Tiramisu: Fast and general network verification. *arXiv preprint arXiv:1906.02043* (2019).

[6] AL-SHAER, E., AND AL-HAJ, S. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration* (2010), pp. 37–44.

[7] AL-SHAER, E., MARRERO, W., EL-ATAWY, A., AND ELBADAWI, K. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *2009 17th IEEE International Conference on Network Protocols* (2009), IEEE, pp. 123–132.

[8] ALPERNAS, K., MANEVICH, R., PANDA, A., SAGIV, M., SHENKER, S., SHOHAM, S., AND VELNER, Y. Abstract interpretation of stateful networks. In *Static Analysis: 25th International Symposium, SAS 2018, Freiburg, Germany, August 29–31, 2018, Proceedings 25* (2018), Springer, pp. 86–106.

[9] ANDERSEN, H. R. An introduction to binary decision diagrams. *Lecture notes, available online, IT University of Copenhagen* (1997), 5.

[10] BECKETT, R., GUPTA, A., MAHAJAN, R., AND WALKER, D. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 155–168.

[11] BERDE, P., GEROLA, M., HART, J., HIGUCHI, Y., KOBAYASHI, M., KOIDE, T., LANTZ, B., O'CONNOR, B., RADOSLAVOV, P., SNOW, W., ET AL. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking* (2014), pp. 1–6.

[12] DONNELLY, C. Microsoft 365 outage affecting teams, outlook and azure users blamed on 'networking fault'. Website, 2023. https://www.computerweekly.com/news/252529561/Microsoft-365-outage-affecting-Teams-Outlook-and-Azure-users-blamed-on-networking-fault.

[13] FAYAZ, S. K., SHARMA, T., FOGEL, A., MAHAJAN, R., MILLSTEIN, T. D., SEKAR, V., AND VARGHESE, G. Efficient network reachability analysis using a succinct control plane representation. In *OSDI* (2016), vol. 16, pp. 217–232.

[14] GEMBER-JACOBSON, A., VISWANATHAN, R., AKELLA, A., AND MAHAJAN, R. Fast control plane analysis using an abstract representation. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), pp. 300–313.

[15] GUO, D., CHEN, S., GAO, K., XIANG, Q., ZHANG, Y., AND YANG, Y. R. Flash: fast, consistent data plane verification for large-scale network settings. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 314–335.

[16] HERN, A. Google suffers global outage with gmail, youtube and majority of services affected. Website, 2020. https://www.theguardian.com/technology/2020/dec/14/google-suffers-worldwide-outage-with-gmail-youtube-and-other-services-down.

[17] HORN, A., KHERADMAND, A., AND PRASAD, M. R. Delta-net: Real-time network verification using atoms. In *NSDI* (2017), vol. 17, pp. 735–749.

[18] KAZEMIAN, P., CHAN, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *NSDI* (2013), pp. 99–111.

[19] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)* (2012), pp. 113–126.

[20] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying Network-Wide invariants in real time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, Apr. 2013), USENIX Association, pp. 15–27.

[21] LABUSCHAGNE, H. Big rsaweb outage. Website, 2023. https://mybroadband.co.za/news/fibre/478521-big-rsaweb-outage.html.

[22] MAABAR, M. Trie data structure. https://bioinformatics.cvr.ac.uk/trie-data-structure/, 2014.

[23] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P. B., AND KING, S. T. Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review 41*, 4 (2011), 290–301.

[24] PANDA, A., LAHAV, O., ARGYRAKI, K. J., SAGIV, M., AND SHENKER, S. Verifying reachability in networks with mutable datapaths. In *NSDI* (2017), vol. 17, pp. 699–718.

[25] PEDROSA, A. F. S. F. L., WALRAED-SULLIVAN, M., AND MILLSTEIN, R. G. R. M. T. A general approach to network configuration analysis. In *12th USENIX symposium on networked systems design and implementation (NSDI 15)* (2015), pp. 469–483.

[26] PRABHU, S., CHOU, K. Y., KHERADMAND, A., GODFREY, B., AND CAESAR, M. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 953–967.

[27] SPRING, N., MAHAJAN, R., AND WETHERALL, D. Measuring isp topologies with rocketfuel. *ACM SIGCOMM Computer Communication Review 32*, 4 (2002), 133–145.

[28] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Symnet: static checking for stateful networks. In *Proceedings of the 2013 workshop on Hot topics in middleboxes and network function virtualization* (2013), pp. 31–36.

[29] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), pp. 314–327.

[30] VAHIDI, A. Jdd: a pure java bdd and z-bdd library. https://bitbucket.org/vahidi/jdd, 2003.

[31] VELNER, Y., ALPERNAS, K., PANDA, A., RABINOVICH, A., SAGIV, M., SHENKER, S., AND SHOHAM, S. Some complexity results for stateful network verification. In *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings 22* (2016), Springer, pp. 811–830.

[32] WEITZ, K., WOOS, D., TORLAK, E., ERNST, M. D., KRISHNAMURTHY, A., AND TATLOCK, Z. Scalable verification of border gateway protocol configurations with an smt solver. In *Proceedings of the 2016 acm sigplan international conference on object-oriented programming, systems, languages, and applications* (2016), pp. 765–780.

[33] YANG, H., AND LAM, S. S. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking 24*, 2 (2015), 887–900.

[34] YE, F., YU, D., ZHAI, E., LIU, H. H., TIAN, B., YE, Q., WANG, C., WU, X., GUO, T., JIN, C., ET AL. Accuracy, scalability, coverage: A practical configuration verifier on a global wan. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 599–614.

[35] YUAN, Y., MOON, S.-J., UPPAL, S., JIA, L., AND SEKAR, V. Netsmc: A custom symbolic model checker for stateful network verification. In *NSDI* (2020), pp. 181–200.

[36] ZENG, H., ZHANG, S., YE, F., JEYAKUMAR, V., JU, M., LIU, J., MCKEOWN, N., AND VAHDAT, A. Libra: Divide and conquer to verify forwarding tables in huge networks. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)* (2014), pp. 87–99.

[37] ZHANG, P., LIU, X., YANG, H., KANG, N., GU, Z., AND LI, H. Apkeep: Realtime verification for real networks. In *NSDI* (2020), pp. 241–255.

Netcastle: Network Infrastructure Testing At Scale

Rob Sherwood [⋆], *Jinghao Shi* [†], *Ying Zhang* [†], *Neil Spring* [†], *Srikanth Sundaresan* [†], *Jasmeet Bagga* [†]
Prathyusha Peddi [†], *Vineela Kukkadapu* [†], *Rashmi Shrivastava* [†], *Manikantan KR* [†], *Pavan Patil* [†]
Srikrishna Gopu [†], *Varun Varadan* [†], *Ethan Shi* [†], *Hany Morsy* [†], *Yuting Bu* [†], *Renjie Yang* [†]
Rasmus Jönsson [†], *Wei Zhang* [†], *Jesus Jussepen Arredondo* [†], *Diana Saha* [†], *Sean Choi* [‡]

[⋆]*NetDebug.com* [†]*Meta Platforms Inc.* [‡]*Santa Clara University*

Abstract

Network operators have long struggled to achieve reliability. Increased complexity risks surprising interactions, increased downtime, and lost person-hours trying to debug correctness and performance problems in large systems. For these reasons, network operators have also long pushed back on deploying promising network research, fearing the unexpected consequences of increased network complexity. Despite the changes' potential benefits, the corresponding increase in complexity may result in a net loss.

The method to build reliability despite complexity in Software Engineering is testing. In this paper, we use statistics from a large-scale network to identify unique challenges in network testing. To tackle the challenges, we develop Netcastle: a system that provides continuous integration/continuous deployment (CI/CD) *network testing as a service* for 11 different networking teams, across 68 different use-cases, and O(1k) of test devices. Netcastle supports comprehensive network testing, including device-level firmware, datacenter distributed control planes, and backbone centralized controllers, and runs 500K+ network tests per day, a scale and depth of test coverage previously unpublished. We share five years of experiences in building and running Netcastle at Meta.

1 Introduction

Decades of accumulated network deployment experience can be pragmatically summarized as: seemingly simple changes to the network can break things in ways that are both catastrophic and non-obvious even to experts. Other times the breakage is subtle and thus hard to detect and root cause [31]. This unfortunate state of affairs is due to the complex reality of modern networks. Many aspects contribute to complexity including the number and variety of devices, workloads, device types and manufacturers, as well as typical distributed systems problems of state distribution, race conditions, consensus, and propagation delays. Compounding these issues, each of these dimensions of complexity seem to be individually increasing (more devices, more workloads, more device types, etc.) for a multiplicative increase in total complexity.

Any change to the system risks non-obvious performance impact or even outages; increased complexity only exacerbates this danger. Thus, it is natural for network operators to want to limit changes to the network. However, this is not practical in today's world. Many changes are initiated by network operators out of necessity, e.g., for network growth or to replace obsolete, inefficient, or failed devices. Moreover, many changes happen without consultation or control of network operators: new users join the system, usage patterns change, and new services are regularly deployed—all assuming that the network will "just work". Thus, increased complexity coupled with regular, sometimes unvetted, change requires constant vigilance of operators to keep networks reliable.

Modern software engineering promises us that complexity can be managed with increased testing and test automation [2]. This is particularly true in software-as-a-service ("SaaS") models where large distributed systems with thousands of active developers continuously change under the covers, unnoticed by end-users, and rarely introducing bugs. These systems rely heavily on various levels of unit, regression, performance, stress, and end-to-end testing to continuously ensure that the proposed changes do not break desired functionality. Only changes that do not break tests are allowed to land ("continuous integration" - CI) and automated tooling regularly applies these changes to the production systems and monitors the state of new deployments ("continuous deployment" - CD) for regressions. With sufficient test coverage, even non-trivial changes (e.g., a large refactoring) can be performed safely.

It should, in theory be possible to run a network with both high-reliability and a high rate of change, using SaaS principles. However, this assumes that it is possible to verify by exhaustion that the network works correctly before and after a proposed change, which in turn requires writing an offline non-production test for every conceivable situation. This simple assumption is challenging to meet in practice. In this paper, we first use the well-known testing matrix [21] concept to quantify the complexity of network testing in production. In

addition to the theoretical analysis, we reveal the real-world challenges of testing large network systems. Motivated by them, we share the development and operation of a single automated, centralized network testing system: Netcastle, which has been in production for five years at Meta. This paper provides the following contributions:

First, we analyze the complexity of network testing formally using the concept of the testing matrix and quantify testing complexity in a production network. Further, we review the limitations of our original test methodologies and use them to derive the requirements for an idealized *network testing as a service* system (§2) including the need for fungible shared hardware resources, dynamic L1 topologies, and a testing pyramid hierarchy that trade-off control vs. coverage. This is the first work that provides systematic and quantitative analysis of the network testing problem.

Second, we present Netcastle, a collection of software and hardware systems that provide network testing as a service (§4). It hides low-level resource management details with high-level abstractions that ease test development. Its management layer automatically handles test scheduling to maximize resource utilization and minimize noise. In addition, it leverages advanced hardware modules such as optical switch and smart power distribution units (PDUs) to provide flexible topology and isolation in a common physical infrastructure.

Third, We share five concrete testing use cases built on top of Netcastle infrastructure, including data center network testing, wide-area Backbone network testing, FBOSS white-box switch testing, and OpenBMC firmware testing (§5). They not only provide details of real-world network tests but also demonstrate how Netcastle enables them flexibly.

Fourth, We evaluate Netcastle's scalability and usability in §6. It comprises over 500 racks with 3MW power and thousands of switches. It supports 171 test scenarios across multiple teams in Meta. It manages 87K assets and receives 495K reservations per day, which is around 6 reservations per second. The services handle 37M requests daily at an average of 428 queries per second. Moreover, we show its impact with five severe bugs that were caught by Netcastle which would otherwise cause catastrophic outages.

Fifth, as the first to propose and share the *network testing as a service* framework with academia, we discuss the testing implications for network complexity and possible future research directions in §7.

2 Motivation

We first motivate the problem by leveraging a well-known metric in software engineering, the test matrix [21]. Intuitively, complexity is a function of the building blocks of the network: e.g., features, components, and device types. In a well-run system, all individual elements will have corresponding tests to ensure they are functioning correctly; as will the interaction between the elements. Thus, the more complex a system, the more tests needed to validate correctness, and thus the larger the test matrix. For the largest networks, such as at Meta, the testing system complexity itself can limit the rate of change of the network and is therefore worth studying on its own.

To illustrate the above, in this section, we quantify the test matrix for a single software system and show how it can grow multiplicatively over features, tests, etc. We then describe how the composition of systems in a network make the overall test matrix grow exponentially. This untrammelled growth serves as the primary motivation for Netcastle; the bigger the test matrix, the more engineering that is required to tame it. More discussions of test matrix are in §A.

2.1 A Single Project's Testing Matrix

FBOSS [5, 29] is the software that manages the switch hardware of Meta's data centers. Its many functions include translating high-level messages from the routing stack to the underlying hardware, exporting statistics, and sourcing network alerts (e.g., port down). FBOSS is updated across the global fleet at least monthly. With each new code or configuration change, the FBOSS binary is tested against all elements of its test matrix to ensure none of the new changes have (a) broken other features (e.g., does ARP expiration still work?), (b) caused a performance problem (e.g., did the route insertion rate drop below a threshold), or (c) caused a regression in a critical scaling dimension (e.g., maximum routes supported). The FBOSS test matrix has the following dimensions:

- Each switch hardware platform deployed in production [1]: e.g., Wedge40/Wedge100/Wedge400 (rack switches), Minipack/Minipack2 (100G/200G fabric switch)
- Each Vendor SDK type/version: e.g., Broadcom SDK 6.5.17, 6.5.18, SAI
- Each boot mode: cold (empty ASIC memory), warm (pre-populated ASIC memory)
- Each feature test: 800+ different features – see "fboss/fboss/agent/hw/bcm/tests/*" in [29]

Validating that a given change has not caused a regression requires all combinations of (6 hardware types) $\times$ (3 SDK versions) $\times$ (2 boot modes) $\times$ (800 feature tests) $=$ 28,800 tests. Each test can take from a few seconds to tens of minutes to run; a serial run of all FBOSS tests would take (impractically) days to finish. Developers may launch several test runs daily and there are dozens of developers, resulting in hundreds of thousands of test runs per day from this single software project (more data in §6.1.1 and Figure 8).

While it might be tempting to take short cuts in this test matrix, e.g., test more intelligently than the brute-force combination of all dimensions, experience has shown that this can lead to bugs being deployed into production. For example, it is not uncommon for a bug to only affect a single feature on one SDK version and one switch type. Predicting these unlikely interactions in advance is hard and predicting incorrectly can

cause an outage, so our hard-won lesson is that brute-force testing all combinations is necessary.

Software engineering best practices dictate that every new feature or change should add its own tests, thus adding complexity to the system. How much complexity a change adds depends exactly on how it affects the test matrix:

- *New element to an existing dimension*: When the FBOSS team introduced a new switch type, Minipack, they increased the "switch hardware" testing dimension from 5 to 6 devices, resulting in a corresponding 20% increase in tests from 24,000 per run to 28,800 tests per run. This is the least complex type of change.
- *Adding a new test matrix dimension*: This is more complex—even a simple dimension with a binary element doubles the size of the test matrix; more elements would result in a correspondingly larger matrix. For example, the FBOSS team experimented with the Algorithmic Longest Prefix Match (ALPM) route memory storage algorithm which increases route capacity. Since ALPM potentially affected all tests and was not initially deployed pervasively, it required a new binary dimension to the test matrix, ALPM off *vs*. ALPM on, or a 100% increase in test matrix size.
- *Adding multiple new dimensions to the test matrix*: An example of a dimension with potentially more elements is the parameter values for Explicit Congestion Notification (ECN) support. ECN has a threshold parameter for when to mark packets and every queue has a different allocation of the shared buffer—forming two independent testing dimensions. In practice, we explored 4 different ECN threshold values and 3 different queue allocations for a net 1200% increase in test matrix size.

2.2 Multiple Project Combinatorial Explosion

Above, we showed that a single network component can have a very large number of tests; testing the network as a whole across different software components results in a combinatorial explosion of tests, e.g., does feature A in version B of FBOSS correctly inter-operate with feature X in version Y of BGP? For the number of software systems and features that interact, this testing matrix quickly become intractable. To see why, we note that FBOSS updates the global data center fleet software every two weeks, and the Backbone controller and switch firmware (details in §5) every 3-6 weeks. Such frequent large-scale releases are only possible if we can have safe, efficient, and extensive testing.

3 Evolution and Challenges

Before Netcastle, every network team ran their own independent test lab and processes. As a result, we have a number of different physical facilities for data center network testing, new hardware testing, backbone network testing, etc. Each of them is set up manually with a small number of switches under a fixed wiring, and hard-coded with static configurations. Making changes to the testbed often requires physical access, together with manual and ad-hoc efforts.

We term these disparate labs "traditional" and in this section, describe key problems with that approach that motivated the Netcastle design. Network testing labs must be isolated from the production network, but this isolation and non-standard testing configurations lead to maintenance challenges. Thus a common theme of traditional test lab problems is "lab rot"— the tendency for labs to accumulate old configurations, forgotten topologies, unmanaged devices, and unclear ownership. Lab rot increases uncertainty in the test signal (is it my code or the lab that is broken?), risks security problems (e.g., missing patches), and wastes resources that users hoard to protect against the misconfigurations of others. This generation of testing exposed several challenges that leads to the Netcastle design, which is summarized in Table 1.

3.1 Physical resource limits

Limited Lab Scale, Unbounded Device Heterogeneity. Test labs are an extra expense, so are typically much smaller than production networks. When a test lab is allocated to each individual team, they are particularly limited in size, making it difficult to find problems that only appear at production scale (e.g., table exhaustion) or infrequently (e.g., race conditions).

At the same time, network testing is meant to confirm that device A will interoperate with device B, across all deployed device types and all relevant configurations. Compounding this interoperability problem is that meeting the relevant specification or RFC alone is not necessarily sufficient; real interoperability requires "bug-level compatibility" testing which requires all n^2 pairs. Further, more device types are added to the network faster than old device types are decommissioned, resulting in unbounded device heterogeneity, and high demand for testing. Because individual teams maintain small scale test labs, this results in compromises to test coverage.

Device, Power, and Space Fragmentation. Physical realities mean that at lab construction time, each row and each rack within that row is allocated an amount of space (rack units) and electrical power (kilowatts) that is cost prohibitive to change. However, many test setups require locality in the form of direct physical connection. This means that when constructing topologies of nearby resources that can be directly connected, fragmentation is possible: devices in a row that are not used by one test setup may be too few to be usable by another. This fragmentation can be aggravated if test device use is of unbounded duration or devices cannot be reclaimed.

3.2 Automating test configurations

Static, Manually Maintained Topologies. A hardware test network must support many topologies: physical cables that di-

Network Testing Challenges		Addressed By Netcastle
Physical resource limits	Disparate Testing Labs	Automated Device Enumeration + Lab Syncer (§4.5)
	Limited Scale	Fungible Centralized Resource Pool (§4.1)
	Device, Power, Space Fragmentation	Centralized Fiber Infrastructure (§4.2, §4.3)
Automate config	Static, Manual Topologies	Any-To-Any Optical Interconnects (§4.3)
Manage and schedule tests	Large Testing Matrices	Netcastle Test Runner (§4.4) parallelization
	Test Harness Hording	Finite Reservations + Usage Monitoring (§4.1)
	Testing Conflicts	Reservations (§4.1, §7.7)
Reduce test noise	Broken Testing Devices	Lab Doctor (§4.6)
	Test devices not maintained	Lab Doctor (§4.6) + Finite Reservations (§4.1, §7.5)
Debug tests	Network Testing Distributed System	Respect the Testing Pyramid (§7.3)

Table 1: Summary of Problems Resolved By Netcastle Design

rectly connect the devices under test. It is not practical to manually rearrange cables at the time scales of when tests run: cable moves can take up to a day to schedule from onsite personnel, while tests run many times per second. Instead, available devices are partitioned into different manually configured topologies, e.g., Clos, star, or snake. With this partitioning, it is not sufficient to have a device of type A available; it must also be cabled appropriately. Ultimately, this means more device types are needed for tests with different topologies.

At the same time, the manual maintenance that supports standard topologies makes it more difficult to manage one-off, experimental topologies. "Innovative" cabling decisions (i.e., "who ran this fiber through the ceiling!? and what will break if I unplug it?") meant to support these topologies can become difficult to manage and contribute to lab rot.

3.3 Managing and scheduling tests

Test Equipment Hoarding. Test lab users tend to hoard equipment, not through malice, but as a practical reaction to the difficulty of acquiring reliable test gear. Once a user has gone through the significant effort to get a test harness installed, debugged, and working correctly, they are rightly hesitant to release ownership because the cost of setting it all back up again is high. Hoarding aggravates the problem of limited lab scale (fewer available resources of specific types) and fragmentation (fewer available resources in specific racks).
Test conflicts. Unfortunately, one test may adversely impact another. Although labs are designed for test isolation, due to hidden dependencies on external services, unclear ownership, and raw complexity it is inevitable that some amount of test breakage occurs due to stepping on toes. A simple example is mistakenly believing a device/cable was not in use and changing it in the middle of someone else's test . More subtly, networking devices often have non-obvious inter-dependencies: the Wedge series of switches have a main CPU complex running Linux (the "microserver") and a completely separate baseboard management controller (BMC [10]). This architecture allowed a different developer to mistakenly log into each system on the same switch and run tests in parallel with apparent but incomplete isolation. In most cases a test on the microserver would not affect a test on the BMC, but under certain test workloads, e.g., PCI controller reset, it would cause subtle and hard to debug test breakage. Also, we discuss in § 7.7 a hard-to-fix architectural issue in traditional labs where test equipment often depended on other test equipment.

3.4 Reducing testing signal noise

Tests leave devices in complex, broken states. Tests intentionally run on questionable code, configurations, and topologies. As a result, they can break devices in obscure and hard to revert ways. For example, a test configuration may disconnect a switch from its console server and management interfaces, making it unreachable without onsite intervention. In extreme cases, e.g., bad firmware, it is possible to cause permanent physical damage to the hardware. This need to verify and debug test equipment before use both justifies the desire to hoard test equipment and also takes resources offline, compounding problems of scale and fragmentation.
Test devices are not maintained like production devices. Large scale production networks are maintained with ruthless homogeneity and pervasive automation. Because lab networks are heterogeneous and transient, they are both hard to automate and cannot be easily directly managed with existing production management tooling. As a result, software upgrades, security patches, system problem detection ("did the disk fill up?", "did the optic stop working?"), hardware fault detection, etc., typically lag behind the production network, leaving devices in various states of disrepair.

3.5 Debugging complex tests

Network testing is distributed systems. While a test setup is simple to sketch on paper, e.g., "n_1 nodes from vendor X, n_2 nodes from vendor Y, running software of a certain version and a certain topology, send messages $m_1 \ldots m_n$, and a link goes down at time t", actually deploying and validating that these parameters are setup and timed correctly is a complex distributed systems problem. Test devices can be unavailable,

temporarily lose connectivity, have bugs independent from the target of the test (e.g., from previous tests), etc. Similarly, correctly sequencing the control signals that cause test events, e.g., "send a BGP HELLO and then the link goes down", and read their results so that the test runner's understanding of the current state correctly reflects the physical reality ("did the link actually go down? Before or after the HELLO?") reduces directly to standard distributed systems consensus problems. And while one might expect that test labs do not run in a byzantine adversarial model and thus consensus should be easy, practical experience shows that significant time is spent debugging the tests themselves; and the bigger and more complicated the test, the more things that can go wrong. Ultimately, tests need to be highly repeatable and problems in the test, e.g., 1 in 1000 race conditions in the test code adds unacceptable noise if it is supposed to catch even less frequent race conditions in the systems under test.

4 Netcastle Testing Infrastructure

The Netcastle team provides "network test infrastructure as a service". The team does not write the tests, instead they provide and maintain the backend software and lab resources, and document best practices so that partner teams (in § 5) can focus on testing their own code instead of on the challenges in § 2. Figure 1 shows this interface. In this section, we describe how Netcastle addresses the problems described in § 3. The relationship between the problems and features to address them is summarized in Table 1. By using these complementary solutions, more and larger tests are made possible, often by avoiding "lab rot".

The goal of Netcastle is to combine independent, smaller test labs from different teams into a communal pool of test resources, eliminating fragmentation and scale limitations. The project aims to replicate the cloud IaaS experience where users can reserve virtual machines via API without worrying about the physical location or available resources. Netcastle offers the same experience for reserving test equipment with strong test isolation, freeing developers from worrying about backend infrastructure maintenance and allowing them to focus on writing tests. To accomplish this vision, Netcastle provides many tools, processes, and points of integration to cover in detail, so we summarize the most innovative elements here and illustrate how they are used in Figure 3.

4.1 Centralized Reservations Guard Resources

In Netcastle, all test equipment must be reserved for finite time through a centralized system; users can query and update this reservation system via CLI, web interface, or a remote API that supports automation. Resources are annotated with key/-value pair attributes such as CPU/device type, kernel version, etc., enabling queries that find resources by these attributes. Individual resources can be grouped into larger atomically reservable units ("ensembles"), e.g., reserving an entire Clos topology rather than individual devices. This simplifies the user interaction and avoids reservation deadlock.

The centralized reservation system also has less obvious benefits. If the requested resources are unavailable, the caller can queue to wait until they become available, which improves resource utilization, particularly for automated testing (see § 7.5 for details and where this can go wrong). Keeping records of these wait times helps to identify which resource types are in high enough demand to warrant adding more, and instances when a component has failed when a queue is unexpectedly stalled. Automated tools can programmatically determine whether a device is participating in a test and act appropriately, for example, alerts, software updates, and other maintenance activities are suppressed for lab devices that are are participating in a test. Requiring each reservation to be of limited duration reduces resource hoarding and allows better sharing of finite resources. Similarly the reservation system provides instrumentation to monitor end-to-end test system performance (e.g., "the queue for routers of type Y is now 45 minutes long—something must be wrong," e.g., like § 7.5).

Given the importance of the centralized reservation system for Netcastle, it needs to have high performance (130k+ reservations a day, see § 6.1.1) and high availability (downtime impacts developer productivity). While intuitively trivial to build, the reservation system had to be redesigned/replaced three times as load increased over time.

4.2 Centralized Fiber Infrastructure

To remove physical locality as a constraint, and reduce resource waste through fragmentation, we deploy centralized passive fiber infrastructure to all test racks. That is, from each rack in the lab, we pull many pairs of fiber (96 or 192 pairs, depending on rack type) to a single central passive "fiber row" that is shared by and sized for the whole lab. By attaching a passive fiber patch cable from the corresponding ports in the central fiber row, any two devices anywhere in the lab can be directly L1 connected. This enables lab devices to be placed wherever there is physical space and power, making the devices interchangeable and reducing fragmentation.

The centralized fiber row is also an ideal place to deploy test equipment shared between teams, including packet generators and passive optical switches. Decoupling physical locality also reduces the number of variations of device deployments, which simplifies automation and reduces lab rot.

4.3 Any-To-Any Optical Interconnects

In addition to manually connecting patch cables in the central fiber row, Netcastle also supports the ability to dynamically, programmatically, create L1 optical topologies. Despite the investment in the central fiber row and the existence of commercial-off-the-shelf programmable passive optical

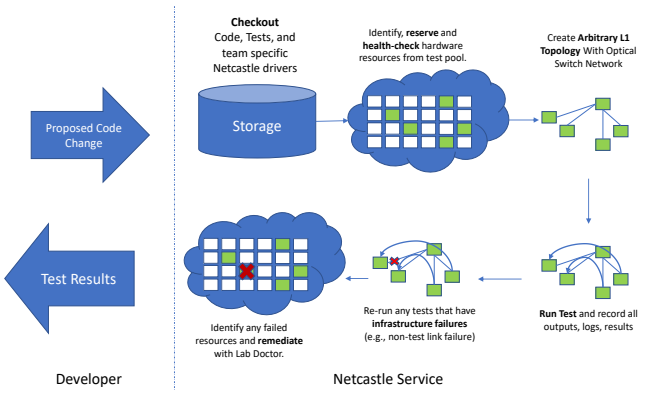


Figure 1: Netcastle testing as a service model.

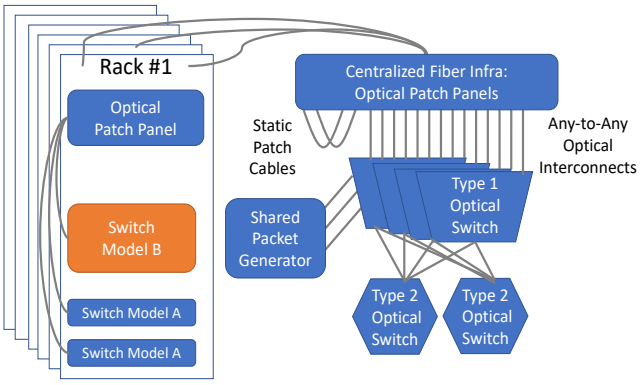


Figure 2: Physical Infrastructure of a Netcastle Lab

switches [14, 15, 33], this is non-trivial due to scale and link setup latencies. Specifically, in order for the link to come up between two devices, the signal to noise ratio, as measured by "light loss level", must be lower than manufacturer specifications. For example, MSA, a common 100Gbps optical standard, allows for a light loss budget of up to 5dB total noise [35] between two devices, including passive connectors (∼0.15–0.5 dB per connector) and the fiber itself (0.5 dB/KM for single mode or 1.0 dB/KM for multimode), must be less than 5dB in order to achieve a high quality connection [24].

Existing commercial passive optical switches fall in to two broad groups. The first use Micro-Electromechanical Systems (MEMS) mirrors or crystals to bend light [14,33]; establishing a link relatively fast(∼10ms) but adding significant light loss (2.5–3dB). They have medium port densities (200–400 ports per device). The second use robotic arms to manage physical passive fabric cables [15]; these can take minutes to establish each link but have light loss equivalent to passive gear (0.5dB per connector), and support high port densities of over 1000 ports per device. Our challenge in making these pieces scale is twofold: first, developer time is important, so it would be prohibitive to have to wait hours to setup a large network topology using only robotic arm based switches. Second, for the number of devices in our labs, there is not a single device big enough to support any-to-any connectivity. We cannot use a "multi-hop" passive optical network with chained optical switches, because it would exceed our light loss budget (e.g., 2 devices each at 2.5dB loss, plus optical connectors, exceeds 5.0 dB). Complicating matters, our data centers prefer OCP [32] standard optics which are cost optimized and have a smaller light loss budget of 3.5dB. We could use different optics (e.g., Multi-Source Agreement or other standards) in our lab, but then we risk testing on equipment that are not used in production, which could undermine our test results.

Faced with these multi-dimensional problems, we developed a hybrid solution that allows limited any-to-any connectivity in our labs (Figure 2). The first insight is that the light loss budget for a given standard (3.5dB for OCP, 5.0 for MSA) is really a *minimum* guarantee and, due to manufacturing variance, a device may support somewhat higher light loss. In practice, we found that most of our nominally 3.5dB OCP optics in fact supported a light loss budget of 5dB. Second, after discussion with multiple MEMS manufacturers, we found that stated light loss for an optical switch (e.g., 2.5–3.0 dB) is a conservative *maximum*; real light loss varies depending on the distance across the device from the input port to the output port. In practice, by carefully picking input and output fiber ports that are physically close to each other, we can significantly reduce light loss. Last, to balance the trade-off between high density, but slow, and and medium density, but fast, connectivity switches, we build a hybrid leaf-spine fabric of optical switches. The leaf nodes which connect directly to the devices under test use the medium density fast connectivity optical switches. We use the high density with slow connectivity switches as spine switches. Then, similar to leaf-spine topologies with electrical-optical switches in the DC, we connect every leaf optical switch to every spine optical switch. As a result, we can achieve any-to-any connectivity across our leaf-spine optical switch fabric with a centralized scheduler: when a developer wants to create an L1 connection between two devices, the centralized scheduler looks up which leaf switches and ports the devices are connected to and allocates a leaf-to-spine-to-leaf optical circuit that is already set up and physically close to the input ports. Thus with this design, we get all three of our desired properties of scale, meeting our light loss budget, and quick link setup.

This any-to-any topology-on-demand feature has enabled a number of network-wide test use cases. For example, examining different generations of data center networks (§5, §6) require creating arbitrary topologies quickly, running tests, and tearing the topologies down. However, topology dynamics introduce additional complexity and potential failures (§7).

4.4 Netcastle Test Runner

Netcastle test "runner" is a unified interface for launching tests and parsing results. The runner combines an array of workflows, including one-off interactive manual testing on

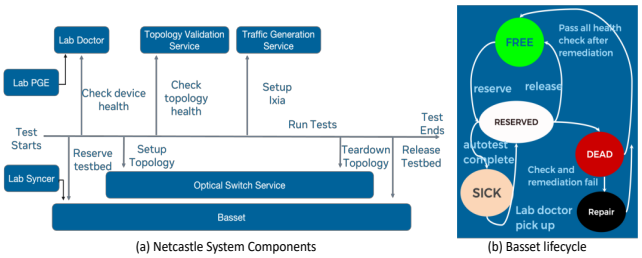


Figure 3: Netcastle Components.

single devices, batch parallel processing across thousands of devices ("test this code with all feature combinations on all types of devices"), to long running stress runs ("run this test 10k times and count failed runs"). The runner automates calls to the reservation system and lab doctor, copies test code and configurations into place, parses the test output, and reports the results. In addition to its own internal accounting databases, the runner must integrate with the graphical tools that developers use to review code, the company-wide test result database that monitors the test results history (e.g., to differentiate between flaky tests vs. tests that are newly consistently failing), and the CI/CD systems that decide if a given build is stable enough to move to the next step in the deployment cycle. For large test runs, the runner also handles sharding across multiple assets in parallel and retrying tests when assets fail for infrastructure reasons. It parses the test output and decides if a given test failure is legitimate (e.g., a bug) vs. a failure in the underlying test infrastructure (§ 7).

Like the centralized reservation system, Netcastle runner is intuitively simple; however, the combinations of use-cases, systems that it must integrate with, and the distributed systems problems associated with setting up and running multi-asset tests, has made it a non-trivial piece of software (§ 7.5).

4.5 Lab Syncer: Asset Verification

To combat lab rot, Netcastle deploys many pieces of automated tooling. *Lab syncer* compares dynamically inferred data (e.g., from connecting to every port on a console server, and from screen scraping management switches) to a centralized configuration to discover devices and report changes to the assets under management. Lab Syncer also verifies that all assets have a common collection of addresses and metadata, e.g., that the reservation tags (device type, kernel version, etc.) match reality and that every device has a proper DNS name.

4.6 Lab Doctor: Device Remediation

The *lab doctor* performs health checks on devices as they exit a reservation to prevent broken resources from returning to the resource pool. It applies a well-known set of remediations to unhealthy devices; it also runs periodically on unreserved assets to ensure that they are healthy. Unhealthy assets that cannot be automatically fixed are marked *DEAD* until manual investigation. Figure 3(b) shows the complete asset life cycle.

```
# 1. Reserve testbed
testbed = Basset.reserve("dc_test_pool", purpose="Testing")
# 2. Check individual device health
is_healty, details = LabDoctor.check_health(testbed)
if not is_healthy:
  # this will be classified as infra error
  raise TestbedError(f"Testbed is not healthy: {details}")
# 3. Reconfigure connections to achieve desired topology
OpticalSwitchService.setUp(testbed, desired_topology)
# 4. Check topology level healthiness
TopologyValidationService.validate(testbed)
# 5. Connect to Ixia traffic generator
TrafficGenerationService.setup(testbed)
# 6. Run tests, example: warmboot without packet loss
# 6.1 Start traffic
TrafficGenerationService.start_traffic(testbed, TRAFFIC_SPEC)
# 6.2 Perform warmboot
forwarding_stack_service.clear_counter("pkt_loss")
forwarding_stack_service.perform_warmboot()
# 6.3 Assert no packet loss
pkt_loss = forwarding_stack_service.get_counter("pkt_loss")
TrafficGenerationService.stop_traffic(testbed)
assertEqual(pkt_loss, 0)
# 7. Tear down topology and release testbed
OpticalSwitchService.tearDown(testbed)
Basset.release(testbed)
```

Figure 4: Example Netcastle Test Case pseudocode.

4.7 Putting it together

Figure 3(a) shows how each component interacts with the test life cycle. Figure 4 is a pseudocode example illustrating how a test developer utilizes Netcastle. The process begins by reserving a test pool of resources and calling on Lab Doctor to check device health status. Then, the optical switch is configured to set up the topology and the traffic generation service (step 5), followed by a measurement to verify its accuracy. Next, in step 6, the test developer runs a set of tests, which we will explain further in §5. Finally, the topology is dismantled, and all assets are returned to the pool. This process ensures that resources are efficiently used and that the test environment is well-maintained throughout the process.

5 Netcastle Use Cases

In this section, we present five test scenarios that use Netcastle and how they relate to experiences from § 7.

5.1 Facebook Open Switch Software Test

FBOSS test aims to achieve rapid, automatic test and deployment for switch software [5, 29]. As we discuss in § 3, every FBOSS code change must be tested across all features, device types, and environments before commit and deployment. FBOSS updates the global data center fleet software every two weeks; this cadence is only possible and safe through extensive testing using Netcastle in the following aspects.

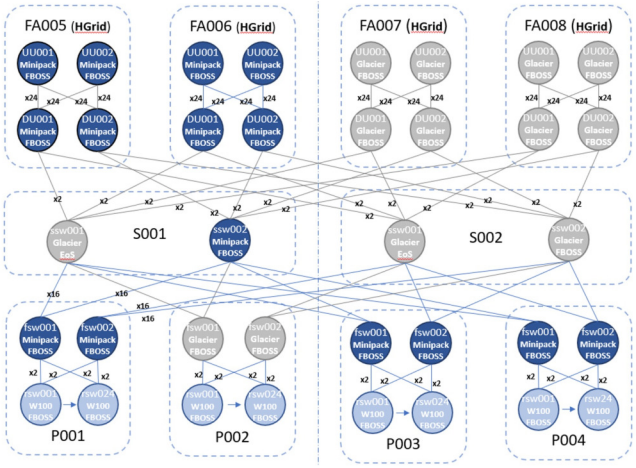


Figure 5: DC Network Testing Usecase.

1. **Hardware functional tests** verify that FBOSS works seamlessly on multiple hardware on a single switch box. About 800 test cases per switch ASIC and SDK combination verify core functions (e.g., QoS, load balancing), they run on every code commit and complete in about an hour.
2. **Hardware benchmark tests** ensure that the current version of FBOSS meets a certain performance threshold (e.g., warmboot time, ECMP shrink time, TX/RX rates). About 20 test cases per switch hardware type and SDK combinations verify that changes meet performance criteria, running once per day.
3. **Integration tests** verify FBOSS switches work seamlessly with other entities in the data center. These tests run on every code commit and completes in about 10 minutes.

Netcastle enables the FBOSS team to focus on writing tests for the software functionality, as Netcastle is able to obtain quality test signals that strictly distinguishes test failures between the software, hardware and the test infrastructure. Furthermore, the FBOSS team is able to have a more predictable deployment cycle, as they can predict how long a test cycle will take and how much coverage the the test signals provide.

5.2 Board Management Controller Test

OpenBMC [10] is an open-source software for managing board management controllers (BMCs) that are embedded in servers and switches to control hardware such as fan speed and remote server access. As the code is shared among switches and servers, thorough testing of OpenBMC is needed to prevent bad code from causing extensive damage. OpenBMC releases new versions 4-6 times per platform per year. The main challenges of testing OpenBMC are the diverse hardware it supports—over 19 different types at Meta—and the need to provide test signals publicly as it is open-source.

Netcastle solves these challenges as follows. For every pull request, Netcastle runs OpenBMC tests on variety of hardware types to catch regressions, and makes the test signals available to the open-source repository, facilitating debugging

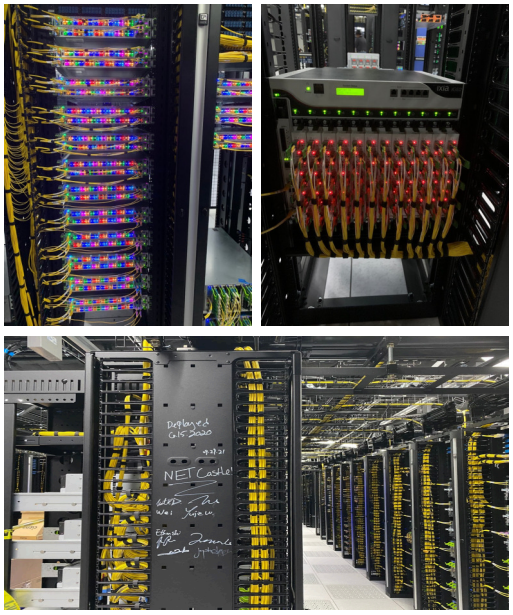


Figure 6: Netcastle Hardware Platform.

efforts by the community. Finally, Netcastle is used in validating OpenBMC image binaries. Thus, by utilizing Netcastle, OpenBMC is able to avoid building another set of testing infrastructure and a build pipeline, and is simultaneously able to reap the benefits of obtaining high quality test signals.

5.3 DC Network Testing

The Data Center Network Engineering team conducts multi-device testing on FBOSS switches and other vendor switches, focusing on network-wide objectives and configuration correctness. This includes designing and validating data center topology, managing network capacity, and more. Figure 5 shows a common set up for DC network testing. It faithfully mimics the production fabric topology in a smaller scale [5]: four pods are set up with two ToRs, connecting to two fabric switches, interconnected by two spine switches at the aggregation layer, and four aggregation switches at the core layer. The tests mainly consists of the following steps: test configuration setup, inducing disruptive events (e.g., software/hardware reboot/crashes, switch drain/undrain, link flaps), and recording the results (e.g., resource utilization and packet loss during the disruptive event). In the example in Figure 4, step 6 tests the packet loss during switch warmboot operations.

Testing new features presents challenges such as isolating the correct topology, and filtering false test signals from manually induced test failures. Netcastle addresses these challenges by enabling easy isolation of the testing topology through simple scripts for controlling switch interfaces, allowing configuration to reserve and use specific switches for testing, and filtering bad signals that are manually induced.

5.4 Express Backbone Controller

The EBB Controller [17] is the SDN WAN controller that manages Meta's inter-region global backbone network. The team has integrated Netcastle into its CI/CD pipeline since 2019 and ships a new release every 3-4 weeks. They required external traffic generators for their test cases; this pushed the Netcastle team to automate reservations and access to third party traffic generation tools. In this topology, software VLANs mimic edge data centers. Five lab express backbone (leb) routers are interconnected with parallel 200G physical links. This setting is used to test new controller software, traffic engineering algorithms, new EB router platforms, and SDKs, among other operations.

5.5 Hardware NPI

New Product Introduction (NPI) validates prototype networking gear and verifies that the hardware works correctly. This includes, among others, environmental tests, sensor and power checks, and data plane performance tests. Data plane testing is simplified by connecting ports in a "snake" topology: Port i is connected to port $i+1 \; \forall \; i \mod 2 = 0$, and traffic sent out of one port is received by the next. This reduces the traffic generator requirements from the full number of ports to only two; one source and one sink.

6 Evaluation

In this section, we demonstrate the impact of Netcastle with both real-world statistics and incidents caught.

6.1 System Evaluation

6.1.1 Scale: Tests, Daily Tests, and Reservations

Netcastle has grown to be the primary testing infrastructure for the Meta network. Figure 7 shows the cumulative number of tests developed on Netcastle over three years. From §2.1, the FBOSS team alone has 800+ different tests before considering different hardware and software configurations. In total, there are close to 4500 total tests developed across 20+ teams. The significant increase from week 40-80 represents Netcastle's largest adoption period. The steady increase after week 80 shows its continuous benefit as the network grows.

Another metric of scale is the number of test runs per day. Figure 8 shows the amount of daily tests in the course of three months, ranging from 300K to as high as 700K. The average daily test runs is around 500K, demonstrating its scale.

We further demonstrate the lab reservation's usage in Figure 9 for a period of recent 4 months. The reservation system can scale to as large as 130K reservations per day. The noticeable drop around day 90 is caused by the ramping down of development activity close to holidays.

6.1.2 Effectiveness: Failed Tests and Errors

The purpose of Netcastle is to catch test failures before bugs manifest in production. Because Netcastle is integrated in both code review stages and continuous deployment stages, we can identify test failures at each stage. Figure 10 shows that in the past four months, Netcastle caught over 500K test failures just during code commit time. Further, the dashed curve represents the necessity of testing during deployment, as it captured issues with other dependent systems. Such failures are orders of magnitude fewer than the failures that manifest at code commit time. Yet, the volume is high—100K.

Not only are the test failures caught by Netcastle abundant but they are also high-quality. Tests in Netcastle are grouped into jobs for efficiency. A Netcastle test job can have 3 outcomes: succeed (all tests passed), test failure, or infra error. Jobs with infra errors are automatically retried and reported as "Infra Error - No Signal" when retry limit is exhausted. This helps developers focus on legitimate test failures that indicate code bugs, rather than noise caused by infra issues.

Figure 11 shows the fraction of cumulative number of Netcastle jobs that fail because of infra error and other reasons (true failures). We observed the fraction of jobs with infra errors is higher than jobs with legitimate test failures. Infra errors increase as the testing framework scales and becomes more complex, while the legitimate test failures grows flatter, demonstrating both the necessity and effectiveness of distinguishing legitimate test failures from infra issues, especially as the physical infrastructure of Netcastle grows.

6.1.3 Optical Switch Reconfiguration

Finally, we present the need for topology reconfiguration in Figure 12. The number of reconfigurations per day ranges from 40 to 160. The high frequency is due to multiple teams co-sharing the testing facility, especially the traffic generator which relies on the optical reconfiguration to inject high volume of traffic at different parts of the topology.

6.2 Five Bugs Kept Out of Production

Netcastle tests integrate with the Meta code commit process. It generates 150K test failures yearly, some of which have prevented catastrophic failures. We share five examples below.

6.2.1 OpenBMC Watchdog Didn't Reset

In January 2021, two changes from the open source OpenBMC upstream were pulled and submitted for review. Each change was tested and landed successfully. However, the two changes were not tested together. With both changes, OpenBMC failed to reset the on-board watch dog timer, and rebooted every 5 minutes. This failure resulted in a lab outage, as all lab devices were rebooting repeatedly and all OpenBMC

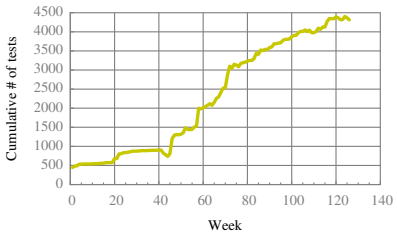


Figure 7: Tests increase weekly.

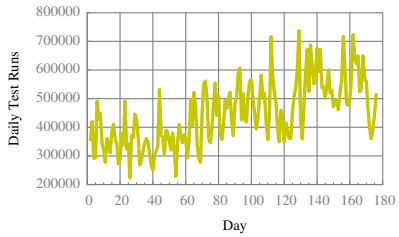


Figure 8: Netcastle test runs per day.

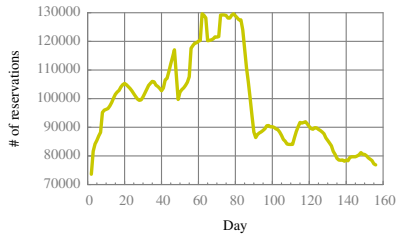


Figure 9: Daily lab reservations.

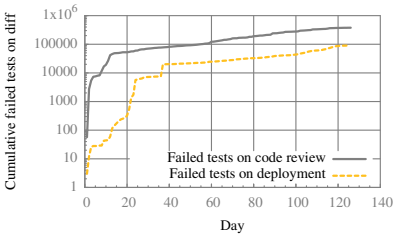


Figure 10: Failed tests by stage.

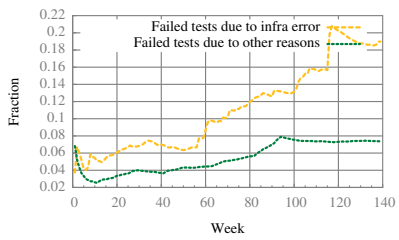


Figure 11: Infra error and test failures.

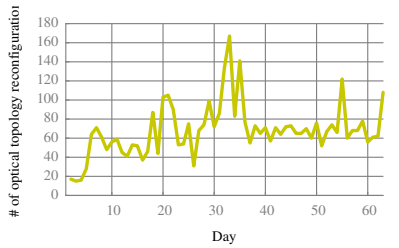


Figure 12: Optical reconfigurations.

tests failed. This bug was caught well before reaching production, and since the test failures were immediate and substantial, it took less than an hour to root cause. Before Netcastle, a similar bug took months to root cause.

6.2.2 FBOSS RIBv2 Redesign Missed Features

The Routing Information Base (RIB) is a critical data structure in FBOSS that stores routes learned from routing protocols and prepares the Forwarding Information Base (FIB) for programming the switch ASICs. Over time, FBOSS has added many use cases and optimizations to the original RIB, including multi-protocol support, fast lookups via radix trees, UCMP, and MPLS. However, the implementation of these features increased the difficulty of adding new use cases and scaling needs. To address this challenge, the FBOSS team initiated a complete redesign of this core data structure. Despite being reimplemented by experienced engineers, massive errors were found in the code: MPLS, mirroring, and traffic engineering did not work as expected. With tests in place, issues were quickly fixed.

6.2.3 Warmboot Packet Loss due to Agent Race

Warmboot is a switch ASIC feature that allows forwarding packets while we we update the FBOSS agent software. Netcastle has the ability to run tests after warmboot from one commit to another, allowing explicit testing of upgrades and rollbacks. In one upgrade, we discovered that a subset of prefixes were experiencing high input discards, and further investigation revealed that a drop rule had been installed in the ASIC but not in the control plane. We also observed that changing the traffic pattern caused previously functioning destination prefixes to fail. We traced the issue to a race condition in the agent software that occurred when adding and removing FDB entries during warmboot with traffic running. A hidden drop rule caused by a race condition would have been extremely disruptive. This issue was challenging to reproduce even in a controlled lab setup; in production, without control over the traffic, it would have been nearly impossible.

6.2.4 Micro-Loop Causing Packet Loss During Drains

"Draining" redirects traffic away from a switch that is being taken offline for maintenance. Our tests check that packet loss during drains lasts less than 30ms. However, during one incident, packet loss continued for 300ms. The root cause of the issue was a micro-loop that formed during routing convergence. When the switch is drained, its transit neighbors are expected to send BGP withdraw messages to the spine to stop attracting traffic. However, the routes were unprogrammed in the Forwarding Information Base (FIB) before the BGP withdrawal was sent to the spine. As a result, the traffic coming from the spine was sent back by the transit switches. Packets were discarded when their time-to-live (TTL) expired.

If this issue had not been detected in the lab, it could have caused significant problems during drain operations, which are performed hundreds of times a day. This black-hole of traffic for 300ms will be handled differently by applications, with responses ranging from normal retransmission and congestion response to abandoning connections and failing queries. The small scale and duration makes such events difficult to detect and troubleshoot in a production environment. Fortunately, using the granular metrics provided by Netcastle, we were able to consistently reproduce and test various hypotheses.

6.2.5 "Factory Reset" Backbone Lab Devices

"Provisioning" is the process of configuring devices (switches or servers) to an operational state. Recently, a faulty provision-

ing workflow led to network devices entering a mode called zero-touch-provisioning: essentially a factory reset state. This state lacked the necessary boot configuration required for operation within our backbone network. This problem was first observed in the Netcastle infrastructure since we were testing the new workflow in the lab. However, we promptly blocked the code change responsible for the misconfiguration to prevent any impact on production. If this issue had not been discovered in the Netcastle and the new provisioning workflow had been rolled out, it would have resulted in a complex and lengthy recovery process. Currently, in EBB, we perform per-plane rebuilds, which involves rebuilding a network shard of 42 devices in parallel. In the worst-case scenario, the bug would have brought down 42 backbone devices, resulting in the loss of hundreds of Terabytes of capacity.

These examples illustrate Netcastle's capability to identify both production tooling issues and device software bugs.

7 Experiences

7.1 Test Signal Quality Is Paramount

Obvious to state but infuriating in the details, the testing signal, e.g., did a given test 'pass' or 'fail', as relayed to the developers/automated tooling must have extremely low false positive and false negative rates. A false negative, say due to a failure in the infrastructure rather than a regression in the code, is often hard to distinguish from a rare bug which can cause developers to waste significant time. A false positive, say when the test runner fails to catch that a test is actually failing but reports as "pass", allows buggy code to be pushed deeper into the continuous deployment ("CD") pipeline. Deeper in the CD pipeline has more noise and variance and so bugs there are harder to replicate, isolate, and debug, and ultimately risk being pushed into production. Worse, any noise in the test signal degrades developer confidence and can cause them to ignore test results or make them less inclined to write more tests in the future. The key insight is, intuitively similar to the famous Nyquist-Shannon sampling theorem, if one wants to catch a code that fails at a certain rate, e.g., 1 in 1000 run race condition, then the corresponding test signal must have proportionately much higher sampling rate/stability (e.g., 1 in 100k test runs result in a false positive). Getting the test infrastructure to this level of engineering confidence is no small effort. Most of the experiences shared below derive from this main point about test signal quality.

7.2 The Test Infrastructure Itself Must Be Testable

The potential for test signal noise is highest when there are bugs in the Netcastle software itself, such as in critical components like the runner. To prevent this, Netcastle has 100% unit test coverage and a complex system of tests that run with each release. In one incident, a bug in the centralized resource reservation system caused conflicting tests to produce a large amount of noise and risked the project's reputation in its early phases. This testing bug resulted in a significant loss of developers' trust, which put the project at risk.

7.3 Respect the Testing Pyramid

The Testing Pyramid [36] is a widely accepted testing abstraction that emphasizes the importance of writing simple tests with fewer moving parts. The number of tests for each type (e.g., unit, component, system, end-to-end) should be inversely proportional to their complexity, creating a pyramid shape. This principle has significant implications for Netcastle and network testing. Despite networking being a distributed system, writing distributed tests directly, such as designing a test with 20 different routers to verify BGP convergence properties, introduces significant testing noise. Therefore, Netcastle and developer teams constantly simplify tests to reduce noise and test the same logic with fewer moving parts, further down the pyramid. An example of simplifying tests is the FBOSS hardware test, where instead of using external devices to source/sink packets for data center switch functional testing, individual switch ports are put into a temporary loopback mode. This replaces an earlier setup where switches were connected to physical servers and test packets were sourced/sinked via remote procedure call (RPC). The move from a distributed system with complex RPC calls to a single machine with only local processing reduced testing noise and eliminated the need to maintain servers.

7.4 Infrastructure vs. Code Errors

When infrastructure errors occur in Netcastle, effort is put into distinguishing them from code errors to avoid disrupting developers. Skipped tests can be retried on different devices to produce a more reliable result, and infrastructure errors are handled by the Netcastle oncall team while code errors are sent to individual developer teams. Lab doctor tests are used to check device health before and after each run, and any changes in device health can help determine whether an error is due to infrastructure or code issues. However, there is no perfect way to separate infrastructure from code errors, and manual triage is often required to determine the root cause.

7.5 Do Not Under Estimate/Engineer Complexity of Testing Workloads

Many of the initial false steps in the project can be attributed to under-estimating and ultimately under-engineering the Netcastle software systems. The team initially underestimated the need for production-level performance, monitoring, and alerting in the testing system, assuming that running tests were not

production workloads. However, the team realized the importance of testing after encountering bugs in the Netcastle code that delayed test signal for code needed to fix a bug in production. The root cause was the lack of a service-level agreement (SLA) for test result return time. Now, Netcastle has a firm SLA with each developer team, and violations trigger SEV alerts [9] and standard production system management procedures to prevent future incidents.

7.6 Manage Latency vs. Completeness

As test coverage grew, we found that not all tests could be run and reported to developers in a timely manner, such as within 30 minutes while they waited for test results to see if their code could be deployed. To compromise, we created a hierarchy of tests that ran at different time scales and with varying levels of coverage. This model mirrors the testing pyramid, with tests categorized as quick and low coverage, periodic and medium coverage, and comprehensive and high coverage. This classification evolved over time due to the need for faster test signal and the growing number of tests, making manual classification impractical. Had we used hierarchies from the beginning, we could have avoided the need for the clunky one-test-per-class heuristic.

7.7 Test Equipment Should Not Depend On Other Test Equipment

In our earlier test labs, we discovered that managing a collection of switches in a standard Clos/leaf-spine topology as individual test equipment can cause interference between tests running on different switches. For instance, tests running on spine switches could unexpectedly disrupt tests on leaf switches and vice versa, leading to incorrect assumptions about network connectivity and causing test failures. Additionally, some leaf nodes had servers attached to them that were only accessible if the leaf switch was running all necessary software, which was not always the case.

To address this issue, we took two steps. First, we designated upstream devices, such as spines connected to leaves, as "production" test gear, meaning they were not reservable and stayed functional during adjacent testing. Second, for developers who wanted to test pairs of gear, we used the concept of an "ensemble" to create atomic units of the desired gear. This prevented a single test device from being reserved, which depended on a test device in someone else's reservation.

8 Related Work

Network Testing Frameworks Testing new networking research ideas is complex and requires consideration of various circumstances such as network size, hardware specifications, and protocols used. Due to the high cost and proprietary nature of network testing frameworks built by owners of large networks, publicly available network testing frameworks based on emulators are becoming more popular. Emulators such as Flexplane [25], NIST Net [4], Mininet [20], and NS3 [28] are widely used tools to emulate large scale networks and test new networking concepts.

In addition to these publicly available tools, there are numerous firms with the business model of providing network testing tools [13]. Some notable firms include, Ixia (Keysight) [34], Forward Networks [11] and Fluke Networks [8]. However, while these tools can be used to detect anomalies in the current network, they are not built with the purpose of testing newly added features, thus they do not naturally support incremental testing of new features.

Network Testing/Verification Methodology. First of all, there are numerous works on verifying correctness of the network [16, 18, 19, 26, 30]. For example, Reitblatt et al. [27] discusses formal method of verifying the correctness network transitions. Similarly, Veriflow [18] is built to verify invariants of the network in real-time. CrystalNet [22] builds an emulation platform for production. However, these methods only check for issues in the software states, making it hard to detect and report any hardware related issues.

Some articles discuss industry methods for network testing, such as Spirent's best practices outlined in [6]. These practices include Vendor Performance Testing, Network Failure Threshold Testing, Configuration Defect Testing, and PASS Methodology Testing. While these practices offer practical guidance, they do not provide a formal theoretical approach for testing networks, which can make it difficult to quantify testing difficulties.

9 Conclusion

Large network operators have real, but hard to quantify, concerns with managing complexity in their networks. In this paper, we first quantify network testing complexity and share the production challenges. We then present the architecture from the Netcastle "test infrastructure as a service" model as well as lessons learned from over three years of growth. Looking forward, we hope that the research community will leverage our "testing matrix as complexity" model (§2.1) and use it in their research to understand which changes operators are more or less likely to deploy in production. We hope that researchers consider these effects when considering new schemes/research as their choices have real impact on network complexity: the people testing it will have to try to tame that complexity.

This work does not raise any ethical issues.

References

[1] Facebook datacenter network. https://engineering.fb.com/2019/03/14/data-center-engineering/f16-minipack/.

[2] K. Beck. *Test Driven Development. By Example (Addison-Wesley Signature)*. Addison-Wesley Longman, Amsterdam, 2002.

[3] R. Bush. Into the future with the internet vendor task force a very curmudgeonly view or testing spaghetti: A wall's point of view. *SIGCOMM Comput. Commun. Rev.*, 35(5):67–68, Oct. 2005.

[4] M. Carson and D. Santay. Nist net: A linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33(3):111–126, July 2003.

[5] S. Choi, B. Burkov, A. Eckert, T. Fang, S. Kazemkhani, R. Sherwood, Y. Zhang, and H. Zeng. Fboss: Building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 342–356, New York, NY, USA, 2018. Association for Computing Machinery.

[6] S. Communications. Testing the data center network: Best practices. https://www.infopoint-security.de/medien/testing_the_data_center_network-best_practices_whitepaper.pdf, November 2013.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[8] F. Corporation. Fluke networks. https://www.flukenetworks.com/, Jan. 2021.

[9] G. Eason. Incident response @ FB, facebook's SEV process, July 2016.

[10] T. Fang. Introducing openbmc: an open software framework for next-generation system management. https://code.facebook.com/posts/1601610310055392, Mar. 2015.

[11] I. Forward Networks. Forward networks. https://forwardnetworks.com/, 2021.

[12] T. Griffin. RFC4264 BGP wedgies. https://tools.ietf.org/html/rfc4264, Nov. 2005.

[13] S. T. Help. Top 30 network testing tools (network performance diagnostic tools). https://www.softwaretestinghelp.com/network-testing-tools/, Jan. 2021.

[14] HUBER+SUHNER. Polatis series 7000 optical switch. https://www.polatis.com/series-7000-384x384-port-software-controlled-optical-circuit-switch-sdn-enabled.asp, Jan. 2021.

[15] T. Inc. Telescent physical layer switching. https://www.telescent.com/, 2021.

[16] K. Jayaraman, N. Bjørner, J. Padhye, A. Agrawal, A. Bhargava, P.-A. C. Bissonnette, S. Foster, A. Helwer, M. Kasten, I. Lee, A. Namdhari, H. Niaz, A. Parkhi, H. Pinnamraju, A. Power, N. M. Raje, and P. Sharma. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 200–213, New York, NY, USA, 2019. Association for Computing Machinery.

[17] M. Jimenez and H. Kwok. Building express backbone: Facebook's new long-haul network. https://engineering.fb.com/2017/05/01/data-center-engineering/building-express-backbone-facebook-s-new-long-haul-network/, May 2017.

[18] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27, Lombard, IL, Apr. 2013. USENIX Association.

[19] D. Kim, J. Nelson, D. R. K. Ports, V. Sekar, and S. Seshan. Redplane: Enabling fault-tolerant stateful in-switch applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 223–244, New York, NY, USA, 2021. Association for Computing Machinery.

[20] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, New York, NY, USA, 2010. Association for Computing Machinery.

[21] W. E. Lewis. *Software Testing and Continuous Quality Improvement, Third Edition*. Auerbach Publications, USA, 2nd edition, 2008.

[22] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 599–613, New York, NY, USA, 2017. Association for Computing Machinery.

[23] Z. M. Mao, R. Govindan, G. Varghese, and R. H. Katz. Route flap damping exacerbates internet routing convergence. *SIGCOMM Comput. Commun. Rev.*, 32(4):221–233, Aug. 2002.

[24] J. Networks. How to calculate power margin for fiber-optic cable. https://www.juniper.net/documentation/en_US/release-independent/junos/topics/task/installation/fiber-optic-cable-budget-margin-calculating.html, July 2020.

[25] A. Ousterhout, J. Perry, H. Balakrishnan, and P. Lapukhov. Flexplane: An experimentation platform for resource management in datacenters. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 438–451, Boston, MA, Mar. 2017. USENIX Association.

[26] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Feb. 2020.

[27] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. *ACM SIGCOMM Computer Communication Review*, 42(4):323–334, 2012.

[28] G. F. Riley and T. R. Henderson. The ns-3 network simulator. In *Modeling and tools for network simulation*, pages 15–34. Springer, 2010.

[29] F. O. Source. Fboss open source. https://github.com/facebook/fboss, 2021.

[30] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu. Debugging p4 programs with vera. SIGCOMM '18, page 518–532, New York, NY, USA, 2018. Association for Computing Machinery.

[31] J. Stone and C. Partridge. When the crc and tcp checksum disagree. *SIGCOMM Comput. Commun. Rev.*, 30(4):309–319, Aug. 2000.

[32] J. Taylor. Facebook's data center infrastructure: Open compute, disaggregated rack, and beyond. In *Optical Fiber Communication Conference*, page W1D.5. Optical Society of America, 2015.

[33] C. Technologies. Calient s series photonic switch. https://www.calient.net/products/s-series-photonic-switch/, 2021.

[34] K. Technologies. Connect and secure your network with keysight. https://www.keysight.com/us/en/cmp/2020/network-visibility-network-test.html, Jan. 2021.

[35] Wikipedia. Multiple source agreement optics. https://en.wikipedia.org/wiki/Optical_module, July 2017.

[36] J. Willet. Evolution of the testing pyramid. https://www.james-willett.com/the-evolution-of-the-testing-pyramid/, Sept. 2016.

A Appendix: Network Testing Is Hard In Theory

In this section, we try to quantify why network testing is hard. Specifically, we define a formal metric for system complexity based on the number of states and allowed state transitions. We then argue that in non-trivial systems, the only way to enumerate the actually allowed state transitions (including bugs, surprising interactions, the halting problem, etc.) is through testing. A testing matrix is a common software engineering term for the multi-dimensional space of all possible interactions in a system. We then introduce the concept of a theoretically complete testing matrix: that is, a conservative/worst case set of conditions/states that if one were to theoretically test all elements of the matrix, the system could be proven by exhaustion to be bug free. We then combine these two definitions to show that network testing is the combinatorial combination of software and hardware systems testing and that by our metrics is uncountably infinite in all but the most trivial of cases.

A.1 Quantifying System Complexity

From our literature review, there does not appear to be a commonly agreed upon method to quantify the complexity of a general system, so we define our own. From computational complexity theory, formally the complexity of an algorithm is the minimum number of bits needed to represent the algorithm and its inputs [7]. Inspired by this definition, we quantify the complexity of a system as:

Definition A.1 ***Measure of System Complexity****: the minimum number bits required to describe all possible states and the allowed state transitions of a system.*

Consider a system with n different states where the allowable state transitions can be (possibly inefficiently) represented as an $n \times n$ matrix of bits denoting "state transition $i \rightarrow j$ is allowed". By this definition, naively this system would require $\log(n)$ bits to describe the possible states plus n^2 bits to describe the state transitions or $\log(n)+n^2$ bits to capture this system's complexity. However, if this system's allowed state transition matrix was sparse, summarizable with few rules or otherwise compressible, then the minimum number of bits required to represent it could be much smaller than n^2. From this definition, and matching our intuition, a system gets more complex as it adds more states or as the allowed state transitions become harder to describe.

This definition captures a critical combinatorial property of system complexity: when two or more smaller systems ("components") are coupled into a larger system, the complexity

increases multiplicatively. That is, if one couples two systems that have m and n states respectively, then the resulting system C has $n \times m$ different possible states and thus $(n+m)^2$ possible state transitions. As above, the state transition matrix of C may still be compressible and thus the minimum bits needed to describe them may be smaller than $(n+m)^2$ in practice. That said, as the number of coupled components increase, the ability to succinctly summarize the allowed state transitions of the larger becomes harder, e.g., it requires a combinatorial multiplication of rules.

Note crucially that "allowed state transitions" are the ones allowed by the actual deployed running implementation and not limited to the transitions intended by the original design intent. A network architect may believe they have designed a simple system with few possible states and limited state transition rules ("no persistent loops", "control plane processes never reach the segmentation fault state", etc.). However, the implementation of the actual system including bugs, hardware failure, environmental effects, unforeseen vendor incompatibilities, etc. may in fact be much more complex. Thus, for any large, practical system, there may be unintended and surprising state transitions: i.e., the allowed state transitions an emergent property of the coupling of many systems. More so, the "halt" state is important to any system, so enumerating the states that can transition to 'halt' is directly equivalent to the well-known and undecidable halting problem. As a result, we claim that the only way to even approximately enumerate possible state transitions, and thus quantify the system's complexity, is through providing specific inputs and testing which state transitions are and are not actually allowed.

A.2 Theoretically Complete Testing Matrix

We define a *theoretically complete testing matrix* as a test matrix where if all possible tests were evaluated, then the components/system under test would be proven to be correct by exhaustion. As we will show, this is often not a practical tool as many of the dimensions of the theoretically complete testing matrix will be infinite in practice, we believe it is still a useful vehicle for quantifying network complexity.

There are two properties of a theoretically complete test matrix:

1. It makes no simplifying assumptions about the implementation of the components under test. As a result:

2. Similar to Definition A.1, each dimension or component added to the system increases the size of the testing matrix multiplicatively.

Quantifying Software Systems Testing First, let's consider constructing the simplest possible theoretically complete test matrix. Consider a toy, single threaded user-space process (the "process under test") that has n different possible states and the only information it stores is the value of the current state. The test matrix for such a software system is $O(n^2)$, as the programmer needs to test all possible state transitions to make sure each works as expected. Note that it is important to test not only the valid state transitions (functional tests) but also test the invalid transitions (negative testing) to verify that they are correctly prevented. The current state and the next state form the first two dimensions of our testing matrix and each cell in the matrix is a single bit representing the pass/fail result of running the test.

Next, let us relax our toy application assumption by assuming that it keeps non-trivial information across state transitions (e.g., allocates memory, opens a file, etc.) such that we now need to consider the history of all state transitions - another dimension to our matrix. This new dimension is potentially infinite in size (all permutations of all possible paths through the state diagram) and practically infinite for long running programs. While seemingly abstract, the history of all possible state transitions is where we catch the most common types of bugs: memory corruption, resource leaks, privilege elevation, etc. Then, let us further relax our single-threaded assumption by allowing the process to use threads, asynchronous I/O, or other form of scheduling non-determinism. This adds scheduling order as another dimension to our test matrix which is exponential in the number of threads for each state change. Now relax the single process assumption and allow the process under test to have non-trivial communication with both the kernel and other processes (local or remote). Thus, the theoretically complete testing matrix for a distributed system of software processes is the multiplicative product of each software component in the system. In other words, if one wants to be absolutely certain to prevent bugs in a non-trivial multi-threaded distributed system, they need to write sufficient tests that cover the multiplicative product of: all possible state transitions $\times$ all possible state transitions histories $\times$ all possible schedule orderings $\times$ all possible states of the other components in the system.

Quantifying Hardware Systems Testing Hardware systems have a superset of possible bugs as compared to software systems and thus their testing matrices are correspondingly larger. Consider the logical design of a single integrated circuit or component. Often before construction, such components are modeled in a software simulator, and similar to software testing, are tested across all possible inputs/outputs and state transitions. As a result, all of the above analysis for software test matrices applies to hardware testing matrices as well. Additionally, hardware systems must test for manufacturing repeatability (e.g., if a component requires a physical size tolerance of $\pm X\%$, what is the distribution in practice) and manufacturing process changes (e.g., different vendors of solder, electronics components). Hardware systems must test under different operating conditions (e.g., temperature, humidity, vibrations, electromagnetic interference) and typically ship with specified operating tolerances (e.g., run between

50-85 degrees *F*). All of these aspects add new dimensions to the theoretically complete testing matrix.

A.3 Testing Network Systems

Computer networks are, by definition, transitively coupled hardware and software systems. That is, state from any one node could in theory be transferred to any/every other node in the system. As a result, the complexity (Definition A.1) and theoretically complete testing matrix of a network is the combinatorial explosion of all possible testing matrices of all connected hardware and software systems. More than just a theoretical concept, large network outages happen in practice because of rapid distribution of state from a single failed component. A typical example is a single faulty electrical component (e.g., a capacitor) in one optical interface on a single switch can cause a shared link to flap up and down, which various control plane software systems (e.g., spanning tree protocol in L2, BGP in L3) then propagates the new forwarding state to other nodes in the system, which each try to adapt to the new state based on local policy. The result of these transitive interactions, despite decades of experience and armies of network engineering experts, are often surprising and detrimental [12, 23].

Unique to networking, practical deployments are often the worst case for many software and hardware testing scenarios. For example, device up-time is typically long (over a year in some routers) so the set of all possible paths through the state machine is in practice large. Networking gear is expected to operate in a variety of environments on the edges of their operating tolerances from (in theory) climate controlled data centers to road-side optical devices to the WiFi access point under your desk that collects dust bunnies. Despite formal protocol standards which should in theory ensure interoperability, in practice all-to-all interoperability tests are still required between different implementations and many deployments effectively require "bug-level" compatibility. Last, for reasons of compatibility as well as practical software engineering and business realities [3], networking devices tend to monotonically accumulate features over time (e.g., Apple talk support, RIP, etc.). All of these elements add to the complexity of network systems and thus increase the complexity of testing these systems.

MESSI: Behavioral Testing of BGP Implementations

Rathin Singha[1] Rajdeep Mondal[1]
Ryan Beckett[2] Siva Kesava Reddy Kakarla[3] Todd Millstein[1] George Varghese[1]

[1]*University of California, Los Angeles* [2]*Microsoft* [3]*Microsoft Research*

Abstract

Complex network protocols like the Border Gateway Protocol (BGP) are prone to implementation errors that cause unintended behaviors with potentially global consequences. We introduce an approach and tool called MESSI (Modular Exploration of State and Structure Inclusively) to automatically generate tests for black-box BGP implementations. Our approach is *model-based*, leveraging an executable model of BGP to generate behavioral tests. However, doing so effectively requires addressing new challenges such as the stateful nature of BGP and the need to generate complex structures like regular expressions in route maps. We used MESSI to generate roughly 150K tests that capture different aspects of BGP, such as route-map filtering, the decision process, route aggregation, and dynamics. These tests identified 22 correctness bugs across several widely used open-source BGP implementations (FRR, Quagga, GoBGP, BIRD, Batfish) and one closed-source implementation. Eight of these errors have already been fixed. While our models are BGP-specific our approach is not: thus we expect it can be adapted to test other stateful protocols with complex structures.

1 Introduction

The Border Gateway Protocol (BGP) is the backbone of modern internet routing, as it connects the various autonomous systems to one another [47]. Due to its ability to enforce expressive policies, BGP is also widely used by organizations, for example, to establish reliable communication in data centers and to shape policies within enterprises. Due to its ubiquity and large blast radius, BGP errors can have large-scale effects across the globe. Unfortunately, BGP-related outages are quite common [7, 18, 31, 36, 41, 43, 50]. While there has been significant research on identifying BGP *configuration* errors (*e.g.*, [5, 24, 26, 34, 48]), there has been less work on automatically identifying BGP protocol *implementation* errors, which occur frequently in the various widely used BGP implementations and are the focus of this work [8, 9, 19]. According to a study conducted in [39], 36% of the significant and customer-impacting incidents in Microsoft's network are caused by software implementation bugs.

Our goal is to automatically generate tests for BGP implementations that cover a wide range of BGP behaviors. Doing so is challenging because each test case consists of a triplet of an incoming route announcement, a configuration that indicates how incoming announcements should be treated, and the current BGP state of the router. Traditional test-generation approaches are not sufficient: for example, fuzzing is unlikely to find combinations of these inputs that explore a wide variety of interesting behaviors. Instead, we are inspired by recent work that employs *model-based testing* to find errors in black-box protocol implementations [33]. The basic idea of that approach, called SCALE, is to build an executable model of a protocol and then symbolically execute the model in order to generate tests. The model encodes the intended behavior of the protocol, as specified in RFCs, and the use of symbolic execution ensures that the generated tests cover a wide variety of scenarios represented by different execution paths through the model. The resulting tests can then be executed on multiple black-box protocol implementations, with any differences among them identified as potential errors.

While promising, the SCALE approach targeted DNS nameserver implementations. The BGP setting introduces several new challenges for test generation. First, BGP is *stateful*: the handling of a BGP announcement depends on earlier announcements; by contrast, DNS nameservers are stateless. Handling state dramatically increases the search space for test generation and requires correlating state with other protocol inputs to identify interesting behaviors. Second, DNS configurations consist of a flat set of *records* in a zone file. By contrast, BGP configurations are like little programs, with complex and often hierarchically structured policies defined via *route maps*. Notably, BGP policies depend heavily on regular expressions: but it is infeasible to symbolically generate regular expressions both theoretically (the theory of strings with symbolic regular expressions describes non-regular languages) and in practice. Third, DNS nameserver implementations all accept a standard format for configurations; in contrast, the various BGP implementations like Cisco, Juniper, Quagga, BIRD, and GoBGP have distinct configuration languages that differ in both obvious and subtle ways, making it difficult to perform

differential testing across them.

We present an approach and corresponding tool called MESSI[1] (Modular Exploration of State and Structure Inclusively) to overcome these challenges. First, we handle the complexity of BGP by decomposing it into several modules and generating tests for each module individually. The benefit of this decomposition is that testing each module only requires us to model the portion of BGP inputs, configurations, and states that are relevant to that module, which makes a model-based approach tractable. For example, generating tests for route-map filtering requires us to model the way that a route map processes an input route, but it does not require us to model the state. In contrast, generating tests for the BGP decision process requires us to model router state, but only a specific portion of it — the single best route installed so far to a destination — and does not require us to model complex configurations. In addition to these models, MESSI also includes models for *route aggregation* as well as for *dynamics*, which models router state changes such as updates to a route map and updated route announcements.

Second, we handle test generation in the presence of regular expressions through a hybrid approach. Inspired by enumerative combinatorics [38], we use a form of enumerative testing to generate (not just count) all forms of regular expressions up to a specified size. We then generate positive and negative example strings for each regular expression, doing so in a manner that provides coverage guarantees over the regular expression's finite automaton representation. Finally, our model of route-map filtering treats regular expressions concretely — we supply it with specific regular expressions that we have generated, and the model uses only the given positive and negative example strings when building concrete routes to match against such regular expressions. Finally, to handle the diversity of BGP configuration languages, we have created our own intermediate representation for configurations along with a lightweight shim that translates from this representation to the various vendor-specific languages.

Our tool MESSI generated roughly 150K test cases for the BGP decision process, Route filtering, Aggregation, and Dynamics within two days. From 150K tests, we found around 1500 failed test cases across all implementations. We added tags to each test case based on the path they traverse in the code logic and thus grouped equivalent failed cases into buckets. Although the BGP implementations we tested are well maintained and in use for several years, we still found 22 bugs across different BGP implementations, including FRR [27], Quagga [45], GoBGP [28], and Batfish [3]. 11 of the bugs are acknowledged by the developers and 8 of them are already fixed. Our tool is able to find 18 bugs that were unknown previously, while the other 4 had been reported earlier. These bugs cover a range of BGP features, *e.g.*, route map logic, route aggregation, community lists, etc.

[1] https://github.com/rsingha108/MESSI

The rest of this paper is structured as follows. §2 describes the background and motivation of this work using some noteworthy bugs that were automatically found by our tool. §3 describes our solutions to the new challenges that automated testing of BGP introduces, such as regex undecidability and statefulness. Then in §4.2, we describe the experimental setup and methodology, showing how our solutions were executed to test black-box BGP implementations. § 4.3 presents our results – a list of major bugs, with their descriptions, that we were able to find using our tool. §6 surveys related work, while §7 outlines future work and draws conclusions.

Contributions: This paper's contributions are:

- The first automated approach and tool MESSI to identify RFC violations in black-box BGP implementations.
- Modular exploration to deal with protocol complexity.
- Efficient enumerative testing of regular expression inspired by enumerative combinatorics, which cannot otherwise be handled by symbolic testing.
- A testing framework to catch bugs that are generated due to dynamics of BGP often caused by incorrect implementation attempts to do an incremental computation

2 Background & Motivation

The Border Gateway Protocol (BGP) operates as an essential component of the Internet's control plane that connects different autonomous systems (ASes). The BGP protocol has many different aspects. We focus on four specific aspects — the decision process, route filtering, route aggregation, and dynamics — as these are complex and commonly used, and we observed many correctness bugs due to them in the bug databases for popular BGP implementations.

The *decision process* refers to the determination by a BGP router of the best route to some destination, which involves a comparison of route attributes (Local preferences, AS path length, Multi-Exit Discriminator, etc.). The best route is added to the routing information base (RIB) of the BGP router and advertised to neighbors. *Route filtering* empowers administrators to tailor route advertisements based on desired policies. It is performed by *route maps*, which are effectively functions that permit or deny route advertisements based on the properties of those advertisements. Route maps can also transform accepted routes by updating their attributes. Accepted routes then go through the decision process described above. *Route aggregation* is a feature that enables the representation of routes to multiple contiguous IP prefixes as a single summarized route, thus enhancing routing efficiency. Finally, BGP routers must constantly adapt to changes in the environment, for example, the withdrawal of route advertisements and updates to routing policies. We refer to such changes as BGP *dynamics*.

In this section, we demonstrate through concrete examples the challenges for testing BGP implementations as well as the capabilities of our MESSI approach and tool.

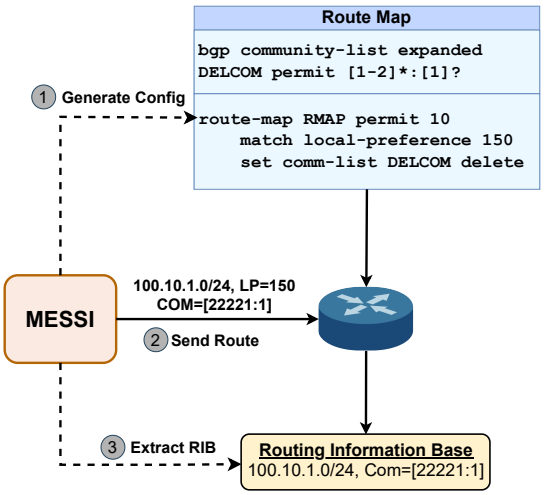


Figure 1: A MESSI test for the BGP route filtering with regular expression.

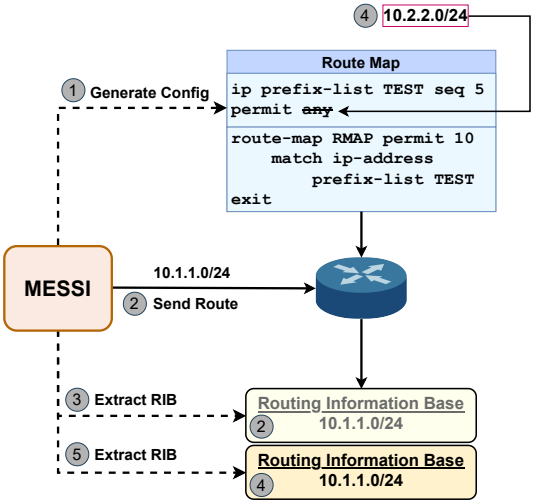


Figure 2: A MESSI test for route filtering with dynamics.

2.1 Route Filtering and Regular Expressions

Figure 1 shows a pair of a route map and route announcement that MESSI automatically generated to test BGP's route filtering behavior. The route map permits routes that have local preference 150 and deletes specific communities from such routes, namely those matching the regular expression in the community list `DELCOM`, which MESSI also generated. The generated route announcement should be permitted by the route map since it has the right local preference, and its community `22221:1` should be deleted since it matches the regular expression `[1-2]*:[1]?`. However, as shown in the figure, when we tested this route map and route announcement on the FRR BGP implementation, we found that the route was accepted, but the community was not deleted.

Upon investigation, we learned that the bug arises only when the matched regular expression has only one ? symbol in it that appears at the end, and the configuration is installed via the command line because this symbol is also overloaded as the "help" command in the configuration console. The result is that the `permit` line in the configuration is completely ignored. Finding this bug is non-trivial: it requires a configuration containing a regex that ends with ?, and there be no other ? symbol in it; it also needs a route announcement that matches the corresponding route-map stanza and matches the regex. When we raised this issue on FRR's GitHub page, they acknowledged it and suggested two workarounds. The first is to use `^+V+?` on the CLI to escape it. The second is to enter a blank space directly after the ?.

2.2 BGP Dynamics

Many BGP implementation bugs lurk in the dynamics of BGP. For example, when a router configuration is changed, the router may not behave as intended after the change is made. This is often caused by (incorrect) implementation attempts to do more efficient incremental computation. We describe an example of this kind that MESSI found automatically.

Dynamics Bug in Route Filtering. Figure 2 shows a pair of a route map and route generated by MESSI. In the FRR BGP implementation, initially, because of the `permit any` line in the generated prefix list, the announcement for `10.1.1.0/24` is permitted. Subsequently, suppose the definition of the prefix list is changed to only permit the prefix `10.2.2.0/24`. Therefore, when a route announcement to `10.1.1.0/24` is sent, it should now be rejected, but we found it was (erroneously) still accepted. The bug was caused by an `ANY` flag that the implementation maintained that was not properly reset upon configuration updates because of an incorrect (but efficient!) incremental computation. The bug was acknowledged and fixed by the developers.

2.3 Route Map Semantics

Figure 3 shows a test case generated by MESSI where it simultaneously generated a route map and a route. The route map refers to a community list with two communities `0.11` and `0:222` and permits any route that has one of those communities. MESSI generated a route with community `0:222` for this test, which should match with community-list `COMM`, and the route should be accepted by the router.

When we ran this test case on the FRR router, FRR did not accept the route. Further investigation showed that the FRR documentation stated the semantics of the community list to be "OR" – i.e., if the incoming route mentions *any* of the communities in the community list, then it should match the community list. However, the implementation displayed "AND" semantics: it only matched when the route contains *all* the communities in the community list. This was evident when we tested the same route map with an incoming route that has community `[0:11 0:222]`. When we posted this issue on the GitHub page of FRR, the developers confirmed it was a documentation bug and fixed the issue.

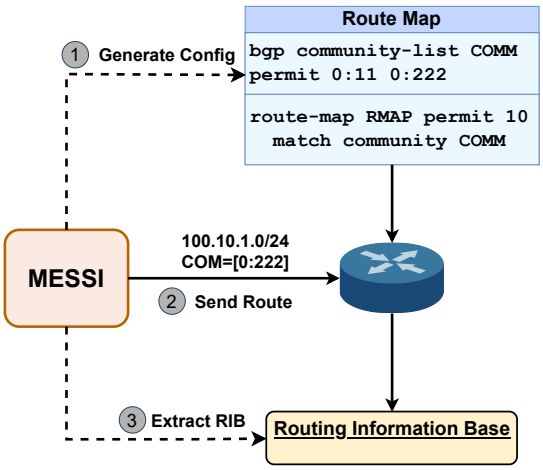


Figure 3: A MESSI test showing buggy behavior of Community list in Route Maps

3 Model Based Testing for BGP

This section describes our approaches to addressing the challenges of test generation for BGP in MESSI. First, we overview our approach at a high level. We then explain in detail how MESSI generates tests for each of the four key aspects of BGP on which we focus.

3.1 Overview

The MESSI approach builds on prior work on model-based testing for network protocol implementations [33]. Specifically, we built an executable model of (parts of) the BGP protocol based on the relevant RFCs. We generally adopted the behaviors that are most commonly used across all the implementations based on our interpretation of the documentation. To generate tests, we *symbolically execute* the model [35], which involves picking execution paths through the model, determining the constraints under which each such path will be taken, and then solving the constraints to generate a concrete test that traverses the path. If any unexpected behavior is observed during testing, we check whether it is a genuine bug or the software implementer's choice. It is a cyclic and iterative process, where we manually validate the model based on our reading of the RFCs but also through our testing setup – unexpected behaviors can indicate bugs in our model, which is then refined.

The model is built in C# using a constraint-solving library called Zen [4, 6]. Zen allows the model to be written as regular C# code but with some inputs declared as *symbolic*. It then performs symbolic execution to solve for these inputs automatically, leveraging Satisfiability Modulo Theories (SMT) solvers like Z3 [21] to solve the constraints. A model would typically have an unbounded number of paths, but Zen offers various ways to limit the set of desired paths, for example, those whose size is bounded by a depth.

Prior work that used this approach built a single monolithic model of the network protocol, which in that case was a DNS nameserver [33]. However, BGP is significantly more complex than DNS nameservers. For example, BGP is stateful, whereas DNS nameservers are stateless, and BGP configurations are much closer to programs — containing forms of conditionals, function calls, and nested structure, for example — than to DNS configurations, which are a sequence of flat records. Symbolic execution on a monolithic model of BGP is infeasible due to *path explosion*: the number of paths grows exponentially in the model size. Thus a monolithic model will have a prohibitively large number of paths, allowing only a small subset of protocol behaviors to be tested in practice.

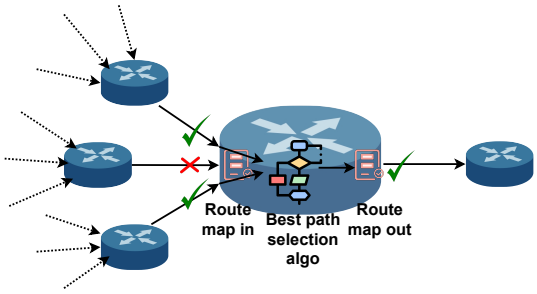


Figure 4: An overview of BGP route selection.

We address this challenge via modular exploration. Instead of a single monolithic model of BGP, we built separate models for key aspects of BGP. Figure 4 shows the BGP route selection process. When a route is received, a route map defined in the BGP configuration is used to determine whether the route is permitted or denied (as shown on the left of the main router); this is called *route filtering*. Subsequently, the best route to a destination is selected and installed in the RIB; this is called the *decision process*. Finally, the selected route is sent to neighbors after passing through a route filtering step.

Leveraging this structure, we built separate models for route filtering and for the decision process. In general, if there are m possible execution paths for route filtering and n paths for the decision process, then there can be $m \cdot n$ paths for a model that combines them, whereas in our approach, there are only $m + n$ paths to explore. Modular exploration also enables us to tailor each model's symbolic inputs to only the parts of BGP that are relevant to that feature, which further simplifies each model and makes test generation feasible. For example, route filtering involves a BGP route map and an input route to be processed, but it is independent of the state of the router, which, therefore, need not be modeled. On the other hand, the BGP decision process depends on the current router state but is independent of the route maps in the BGP configuration.

Figure 5 overviews the architecture of MESSI. The symbolic module contains separate executable models for the BGP decision process, route filtering, route aggregation, and forms of protocol dynamics, which we discuss in turn in the following subsections. As shown in the figure (and discussed further below), each such model has a different set of inputs and outputs tailored to that feature's role in the overall BGP process. In each case, we use Zen to generate concrete test cases (indicated

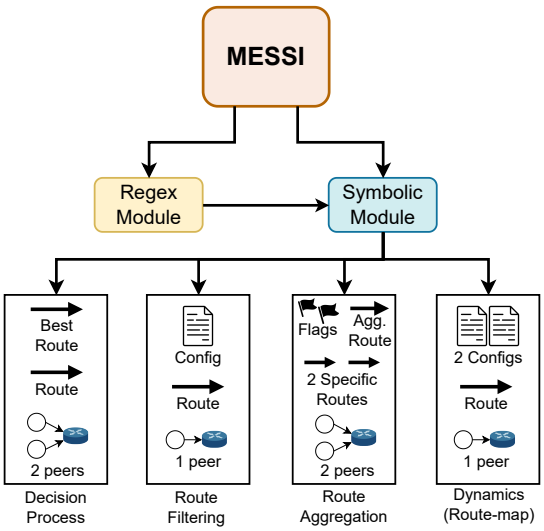


Figure 5: The architecture of MESSI

in the figure as the Symbolic Module), which we then execute on black-box BGP implementations. A particular challenge for test generation of route filtering is the need to generate regular expressions, as these are commonly used in route maps to match on communities and AS paths in routes. We have developed a special technique to handle regular expressions (shown as the Regex Module) that we describe in detail below.

3.2 Decision Process

As shown in Figure 5, we model the decision process as a function that takes two routes to a single destination and returns the route that is more preferred. We argue that these inputs are sufficient to identify any error in the BGP decision process. The decision process is used whenever a router receives an incoming route announcement and decides to permit it (possibly with some modifications). At that point, the newly permitted route is compared against the current best route to that destination in the RIB to determine whether the RIB should be updated. Our model captures exactly this scenario, where one of the two input routes models the current best route in the RIB and the other models the new route.

The BGP decision process involves comparing the various attributes of the two given routes in a particular order of precedence, in order to decide which route is more preferred. For example, first, the attribute called "local preference" of the two routes is checked, and the route with the higher local preference is preferred. If the local preference values are equal, then the BGP decision process moves on to check the values of other attributes, and so on.

Our Zen model is, therefore, simply a function that takes two routes and compares their attribute values in the correct precedence order to determine which route is preferred. Marking the two routes as *symbolic* ensures that Zen will explore all paths through this function and, for each path, will generate two concrete input routes that cause execution

```
1  Zen<Route> Compare(Zen<Route> r1, Zen<Route> r2){
2      // compare attributes
3      var gtLp = r1.LocalPref() > r2.LocalPref();
4      var neLp = r1.LocalPref() != r2.LocalPref();
5      var ltLen = r1.PathLen() < r2.PathLen();
6      var neLen = r1.PathLen() != r2.PathLen();
7      ...
8      var ltRid = r1.RouterId() < r2.RouterId();
9      var neRid = r1.RouterId() != r2.RouterId();
10
11     // compare the routes
12     var ridCmp = If(ltRid, r1, If(neRid, r2, r1));
13     ...
14     var lenCmp = If(ltLen, r1, If(neLen, r2, ...));
15     var lpCmp = If(gtLp, r1, If(neRid, r2, lenCmp));
16     return lpCmp;
17 }
```

Figure 6: Pseudo-code for BGP decision process.

to follow that path. For example, Zen will generate a pair of input routes for the case when the first route has a higher local preference than the second one, another pair for the case when the second route has a higher local preference than the first one, and many pairs where the local preferences of the two routes are equal so that other paths will be explored.

In Figure 6, we show the pseudo-code for the BGP decision process. Note that we assume that the `always-compare-med` and `compare-routerid` flags are enabled. Turning the first flag on allows us to have a complete ordering between routes. However, we can easily modify our model to accommodate other comparison modes for MED. In Lines 3 to 9, we first compare the routes' attributes according to the decision process preference. Next, we order these decisions Lines 12 to 15 by comparing the attributes in the correct order. We build the hierarchy of attribute comparisons in a bottom-up fashion using nested if-then-else expressions. If the local preference of the first route is higher, then we return the first route. Otherwise, if the second is preferred, we return it instead. If they are equal, then we move on to the AS path length. In this way, we mimic the BGP decision process by sequentially comparing the attributes of the two routes in order of preference.

3.3 Route Filtering

To test route filtering, we employ a *hybrid* approach, which leverages both an executable model for symbolic execution and a special approach to support generation of regular expressions. We describe each in turn.

3.3.1 Zen Model

As shown in Figure 5, we model the route filtering process in Zen as a function that takes two inputs: a configuration, specifically a route map and associated definitions like prefix lists and community lists; and an incoming route. The route filtering process applies the route map to the route to determine if the route is permitted or denied; in the case that it

is permitted, the route is also modified as specified in the route map. Notably, route filtering is *stateless*: the processing of an incoming route by a route map is independent of the current router state, such as the contents of the RIB. Hence there is no need to model that state. The previous section showed examples of pairs of route maps and routes that MESSI automatically generated from its route filtering model.

In general, a route map consists of one or more *stanzas*. Each stanza consists of an action (permit or deny), zero or more *match* statements, which indicate the conditions under which the stanza matches the input route, and zero or more *set* statements, which update the incoming route if it is permitted. Stanzas are processed sequentially until finding a stanza that matches the input route, in which case the corresponding action and set statements are applied. Route maps can also contain more sophisticated control flow; for instance, stanzas can be chained together, with one falling through to the next, and route maps can invoke other route maps.

Handling the full complexity of route maps described above is simply infeasible. Therefore, we take our cue from prior work on model-based testing for protocols, which found that small configurations were sufficient to surface many interesting implementation errors [33]. Our Zen model considers a route map to consist of exactly one stanza. While this is a significant simplification, it nonetheless creates a rich space of policies to explore. Within a stanza, we allow any number of match statements, and they can match on all of the different attributes of a route. We similarly allow any number of set statements, which can update those attributes in various ways, for example, adding and deleting communities. Finally, as shown in the earlier examples, our model also includes auxiliary structures like prefix lists and community lists, which can be generated by Zen and referred to in route maps.

3.3.2 Generating Regular Expressions

One significant source of complexity in route maps that we *cannot* avoid is the use of regular expressions to match on communities and AS paths. Their usage is ubiquitous, and regular expressions are also a common source of errors; thus, any test generation approach for route maps must support them. A naive approach would be for our Zen model to simply include the logic for checking whether a string satisfies a regular expression and ask Zen to symbolically execute this code. However, doing so would dramatically explode the number of paths through our model (since, for example, it would include the logic to convert a regex to a DFA). Further, solvers [21] cannot handle the theory of strings with *symbolic* regular expressions since they can describe non-regular languages [46].

Instead, we developed a hybrid approach to generating route maps containing regular expressions that avoids the need to symbolically execute the regex matching process while also obtaining useful coverage guarantees for both regexes and the strings that are matched against them. First, we exhaustively enumerate *regex structures* up to some size (a form of coverage), which are regexes that are parameterized by the base regexes within them. Next, we employ random generation to produce the base regexes for each regex structure, yielding concrete regexes. Third, for each such regex we generate a set of positive and negative example strings that together cover all nodes and edges in the DFA representation of the regex. Finally, these regexes and their associated examples are given as concrete inputs to our Zen model to use when building route maps and routes. Next, we describe these steps in more detail.

Algorithm 1 Enumerating regexes at a given level

```
1:  function RegexEnumerator(R_{n-1})
2:      R_n ← []
3:      for i in range(len(R_{n-1})) do
4:          r ← R_{n-1}[i]
5:          R_n ← R_n + [r, (r)*, (r)+, (r)?]
6:          for s in (R_{n-1}[i:]) do
7:              R_n ← R_n + [(r)|(s)]          ▷ Alternation
8:              R_n ← R_n + [(r)(s)]           ▷ Concatenation
9:          end for
10:     end for
11:     return R_n
12: end function
```

Regex enumeration algorithm. We are inspired by work in enumerative combinatorics to count [38] (not generate) regular expressions. Similar ideas can apply to other complex protocol structures for which a recursive formulation is feasible. We use the algorithm shown in Algorithm 1 to enumerate regex structures exhaustively up to some level n, given the regex structures up to level $n-1$.

We use the character 'x', which represents a regex to be filled in later, as the sole regex structure at level 0. According to the algorithm, level 1 will consist of the regex structures {`x`, `(x)*`, `(x)+`, `(x)?`}. Similarly at level 2 we will have regex structures like `(x*)|(x+)`, `(x+)?`, and so on. The number of regex structures at every level increases exponentially, so we stop at level 3 with 4686 such structures.

At the nth level the complexity of Regex Generation is $O(R_{n-1}^2)$ where R_{n-1} is number of regexes generated at the $(n-1)^{th}$ level. If we solve the recursion $T(n) = T(n-1) * T(n-1)$ then we get $T(n) = O(2^{2^n})$

Random regex generation. For each of these structures, we randomly generate valid community regexes to fill in their placeholder values 'x', via the following grammar:

T	::=	$(\mathtt{E}_1 \mid \ldots \mid \mathtt{E}_k)$	*(top-level-regex)*
E	::=	`^S$`	*(end-to-end-regex)*
S	::=	$(\mathtt{C}_1 \ldots \mathtt{C}_k)$	*(sequence-regex)*
C	::=	`P:P`	*(community-regex)*
P	::=	$(\mathtt{B}_1 \ldots \mathtt{B}_k)$	*(community-segment-regex)*
B	::=	`R` | `R*` | `R+` | `R?`	*(base-regex)*
R	::=	`[N-N]`	*(regex-number-range)*
N	::=	`0` | `1` | ... | `9`	*(number)*

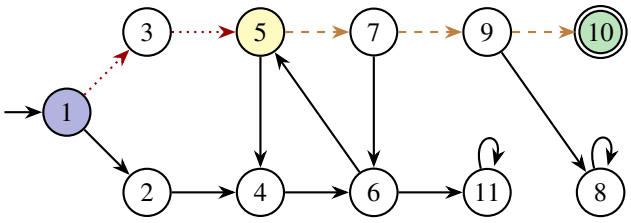


Figure 7: Node Coverage on the DFA: To generate a positive example that covers state 5 in the above figure, we first find a path P1 (dotted maroon) from the starting state (1) to state (5). In this case, it is (1) → (3) → (5). Then we find another path P2 (dashed brown) from state (5) to the accepting state (10).

An example random regex generated from this grammar is:

```
(^[2-4]+[3-5]*:[2] [3]:[1-2]?$)|(^[5-6]+:[4]*$)
```

After plugging these randomly generated strings into the regex structures generated by the enumeration algorithm, we obtain a set of concrete regexes.

String generation For each regex that the Zen model uses in a community list, we require both positive and negative examples to use as concrete community values in routes, and we would like to choose these examples in a way that covers many interesting behaviors of the regex. We have devised a simple and effective approach to doing this, through efficient graph algorithms on the DFA representation of each regular expression. Specifically, we generate positive examples that together cover all nodes as well as all edges of the DFA that are reachable, and similarly for the negative examples.

Figure 7 illustrates the approach on an example DFA. Suppose we wish to identify a positive example that includes node (5). First, we use depth-first search (DFS) to find a path from the start node (1) to node (5) (shown in the figure as the dotted path). Then we use DFS again to find a path from node (5) to the accepting node (10) (the dashed path). Each DFA edge is labeled with the input character that causes that edge to be traversed, as usual (not shown in the figure). Hence, we traverse the concatenation of these paths to collect these labels, yielding a positive example. We iterate this process over all nodes, keeping track at each iteration which nodes have already been covered. We use an analogous approach to cover each edge, using DFS to identify a path from the start node to the edge's source and a path from the edge's target to the accepting node. Finally, to generate negative examples, we use the same procedure but on the DFA that represents the complement of the original regex.

3.3.3 Regex Testing

Using the algorithms discussed above, we obtain a total of 253,958 test cases, each consisting of a regular expression and a string that should be either a positive or negative example. We use these test cases in two ways. First, we generate a route-filtering test case for each one to test the regex-matching logic of the BGP implementations. For example, given the regex `^[2-4]+[3-5]*:[2]+$` and positive example `2335:22`, we generate the following route-map, which accepts routes whose community matches our regex, as well as an input route whose community value is `[2335:22]`:

```
ip community-list 101 permit ^[2-4]+[3-5]*:[2]+$
route map FILTER_ROUTES permit 10
       match community 101
```

Second, we randomly choose a small subset of the generated regexes (4 in our experiments) and "hardcode" these into our Zen model. For each such regex, we also randomly choose a small set (3 in our experiments) of associated positive and negative examples to use in community lists. When symbolic execution explores a path that successfully matches the input route against a community list, Zen selects one of the predetermined regexes to use in the generated route map and one of its positive examples to use in the generated input route. This allows us to generate complex tests involving regexes (e.g. tests with multiple match and set statements), without modeling and symbolically reasoning about regex matching logic. The Zen model also uses this pool of regexes and examples to generate tests for community deletion.

We have described our regex generation process specifically for community regexes, but we use a similar approach to generate regexes over AS paths and the corresponding positive and negative example AS path values. Note that each component of our hybrid approach (symbolic, random, enumerative) has coverage guarantees, although of different forms; one can increase test coverage by increasing resources.

3.4 Route Aggregation

As shown in Figure 5, we model the route aggregation process in Zen as a function that takes two routes, the aggregate prefix, as well as some aggregation-specific flags, and produces (possibly multiple) output routes. Our Zen aggregation model first checks whether each of the two incoming routes matches the aggregation prefix. Based on this information and the values of various aggregation-specific flags, it then determines the set of routes to return. Aside from the `matching-med-only` flag, another example flag is `summary-only`, which, if set, indicates that only the aggregate route should be advertised. Conversely, if this flag is not set, then both the individual routes and aggregate route should be advertised.

3.5 Testing BGP Dynamics

BGP routers must constantly respond to changes in the environment — a link can go down, a route can be withdrawn, a route policy can be updated, etc. Efficiently handling these kinds of changes is challenging for developers. We have observed that erroneous incremental computation is at the heart of many identified implementation bugs (e.g., [1, 11, 12, 23]). We have developed an approach in

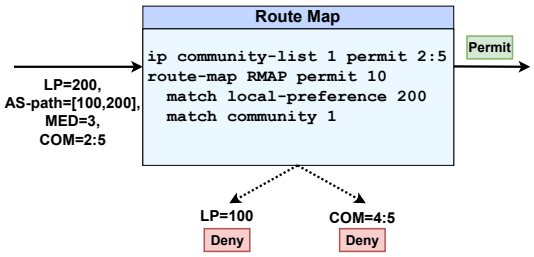


Figure 8: Changes to route map to alter the route-map decision.

MESSI to generate tests to identify incorrect incremental computation. We consider how the route filtering process is affected by a limited form of dynamics: either a modified route map or a modified (withdrawn and newly received) route.[2]

The setup for testing a modified route map is shown in Figure 8. Now our Zen model takes *three* inputs: an original route map, a route to match against, and an updated route map. The goal is to produce test cases whereby the route matches the original route map but not the updated one and vice versa.

Figure 9 similarly shows the setup for testing the case of modified routes. Now a test case consists of a route map, an original route, and an updated route, and we require that the original and updated routes are treated differently by the route map.

The key challenge in generating tests (for even the very limited dynamics we focus on) is the large size of the search space. For example, consider the Zen model for testing route-map changes, which takes as input two route maps and a route. If we perform symbolic execution naively, Zen must explore the space of all possible pairs of route maps that behave differently on the given route. This space is too large, so we will end up exploring a small subset of the space, but with no clear guarantees on coverage.

Instead, we leverage Zen to perform a form of exhaustive *local search*. Specifically, we define a *distance metric* between route maps, which intuitively represents the number of changes to individual stanza actions and match statements that must be made to transform one route map into another akin to Hamming distance. Then we ask our Zen model to only generate pairs of route maps that are within distance k of one another ($k = 1$ in our experiments), which can be done exhaustively in a reasonable amount of time.

For example, consider the following route map:

```
ip community-list 1 permit [1-2]+:[1-2][3]*
route map FILTER_ROUTES permit 10
       match community 1
```

We consider changing the action from `permit` to `deny` as a cost of one. Similarly, changing the regex in community list 1 to another regex is considered a cost of one. Notably, we treat match or set statements on different attributes as being infinitely far apart. For example, if the `match community` statement in our example is changed to a `match metric` statement, then the distance is infinite. In short, our distance metric models the kinds of local changes that network operators often make to route maps, in our experience, and it also makes symbolic test-case generation feasible.

[2]Our use of "small step" BGP dynamics should not be confused with "multi-step" BGP dynamics that manifest in phenomena such as BGP looping and route flap damping.

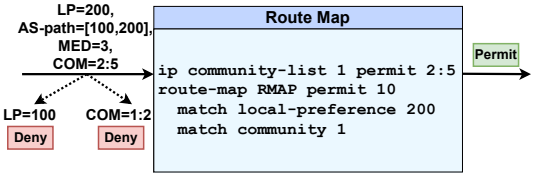


Figure 9: Dynamic Testing by changing route attributes to alter the route-map decision

We use a similar approach to test a modification to the incoming route. Namely, we define a distance metric on route announcements, where each attribute modification has a cost of one, and the Zen model exhaustively explores pairs of routes that have distance at most k ($k = 1$ in our experiments).

We also tested the dynamics of route aggregation similarly, where we allow changes to either the route attributes in one setting or the values of aggregation-specific flags and aggregation prefix (config) in another setting. The distance metric is defined in a similar fashion as above: changing the aggregate prefix, the flags, or a route attribute is considered a cost of one. In our experiments, we constrained this cost to be at most 1.

While supported by our model, increasing distance beyond 1 increases both test cases and generation time exponentially.

3.6 Vendor-specific Translators

BGP is a complicated protocol, and vendors tend to implement their own configuration languages, which differ in syntax, semantics, and features. For instance, both FRR and Quagga support prefix lists, community lists, and AS path lists. However, GoBGP supports *sets* of prefixes, communities, and AS paths, as opposed to lists. The fundamental difference between these two representations is that elements within a list have sequential priority, which allows for overlapping permitted and denied ranges, whereas set members are unordered. We address these differences through special-purpose translators that convert our Zen-generated tests to the input language of a particular implementation.

As an example, suppose Zen generates the following prefix list, denoted in FRR syntax:

```
ip prefix-list PR seq 1 deny 10.1.0.0/15
ip prefix-list PR seq 2 permit 10.0.0.0/8 ge 15 le 18
```

Note that the prefix on line 1 is in the prefix range on line 2. Our GoBGP translator converts this list into the following set of all prefix ranges permitted by line 2 but not denied by line 1:

```
NAME  PREFIX
PR    10.0.0.0/8 16..18
      10.2.0.0/15 15..15
```

Implementation	Language	Description
FRR [27]	C	Used by ISPs, DCs
Quagga [45]	C	Used in Linux
GoBGP [28]	Go	Used by ISPs, research
Batfish [3, 26]	Java	Open-source simulator
Fastplane [40]	C++	Fast BGP simulator
BIRD [13]	C	Dynamic IP routing daemon

Table 1: Implementations tested by MESSI.

```
10.4.0.0/14 15..15
10.8.0.0/13 15..15
10.16.0.0/12 15..15
10.32.0.0/11 15..15
10.64.0.0/10 15..15
10.128.0.0/9 15..15
```

The translator produces prefix sets whose elements are mutually exclusive, which simplifies translation; but it implies we never test GoBGP prefix sets with overlapping elements.

Note that our test cases cover a broad range of settings and attributes, some of which might not be supported by every implementation. For example, GoBGP does not support setting the MED attribute or the removal of AS numbers from an AS path, while both FRR and Quagga do. We also did not find documentation for route aggregation in GoBGP, while both FRR and Quagga have extensive support for it with special flags such as `matching-med-only` and `summary-only`. Similarly, BIRD only offers basic support for regular expressions. Thus while translating test cases for a specific implementation, we ignore inapplicable tests.

3.7 Implementation Testing

We use an automated testing setup that takes as input the translated test cases and sets up a network of Docker containers, one of which runs the implementation under test. The other containers use ExaBGP to send route announcements to this container. We parse the RIB of the implementation under test to obtain the installed routes that result from each test; we consider a test to fail if an implementation's RIB differs from that expected by our model. We group failed tests to reduce the manual effort of debugging. During test generation, our Zen code produces a tag for each test case based on the path it traverses in the code, which roughly indicates the kinds of match and set statements that the test exercises. Failed tests are grouped into *buckets* by tag, and one representative test from each bucket is manually investigated to confirm whether it is a bug or not. This approach is analogous to the bucketing technique from our prior work on test generation for DNS [33].

FRR [27]	
Community list AND/OR matching semantics	Fixed
Filtering routes tagged with Internet Community	Fixed
Delete Community Regex not working	Fixed
Route map permits all routes if internet community permitted	Fixed
Prefix list matches mask greater than or equals	Acked
Changing MED results in incorrect aggregation	Fixed
Route map change from permit any prefix results in incorrect route installation	Fixed
Quagga [45]	
Community list matching AND/OR semantics	Found
Filtering routes tagged with Internet Community	Found
Delete Community Regex not working	Found
Route map permits all internet community routes	Found
Changing MED results in incorrect aggregation	Found
GoBGP [28]	
Prefix set matching with zero masklength but nonzero range	Acked
Withdrawn route is still advertised	Fixed
Route installed in RIB with incorrect MED	Fixed
Batfish [3, 26]	
Space in Community regex giving error	Acked
Internet community route filtering	Found
Fastplane [40]	
Set MED not advertised	Found
AS prepend with 0	Found
AS path regex parsing issue	Found
AS path regex matching issue	Found
Community Regex matching issue	Found

Table 2: Bugs found for different BGP implementations.

Stats	Decision	Filtering	Aggregation	Dynamics
Tests	18	142916	128	11246
Failed	5	457	0	12
Buckets	2	204	0	4
Bugs	1	17	0	4
Acked	1	7	0	3
Fixed	1	4	0	3

Table 3: Experimental results.

4 Results

4.1 System Overview

We generate tests for each module described earlier using the corresponding Zen model. We test implementations using Docker [42] by creating working Docker images of each implementation. For route-map filtering, test generation takes around 2 days; hence, we pipelined test generation by testing the implementations. We used Python scripts for vendor-specific translation (§3.6) and response grouping (§3.7).

4.2 Experiments

We tested six BGP implementations: FRR, Quagga, GoBGP, Batfish, Fastplane, and BIRD. Table 1 shows their source code language and a brief description of their focus or how they are used. Table 2 catalogues all the bugs we found. In Table 3, the first three rows show the number of Zen-generated test cases , failed tests and buckets respectively. The last three rows list the number of bugs found, acknowledged and fixed across all implementations respectively. We found at least 2 bugs in each of the 5 implementations except BIRD, in which we found no bugs. Many of our identified bugs relate to regexes or dynamics. However, BIRD has rudimentary support for regexes, making many of our test cases inapplicable. Further, unlike FRR and Quagga, BIRD restarts the daemon upon config changes, making incremental update errors unlikely.

We next describe the results for each module individually.

Decision Process: For testing the decision process of BGP, we made a model of the BGP decision algorithm in Zen. It generated 18 test cases corresponding to every path (pseudocode shown in Figure 6). We found 1 bug in the decision process of GoBGP and found 4 discrepancies across FRR and Quagga, which revealed some implementation-specific behavior – i.e., for Quagga to choose the route with lower router-id, the bgp `bestpath compare-routerid` flag has to be enabled; by contrast, this is enabled by default for FRR.

Route Filtering: To test BGP route filtering, MESSI simultaneously generates an incoming route along with an inbound routing policy (route map) as a test case. A route map, in general, can have multiple stanzas, with each stanza having its own match and set statements. However, as stated earlier, MESSI generates tests with only one stanza.

We encoded the route map evaluation logic (explained in § 3.3) in Zen for test generation. We also restrict each field of the route and the route map with some validity constraints. The regex testing was discussed earlier in § 3.3.3. MESSI generated 4686 regexes (up to level 3) using enumeration Algorithm 1 and generated positive and negative examples using the coverage algorithm (Figure 7) with a total of 253,958 test cases for regex testing.

In the route map, we only allowed prefix lists to have 3 entries. We did this not only to reduce the search space but also to ensure that we had test cases that could check whether the BGP implementations were following the sequential order while matching entries within the prefix list. IP addresses were limited to unsigned integers within a certain specified range. This is a necessary step so that Zen does not generate invalid IP addresses such as `0.0.25.203` that do not follow CIDR rules.

Second, for ease of computation, we represented subnet masks in their unsigned integer form. The primary reason for doing this is that Zen does not support shift operations like « and ». Thus instead of representing the masks internally as a number in the range 0-32, we converted them to their corresponding unsigned integers. This made it easier to `AND` the subnet masks with the IP addresses to get the prefix value.

```
1  Zen<bool> RMapDynamics(Zen<Route> route,
2      Zen<RouteMap> r1, Zen<RouteMap> r2)
3  {
4      // Limit maximum distance to 1
5      var cond1 = GetDifference(r1, r2) <= 1;
6      // get decisions from the two route maps
7      var dec1 = r1.Match(route);
8      var dec2 = r2.Match(route);
9      // check whether they have different results
10     var cond2 = Not(dec1 == dec2);
11     return And(cond1, cond2);
12 }
```

Figure 10: Constraints for route map dynamics

We also constrained the prefix mask, LE, and GE values to obey the rule `mask <= GE <= LE`. Additionally, to avoid repetition and redundancy, all entries in the prefix list were made unique. AS path and Community lists were limited to 1 entry each. This tactic was again taken to reduce the number of test cases. Finally, local preference and MED values were restricted within a pre-specified range.

This setup generated 142,916 tests, revealing 15 previously unknown bugs across the tested implementations.

Route Aggregation: MESSI uses the model described in §3.4 to simultaneously generate the aggregate prefix, the flags, and two individual prefixes for a test case.

Zen generates 128 test cases to cover all paths in this module. We did not find any bugs in the route aggregation logic across all implementations.

BGP Dynamics: We also tested the dynamics (explained in §3.5) related to route map filtering and route aggregation. The dynamics of route map filtering are further classified into two categories - (1) dynamic changes in the route map and (2) dynamic changes in the incoming route. In the experiments, we use the distance metric (how close route maps or routes are to each other) defined in §3.5.

For (1), we wrote a wrapper method around the route map evaluation module to generate two route maps and an incoming route, subject to the constraints that the route maps should differ by, at most, a unit distance, and one of them should allow the route while the other denies the route. This setup generated 7404 test cases. Figure 10 shows the logic for implementing the wrapper. First Line 5 limits the maximum distance between two route maps to 1. Next, Lines 7 to 8 evaluates the incoming route against the two route maps. Finally, in Line 10, we check to see whether the two decisions are different. For a particular test case, both these conditions must be satisfied.

For (2) we wrote another Zen wrapper to simultaneously generate two routes and one route map. The constraints it is subjected to are: first, the two routes should differ by at most a unit distance; and second, the route map should give different decisions for the two routes. This setup generated 28 test cases.

We also tested route aggregation dynamics, simultaneously generating configs (aggregate prefix + flags) and incoming

routes for two scenarios: changing routes and changing config. Every change is of unit distance. In both scenarios, our constraints ensured the published set of routes are different for the two different settings. In the changing routes scenario, we altered one route keeping a constant config. Conversely, in the changing config scenario, we modified the config while maintaining identical routes. This setup generated 3814 test cases.

Additionally, with each test case, we tested the effect of withdrawal. Dynamic testing revealed 4 bugs across all implementations.

4.3 Example Bugs

In this subsection, we provide more examples of the bugs that MESSI identified, in addition to the ones presented in §2.

Bug #4: The following test was generated by MESSI:

```
Route Map:
ip community-list 1 permit internet
route map FILTER_ROUTES permit 10
       match community 1
Route: 100.10.0.0/16, COM = [0:11]
```

In the above test case, the route map is configured in a way that only permits routes with the Internet community. Although it is expected that only routes containing the internet community (0:0) will be permitted, we observed that FRR and Quagga permit all incoming routes due to a bug in handling the community list. In the source code, there was a code snippet that was setting the decision as `permit` if `internet` community was contained in the community list. The developers in FRR acknowledged and fixed this bug.

Bug #5: MESSI generated the following test case:

```
Route Map:
ip community-list 1 permit 0:0
route map FILTER_ROUTES permit 10
       match community 1
Route: 100.10.0.0/16, COM = [0:0]
```

Here, the route map permits routes tagged with 0:0 (`internet`) community, and the incoming route has community 0:0. Therefore, the expected decision is `PERMIT`. For this particular test case, we observed some discrepancies across different implementations. When we ran this test on FRR, it denied the route, although Fastplane and Batfish allowed it (as expected). If we use `internet` in the route map, and the incoming route has `0:0` as a community, then FRR gives the expected output. But, Batfish doesn't give the expected output. When we filed this issue, FRR developers fixed it by deprecating the `internet` community. We have reported this to the Batfish developers and are waiting for their response.

Bug #6: In another test case generated by MESSI the route map permits all routes that match with default route `0.0.0.0/0` with a mask length greater than or equal to zero. But in FRR, we observed that it does not work. It works as expected in Batfish and Fastplane. However, `ge 1` does work, and it was given as a workaround by the FRR developers.

```
Route Map:
ip prefix-list PFXL seq 5 permit 0.0.0.0/0 ge 0
route-map FILTER_ROUTES permit 10
       match ip address prefix-list PFXL
Route: 100.10.0.0/16...
```

The documentation does not explain why `ge 0` does not work. Neither does FRR throw any error if `ge 0` is provided in the configuration. GoBGP had a similar issue: if there is a prefix set with prefix `0.0.0.0/0` but with a range on mask length, it should match all routes within that mask length range but did not match a prefix generated by MESSI. The GoBGP developers acknowledged this and will hopefully fix this soon. The failed test case generated by MESSI was:

```
prefix-set ps1:
  ip-prefix: 0.0.0.0/0, mask-range: 10..10
policy-definitions:
    conditions:
      match-prefix-set: ps1
Route: 100.10.0.0/10
```

Bug #7: A route map contains a set statement that deletes specific communities from routes that match with some community list. The route communities are not deleted as expected because of an additional `delete` tag that was added to the name of the community list, which made the delete community not work. We found this bug in Quagga and in earlier versions of FRR (fixed later). The failing test case was:

```
Route Map:
ip community-list 101 permit [1-2]+:[3-4]
route map FILTER_ROUTES permit 10
    set coom-list 101 delete
Route: COM = [11:4 0:1]
```

The expected result is the deletion of community `11:4` from the route but it did not delete any communities.

Bug #8: The following test was generated by MESSI:

```
Route Map:
ip prefix-list PFXL seq 5 permit 99.0.0.0/8 le 31
route-map FILTER_ROUTES permit 10
    match ip address prefix-list PFXL
Route: 99.10.11.0/24, MED = 4
```

This test case failed for GoBGP and Fastplane. Here, the incoming route has a `MED 4`, and the inbound route map accepts the route without modifying it. But the installed route had `MED 0`. This error was fixed in GoBGP and is under review by the Fastplane developers. The same test case was also generated by our decision process setup.

Bug #9: Another test case generated by MESSI revealed a bug in Fastplane. The test case is as follows:

```
Route Map:
route-map FILTER_ROUTES permit 10
    set as-path prepend 0 100
Route: AS-path = 200
```

Here the inbound route map has a set statement that should add AS 0 and AS 100 to the AS path of the route, but it only appended AS 100 to the existing AS path. All other implementations appended it correctly. This issue is likely due to a highly compact zero-terminated representation that Fastplane uses to represent the AS path for BGP routes.

5 Limitations

Our model currently focuses on RFC compliance errors only; we do not consider performance bugs or coding bugs such as overflow errors. We do not test route reflectors, confederations, reserved ASNs, and well-known communities, and some regex features such as constraining a route's community set size.

Modular exploration does not test possibly complicated interactions among multiple BGP features. However, the modular approach is more scalable, allowing each feature to be tested extensively; it also allows easily adding support for new features without changing existing portions of the model significantly. We can also use our approach to test the integration of multiple modules when desired.

MESSI automatically configures routers through the command line interface. Certain bugs (e.g., §2.1) would not be revealed by alternative testing approaches, such as directly writing the configuration file. In future work, we could employ multiple test setups to widen the scope of errors found.

6 Related Work

We classify related work into the following categories.

Model-based verification: This approach uses a formal model to verify software. For example, Bagpipe [51] verifies BGP configurations against a formal model of BGP. Verification scales for configurations but not for implementations.

Fuzz testing: Fuzz testing is widely used for software testing in general [2, 14, 37, 52] and specifically for BGP implementations (e.g., [20, 22]). Although fuzzers are effective at finding parser bugs, they are less effective at finding behavioral bugs such as those described in §2.

Symbolic execution: Symbolic execution invokes SMT solvers to generate test cases for as many execution paths of a program as possible (e.g., [17, 29, 30]). BGP implementations contain thousands of lines of low-level code, which makes symbolic execution infeasible in practice.

Model learning: Recent work uses active learning to create abstract models from black-box protocol implementations [25]; errors are detected by comparing the models of different implementations. Compared to our approach, the drawback is the need to learn a model for each protocol implementation; we generate tests for arbitrary protocol implementations from a single reference model. However, their models can be used for purposes other than error detection.

Model-based testing: This general category uses an abstract model of a system to generate tests [16,44,49]. MESSI extends the SCALE model-based testing approach for DNS implementations [33] to account for BGP's complexity and statefulness.

Enumerative Testing: This approach generates inputs up to a given size that meet a specified predicate [15]. We use a form of enumeration to generate regexes for BGP configurations.

7 Future Work and Conclusion

We plan to extend MESSI to test other BGP features such as route redistribution and route reflection. We also aspire to *automatically* derive the constraints from the BGP RFCs, say using large language models like GPT-4, to alleviate the current burden of manually building the model.

The ideas in MESSI should extend to other stateful protocols with complex structures, as well as to other software systems, such as web servers. BGP has a comparatively simple stateful model: if one excludes dynamics, the forwarding of a route depends only on the previous best stored route. However, the state of other protocols may depend on many past messages; this raises the question of efficiently driving such protocols to specified states. MESSI's combination of symbolic, random, and enumerative testing of structures should extend to other protocol structures.

A common slogan when an Internet outage occurs is: "It's always DNS ... except when its BGP" [10] because the majority of major Internet incidents [10] are caused by bugs in DNS (zone files or implementations) or BGP (configurations or implementations). Previous work targets DNS zone file bugs [32], DNS implementation bugs [33], and BGP configuration bugs (e.g., [26]); MESSI fills the remaining gap by targeting BGP implementation bugs.

In terms of ideas, MESSI adds techniques to the repertoire of model-based testing approaches for protocol implementations (e.g., SCALE [33]) to deal with: **1.** complexity (modular exploration), **2.** protocol statefulness (generating both the best route and a new route); **3.** implementation statefulness due to incremental computation (generating two configurations that differ by a single change); **4.** complex structures (combining symbolic with enumerative testing for regexes).

Inspired by the famed soccer player, MESSI tries to attack the goal of BGP testing from many angles, generating over 150K test cases automatically that have already led to the discovery of 22 new bugs in 5 BGP implementations.

Acknowledgements

We thank our shepherd Eric Eide and the anonymous reviewers for their insightful comments. We also thank the BGP developers for their feedback on the bug reports. This work was partially supported by NSF grant CNS-1901510 and by Cisco.

References

[1] About the real-time effect of BGP policy configuration. https://github.com/osrg/gobgp/issues/2164.

[2] American fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[3] Batfish network configuration analyzer. https://github.com/batfish/batfish.

[4] Ryan Beckett. Zen. https://github.com/microsoft/Zen/tree/master.

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 155–168, New York, NY, USA, 2017. ACM.

[6] Ryan Beckett and Ratul Mahajan. A general framework for compositional network modeling. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 8–15, 2020.

[7] Ann Bednarz. Global microsoft cloud-service outage traced to rapid bgp router updates. *Network World*, 2023.

[8] BGP 'address-family ipv4' sub-configuration is not shown in the running configuration. https://quickview.cloudapps.cisco.com/quickview/bug/CSCvk45884.

[9] BGP bug bites juniper software. https://www.networkworld.com/article/2289950/bgp-bug-bites-juniper-software.html.

[10] BGP, DNS, and the fragility of our critical systems. https://www.f5.com/labs/articles/cisotociso/bgp-dns-and-the-fragility-of-our-critical-systems.

[11] BGP route-map work incorrectly when prefix-list modify from deny any to permit. https://github.com/FRRouting/frr/issues/13007.

[12] BGPD: aggregate-address with summary-only and matching-med-only does not work when metric is changed. https://github.com/FRRouting/frr/issues/11912.

[13] BIRD routing software. https://bird.network.cz/?get_doc&v=20&f=bird-5.html.

[14] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[15] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In Phyllis G. Frankl, editor, *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, pages 123–133. ACM, 2002.

[16] Josip Bozic, Lina Marsso, Radu Mateescu, and Franz Wotawa. A formal tls handshake model in lnt. *arXiv preprint arXiv:1803.10319*, 2018.

[17] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[18] Tom Strickx Celso Martinho. Understanding how facebook disappeared from the internet. *Cloud Flare*, 2021.

[19] Cisco ios xe software bgp resource public key infrastructure dos vulnerability. https://bst.cisco.com/quickview/bug/CSCvz55292.

[20] Stanislav Dashevskyi. A simple BGP fuzzer based on boofuzz. *Github*, 2023. https://github.com/Forescout/bgp_boofuzzer.

[21] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] Donatas Abraitis et al. Donald Sharp. fuzzing targets and supported fuzzers available in frr. *Github*, 2023. https://docs.frrouting.org/projects/dev-guide/en/latest/fuzzing.html.

[23] eBGP doesn't withdraw routes to other ebgp peers when peer goes down. https://github.com/osrg/gobgp/issues/2208.

[24] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 43–56, Berkeley, CA, USA, 2005. USENIX Association.

[25] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. Prognosis: closed-box analysis of network protocol implementations. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 762–774, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, Oakland, CA, 2015. USENIX Association.

[27] FRR routing software. https://github.com/FRRouting/frr.

[28] GoBGP routing software. https://github.com/osrg/gobgp.

[29] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, pages 206–215, 2008.

[30] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[31] DAN GOODIN. Google goes down after major bgp mishap routes traffic through china. *ARS Technica*, 2018.

[32] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. Groot: Proactive verification of dns configurations. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 310–328, New York, NY, USA, 2020. Association for Computing Machinery.

[33] Siva Kesava Reddy Kakarla, Ryan Beckett, Todd Millstein, and George Varghese. {SCALE}: Automatically finding {RFC} compliance bugs in {DNS} nameservers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 307–323, 2022.

[34] Siva Kesava Reddy Kakarla, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir, and George Varghese. Finding network misconfigurations by automatic template inference. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 999–1013, 2020.

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

[36] Eduard Kovacs. Bgp flaw can be exploited for prolonged internet outages. *Security Week*, 2023.

[37] Hyojeong Lee, Jeff Seibert, Dylan Fistrovic, Charles Killian, and Cristina Nita-Rotaru. Gatling: Automatic performance attack discovery in large-scale distributed systems. *ACM Trans. Inf. Syst. Secur.*, 17(4), apr 2015.

[38] Jonathan Lee and Jeffrey Shallit. Enumerating regular expressions and their languages. In *Proceedings of the 9th International Conference on Implementation and Application of Automata*, CIAA'04, page 2–22, Berlin, Heidelberg, 2004. Springer-Verlag.

[39] Hongqiang Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *SOSP '17 Proceedings of the 26th Symposium on Operating Systems Principles*, pages 599–613. ACM, October 2017.

[40] Nuno Lopes and Andrey Rybalchenko. Fast bgp simulation of large datacenters. In *VMCAI: Verification, Model Checking, and Abstract Interpretation*, January 2019.

[41] Robert McMillan. Youtube outage underscores big internet problem. *Info World*, 2008.

[42] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux j*, 239(2):2, 2014.

[43] Sebastian Moss. Verizon bgp route leak causes cloudflare customer outages, aws issues. *Data Center Dynamics*, 2019.

[44] Javier Paris and Thomas Arts. Automatic testing of tcp/ip implementations using quickcheck. In *Proceedings of the 8th ACM SIGPLAN Workshop on Erlang*, pages 83–92, 2009.

[45] Quagga routing software. https://www.nongnu.org/quagga/.

[46] Regular expressions. https://microsoft.github.io/z3guide/docs/theories/Regular%20Expressions/.

[47] Yakov Rekhter, Susan Hares, and Tony Li. A Border Gateway Protocol 4 (BGP-4). RFC 4271, January 2006.

[48] Alan Tang, Siva Kesava Reddy Kakarla, Ryan Beckett, Ennan Zhai, Matt Brown, Todd Millstein, Yuval Tamir, and George Varghese. Campion: Debugging router configuration differences. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 748–761, New York, NY, USA, 2021. Association for Computing Machinery.

[49] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive

systems with spec explorer. *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*, pages 39–76, 2008.

[50] Brandon Vigliarolo. Faa grounds all us departures after notam goes down. *The Register*, 2023.

[51] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Bagpipe: Verified bgp configuration checking. In *Proc. OOPSLA*, 2016.

[52] zzuf - multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf.

A High-Performance Design, Implementation, Deployment, and Evaluation of The Slim Fly Network

Nils Blach[1], Maciej Besta[1], Daniele De Sensi[1,2], Jens Domke[3],
Hussein Harake[5], Shigang Li[1,4], Patrick Iff[1], Marek Konieczny[6], Kartik Lakhotia[7],
Ales Kubicek[1], Marcel Ferrari[1], Fabrizio Petrini[7], Torsten Hoefler[1]

[1] *ETH Zürich* [2] *Sapienza University of Rome* [3] *RIKEN Center for Computational Science (R-CCS)*
[4] *BUPT, Beijing* [5] *Swiss National Supercomputing Centre (CSCS)* [6] *AGH-UST* [7] *Intel Labs*

{ nils.blach, maciej.besta, htor } @ inf.ethz.ch

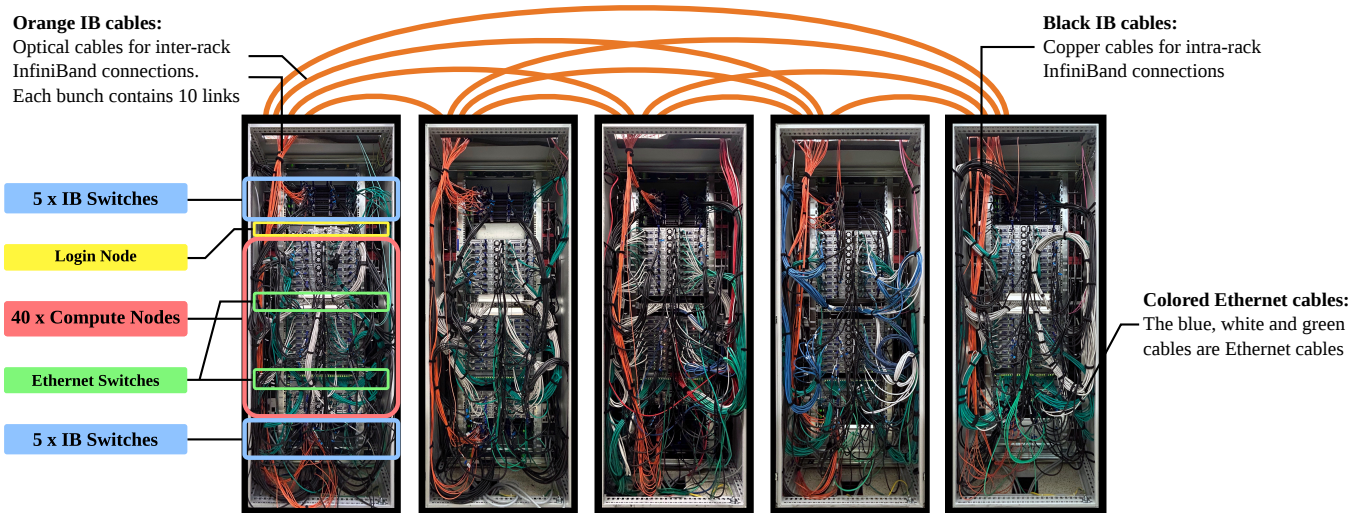


Figure 1: First real-world deployment of the Slim Fly topology. The left-most rack displays labels detailing the arrangement of various components such as InfiniBand (IB) switches, compute nodes and Ethernet switches. Two types of IB links are present: black copper links for intra-rack connections and orange optical fiber links for inter-rack connections. The orange lines above the racks represent bundles of ten optical fiber links each. Additionally, blue, white and green (arbitrary color scheme) Ethernet cables are visible within the racks, which establish the cluster management network together with the Ethernet switches.

Abstract

Novel low-diameter network topologies such as Slim Fly (SF) offer significant cost and power advantages over the established Fat Tree, Clos, or Dragonfly. To spearhead the adoption of low-diameter networks, we design, implement, deploy, and evaluate the first real-world SF installation. We focus on deployment, management, and operational aspects of our test cluster with 200 servers and carefully analyze performance. We demonstrate techniques for simple cabling and cabling validation as well as a novel high-performance routing architecture for InfiniBand-based low-diameter topologies. Our real-world benchmarks show SF's strong performance for many modern workloads such as deep neural network training, graph analytics, or linear algebra kernels. SF outperforms non-blocking Fat Trees in scalability while offering comparable or better performance and lower cost for large network sizes. Our work can facilitate deploying SF while the associated (open-source)[1] routing architecture is fully portable and applicable to accelerate any low-diameter interconnect.

[1] https://github.com/spcl/opensm

1 INTRODUCTION

Low-diameter[2] network topologies such as Slim Fly (SF) [1] have gained significant traction during the last decade. Initial designs in that line of work, Dragonfly (DF) [2] and Flattened Butterfly [3], both with diameter three, focused on improving latency and physical layout. After that, SF lowered the diameter to two, based on an observation that low-diameter does not only improve performance by reducing end-to-end latencies, but *it also reduces cost and power consumption*. This is because, when the diameter is lower, packets on average traverse fewer switches, switch buffers, and links. Thus, fewer links and buffers are needed to construct the network (for a fixed bandwidth), and less dynamic power is needed for moving the packets through the network.

SF's construction costs, consumed power, and latency are lower than those of Clos and Fat Tree (FT) by respectively, ≈25-30%, ≈25-30%, and ≈50% [1]. However, SF has still not seen a real physical deployment, and it is uncertain how to

[2] Network diameter is the maximum distance between any two switches.

deploy SF in practice. To spearhead the practical development of low-diameter networks and show the state-of-the-practice, we design, implement, deploy, and evaluate the first SF installation that includes switches and endpoints, as shown in Fig. 1. We discuss the encountered challenges, and we show that the construction process is straightforward and comparable to established designs such as Clos.

Moreover, to maximize performance benefits from using SF, we design and implement a novel high-performance multipath routing scheme for general low-diameter networks, and we install and use it with the deployed SF cluster. Our routing shows superior performance over the state-of-the-art, and it is independent of the underlying topology details and of the interconnect architecture. Thus, it could be portably used on different topologies (e.g., Xpander [4]) and on different architectures (e.g., Ethernet or InfiniBand [5]).

The equipment available to us is based on the InfiniBand (IB) architecture [5]. IB enables a high-speed switched fabric with hardware (HW) support for remote direct memory access (RDMA) [6, 7]. IB is widely used in high-performance systems, for example four out of ten most powerful systems in the Top500 list (Jun. 2023 issue) [8], manufactured by IBM, Nvidia, and Atos, use the IB interconnect. We use our routing protocol with the IB networking stack; our whole implementation is publicly available to foster future research into multipath routing. Importantly, we provide the first multipathing for IB that can use arbitrary paths (including non-minimal and disjoint ones) and that is independent of the structure details of the underlying network [9, 10].

In our evaluation, we consider a broad range of communication-intense applications that represent traditional dense computations (like physics simulations), sparse graph processing [11, 12, 13, 14, 15, 16], deep neural network (DNN) training [17, 18, 19], and a number of microbenchmarks testing particular popular communication patterns. Our results showcase that SF delivers high performance while achieving optimal, or near optimal scalability, which directly translates to low construction costs. To further reinforce these outcomes, we also conduct a comprehensive comparison between SF and a non-blocking FT that we deploy using the same hardware. Here, SF offers comparable or better performance to FT in a majority of used applications. Simultaneously, its superior scalability ensures up to 50% cost improvements over FT, particularly for large installation sizes [1].

2 NETWORK MODEL & TOPOLOGIES

We start with fundamental concepts and notation. We model a network as an undirected graph $G = (V, E)$; V is a set of switches[3] ($|V| = N_r$) and E is a set of full-duplex inter-switch cables (we do not model endpoints explicitly). A network has N endpoints, with p endpoints attached to each switch (*concentration*). We also use the term *node* to refer to either a switch or any of its endpoints, when the discussion is generic. Total port count in a switch (*radix*) is $k = k' + p$, where k' is the number of channels from a switch to other switches (*network radix*). The diameter is D. All the symbols are listed in Tab. 1.

[3]We abstract away HW details and denote switches and routers with a common term "switch". However, we use a term "routing" when referring to determining a path, because IB switches in our physical implementation have routing capabilities.

Table 1: The most important symbols used in this work.

V, E	Sets of vertices/edges (switches/links, $V = \{0, \ldots, N_r - 1\}$).
N	The number of endpoints in the network.
N_r	The number of switches in the network ($N_r = \|V\|$).
p	The number of endpoints attached to a switch.
k'	The number of channels from a switch to other switches.
k	*Switch radix* ($k = k' + p$).
D, d	Network diameter and the average path length.

We overview SF's structure in Fig. 2, and compare it to a 3–level Fat Tree with diameter four, as they are widely used in medium and large installations [20, 21], and to a diameter-3 Dragonfly, which has also been deployed in practice [22, 23]. SF has >50% fewer switches and >55% fewer cables than a full-bandwidth non-blocking FT of a comparable size. Second, SF's switches form *groups* that are not necessarily fully connected; FT's edge and aggregation switches form *pods*, DF's groups are fully connected. Third, both SF and DF are *direct* topologies (each switch is attached to some number of servers), while in a FT, only edge switches attach to servers.

3 FIRST AT-SCALE SF INSTALLATION

We start by discussing the deployment of the first SF cluster, illustrating the simplicity of its construction and arguing why deploying other SFs would also be straightforward. The cluster is hosted by the Swiss National Supercomputing Centre (CSCS).

3.1 Deployed Hardware Equipment

We use 50 36-port, 56Gb/s IB SX6036 switches and 200 compute endpoints. Each endpoint hosts two 20-core Intel Xeon CPUs and 32 GiB RAM, split equally in a Non-Uniform Memory Access (NUMA) configuration, and a single Mellanox ConnectX-3 MT4099 HCA, which implements the IB Architecture Specification Volume 1, Release 1.2. Copper and optical cables are used for intra and inter-rack switch connections, respectively.

3.2 Topology Structure and Construction

We use a SF based on the graphs by McKay, Miller, and Širáň [24]. We outline its structure, the details are in Appendix A and in the original SF paper [1]. The complete SF installation is shown in Fig. 1 with a highlighted view of the group structure in Fig. 3. One first chooses a prime power q; q is an input parameter that determines the whole topology structure. For example, the number of vertices (switches) is $N_r = 2q^2$ and the network radix $k' = \frac{3q-\delta}{2}$. In our case, $N_r = 50$, thus $q = 5$ and $k' = 7$ (every switch connects to 7 other switches). Interestingly, this construction forms the

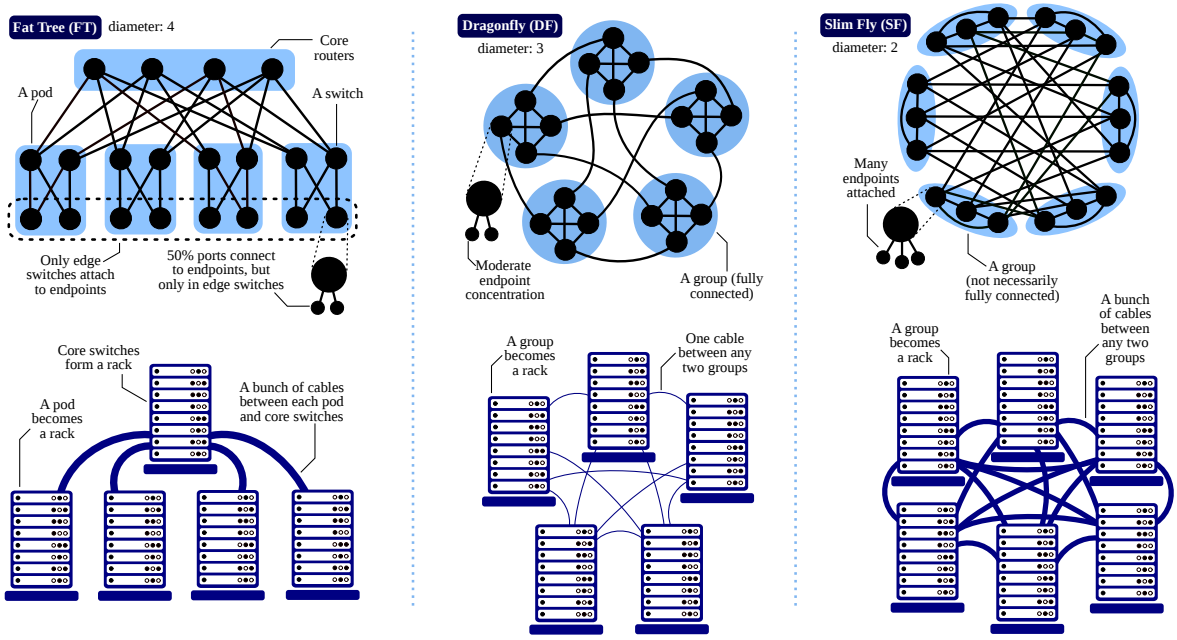


Figure 2: The structure of a small example Fat Tree (FT), Dragonfly (DF), and Slim Fly (SF), and the corresponding installations. Each topology comes with a modular design, where switches form groups (SF, DF) or pods (FT). Such groups can become racks in a physical installation.

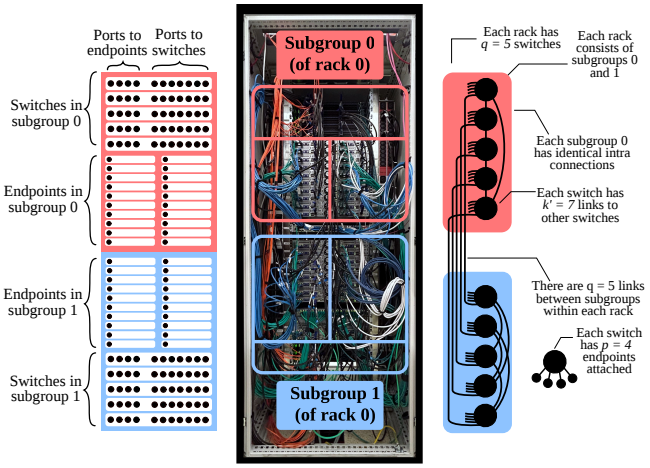


Figure 3: Internal organization of a rack. The image displays a side-by-side comparison of a theoretical diagram and an actual photograph of a single rack in the cluster. The rack consists of two distinct subgroups, each housing 5 IB switches and 40 compute nodes (endpoints). Each IB switch is connected to 4 endpoints and 7 other IB switches.

famous Hoffman-Singleton graph [25, 26], which is *optimal* with respect to the Moore Bound [27]. Finally, one uses $p = \left\lceil \frac{k'}{2} \right\rceil$ endpoints connected to each switch to ensure *full global bandwidth* [1]. In our case, $p = 4$. Note that, while the switch port count in the considered SF is $k' + p = 11$ (and 11-port switches would be the appropriate selection when building the SF from scratch), we use 36-port switches because this has been the only HW equipment available to us.

The whole installation consists of five identical racks. Every two racks are connected with the same number of $2q = 10$ cables. There are $2q = 10$ switches in each rack. Each rack consists of two *subgroups*, *subgroup 0* and *subgroup 1*. All subgroups 0 and all subgroups 1 are identical, but a subgroup 0 and 1 are usually different. We place switches from subgroup 0, together with their attached endpoints, at the top of each rack; subgroup 1 goes to the bottom of the rack. The details on how any two switches are connected is determined by the underlying algebraic structure of the SF topology. We offer full details in Appendix A, with Appendix A.3 explaining the three simple equations that determine switch connectivity; here, we stress that the deployment is straightforward.

3.3 Deployment Efficiency and Ease

To facilitate deployment, we develop scripts that outline both intra- and inter-rack connections. The output of these scripts can be used to create diagrams for every rack pair to ensure a smooth wiring process. Thanks to the algebraic structure of the SF topology, such descriptions for any SF can be automatically generated, providing concrete port-to-port link descriptions and rack placements for each switch. We illustrate an example diagram of connections between racks 0 and 1, and between 0 and 2, that was created based on these generated descriptions, in Fig. 4.

We use our scripts as a basis of an efficient 3-step wiring process. First, we wire intra-subgroup connections; they are identical across all racks for each of the two subgroups. The second step consists of connecting each switch from subgroup 0 to its neighboring switches in subgroup 1 within the same rack. As the subgroups are of equal size, an incorrectly connected pair will result in easily recognizable errors, which break that symmetry. Lastly, the inter-rack connections are established. Hereby, the fact that each switch in a rack uses the same port to connect to the switches in another rack, enables straightforward connection of rack-pairs.

The simplicity of the wiring process can mainly be attributed to the scalable three-step approach, which is equally applicable to larger SF topologies, enabling the efficient deployment of SF clusters. Overall, stripping the previous system and executing the 3-step wiring process were completed within 3 days by a team of two.

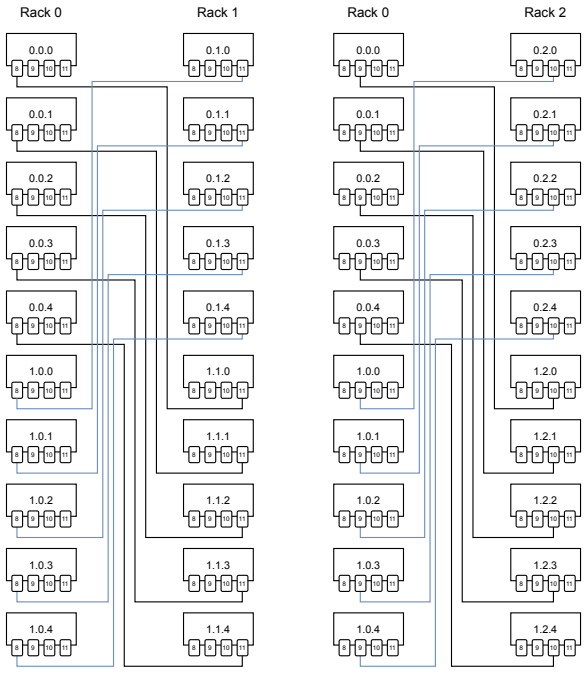


Figure 4: Illustration of the example diagrams created from the output of our scripts, facilitating the cabling process. The diagrams show all the inter-rack connections and the corresponding ports in switches. Each switch is labeled using a triple (S,R,I), where $S \in \{0,1\}$ indicates the subgroup type, $R \in \{0,...,4\}$ indicates the rack, and $I \in \{0,...,4\}$ is the consecutive switch ID within a rack/subgroup. Then, we only show ports 8–11; these ports are used to connect racks. Ports 1–4 (for endpoints) and 5–7 (for intra-rack switch-switch links) are omitted for clarity. The equations presented in Appendix A.3 determine which switches are connected based on the assigned labels.

3.4 Correctness Verification

We provide a set of scripts that ensure the correctness of the cabling. These scripts utilize the auto-generated port-to-port link descriptions and rack placements for each switch and compare it with the output of `ibnetdiscover`, an IB command that performs fabric discovery. This allows us to not only identify incorrectly wired cables and provide concrete instructions on how to rectify mistakes, but also detect missing or broken links. These scripts could even be used on a live cluster, while going through the wiring process, to immediately identify and flag errors.

4 HIGH-PERFORMANCE MULTIPATHING

We now propose a novel high-performance multipath routing protocol for low-diameter networks, which we use on the described SF deployment. For this, we extend the recently proposed FatPaths multipath routing protocol [28] so that it offers vastly superior throughput while still ensuring very low latency.

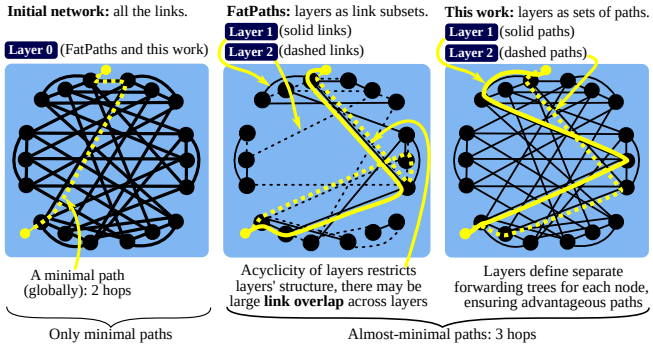


Figure 5: Layered routing in FatPaths and in this work. Traffic is divided and sent using different layers. Our scheme relaxes the requirement in FatPaths for all layers to be trees, as in our scheme deadlock resolution is decoupled from layer creation. **This ensures more flexibility in developing layers, leading to more throughput.** Specifically, while in FatPaths, paths in different layers often overlap (cf. Layer 1 and 2), our routing alleviates this issue and reduces overlap/congestion and increases performance.

4.1 Original FatPaths Routing in Slim Fly

In terms of path diversity, FT has multiple same-length minimal paths between any two edge switches. Thus, one often uses ECMP [29] for multipath routing in FT. In SF (and to some degree in DF [28]), there is usually only one minimal path, but multiple "almost" minimal paths between any switch pair. This makes it challenging to achieve high path diversity in SF using ECMP. To alleviate this and to enable non-minimal high-speed multipathing in SF, the *FatPaths architecture* has recently been proposed [28]. FatPaths harnesses the concept of *layered routing* [30, 31] for low-diameter networks. In layered routing, one first creates *layers*: subsets of switch-switch links. Within one layer, one uses shortest-path routing. However, as a layer does not contain all the links, paths within this layer are usually non-minimal (in the global sense). If two nodes[4] want to communicate using multiple paths, the sending node simply sends its data using paths residing in different layers. Note that multipathing is orthogonal to transport-level issues, and one can use different layers to transfer different flows between two nodes, but also different packets or flowlets within one flow [28]. In FatPaths, selecting links (when constructing layers) is done with simple random uniform sampling; a more elaborate scheme minimizing load imbalance is also provided. Layered routing is summarized in Fig. 5.

4.2 Proposed Multipath Routing: Summary

The central issue in layered routing is *how to divide links into layers*. We aim to minimize the number of layers (which minimizes the usage of HW resources in switches) while simultaneously maximizing the number of disjoint and almost-

[4] Multipathing can be applied both at the switch and at the endpoint level. Thus, we use a term "node" to refer to switches or endpoints when a discussion is generic

minimal paths between any switch pair (for more path diversity). Moreover, a detailed analysis from FatPaths indicates that – to maintain high performance in layered routing in virtually all low-diameter networks and traffic patterns – at least *three* disjoint paths per switch pair are needed [28]. Thus, the main goal of the layer construction algorithm is to find a minimum set of layers that together provide each switch pair with at least three disjoint paths while ensuring minimum overlap between specific layers. Ideally, these three paths include the minimal one (that always exists) and two "almost" minimal ones (in the following, an "almost" minimal path means a path that is longer by one hop than the minimal path between two given switches).

An overview of our proposed layer routing is shown in Fig. 5 (right). The key difference between our scheme and FatPaths is that we do *not* remove links from layers in order to ensure deadlock-freedom or to introduce non-minimal paths. Instead, we decouple deadlock resolution from layer creation, and explicitly construct paths satisfying the appropriate constraints on their count, non-minimal length, and well-balancedness. This facilitates creating layers that result in *much* higher throughput.

4.3 Generating Routing Layers

Our layer construction scheme is detailed in Algorithm 1. The input is the topology of inter-switch connections $G = (V, E)$, and the desired number of layers $|L|$. The output is a set of layers L, where each layer contains a collection of paths connecting different pairs of nodes. These paths together define a separate forwarding tree for each node.

The layer generation starts with assigning all links to layer 1. In layer 1, we only use minimal paths, as we want to ensure that the single minimal path existing between all node pairs is included in at least one layer for each pair. Moreover, a matrix W and a priority queue p are initialized. These structures are used to find advantageous non-minimal paths for each node pair. Intuitively, a priority $p(u,v)$ of a node pair u, v is determined by the number of non-minimal paths already assigned to u, v (and maintained in other layers). The higher $p(u,v)$ is, the lower the priority of u, v is. Hence, when looking for new non-minimal paths, node pairs with fewer paths assigned are prioritized. This facilitates balancing the number of advantageous paths across all pairs of nodes, to eliminate potential hotspots in the network.

Second, each entry $W(r,s)$ in matrix W describes the weight of a link between switches r, s. This weight equals the number of paths (from any layer) that already use this link. The higher $W(r,s)$ is, the more paths use the corresponding link. Hence, when selecting new paths, we use W to balance numbers of paths across single links, minimizing risk of congestion. We also use W to balance the paths in the first layer to ensure minimal overlap of minimal paths.

Then, for every layer $2 \ldots |L|$, and for each node pair in each layer, we find a single almost-minimal path that minimizes overlap with respect to paths already added to any other layer. For this, when finding paths in a layer l, we first copy the current priorities of node pairs into a list that preserves the current state of priorities (*copy_pairs*). Here, node pairs with the same priority are in a random order, but come before any node pair with lower priority. Note that each node pair appears twice in the list, once for each direction. This enables using different paths when routing in different directions, further increasing the flexibility of path selection.

After that, we iterate over each node pair, in an attempt to construct a path for each such pair in each layer. Note that, in principle, it is possible that one cannot find a path for each node pair in each layer (we elaborate on dealing with such rare cases in Appendix B.1; we resolve them with a simple fallback to a minimal path – our evaluation shows that this does not negatively impact throughput).

In each such iteration, we first use the *find_path* routine to try to find an almost-minimal path for a given node pair *pair*, based on already inserted paths for that layer (specified in l) and weights assigned to each link (specified in W). If we are able to find a valid path, we accordingly update priorities p (*update_priorities*) and link weights W (*update_weights*). Finally, we insert the path into layer l (*add_path_to_layer*).

Algorithm 1: Construct routing layers; details are in § 4.3

```
Input : Network topology G = (V,E), number of layers |L|
Result: A set of L routing layers
// W ∈ ℝ^{N_r × N_r} contains weights of links; p is a priority
   queue, with entries being pairs of nodes
1  W = init_link_weight_matrix() // Set all matrix entries to 0
2  p = init_p_queue(G) // Each node pair gets the same priority
3  L = {E} // Layer 0 contains all the links (E)
4  for l = 1 to |L| − 1 do
5      init_layer(l) // Initialize the next layer as empty
6      node_pairs = copy_pairs(p)
7      while node_pairs ≠ ∅ do
8          pair = node_pairs.dequeue()
9          path = find_path(G, W, pair, l)
10         if valid(path) then
11             update_priorities(path, p)
12             update_weights(path, W)
13             add_path_to_layer(path, G, l)
14         end
15     end
16     L = L ∪ {l} // Add a new layer to finalized layers
17 end
```

5 IMPLEMENTATION OF MULTIPATHING

The IB architecture [5] enables a high-speed switched fabric with HW support for RDMA [6, 32] and atomic operations [33]. IB provides lossless destination-based packet forwarding that relies on link-level, credit-based flow control [34]. We now discuss the used IB features.

An IB network usually forms a single subnet consisting of physical IB switches and *Host Channel Adapters (HCAs)* that correspond to Ethernet NICs. All communication up to and including the transport layer is implemented within these two components.

Routing configuration is managed by a centralized *subnet manager (SM)*. The SM configures connected IB devices, appropriately computes the forwarding tables to implement

the used destination-based routing algorithm, and monitors the network for failures. Within an IB subnet, each HCA and each switch receive a unique *local identifier (LID)*, assigned by the SM.

Each physical IB port has several independent *virtual lanes (VLs)*. Each VL has its own receive and transmit buffers and flow control resources. There can be up to 15 VLs per physical port (depending on the equipment) and 1 VL for management traffic. Multiple VLs per port are used for deadlock freedom and to eliminate head-of-line blocking [34] (we discuss deadlocks in more detail in § 5.2).

Each switch provides a forwarding table called the *Linear Forwarding Table* (LFT) that – for a given packet – determines the outgoing port using the destination address (DLID) from the packet header. Then, for a given outgoing port, to determine the outgoing VL for a given packet, the switch uses a four-bit Service Level (SL) field from the packet header, in combination with the incoming and outgoing packet ports, to index into the *SL-to-VL table*. This enables packets to change virtual lanes at each hop and it allows for seamless utilization of switches with potentially different numbers of virtual lanes.

5.1 Routing

OpenSM, our choice of IB compliant SM, provides complete subnet information, including a list containing all nodes (switches, HCAs, routers) and ports, as well as the connections between them. We use this information to create and populate forwarding tables so that they implement the prescribed layered routing.

Multipathing In ECMP, each router stores multiple possible next-hops that each lie on a minimal path towards the destination. This approach of storing multiple next-hops for a given destination is not possible in IB. However, it can be emulated by assigning multiple LIDs to each HCA, a feature that we use to enable multipathing and to implement our layered routing in an IB setting. An HCA can receive a contiguous range of LID addresses. This range is determined by the so called *LID Mask Control* (LMC) value. Specifically, for an LMC equal x, each HCA port hosts a consecutive range of 2^x LIDs. Then, one routes towards each such LID using a *different* path. We use the information provided by OpenSM to appropriately populate forwarding tables so that they implement the layered routing described in § 4.

Implementation of Layers We assign multiple addresses to each node; one address falls into one layer (each layer gets one address from each node). Hence, a layer is physically formed by the assigned addresses and the associated forwarding entries that route traffic to these addresses. The forwarding entries are set according to the specification of layers in the initialization phase. Our scheme for constructing layers provides a data structure *port*, which specifies the output port to be used for a packet traveling to a node d, from a switch s, within a layer l; this output port is denoted with $port[l][s][d]$.

Routing Within Layers The number of layers equals the number of addresses assigned to each node. Thus, we can treat the layer ID as the offset to the base (i.e., to the first) LID of each node. Hence, for instance, routing in the first layer (ID 0) uses the base LID of each node, whereas routing in the second layer uses the base LID plus offset 1.

Populating Forwarding Tables To populate forwarding entries, we add a value $port[l][s][d]$ into the LFT of switch s, as the outgoing port number for packets being routed towards node d. As the destination address, we use the base LID of the node, *increased by the offset* l, to ensure routing within layer l. As the last step, we run a deadlock-resolution scheme that fills all SL-to-VL tables, eliminating the risk of deadlocks (cf. § 5.2).

5.2 Deadlock-Freedom

One downside of IB's credit-based flow control ensuring losslessness is the possibility of *deadlocks*. Specifically, an IB network may enter a state in which packets in different buffers wait for each other indefinitely long to free the buffers, resulting in a deadlock. To overcome this, most routing schemes use different VLs to send packets [35, 36, 37, 38, 39, 40]. By splitting a single port buffer into multiple independent logical VLs, one can break dependencies between waiting packets.

In FatPaths, each layer is acyclic, to ensure no deadlocks within each layer. However, this does not imply global deadlock-freedom on IB because of its lossless design based on channels. Specifically, one has to ensure that dependencies between packets using routes stored in *any* layers are also deadlock-free. Thus, we change the FatPaths approach by decoupling deadlock-avoidance from layer creation. Instead, we apply deadlock-removal *after* the layers are created. This also enables much more throughput because acyclic layers vastly restrict the choice of paths to be taken.

In our IB implementation, we propose and enable the use of two different deadlock-avoidance schemes. Firstly, if a sufficient number of VLs is available, we use the scheme introduced with the Deadlock-Free Single Source Shortest-Path (DFSSSP) [36] algorithm, which is already integrated in IB. Intuitively, given a ready routing (i.e., the populated forwarding tables), DFSSSP first finds all dependencies that could lead to a deadlock, and then it iteratively accommodates these dependencies in a deadlock-free way, by assigning selected routes to use yet unoccupied VLs. If not enough VLs are available, the algorithm fails. If not all VLs are exhausted, DFSSSP additionally balances the number of paths using each VL, for more throughput.

By increasing the number of layers used, the total number of unique paths between node pairs increases, resulting in a higher number of virtual lanes (VLs) required to resolve deadlocks using the DFSSSP scheme. To maximize the number of supported layers, we propose a novel deadlock avoidance scheme based on the Duato's approach [41], that is agnostic to the number of layers and tailored for IB deployments that

rely exclusively on paths of length $<= 3$, such as those based on SF with our multipath routing method. The proposed algorithm ensures that the first, second, and third inter-switch hop of any path connecting two nodes use disjoint subsets of VLs. To achieve this, at least three VLs need to be available, and switches, for a given packet, must be able to identify their respective positions on the path using only the packet's SL, incoming and outgoing port.

To illustrate the algorithm's functionality, we consider each case individually. The first case, which involves paths of length 1 ($sw_1 - sw_2$), can be solved trivially since sw_1 can determine that it is the first hop along the path by checking whether the incoming packet port is connected to an endpoint. This information can then be encoded easily in the SL-to-VL table.

The strategy to address the second case, paths of length 2 ($sw_1 - sw_2 - sw_3$), is the same as the one for case three; therefore, we only present it once. In the third and final case, paths of length 3 ($sw_1 - sw_2 - sw_3 - sw_4$), we treat sw_1 as in case one but use a different approach to differentiate between sw_2 and sw_3. We establish a proper coloring of switches, using at most as many colors as there are available SLs. This color assignment is then mapped to SLs, ensuring each switch has a unique color and SL among its neighbours. By setting the SL of a packet routed along a path of length 2 or 3 to the SL assigned to the second switch (sw_2) along that path, it is guaranteed that the packet's assigned SL matches the SL of the second hop but not the SL of the third. Subsequently, if a switch is neither the first nor last hop on a path – a condition trivially determined through the incoming and outgoing packet ports – then the switch's position along the path can be ascertained by whether the incoming packet's assigned SL matches the SL assigned to the switch. Specifically, if the SLs match, then the given switch must be the second hop; if they don't, then it must be the third. Thus, we can differentiate the second hop from a potential third hop and select the appropriate subset of VLs at each hop accordingly.

If fewer than 3 VLs are available or no proper coloring using the available SLs can be established, the algorithm fails. Similar to the DFSSSP scheme, the disjoint VL subsets can be chosen to balance the number of paths crossing each VL.

5.3 Load Balancing

For load balancing, we rely on the respective protocol higher up in the stack to choose a layer out of the set of possible ones available for a given destination. In our case, this is the Open MPI [42] implementation of the Message Passing Interface (MPI) standard [43]. Open MPI serves as a communication library and directly interfaces with the IB networking API (Verbs). To optimize traffic flow, we utilize Open MPI's default load balancing technique, which distributes traffic evenly across the available paths using a round-robin selection process. More advanced, adaptive schemes can seamlessly be used by changing the selection policy.

For fault tolerance, we rely on IB's subnet manager. We stress that our routing can be seamlessly used with other transport schemes besides the ones used in the deployed cluster.

5.4 Path Diversity vs. Network Size

Increasing the number of different paths between each node pair requires more layers and thus also more addresses assigned to each node (i.e., a larger LMC value). However, using more addresses within one node decreases the maximum number of nodes that can be used in the network overall (because the address field size is fixed to 16 bits). We analyze this trade-off in Tab. 2. We assume the maximum SF network based on {36, 48, 64}-port switches, that guarantees full global bandwidth. The results illustrate that one can use 4 layers without having to make any compromises on the networks size, but anything beyond 4 layers would reduce the maximum network size. At this point, the constraining factor is no longer the switch radix, but the address space. In § 6 and § 7, we show that – fortunately – our routing scheme's performance is already quite substantial with just 4 layers and does not need more than 8 layers for high performance.

Table 2: Maximum number of switches and servers supported by a single-subnet, full global bandwidth, SF-based IB network, with **"#A"** $= 2^{LMC}$ many addresses per node.

	36-port switches				48-port switches				64-port switches			
#A	N_r	N	k'	p	N_r	N	k'	p	N_r	N	k'	p
1	512	6144	24	12	882	14112	31	16	1568	32928	42	21
2	512	6144	24	12	882	14112	31	16	1250	23750	37	19
4	512	6144	24	12	800	12000	30	15	800	12000	30	15
8	450	5400	23	12	450	5400	23	12	450	5400	23	12
16	288	2592	18	9	288	2592	18	9	288	2592	18	9
32	162	1134	13	7	162	1134	13	7	162	1134	13	7
64	98	588	11	6	98	588	11	6	98	588	11	6
128	72	360	9	5	72	360	9	5	72	360	9	5

6 THEORETICAL ANALYSIS

We conduct a theoretical analysis of the developed routing protocols using the deployed SF network as a case study. We focus on how well our routing uses the diversity of non-minimal paths, which is necessary for high performance [28].

Baselines and Parameters We analyze our layered routing that minimizes path overlap (§ 4) and compare it to a simple random layer construction (RUES, Random Uniform Edge Selection) and to the state-of-the-art FatPaths scheme [28].

We vary different parameters, including the fraction p of preserved links in a layer, which refers to the proportion of links from the network that are included in each layer for the RUES scheme (specifically, we consider $p = 40\%$, $p = 60\%$, and $p = 80\%$), and the number of layers used. We focus on the deployed SF with 50 switches, but the results generalize to larger sizes. Overall, we show that the proposed layered routing is superior to the state-of-the-art in crucial metrics: lengths, distribution, and diversity of used paths, and the achieved throughput.

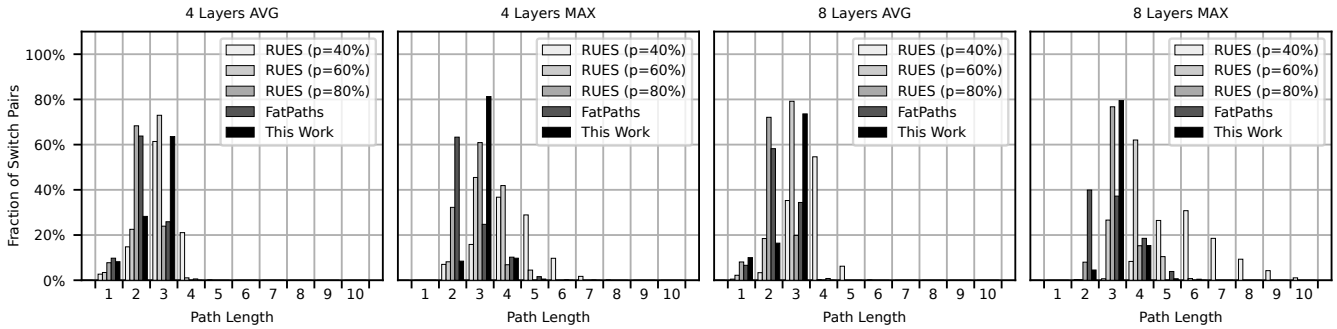


Figure 6: Histograms of average path lengths and maximum path lengths across all layers for each switch pair.

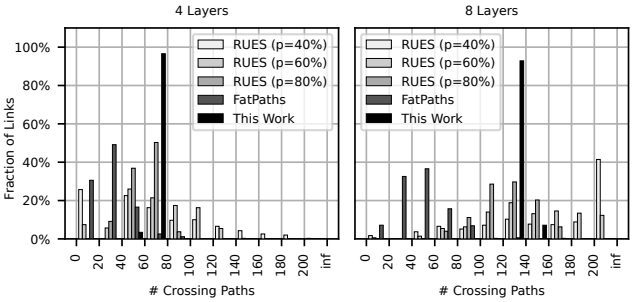


Figure 7: Histograms (bin size = 20) of counts of paths crossing each individual link.

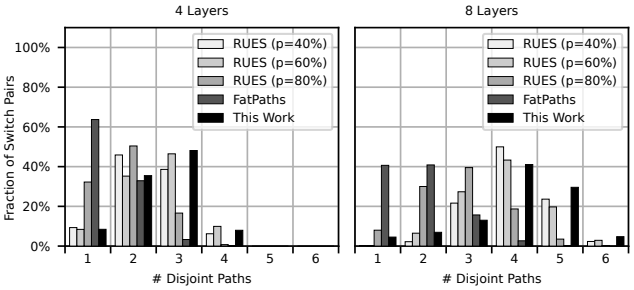


Figure 8: Histograms of counts of disjoint paths for different switch pairs.

6.1 Path Lengths

The first important metric for evaluating routing is *the length of paths constructed using the proposed routing schemes*. Specifically, when routing in SF, one wants to use the single available minimal path (with 1 or 2 hops, depending on picked switch pairs) and the "almost" minimal ones – with 3 hops – as indicated in the FatPaths study [28]. To analyze whether the considered routing ensures this, we compute the average and maximum lengths of the set of paths connecting each individual switch pair, as produced by the respective routing schemes. Fig. 6 shows the analysis results.

Our novel layered scheme outperforms all others, because it ensures that the highest fraction of switch pairs uses the "almost" minimal paths of length *at most* 3. The downside of RUES is that the more randomness is employed, the larger the maximum path length becomes. For a sampling factor $p = 80\%$, there is no switch pair with a path of length more than 4, whereas for $p = 40\%$ some switch pairs have paths of length greater than 8. This indicates large differences in path lengths in different layers for some switch pairs, even if the

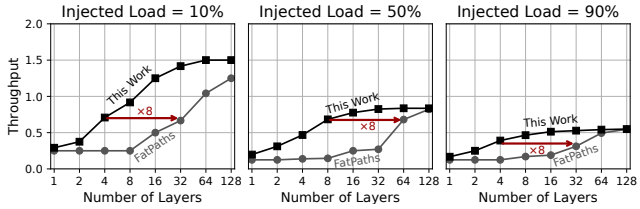


Figure 9: Maximum achievable throughput for the adversarial traffic pattern under three different injection loads (fraction of communicating endpoint pairs).

average path length is between 3 and 4. This can negatively impact load balancing efforts as it becomes more difficult to predict path latency. Then, in FatPaths, large fractions of switch pairs use paths of length 2, which means that these links may likely become congested.

Doubling the number of layers does not change the overall trends and it has mostly no effect on the average path length distributions. Only the maximum path lengths display a small shift to the right. This is because using more layers increases the probability of finding a longer path.

6.2 Path Distribution

We now count the total number of paths that cross each individual link, see Fig. 7. Our layered routing ensures a balanced scenario, i.e., close to equal utilization of each link. This corresponds to a "single bar", i.e., the "tighter" the distribution the better balanced the paths are.

Similarly to the analysis on path length, less randomness leads to better results, which is expected because as layers become less dense, the links that are present will be more utilized. Hence, any link that by chance is included in more than an average number of layers will have a higher number of crossing paths and vice versa. FatPaths performs similarly to RUES for a sampling factor of $p = 80\%$. The distributions for 8 layers are slightly shifted to the right compared to 4 layers, as they have twice as many paths.

6.3 Path Diversity

Two paths are disjoint if they do not share common links. In layered routing, we aim to maximize the number of such paths used by node pairs. Fig. 8 displays counts of disjoint paths between switch pairs. The FatPaths layer construction based on minimizing path overlap underperforms because

of its acyclic layers. Moreover, unlike in previous analyses, more randomness (and thus sparser layers) leads to better result for RUES. For a sampling factor of $p = 40\%$ and 8 layers, $\approx$97.5% of switch pairs have at least the 3 desired disjoint paths. This is the best performing algorithm out of the ones considered. However, this comes at the expense of disadvantageous path lengths and path distribution.

Our scheme does not need to make a similar trade-off because with 8 layers already around 88.5% of switch pairs have at least 3 disjoint paths, which we have verified to grow to almost 100% percent when scaling to the next higher configuration that uses 16 layers. At the same time, the lengths and path distributions over links are highly beneficial.

6.4 Maximum Achievable Throughput

We also analyze the maximum achievable throughout (MAT). MAT is defined as the maximum fraction of traffic demands from all endpoint pairs that can be accommodated simultaneously, while adhering to network and routing constraints. For example, a throughput of 1.5 denotes that the network can sustain 1.5 times the traffic demand of each communicating node pair simultaneously.

Here, we consider an adversarial traffic pattern, which maximizes stress on the interconnect by incorporating several large elephant flows between endpoints that are separated by more than one inter-switch hop, and combining these large flows with many small flows [44]. We use TopoBench [44], a throughput evaluation tool which relies on linear programming to compute MAT. The results are displayed in Fig. 9.

Our algorithm outperforms FatPaths for different traffic intensities and layer counts. This is most important for a small number of layers, which is key for routing on IB hardware as using many layers reduces the supported network sizes (cf. Tab. 2). Our layered routing experiences diminishing returns beyond 16 layers. This is expected, as almost 100% of endpoint pairs have at least 3 disjoint paths for 16 layers (one needs at least that many disjoint paths to ensure high performance with non-minimal routing). Before diminishing returns set in, FatPaths requires 8$\times$ as many layers to reach equivalent performance, making our design much more practical.

6.5 Insights & Takeaways - Theoretical Results

Our novel IB layered routing achieves superior performance in all considered path quality measures and especially in MAT. Almost around 60% of switch pairs have at least 3 disjoint non-minimal paths when using only 4 layers, which grows to 88.5% with 8 layers. Furthermore, we achieve the most balanced distribution of paths over the links in the network. FatPaths performs similarly in terms of average and maximum path lengths, but underperforms in the available number of disjoint paths per switch pair. For RUES, a sampling factor of $p = 60\%$ achieved the most balanced results across all metrics, but RUES performs much worse in comparison to FatPaths and our work overall.

7 EVALUATION

We now illustrate the feasibility of our SF installation by evaluating a broad set of applications from numerous domains against a comparable FT installation.

7.1 2-Level Non-Blocking Fat-Tree

FT topologies have historically been the usual choice for large-scale computing systems, largely due to their predictable behavior and full-bandwidth capabilities, when configured in a non-blocking manner. However, their high cost often leads to oversubscribed deployments at the tree's lowest level, reducing construction costs at the expense of bisection bandwidth.

To ensure a fair performance comparison with our SF installation, we construct a 2-level non-blocking FT, reusing the same hardware. The FT and SF both share the same network diameter and full-bandwidth capabilities. Our FT configuration employs 6 core and 12 leaf switches, compatible with our 36-port switches. Each leaf switch connects to each core switch through 3 links, and the remaining ports link to evenly distributed endpoints. This configuration supports up to 216 endpoints, making the FT marginally under-subscribed and thus strengthening the fairness of our comparison.

7.2 Workloads & Configurations

We utilize a significant subset of the benchmarks included in the TSUBAME2 HyperX (t2hx) benchmark suite [10] and enhance them with a custom implementation of MPI_Alltoall[5], as well as three DNN proxies introduced by Hoefler et al. [57]. The configuration of each benchmark is provided in Tab. 3. Our analysis includes three classes of benchmarks:

Microbenchmarks We evaluate the system's bandwidth using Intel MPI Benchmarks' (IMB) measurements of the Allreduce and Bcast collectives [45], and a custom alltoall. We also assess the effective bisection bandwidth (ebb) of the system using Netgauge's eBB benchmark [46].

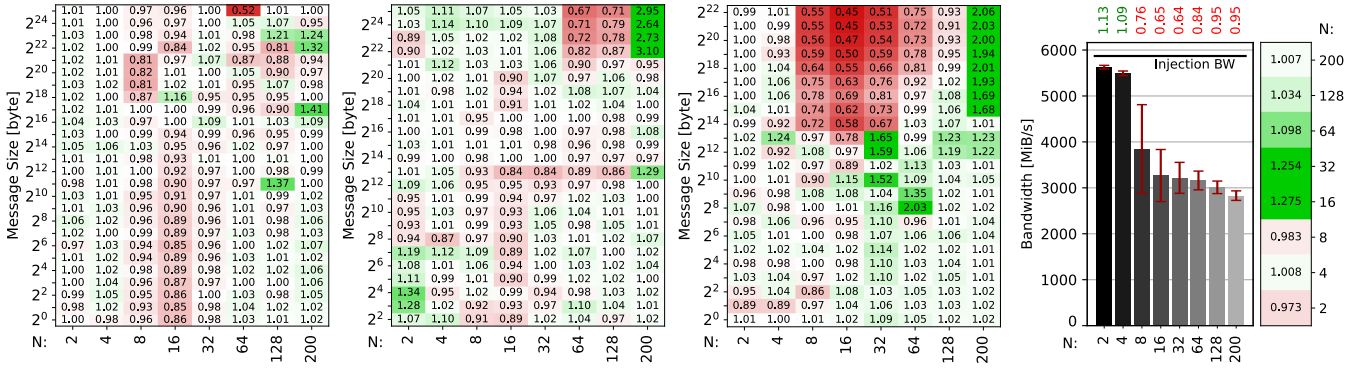
Scientific Application & HPC Benchmarks We evaluate a wide range of benchmarks, covering various scientific applications, all of which are listed in Tab. 3 and taken directly from the t2hx benchmark suite. We also analyze the performance of the High Performance Linpack (HPL) [55] benchmark and of the breadth-first search (BFS) [60] in the Graph 500 Benchmark [53]. Additionally, we extend the BFS performance analysis by changing the average degree of the vertices (edgefactor), while scaling the number of vertices linearly with the number of participating compute nodes. Specifically, we consider edgefactors 16, 128 and 1024.

DNN Proxies The DNN proxies evaluated on SF include ResNet152 [56] (pure data parallelism), CosmoFlow [58] (data and operator parallelism) and GPT-3 [59] (data, operator, and pipeline parallelism), as outlined in Tab. 3. For GPT-3, each pipeline stage processes one DNN-layer.

[5]Details on the performance improvements for the custom alltoall collective, over the default, can be found in the appendix (Sec C.1).

Table 3: Workload Configurations.

Workload	Configuration	# Nodes (N)	Scaling	Metric
Custom Alltoall	Message Sizes: 1B → 4MiB	2,4,8,16,32,64,128,200	Weak	Bandwidth [MiB/s]
IMB Bcast/Allreduce [45]	Message Sizes: 1B → 32MiB	2,4,8,16,32,64,128,200	Weak	Bandwidth [MiB/s]
eBB [46]	Message Size: 128MiB	2,4,8,16,32,64,128,200	Strong	Bandwidth [MiB/s]
CoMD [47]	100^3 Atoms per Process	25,50,100,200	Weak	Time [s]
FFVC [48]	128^3 Cuboid per Process for ≤ 64 processes, else 64^3	25,50,100,200	Weak	Time [s]
mVMC [49]	Unmodified *job_middle* weak-scaling test	25,50,100,200	Weak	Time [s]
MILC [50, 51]	*benchmark_n8* Input	25,50,100,200	Weak	Time [s]
NTChem [52]	*taxol* Model	25,50,100,200	Strong	Time [s]
BFS_{16} [53, 54]	# Vertices: 2^{23}, 2^{24}, 2^{25}, 2^{26} Avg. Degree: 16	25,50,100,200	Weak	Giga-Traversed Edges per Second [GTEPS]
BFS_{128} [53, 54]	# Vertices: 2^{23}, 2^{24}, 2^{25}, 2^{26} Avg. Degree: 128	25,50,100,200	Weak	Giga-Traversed Edges per Second [GTEPS]
BFS_{1024} [53, 54]	# Vertices: 2^{23}, 2^{24}, 2^{25}, 2^{26} Avg. Degree: 1024	25,50,100,200	Weak	Giga-Traversed Edges per Second [GTEPS]
HPL [55]	Matrix $A \approx 1$ GiB ,1 GiB ,1 GiB and 0.25 GiB pre Process	25,50,100,200	Weak	Giga-Floating point OP/s [GFLOPS]
ResNet152 [56, 57]	Pure Data Parallelism	40,80,120,160,200	Weak	Iteration Time [s]
Cosmoflow [57, 58]	Model Shards: 4 Data Shards: $\frac{\#\text{Nodes}}{4}$	40,80,120,160,200	Weak	Iteration Time [s]
GPT-3 [59, 57]	Pipeline Stages (layers): 10 Model Shards: 4 Data Shards: $\frac{\#\text{Nodes}}{40}$	40,80,120,160,200	Weak	Iteration Time [s]



(a) MPI Bcast - SF L vs. FT (b) MPI Allreduce - SF L vs. FT (c) Custom Alltoall - SF L vs. FT (d) eBB - SF L vs. FT

Figure 10: Relative performance difference of SF (linear placement strategy) over FT for various Microbenchmarks; eBB performance of SF L in comparison to maximum bandwidth and FT performance (higher is better), including routing improvement of this work over DFSSSP (heatmap).

7.3 Execution Environment

To ensure consistency and reproducibility, all benchmarks were compiled using GCC v4.8.5 and executed using Open-MPI v1.10.7. We use one MPI rank per node and assign one OpenMP thread per physical core on Socket 1 of the dual-socket system (pinning on Socket 2 introduces non-negligible slowdowns due to inter-socket communication).

We investigate two MPI rank placement strategies: linear and random. The linear strategy places rank j on node j, a commonly used approach that enhances latency and traffic locality, especially for FTs [61, 62]. This strategy also models a system with minimal fragmentation. In contrast, the random strategy represents systems with significant fragmentation. It randomizes rank placement to potentially reduce network bottlenecks on SF, albeit at the cost of increased latency. For FT, the linear placement significantly outperformed its random counterpart in all microbenchmarks and exhibited comparable performance in the remaining tests. Consequently, we report SF performance relative to the FT's linear placement only.

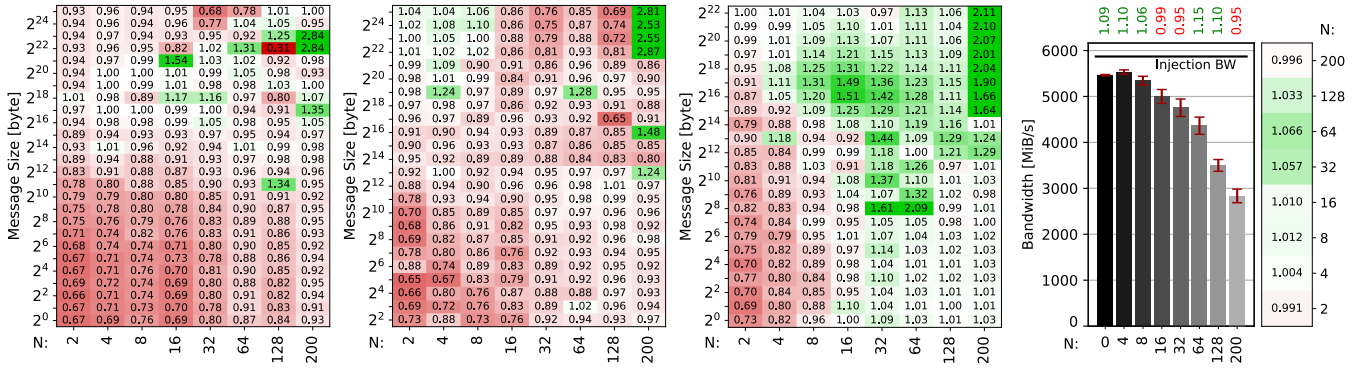
Each benchmark configuration is executed five times; microbenchmarks are executed for at least 100 iterations. We assess all SF benchmarks using our new multipath routing algorithm based on both minimal and almost minimal paths, as well as the defacto standard multipath routing algorithm in IB (DFSSSP), that leverages minimal paths only [63]. We instantiate each routing algorithm once with 1, 2, 4, and 8 layers, respectively, but only report the results of the best-performing variant for each benchmark configuration. For all FT benchmarks we choose the commonly used ftree routing [64]. Mean and standard deviation of the results are reported, with the latter indicated using red error bars for all bar plots. Relative performance differences of SF over FT are annotated above each bar. Any significant performance gains or losses of our novel routing algorithm in comparison to DFSSSP for any benchmark are either explicitly stated in the text or visualized using heatmaps.

In the main text, we present comprehensive results for SF using the linear placement strategy, and include only microbenchmark results for the random placement strategy due to space considerations. Detailed results of the random strategy for other benchmarks, which largely mirror those obtained with the linear strategy, are in Appendix C.

7.4 Microbenchmarks

Fig. 10a–10c illustrate the relative performance differences of SF with linear placement over FT and Fig. 11a–11c of SF with random placement over FT for MPI collectives bcast, allreduce, and custom alltoall.

Generally, SF's performance using the linear placement strategy closely matches that of the FT, with FT only displaying minor advantages in bcast and allreduce for 8 and



(a) MPI Bcast - SF R vs. FT (b) MPI Allreduce - SF R vs. FT (c) Custom Alltoall - SF R vs. FT (d) eBB - SF R vs. FT

Figure 11: Relative performance difference of SF (random placement strategy) over FT for various Microbenchmarks; eBB performance of SF R in comparison to maximum bandwidth and FT performance (higher is better), including routing improvement of this work over DFSSSP (heatmap).

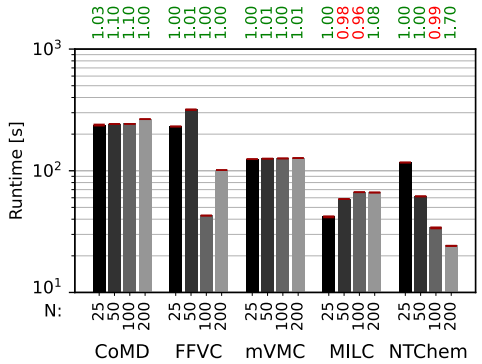


Figure 12: Runtime of scientific workloads (lower is better) - SF L vs. FT

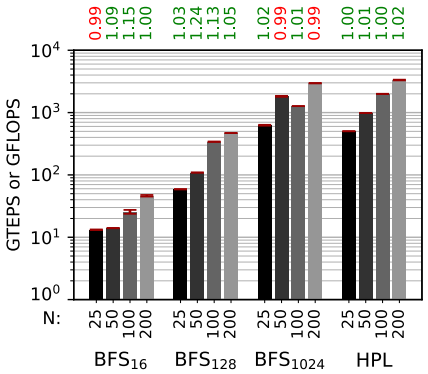


Figure 13: Performance of HPC benchmarks (higher is better) - SF L vs. FT

16 node configurations at smaller, latency-sensitive message sizes. This marginal edge of FT in specific configurations is due to its architecture, wherein leaf switches connect to at least 16 nodes, facilitating localized communication with zero inter-switch hops, thus minimizing latency. While SF, under linear placement, enjoys the benefits of zero inter-switch hops mostly for configurations of up to 4 nodes – owing to its design of connecting 4 nodes per switch – random placement generally does not benefit from this localized communication advantage. As a result, SF experiences marginally lower performance in comparison to FT for these latency-sensitive scenarios with the random placement strategy.

In contrast, for the communication-intensive alltoall collective, SF's performance closely mirrors, or even slightly surpasses, that of the FT for small message sizes when employing the linear placement strategy (cf. Fig. 10c). However, in 8, 16, and 32 node configurations, particularly with bandwidth-critical message sizes, SF lags due to congestion caused by all inter-switch communication occurring between 2, 4, or 8 switches, respectively. This leads to traffic bottlenecks on the often single shortest path between these switches. While our new routing scheme, as discussed in § 6, theoretically mitigates this congestion, the absence of adaptive load balancing limits practical improvements to at most 7% over DFSSSP.

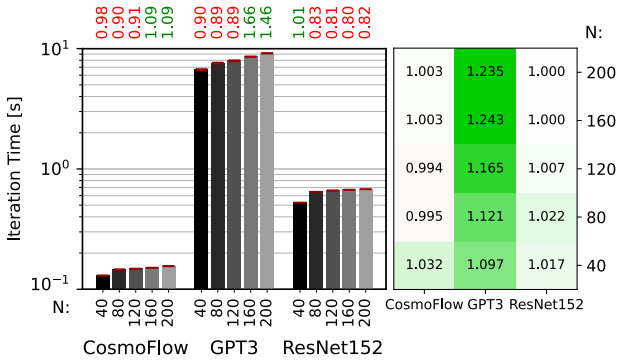


Figure 14: Iteration time of DNN proxy workloads (lower is better) SF L vs. FT and routing improvement of this work over DFSSSP (heatmap) for SF L.

Switching to the random placement strategy markedly improves SF's performance for the alltoall collective, as shown in Fig. 11c. This strategy not only overcomes the noted bottlenecks but also enables SF to significantly outperform FT. This improvement results from the random placement strategy's enhanced traffic distribution across the network, showcasing the trade-off between increased latency for smaller message sizes and superior traffic balancing within the SF topology. These findings imply that the integration of adaptive load balancing with our routing scheme could effectively address the congestion issues identified with linear placement, underscoring the potential of our routing scheme to optimize network performance for demanding communication patterns.

Lastly, in Fig. 10d and Fig. 11d, we present the ebb across various node counts for the linear and random placement strategy, respectively. At maximum node count we achieve

approximately half of the injection bandwidth, equating to 75% of the theoretical bisection bandwidth optimum [1], with both strategies. Though the FT matches SF's full-system ebb, it outperforms SF with linear placement for the 8, 16, and 32 nodes configurations. This discrepancy mirrors the observations for the alltoall collective and is similarly overcome with the random placement strategy (cf. Fig. 11d).

In the right section of both Fig. 10d and Fig. 11d, heatmaps display the performance gains of our new routing scheme over DFSSSP for the eBB benchmark. Notably, for the linear placement strategy, improvements of up to 28% are observed for the earlier described node configurations, which are especially prone to congestion. Under the random placement strategy, the level of improvement is less significant, with only up to 7%, suggesting that this strategy's primary advantage lies in its ability to distribute traffic more evenly, even in the absence of adaptive load balancing.

7.5 Scientific Workloads & HPC Benchmarks

In Fig. 12, we present the runtime and relative performance of the solver/kernel for each of the scientific workloads on SF, using the linear placement strategy. The scaling behavior of each workload, based on their configurations detailed in Tab. 3, is evident. Notably, the drop in runtime for FFVC when scaling from 50 to 100 nodes is due to the decrease in the workload's problem size when running on > 64 nodes. Utilizing almost minimal paths in combination with minimal paths does not generate any significant speedup for these workloads over pure minimal routing (DFSSSP), and generally results in only small runtime variances of $< 1\%$. This is due to the communication time only constituting a small fraction of the overall runtime for these scientific workloads [10, 65].

Fig. 13 shows the performance of the HPC benchmarks, which display similar weak-scaling behavior as the scientific workloads. HPL exhibits almost linear scaling performance when increasing the number of nodes from 25 to 50 or 100 nodes, indicating that the overhead introduced by the increased amount of communication is negligible. Consistent with these results, introducing almost minimal paths to the routing impacts performance by less than 1% for the HPL benchmark. The only exception is the 200 node setting, where the decrease in the problem size (per node) is likely the main cause for the deviation from the linear scaling observed.

In the case of the Graph 500 - BFS benchmark, we experienced high variance with the default implementation. To mitigate this, we fixed the seed for the graph generation and used the same source vertex for each BFS run. The BFS scaling results show more fluctuations in comparison to the HPL results, particularly for the sparser variant. This is accompanied by greater variability in speedup through almost minimal paths, which ranged from -5% to +1%. It is not clear whether this can be attributed purely to network communication or to other factors such as caching effects and system noise.

Overall, our experiments show SF competes effectively with FT in terms of performance, while being very effective for scaling both scientific workloads and HPC benchmarks, even when limited to minimal paths.

7.6 Deep Learning Workloads

The left part of Fig. 14 shows the runtime and relative performance of the DNN proxies when linearly increasing the number of nodes from 40 to 200. ResNet152 with pure data parallelism only requires allreduce for gradient aggregation. CosmoFlow with a hybrid of data and operator parallelism requires allgather, reduce-scatter, allreduce, and point-to-point communications. GPT-3 with a hybrid of data, operator, and pipeline parallelism requires allreduce and point-to-point communications. As we increase the data shards proportionally to the number of nodes, the scalability is mainly determined by allreduce across the data dimension.

We find that CosmoFlow's runtime on SF is comparable to that on FT. In contrast, GPT-3 notably performs better on SF for configurations with 160 and 200 nodes, while ResNet152 begins to lag as the node count increases. Although both GPT-3 and ResNet152 predominantly rely on allreduces at higher node counts, their diverging performance trends can be attributed to differences in message sizes; GPT-3 handles significantly larger messages than ResNet152. Expectedly, the performance trend of GPT-3 matches the trend of MPI Allreduce for the high node count configurations (cf. Fig. 10b).

The right part of Fig. 14 shows that our work generally outperforms DFSSSP for GPT-3, with up to 24% improvements.

7.7 Insights & Takeaways - Empirical Results

When analyzing communication-intensive workloads on configurations with 8, 16, or 32 nodes, we identified some congestion challenges. These challenges stemmed from the non-adaptive nature of the path selection. However, by employing a random placement strategy, these issues were effectively counteracted. Our findings subsequently indicate that SF consistently achieves performance on par with, or even surpassing, the well-established FT topology, particularly under conditions of full-system utilization. Additionally, SF displays effective scaling capabilities across a diverse range of workloads. In comparison to the established DFSSSP, our novel routing approach exhibited promising performance, registering improvements of over 20%.

7.8 Scalability & Cost Analysis

FT topologies are the preferred choice for large-scale HPC deployments due to their adaptability, adoptable bisection bandwidth, established routing, and isolation advantages. These properties often benefit application performance consistency [66, 67, 68]. However, their low-diameter configurations do not scale as well as contemporary topologies [69].

We compare the scalability and deployment cost of 2-level FTs (FT2), 3-level FTs (FT3), 2-D HyperX (HX2) [10, 70], and SF. Our evaluation, summarized in Tab. 4, includes both

Table 4: Maximal scalability and costs of SF deployments compared to non-blocking FT2, FT2 oversubscribed by 3 (FT2-B), FT3 and 2-D HyperX (HX2) under given port constraints. For the fixed size cluster we use 64-port switches for the FT2 and FT-B, 40-port switch for HX2, and 36-port for SF and FT3.

	36-port switches					40-port switches					64-port switches					2048 nodes clusters				
	FT2	FT2-B	FT3	HX2	SF	FT2	FT2-B	FT3	HX2	SF	FT2	FT2-B	FT3	HX2	SF	FT2	FT2-B	FT3	HX2	SF
Endpoints	648	972	11664	2028	6144	800	1200	16000	2744	7514	2048	3072	65536	10648	32928	2048	2048	2048	2197	2178
Switches	54	45	1620	169	512	60	50	2000	196	578	96	80	5120	484	1568	96	59	303	169	242
Links	648	324	23328	2028	6144	800	400	32000	2548	7225	2048	1024	131072	10164	32928	2048	344	4320	2028	2057
Costs [M$]	1.5	1.1	45	4.5	13.8	2.4	1.7	84.2	7.8	22.4	9	7.2	491	45.5	146	7.5	2.7	8.3	6.4	5.8
Cost/Endp [k$]	2.2	1.2	3.8	2.2	2.2	3	1.5	5.2	2.8	2.9	4.4	2.3	7.5	4.3	4.4	3.6	1.3	4	3.1	2.8

the non-blocking FT2 variant and its 3:1 oversubscribed version (FT2-B). The pricing details are in Appendix D.

Scalability We show that SF networks offer a distinct advantage in scalability by evaluating maximum network size for a HW setup with 36, 40, and 64-port switches. SF can accommodate approximately 10, 6, and 3 times more endpoints than FT2, FT2-B, and HX2 respectively, while maintaining a lower or comparable cost-to-endpoint ratio and the same network diameter of 2. FT3 can accommodate more endpoints than SF, however, this comes at a significantly larger (around 1.75x) cost-to-endpoint ratio and increased network diameter which has an impact on a performance of latency critical applications. This makes SF a compelling choice for large-scale diameter-2 deployments.

Cost When the number of endpoints is predetermined, SF's requirement for fewer port switches can reduce overall deployment costs, while keeping comparable benchmark performance to FT2 as shown in § 7. Tab. 4 further shows an example of fixing a cluster requirement to 2048 endpoints. Realising such a cluster using SF in comparison to FT2, HX2, and FT3 results in absolute cost saving of $1.7M, $0.6M, and $2.5M respectively. While using FT2-B might be cheaper in this scenario, it does not provide the full bandwidth property as SF, FT2, HX2, and FT3.

8 RELATED WORK

Our work touches on different areas. We now outline related works, excluding the ones covered in past sections.

Network Topologies Several recent networks build upon SF. This includes Megafly [71], Bundlefly [72], Galaxyfly [73], and Xpander [4]. Yet, they do not provide diameter-2 and thus none of them are competitive with SF in latency, cost, or power consumption, as observed by recent results [28]. Although PolarFly has shown promising results in recent studies, its advantages over SF can be attributed to the diligent design of routing protocols that leverage its structure to guarantee optimal routing decisions [74, 75]. Some recent designs based on similar principles target on-chip networks only [76, 77].

Physical Interconnect Installations The majority of works on interconnects use simulations for evaluation [1, 2, 3, 4, 71, 72, 73, 78, 79, 80, 81]. However, some topologies have been evaluated with real installations. This includes – for example – HyperX [10] and Dragonfly [22]. Here, we offer the first real evaluation of Slim Fly.

Congestion Control & Load Balancing In general, we do not focus on transport protocols (flow, congestion). Here, we rely on mechanisms from the FatPaths [28] architecture. In layered routing, traffic is balanced across layers. We use simple randomized and round-robin schemes, which results in high performance. Other schemes could also be incorporated, including load balancing based on flows [29, 82, 83, 84, 85, 86, 87, 88, 89], flowcells [90], flowlets [91, 92, 93, 94, 95], and single packets [96, 97, 98, 99, 100, 101].

9 CONCLUSION

Slim Fly (SF) is the first network topology that lowered cost and improved performance by reducing the network diameter to two, promising significant improvement over established interconnects. However, it has not yet been tested in practice. We address this by deploying the first at-scale SF installation and establishing and implementing open-source routines for cabling and physical layout, to guide future deployments and effectively verify cabling. This can foster the adoption of SFs in broad industry and facilitate practical deployments of other low-diameter topologies, including the most recent ones, such as PolarFly or Bundlefly.

We further introduce a novel high-performance routing scheme that improves upon state of the art, achieving up to 24% speedup for the evaluated deep neural network (DNN) workloads over the standard IB multipath routing algorithm (DFSSSP) through non-minimal paths.

We use the first practical, real-world deployment of SF to demonstrate the topology's ability to scalably process a wide selection of modern workloads such as distributed DNN training, graph analytics, or linear algebra kernels. It consistently matches or surpasses the performance of a comparable non-blocking Fat Tree (FT) deployment for a wide selection of workloads, for example, achieving a 66% speedup for distributed deep neural network training. Importantly, SF simultaneously delivers superior scalability. For example, it enables connecting between 3× and 10× the number of servers compared to other diameter-2 topologies like 2-level FT and 2-D HyperX, while maintaining both a comparable cost-to-endpoint ratio and full bandwidth. For larger installation sizes, SF's scalability translates to significant cost advantages, for example, 50% over full bandwidth non-blocking 3-level Fat Tree configurations [1]. Overall, this effort will spearhead future research into more powerful network topologies.

Acknowledgments

We thank Colin McMurtrie, Mark Klein, Angelo Mangili, and the whole CSCS team granting access to the Ault and Daint machines, and for their excellent technical support with the Slim Fly cluster infrastructure. We thank Timo Schneider for help with infrastructure at SPCL. This project received funding from the European Research Council (Project PSAP, No. 101002047), and the European High-Performance Computing Joint Undertaking (JU) under grant agreement No. 955513 (MAELSTROM). This project received funding from the European Union's HE research and innovation programme under the grant agreement No. 101070141 (Project GLACIATION). This project was supported by JSPS KAKENHI Grant Number JP19H04119.

References

[1] Maciej Besta and Torsten Hoefler. 2014. Slim fly: a cost effective low-diameter network topology. In *ACM/IEEE Supercomputing*. New Orleans, Louisana, 348–359. ISBN: 9781479955008. DOI: 10.1109/SC.2014.34.

[2] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. 2008. Technology-Driven, Highly-Scalable Dragonfly Topology. In *Proc. of Intl. Symp. Comp. Arch.* (ISCA '08). IEEE Computer Society, Washington, DC, USA, 77–88. ISBN: 978-0-7695-3174-8. DOI: 10.1109/ISCA.2008.19.

[3] John Kim, William J. Dally, and Dennis Abts. 2007. Flattened Butterfly: A Cost-efficient Topology for High-radix Networks. In *Proc. of Intl. Symp. Comp. Arch.* (ISCA '07). ACM, San Diego, California, USA, 126–137. ISBN: 978-1-59593-706-3. DOI: 10.1145/1250662.1250679.

[4] Asaf Valadarsky, Michael Dinitz, and Michael Schapira. 2015. Xpander: unveiling the secrets of high-performance datacenters. In *ACM HotNets*.

[5] The InfiniBand Trade Association. 2004. *Infiniband Architecture Spec. Vol. 1, Rel. 1.2*. InfiniBand Trade Association.

[6] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. 2013. Enabling highly-scalable remote memory access programming with mpi-3 one sided. In *ACM/IEEE Supercomputing*, 1–12.

[7] Salvatore Di Girolamo et al. 2019. Network-accelerated non-contiguous memory transfers. In *ACM/IEEE Supercomputing*.

[8] Jack J Dongarra, Hans W Meuer, Erich Strohmaier, et al. 1997. Top500 supercomputer sites. *Supercomputer*, 13, 89–111.

[9] Maciej Besta, Jens Domke, Marcel Schneider, Marek Konieczny, Salvatore Di Girolamo, Timo Schneider, Ankit Singla, and Torsten Hoefler. 2020. High-performance routing with multipathing and path diversity in ethernet and hpc networks. *IEEE TPDS*.

[10] Jens Domke et al. 2019. HyperX Topology: First At-Scale Implementation and Comparison to the Fat-Tree. In *ACM/IEEE Supercomputing*.

[11] Maciej Besta et al. 2017. To push or to pull: on reducing communication and synchronization in graph computations. In *ACM HPDC*. ACM. Washington, DC, USA, 93–104. ISBN: 9781450346993. DOI: 10.1145/3078597.3078616.

[12] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2023. Demystifying graph databases: analysis and taxonomy of data organization, system designs, and graph queries. *ACM CSUR*.

[13] Maciej Besta et al. 2021. Sisa: set-centric instruction set architecture for graph mining on processing-in-memory systems. In *ACM MICRO*.

[14] Maciej Besta et al. 2021. Graphminesuite: enabling high-performance and programmable graph mining algorithms with set algebra. *VLDB*.

[15] Maciej Besta et al. 2020. High-performance parallel graph coloring with strong guarantees on work, depth, and quality. In *ACM/IEEE Supercomputing*.

[16] Maciej Besta et al. 2022. Practice of streaming processing of dynamic graphs: concepts, models, and systems. *IEEE TPDS*.

[17] Tal Ben-Nun, Maciej Besta, Simon Huber, Alexandros Nikolaos Ziogas, Daniel Peter, and Torsten Hoefler. 2019. A modular benchmarking infrastructure for high-performance and reproducible deep learning. In *IEEE IPDPS*. IEEE, 66–77.

[18] Maciej Besta and Torsten Hoefler. 2023. Parallel and distributed graph neural networks: an in-depth concurrency analysis. *IEEE TPAMI*.

[19] Maciej Besta et al. 2023. The graph database interface: scaling online transactional and analytical graph workloads to hundreds of thousands of cores. In *ACM/IEEE Supercomputing*.

[20] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review* number 4. Vol. 38. ACM, 63–74.

[21] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. 2009. Portland: a scalable fault-tolerant layer 2 data center network fabric. *ACM SIGCOMM CCR*, 39, 4, 39–50.

[22] Greg Faanes et al. 2012. Cray Cascade: A scalable HPC system based on a Dragonfly network. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)* Article 103. IEEE Computer Society, Salt Lake City, Utah, 103:1–103:9. ISBN: 978-1-4673-0804-5. http://dl.acm.org/citation.cfm?id=2388996.2389136.

[23] Daniele De Sensi, Salvatore Di Girolamo, Kim H. McMahon, Duncan Roweth, and Torsten Hoefler. 2020. An in-depth analysis of the slingshot interconnect. *CoRR*, abs/2008.08886. https://arxiv.org/abs/2008.08886 arXiv: 2008.08886.

[24] Brendan D McKay, Mirka Miller, and Jozef Siran. 1998. A note on large graphs of diameter two and given maximum degree. *J. Comb. Theory Ser. B*, 74, 1, (Sept. 1998), 110–118. DOI: 10.1006/jctb.1998.1828.

[25] Alan J Hoffman and Robert R Singleton. 1960. On moore graphs with diameters 2 and 3. *IBM Journal of Research and Development*, 4, 5, 497–504.

[26] Paul R Hafner. 2003. The hoffman-singleton graph and its automorphisms. *Journal of Algebraic Combinatorics*, 18, 1, 7–12.

[27] Mirka Miller and Jozef vSirávn. 2005. Moore graphs and beyond: a survey of the degree/diameter problem. *Electronic Journal of Combinatorics, Dynamic survey*, 14, (Dec. 2005), 1–61.

[28] Maciej Besta, Marcel Schneider, Karolina Cynk, Marek Konieczny, Erik Henriksson, Salvatore Di Girolamo, Ankit Singla, and Torsten Hoefler. 2020. Fatpaths: routing in supercomputers and data centers when shortest paths fall short. *ACM/IEEE Supercomputing*.

[29] C Hopps. 2000. RFC 2992: Analysis of an Equal-Cost Multi-Path Algorithm. (2000).

[30] Jayaram Mudigonda, Praveen Yalagandula, Mohammad Al-Fares, and Jeffrey C Mogul. 2010. SPAIN: COTS Data-Center Ethernet for Multipathing over Arbitrary Topologies. In *NSDI*, 265–280.

[31] Brent Stephens, Alan Cox, Wes Felter, Colin Dixon, and John Carter. 2012. PAST: Scalable Ethernet for data centers. In *ACM CoNEXT*.

[32] Infiniband Trade Association and others. 2014. Rocev2. (2014).

[33] Hermann Schweizer et al. 2015. Evaluating the cost of atomic operations on modern architectures. In *ACM/IEEE PACT*. IEEE, 445–456.

[34] William Dally and Brian Towles. 2003. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN: 0122007514.

[35] Jens Domke, Torsten Hoefler, and Satoshi Matsuoka. 2016. Routing on the Dependency Graph: A New Approach to Deadlock-Free High-Performance Routing. In *Proceedings of the 25th Symposium on High-Performance Parallel and Distributed Computing (HPDC'16)*. (June 2016).

[36] Jens Domke, Torsten Hoefler, and Wolfgang E. Nagel. 2011. Deadlock-Free Oblivious Routing for Arbitrary Topologies. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, (May 2011), 613–624.

[37] Timo Schneider, Otto Bibartiu, and Torsten Hoefler. 2016. Ensuring deadlock-freedom in low-diameter infiniband networks. In *Proceedings of the IEEE 24th Annual Symposium on High-Performance Interconnects (HOTI)* (Santa Clara, CA, USA).

[38] Keun Sup Shim, Myong Hyon Cho, Michel Kinsy, Tina Wen, Mieszko Lis, G. Edward Suh, and Srinivas Devadas. 2009. Static virtual channel allocation in oblivious routing. In *2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*. IEEE. DOI: 10.1109/nocs.2009.5071443.

[39] Tor Skeie, Olav Lysne, Jose Flich, Pedro Lopez, Antonio Robles, and Jose Duato. 2004. Lash-tor: a generic transition-oriented routing algorithm. In *Proceedings. Tenth International Conference on Parallel and Distributed Systems, 2004. ICPADS 2004.* IEEE, 595–604.

[40] Tor Skeie, Olav Lysne, and Ingebjørg Theiss. 2002. Layered shortest path (lash) routing in irregular system area networks. In *ipdps*. Citeseer, 0162.

[41] Jose Duato, Sudhakar Yalamanchili, and Ni Lionel. 2002. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN: 1558608524.

[42] Edgar Gabriel et al. 2004. Open mpi: goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 97–104.

[43] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. 1994. The mpi message passing interface standard. In *Programming environments for massively parallel distributed systems*. Springer, 213–218.

[44] Sangeetha Abdu Jyothi, Ankit Singla, P Brighten Godfrey, and Alexandra Kolla. 2016. Measuring and understanding throughput of network topologies. In *ACM/IEEE Supercomputing*.

[45] Intel Corporation. 2018. Intel®mpi benchmarks user guide. https://software.intel.com/en-us/imb-user-guide. (2018).

[46] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2008. Multistage switches are not crossbars: effects of static routing in high-performance networks. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan*, 116–125. DOI: 10.1109/CLUSTR.2008.4663762.

[47] ExMatEx. 2012. Comd proxy application. http://www.exmatex.org/comd.html. (2012).

[48] the University of Tokyo Institute of Industrial Science. 2014. Ffvc-mini. https://github.com/fiber-miniapp/ffvc-mini. (2014).

[49] RIKEN Advanced Institute for Computational Science. 2016. Mvmc-mini. https://github.com/fiber-miniapp/mVMC-mini. (2016).

[50] G. Bauer, S. Gottlieb, and T. Hoefler. 2012. Performance Modeling and Comparative Analysis of the MILC Lattice QCD Application su3 rmd. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society, Ottawa, Canada, (May 2012), 652–659. ISBN: 978-0-7695-4691-9.

[51] Steven Gottlieb, W. Liu, William D Toussaint, R. L. Renken, and R. L. Sugar. 1987. Hybrid-molecular-dynamics algorithms for the numerical simulation of quantum chromodynamics. English (US). *Physical review D: Particles and fields*, 35, 8, 2531–2542. DOI: 10.1103/PhysRevD.35.2531.

[52] RIKEN Advanced Institute for Computational Science. 2016. Ntchem-mini. https://github.com/fiber-miniapp/ntchem-mini. (2016).

[53] James A. Ang, Brian W. Barrett, Kyle B. Wheeler, and Richard C. Murphy. 2010. Introducing the graph 500. In.

[54] Koji Ueno, Toyotaro Suzumura, Naoya Maruyama, Katsuki Fujisawa, and Satoshi Matsuoka. 2016. Extreme scale breadth-first search on supercomputers. In *2016 IEEE International Conference on Big Data (Big Data)*, 1040–1047. DOI: 10.1109/BigData.2016.7840705.

[55] Antoine Petitet, R. Whaley, Jack Dongarra, and A. Cleary. 2008. Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers, (Jan. 2008).

[56] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. (2015). arXiv: 1512.03385 [cs.CV].

[57] Torsten Hoefler, Tommaso Bonato, Daniele De Sensi, Salvatore Di Girolamo, Shigang Li, Marco Heddes, Jon Belk, Deepak Goel, and Steve Scott Miguel Castro. 2022. HammingMesh: A Network Topology for Large-Scale Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'22)*. (Nov. 2022).

[58] Amrita Mathuriya et al. 2018. Cosmoflow: using deep learning to learn the universe at scale. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 819–829.

[59] Tom Brown et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33, 1877–1901.

[60] Maciej Besta, Florian Marending, Edgar Solomonik, and Torsten Hoefler. 2017. Slimsell: a vectorizable graph representation for breadth-first search. In *IEEE IPDPS*. IEEE, 32–41.

[61] George Michelogiannakis, Khaled Z Ibrahim, John Shalf, Jeremiah J Wilke, Samuel Knight, and Joseph P Kenny. 2017. Aphid: hierarchical task placement to enable a tapered fat tree topology for lower power and cost in hpc networks. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 228–237.

[62] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. Slurm: simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*. Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, (Eds.), 44–60.

[63] J. Domke, T. Hoefler, and W. Nagel. 2011. Deadlock-Free Oblivious Routing for Arbitrary Topologies. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, Anchorage, AL, USA, (May 2011), 613–624. ISBN: 0-7695-4385-7.

[64] Joan Jacobs. 2010. D-mod-k routing providing non-blocking traffic for shift permutations on real life fat trees. In https://api.semanticscholar.org/CorpusID:1831393.

[65] Benjamin Klenk and Holger Fröning. 2017. An overview of mpi characteristics of exascale proxy applications. In *High Performance Computing: 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, June 18–22, 2017, Proceedings*. Springer-Verlag, Frankfurt, Germany, 217–236. ISBN: 978-3-319-58666-3. DOI: 10.1007/978-3-319-58667-0_12.

[66] Craig B Stunkel, Richard L Graham, Gilad Shainer, Michael Kagan, SS Sharkawi, B Rosenburg, and GA Chochia. 2020. The high-speed networks of the Summit and Sierra supercomputers. *IBM Journal of Research and Development*, 64, 3/4, 3–1.

[67] Sebastien Varrette, Hyacinthe Cartiaux, Teddy Valette, and Abatcha Olloh. 2022. Aggregating and Consolidating two High Performant Network Topologies: The ULHPC Experience. In *Practice and Experience in Advanced Research Computing*, 1–6.

[68] Abhinav Bhatele, Nikhil Jain, Misbah Mubarak, and Todd Gamblin. 2019. Analyzing cost-performance tradeoffs of hpc network designs under different constraints using simulations. In *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, 1–12.

[69] Georgios Kathareios, Cyriel Minkenberg, Bogdan Prisacari, German Rodriguez, and Torsten Hoefler. 2015. Cost-effective diameter-two topologies: analysis and evaluation. In *ACM/IEEE Supercomputing*. ACM, 36.

[70] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S. Schreiber. 2009. HyperX: Topology, Routing, and Packaging of Efficient Large-Scale Networks. *SC*.

[71] Mario Flajslik et al. 2018. Megafly: a topology for exascale systems. In *International Conference on High Performance Computing*. Springer, 289–310.

[72] Fei Lei, Dezun Dong, Xiang-Ke Liao, and José Duato. 2020. Bundlefly: a low-diameter topology for multicore fiber. In *Proceedings of the 2020 International Conference on Supercomputing*. (June 2020), 1–11. DOI: 10.1145/3392717.3392747.

[73] Fei Lei, Dezun Dong, Xiangke Liao, Xing Su, and Cunlu Li. 2016. Galaxyfly: a novel family of flexible-radix low-diameter topologies for large-scales interconnection networks. In *ACM ICS*.

[74] Kartik Lakhotia, Maciej Besta, Laura Monroe, Kelly Isham, Patrick Iff, Torsten Hoefler, and Fabrizio Petrini. 2022. PolarFly: a cost-effective and flexible low-diameter topology. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 1–15.

[75] Kartik Lakhotia, Kelly Isham, Laura Monroe, Maciej Besta, Torsten Hoefler, and Fabrizio Petrini. 2023. In-network allreduce with multiple spanning trees on polarfly. In *ACM SPAA*.

[76] Maciej Besta, Syed Minhaj Hassan, Sudhakar Yalamanchili, Rachata Ausavarungnirun, Onur Mutlu, and Torsten Hoefler. 2018. Slim noc: a low-diameter on-chip network topology for high energy efficiency and scalability. In *ACM ASPLOS* number 2. Vol. 53. ACM New York, NY, USA, 43–55. DOI: 10.1145/3296957.3177158.

[77] Patrick Iff, Maciej Besta, Matheus Cavalcante, Tim Fischer, Luca Benini, and Torsten Hoefler. 2022. Sparse hamming graph: a customizable network-on-chip topology. In *DAC*.

[78] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P Brighten Godfrey. 2012. Jellyfish: Networking data centers randomly. *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[79] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S Schreiber. 2009. HyperX: topology, routing, and packaging of efficient large-scale networks. In *ACM/IEEE Supercomputing*, 41.

[80] Michihiro Koibuchi, Hiroki Matsutani, Hideharu Amano, D. Frank Hsu, and Henri Casanova. 2012. A case for random shortcut topologies for HPC interconnects. In *ISCA'12*. IEEE, 177–188.

[81] Maciej Besta, Marcel Schneider, Salvatore Di Girolamo, Ankit Singla, and Torsten Hoefler. 2021. Towards million-server network simulations on just a laptop. *arXiv preprint arXiv:2105.12663*.

[82] Andrew R Curtis, Wonho Kim, and Praveen Yalagandula. 2011. Mahout: low-overhead datacenter traffic management using end-host-based elephant detection. In *INFOCOM, 2011 Proceedings IEEE*. IEEE, 1629–1637.

[83] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. 2014. Planck: millisecond-scale monitoring and control for commodity networks. In *ACM SIGCOMM Computer Communication Review* number 4. Vol. 44. ACM, 407–418.

[84] Siddhartha Sen, David Shue, Sunghwan Ihm, and Michael J. Freedman. 2013. Scalable, optimal flow routing in datacenters via local link balancing. In *CoNEXT*.

[85] Fung Po Tso, Gregg Hamilton, Rene Weber, Colin Perkins, and Dimitrios P. Pezaros. 2013. Longer is better: exploiting path diversity in data center networks. In *IEEE 33rd International Conference on Distributed Computing Systems, ICDCS*, 430–439.

[86] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. 2011. Microte: fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*. ACM, 8.

[87] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. 2014. Wcmp: weighted cost multipathing for improved fairness in data centers. In *ACm EuroSys*.

[88] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. 2010. Hedera: dynamic flow scheduling for data center networks. In *NSDI*. Vol. 10, 19–19.

[89] Abdul Kabbani, Balajee Vamanan, Jahangir Hasan, and Fabien Duchene. 2014. FlowBender: Flow-level Adaptive Routing for Improved Latency and Throughput in Datacenter Networks. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 149–160.

[90] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John B. Carter, and Aditya Akella. 2015. Presto: edge-based load balancing for fast datacenter networks. In *ACM SIGCOMM*.

[91] Naga Praveen Katta, Mukesh Hira, Aditi Ghag, Changhoon Kim, Isaac Keslassy, and Jennifer Rexford. 2016. CLOVE: how I learned to stop worrying about the core and love the edge. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks, HotNets*, 155–161.

[92] Mohammad Alizadeh et al. 2014. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 503–514.

[93] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. 2017. Let it flow: resilient asymmetric load balancing with flowlet switching. In *NSDI*, 407–420.

[94] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. Hula: scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*. ACM, 10.

[95] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. 2007. Dynamic load balancing without packet reordering. *ACM SIGCOMM Computer Communication Review*, 37, 2, 51–62.

[96] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy H. Katz. 2012. Detail: reducing the flow completion time tail in datacenter networks. In *ACM SIGCOMM*, 139–150.

[97] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wojcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM SIGCOMM*.

[98] Advait Dixit, Pawan Prakash, Y Charlie Hu, and Ramana Rao Kompella. 2013. On the impact of packet spraying in data center networks. In *INFOCOM, 2013 Proceedings IEEE*. IEEE, 2130–2138.

[99] Jiaxin Cao et al. 2013. Per-packet load-balanced, low-latency routing for clos-based data center networks. In *ACM CoNEXT*, 49–60.

[100] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2015. Fastpass: a centralized zero-queue datacenter network. *ACM SIGCOMM Computer Communication Review*, 44, 4, 307–318.

[101] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. 2011. Improving datacenter performance and robustness with multipath TCP. In *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 266–277.

[102] Van Emden Henson and Ulrike Meier Yang. 2002. Boomeramg: a parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41, 1, 155–177. Developments and Trends in Iterative Methods for Large Systems of Equations - in memorium Rudiger Weiss. DOI: https://doi.org/10.1016/S0168-9274(01)00115-5.

[103] Mantevo Project. 2016. Minife finite element mini-application. https://github.com/Mantevo/miniFE. (2016).

A Details of Slim Fly Construction

A.1 Selecting Topology Size, Parametrizing Input

Overall, one first chooses a prime power q that satisfies the equation $q = 4w + \delta$ for some $\delta \in \{-1, 0, 1\}$ and $w \in \mathbb{N}$. q is an input parameter that determines the whole topology structure. For example, the number of vertices (switches) is $N_r = 2q^2$ and the network radix $k' = \frac{3q-\delta}{2}$. In our case, $N_r = 50$, thus $q = 5$, which satisfies the equation $q = 4w + \delta$ for $w = 1$, $\delta = 1$, and $k' = 7$. Hence, every switch is connected to 7 other switches. Interestingly, this construction forms the famous Hoffman-Singleton graph [25, 26], which is *optimal* with respect to the Moore Bound. Finally, as a regular and direct network, it is recommended to attach $p = \left\lceil \frac{k'}{2} \right\rceil$ endpoints to each switch to ensure *full global bandwidth* [1]. In our case, $p = 4$.

A.2 Finding Needed Algebraic Structures

Once q is selected, one uses it to construct several algebraic structures. Specifically, one finds a *base ring* $\mathbb{Z}_q$ (for us, $\mathbb{Z}_5 = \{0, 1, ..., 4\}$), its *primitive element* ξ that generates all elements of $\mathbb{Z}_q$ (for us, $\xi = 2$), and two *generator sets* $X = \{\xi^0, \xi^2, ..., \xi^{q-3}\}$ and $X' = \{\xi^1, \xi^3, ..., \xi^{q-2}\}$ (for our installation, $X = \{1, 4\}$ and $X' = \{2, 3\}$). While not complex, details on these structures are not necessary to understand our Slim Fly deployment. The interested readers may check them in the original publication [1].

A.3 Labeling and Connecting Switches

Each switch receives a 3-tuple label from a set $\{0, 1\} \times \mathbb{Z}_q \times \mathbb{Z}_q$. Thus, SF switches come in two flavors determined by the first elements of their labels: $(0, \cdot, \cdot)$ and $(1, \cdot, \cdot)$. These labels determine how the switches are connected. Specifically, switches with labels $(0, \cdot, \cdot)$ are connected using the following equation [1]:

$$\text{switch } (0,x,y) \text{ is connected to } (0,x,y') \iff y - y' \in X. \quad (1)$$

Symmetrically, switches with labels $(1, \cdot, \cdot)$ use the following equation:

$$\text{switch } (1,m,c) \text{ is connected to } (1,m,c') \iff c - c' \in X'. \quad (2)$$

Lastly, two switches with labels $(0, \cdot, \cdot)$ and $(1, \cdot, \cdot)$, respectively, are connected according to the following equation:

$$\text{switch } (0,x,y) \text{ is connected to } (1,m,c) \iff y = m \cdot x + c \quad (3)$$

A.4 Topology Structure & Physical Layout

The graph underlying the SF topology consists of two same-size subgraphs. One subgraph contains routers $(0,x,y)$, the other consists of routers $(1,m,c)$. Each subgraph contains q identical groups of routers. Groups in different subgraphs usually differ from one another. There are no connections between groups within the same subgraph, i.e., no two routers $(0,x,y)$ from different groups are linked, the same holds for routers $(1,m,c)$. However, each group from one subgraph has connections to *every other group* in the other subgraph; thus the groups form a fully connected bipartite graph.

This property facilitates physical layout and we use it in our installation. Specifically, as recommended in the original work [1], we combine groups from different subgraphs pairwise; these combined groups form racks. In general, this leads to q racks, each with $2q$ routers. In our installation, we have 5 racks, each with 10 routers and 40 compute nodes.

A.5 Constructing Slim Fly with N nodes

As the space of valid SF topologies is quite sparse, we show the simple steps needed to find a SF network with the number of nodes as close to N as possible:

1. Obtain the cube root R of the desired node count N
2. Find prime powers close to R
3. Obtain the corresponding full-bandwidth network configurations (see previous sections)
4. Verify network sizes and select the network that is closest to N in terms of number of supported nodes

B Routing Details

B.1 Details of Layer Generation

We provide more details on crucial parts of layer generation.

B.1.1 Finding Almost-Minimal Paths

We look for almost-minimal paths that are exactly 3 hops long (one hop longer than SF's diameter of two), while balancing the number of paths crossing each link (to avoid highly congested links). We do not target longer paths, in order to conserve network resources (i.e., a flow taking fewer hops occupies fewer buffers).

For this, we design a heuristic based on a modified breadth first search graph traversal starting from the source node *src*, constraining the path length to 3. In theory one could also define a range of valid lengths. The heuristic obtains the set P of all valid paths starting in *src* and ending in the destination node *dst*; $P = \{(u_1, \ldots, u_l) \mid l = 3 \wedge u_1 = src \wedge u_l = dst\}$. Here, a path is considered valid if it satisfies the given length constraint 3 and if its insertion into the layer does not affect any previously inserted paths. Then, we choose a path $p \in P$ that minimizes link weights, i.e., $\forall p' \in P\ \omega(p) \leq \omega(p')$ where $\omega(p)$ is the sum of weights of links included in p.

B.1.2 Node Pair Priority Queue

The order in which the paths are inserted is very important, because it may impact whether we are able to find new paths. If one would first find a given number of paths for a single node pair, and only then proceed to the next node pair, some node pairs might not receive any, or much fewer, paths than other pairs. To alleviate this, we balance the total number of added almost-minimal paths across all node pairs. For this, each node pair is assigned a priority value, equal to its total number of almost-minimal paths across all layers; the lower the value, the more important it is to find a path for this node pair. Therefore, the number of required priority levels is upper-bounded by $|L| - 1$, because each node pair can have at most one almost-minimal path per each of $|L| - 1$ layers, and is initially in the highest priority value (value of 0). The lowest priority level is value $|L| - 1$, which only contains node pairs who have had an almost-minimal path inserted in every layer.

Whenever a path is added to a layer, all of the node pairs that have a non-minimal path inserted have their priority decreased and they move up to the next higher priority layer. For instance, in Fig. 16 by assuming that the minimum length for an almost-minimal path is two, adding the illustrated path to a layer, results in both node pairs (v_1, v_4) and (v_2, v_4) having an almost-minimal path added for them (assuming we allow paths of length 2 and 3 as non-minimal paths and *dst* is one of the receiving nodes). Therefore, both of their priorities would be decreased by 1. This also assumes that the paths were not already in this layer, which could have been the case for (v_2, v_4).

The *node_pairs* list generated from the priority queue p in Algorithm 1 contains the entries of the priority queue in the order of priority value, and randomized within each level. Hence, the layer generation algorithm first tries to add an almost-minimal path for all nodes of priority value 0 in a random order, and then move to the nodes of the next value. Hence, it first processes all node pairs with no inserted paths, then with one inserted path, and so forth, facilitating a balanced path distribution across node pairs.

B.1.3 Path Weighting

A weight update is performed after the insertion of a new path into any layer. The weight of each link in any existing path is increased by the total number of new "routes" that now occupy the link. An example is shown in Fig. 15. The weight of link (v_1, v_2) is increased by 9 because it has 9 new routes using it, as there are 3 sending nodes $(a_1 - a_3)$ and 3 receiving nodes $(b_1 - b_3)$. The weight of link (v_3, v_4) is increased by 27 as there are 9 sending nodes $(a_1 - a_9)$ and 3 receiving nodes $(b_1 - b_3)$.

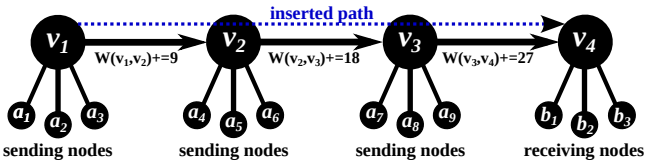


Figure 15: Illustration of the weight update methodology employed by the algorithm. After the insertion of the path from v_1 to v_4, the weights of the links (v_1, v_2), (v_2, v_3) and (v_3, v_4) are increased by 9, 18, and 27, respectively.

B.1.4 Potential Invalidity of Paths

For a given source *src* and destination *dst*, it may happen that $P = \emptyset$, in which case no almost-minimal path is added to a given layer for that node pair. There are two scenarios when this may happen, we illustrate them in Fig. 16 and in Fig. 17. The first one occurs when a path for the node pair is already included in another (previously inserted) path into the layer. For instance, after the path in the figure is inserted into layer l, all sub-paths $((v_2, v_4), (v_3, v_4))$ become included as well, forcing v_2 and v_3 to route along minimal paths towards destination v_4 in layer l.

The second scenario occurs when no path of required length can be found because routing via any of the source node's neighbors would result in a path too short or too long. In our second example, the almost-minimal paths are constrained to have length exactly 3. At first, the two almost-minimal paths $q = (v_1, v_2, v_3, u_3)$ and $q' = (w_1, w_2, w_3, u_3)$ are inserted, which fixes the paths for all node pairs in the set $\{(v_i, u_3), (w_i, u_3) \mid i \in \{1, 2, 3\}\}$. Now any path for the node pair (u_1, u_3) that respects the already inserted paths will have length $l \in \{1, 2, 4\}$ because it would have to come from the following set of paths: $\{(u_1, q), (u_1, q'), (u_1, u_3), (u_1, u_2, u_3), (u_1, u_2, v_2, v_3, u_3), (u_1, u_2, w_2, w_3, u_3)\}$. If this scenario occurs, we route minimally, i.e. path (u_1, u_3).

B.1.5 Specification of Forwarding Tables

In layered routing, each forwarding entry $(l, s, d) \in$ layers $\times$ switches $\times$ switches corresponds to the port that switch s uses

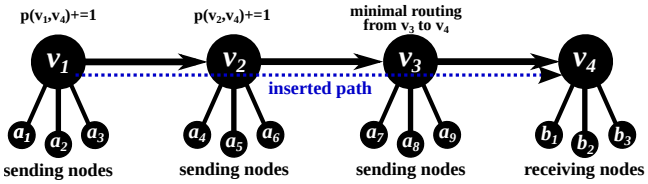


Figure 16: Illustration of an almost-minimal path from v_1 to v_4, which enforces minimal routing from *src* nodes like a_7, located on the sub-paths, to *dst* nodes, i.e. b_1, in this layer.

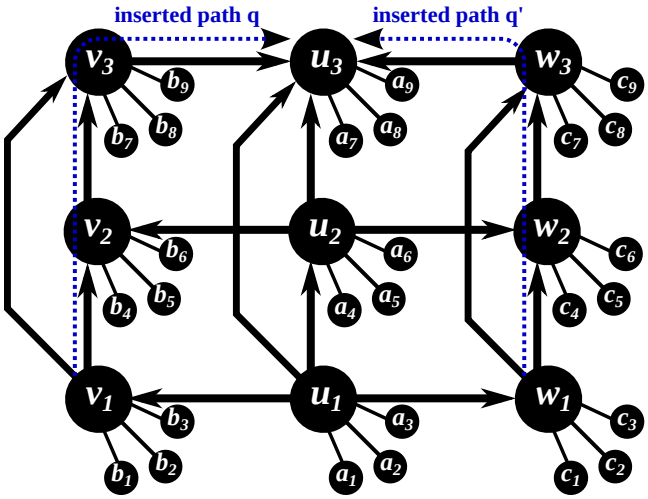


Figure 17: Illustration of a scenario in which no almost-minimal, valid path of length exactly 3 can be found for node pair (u_1, u_3) in the given layer due to the prior insertion of two valid paths.

when routing in layer l and transmitting a packet addressed to a switch d.

C Additional Results

C.1 Changes for Custom Alltoall

We decided not to use the OpenMPI's default implementation of alltoall, as the algorithms it relies on result in sub-optimal performance for the deployed SF. Empirically, we determined that the best-performing alltoall for our system was a simple algorithm that posts all non-blocking send and receive requests simultaneously and then waits for completion. Other collectives did not show a similar impact, and we thus used the default implementations. These issues are not expected with newer hardware.

C.2 Scientific Workloads & HPC Benchmarks

We show in Fig. 18 the runtime and relative performance of the solver/kernel for each of the scientific workloads on SF using the random placement strategy. We observe similar trends as for the linear placement strategy for all scientific workloads and SF's performance aligns closely with FT's, while no significant speedup or slowdown through the use of non-minimal paths could be observed.

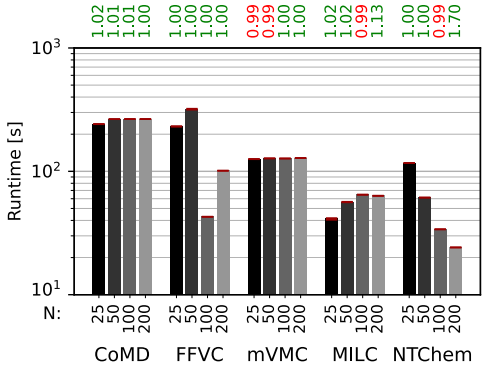
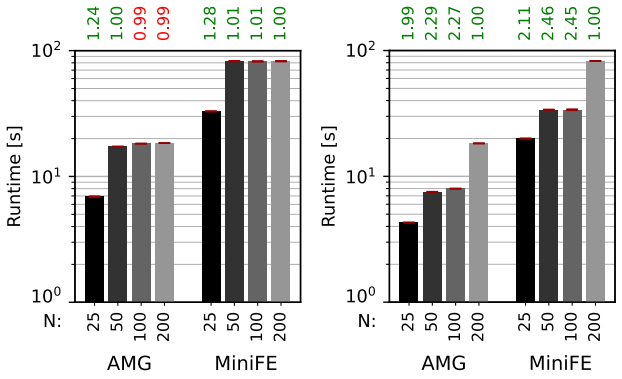


Figure 18: Runtime of scientific workloads (lower is better) - SF R vs. FT



(a) SF R vs. FT (b) SF L vs. FT

Figure 19: Runtime of additional scientific workloads (lower is better)

In Fig.19, we present the relative performance of two additional scientific workloads, AMG[102] and MiniFE [103], on SF, using both placement strategies. For this assessment, AMG was configured with a 128^3 cube per process, while MiniFE was set with grid input dimensions of $n_{x|y|z_b} = 90$. In accordance with these configurations, clear weak-scaling behavior is evident under the random placement strategy. On the other hand, with the linear placement strategy, the observed trends are less distinct, and, there are instances where SF outperformed FT by unexpected margins. We believe that this disparity can't be merely attributed to the variations in communication stemming from the placement strategy, as the applications in consideration aren't generally communication-bound and the FT is fully non-blocking. However, the precise cause remains unclear.

Fig. 20 shows the performance of the HPC benchmarks on SF using the random placement strategy, results that largely mirror those obtained using the linear placement strategy.

C.3 Deep Learning Workloads

The left part of Fig. 21 shows the runtime and relative performance of the DNN proxies with the random placement strategy. The results are also very similar to those obtained using the linear placement strategy, including GPT-3 matching the performance trends of the MPI Allreduce pattern with the random placement strategy and comparable node counfig-

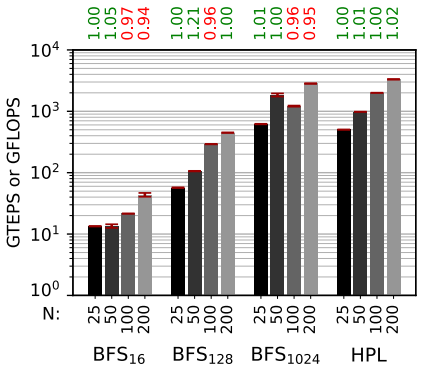


Figure 20: Performance of HPC benchmarks (higher is better) - SF R vs. FT

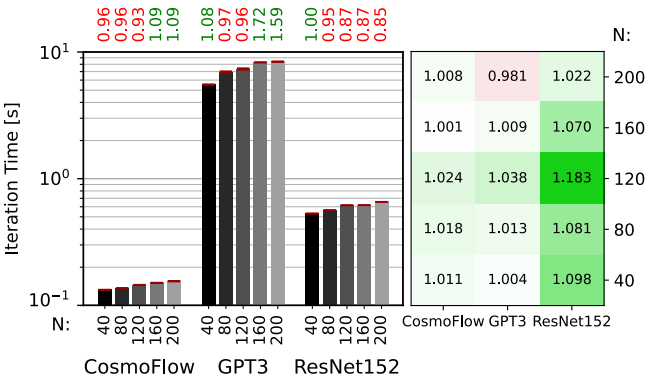


Figure 21: Iteration time of DNN proxy workloads (lower is better) SF R vs. FT and routing improvement of this work over DFSSSP (heatmap) in SF R

urations (cf. Fig. 11b).

However, similar to previous results, the right part of Fig. 21 shows that our work generally matches or outperforms DFSSSP, achieving up to a 1.18x speedup.

D Pricing details

We based our pricing on data colfaxdirect.com[6] and SHI.com[7]. Regarding the equipment selection, we use InfiniBand Topology Configurator [8]. For different switch sizes, we selected different models from current Nvidia offerings. For example, for a 36-port switch, we chose Mellanox SB7800 EDR 100Gb/s[9]. For a 40-port switch, we decided to use Mellanox Quantum QM8700 HDR 200Gb/s [10]. Finally, for a 64-port switch, we use Nvidia QM9700 NDR 400G [11] model. For AoC cables, we selected active fiber links, and for DAC cables, we chose passive copper cables for endpoint connections. Again, we base our estimations on mentioned earlier InfiniBand Topology Configurator online service. However, it can be challenging to determine the cost of networking hardware because the prices of such hardware can vary greatly depending on the quantity ordered, and large orders may be eligible for substantial discounts.

[6] COLFAX DIRECT website
[7] SHI website
[8] Mellanox InfiniBand Topology Generator tool
[9] Mellanox SB7800 EDR 100Gb/s product detail
[10] Mellanox Quantum QM8700 HDR 200Gb/s product detail
[11] Nvidia QM9700 NDR 400G product detail

Crescent: Emulating Heterogeneous Production Network at Scale

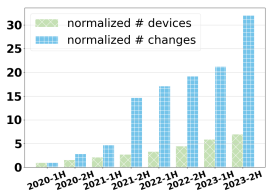
Zhaoyu Gao*, Anubhavnidhi Abhashkumar*, Zhen Sun◇, Weirong Jiang*†, Yi Wang*
ByteDance, Cornell University*◇

Abstract

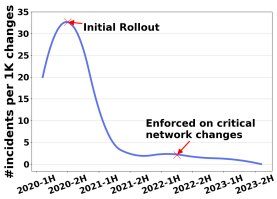
This paper presents the design, implementation, evaluation, and deployment of *Crescent*, ByteDance's network emulation platform, for preventing change-induced network incidents. Inspired by prior art such as CrystalNet, *Crescent* achieves high fidelity by running switch vendor images inside containers. But, we explore a different route to scaling up the emulator with unique challenges. First, we analyze our past network incidents to reveal the difficulty in identifying a safe emulation boundary. Instead of emulating the entire network, we exploit the inherent symmetry and modularity of data center network architectures to strike a balance between coverage and resource cost. Second, we study the node-to-host assignment by formulating it as a graph partitioning problem. Evaluation results show that our partitioning algorithm reduces the testbed bootup time by up to 20× compared with random partitioning. Third, we developed an incremental approach to modify the emulated network on the fly. This approach can be 30× faster than creating a new testbed of the same scale. *Crescent* has been actively used for three and a half years, which led to a significant reduction in change-induced network incidents. We also share *Crescent*'s success in many other use cases and the critical lessons learned from its deployment.

1 Introduction

As one of the largest and fastest-growing global online service providers, ByteDance's physical network infrastructure has expanded rapidly in recent years to meet the explosive business demand [5, 7]. Such a wild expansion has created a hyperscale and heterogeneous global network that consists of medium- and large-scale data centers (DCs), regional and global wide area networks (WANs), points of presence (PoPs) of all sizes, and virtual DCs (vDCs) from multiple public cloud providers. We have network switches from nearly all the major switch vendors and multiple generations of network architectures co-existing across different DCs/PoPs. As shown in Figure 1a, the total number of switches in our networks has grown by 7.5× in the past three and a half years. Meanwhile, the number of changes made on these networks has increased at an even higher rate. The changes, mainly on topology and switch configurations, include modifications to our network structures, routing policies, device software, peering with cloud and Internet service providers (ISPs), etc. Many such changes require network operators' careful designs, detailed Methods of Procedure (MOPs), and cautious executions. But even with those efforts, we still ran into a series of outages due to missed or unforeseen issues. In the second half of 2020, network changes accounted for approximately one-third of our incidents. Since the networks kept growing bigger and increasingly complex, it became a daunting challenge for our network operators to think through all scenarios and identify all possible issues before performing changes to the production networks.



(a) # network devices and changes (normalized using 2020-1H).



(b) Trend of # network incidents per thousand changes.

Figure 1: Growth of network size, network changes, and network incidents in last 3.5 years.

To reverse the alarming trend of the change-induced incidents, we developed *Crescent*, a high-fidelity emulation platform to provide a production-like environment for network operators to test and verify their changes. As shown in Figure 1b, we rolled out *Crescent* during the second half of 2020. Since then, we have witnessed a steady decrease in network incidents along with increased deployment and usage of *Crescent*. Around the mid of 2022, we enforced *Crescent* as a mandatory requirement to conduct critical network changes, which has further bent down the curve.

†corresponding author

We are aware that a plethora of solutions have been proposed in recent years to prevent network incidents caused by configuration changes. One popular alternative to emulation is control plane verifiers (CPV) [12, 29, 55, 57, 59] which employ either simulation or formal modeling to verify the impact of configuration changes on specific reachability properties of the network. But most of them cannot detect vendor-specific behaviors (VSBs), which have caused many of our incidents (§2.2). Hoyan [55] attempts to model VSB by actively tracking the differences between its analysis output and the real production behavior. But, it can only catch VSBs that are present in production and requires extra efforts to update its VSB models. Many VSBs that caused our incidents (§2.2) were neither documented nor seen in production beforehand. Other limitations of CPV have been discussed in [19, 41].

Network emulators [3, 4, 8, 10, 23, 40, 41, 53] overcome the aforementioned CPV limitations by running switch vendor software images in virtual machines (VMs) or containers. Nevertheless, very few emulators can scale to emulate large networks, and most require manual efforts to adapt production configurations to the format supported by the vendor images (§4.2). The closest work to *Crescent* is CrystalNet [41], which leveraged cloud VMs to scale up and proposed reducing the size of the emulated network by finding a *safe static boundary*. But as admitted in [41], the boundary found by CrystalNet is based on certain assumptions from their routing policies that do not apply to everyone. Analysis of our past incidents confirms that such a boundary is much harder to find in reality (§2.2). This led us to a different route from CrystalNet in scaling up the emulator. The design and implementation of *Crescent* aim to answer the following questions, which are also the main contributions of this paper.

First, if it is hard to identify a safe emulation boundary, *does it mean we have to emulate the entire network?* Based on the analysis of our past incidents as well as the experience from other hyperscalers [13, 15, 32, 41, 42, 48, 51], we see that the changes on different devices have different impacts. High-level core devices (e.g., WAN devices) tend to have more complicated configurations, and the incidents due to the changes in these devices tend to be more severe. On the other hand, the devices inside a DC are normally in modular and symmetric architecture, and incidents caused by the changes in these devices have a limited blast radius. Inspired by these observations, we proposed emulating a baseline topology that includes all the core devices while sampling the lower-level non-core devices.

Second, the above baseline topology is still quite large. Because of the high resource requirement to run switch images, we need many hosts to emulate a large network. Then *will it matter how we map the network nodes to different hosts?* CrystalNet mapped nodes to many VM hosts but did not answer this question. We conducted thorough experiments to show that the performance was heavily impacted by the node-to-host assignment. We formulated it as a graph partitioning problem and proposed a variant of the community detection algorithm to solve it. Evaluation results show that our algorithm resulted in 20× improvement in testbed bootup time compared with random partitioning.

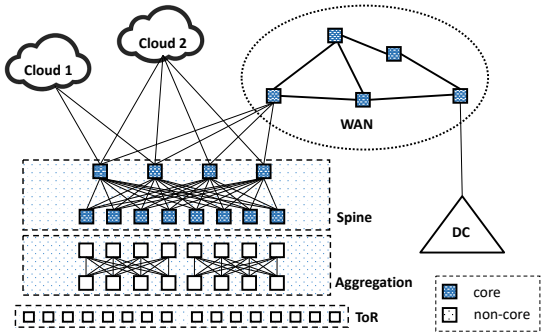


Figure 2: Simplified view of ByteDance's network.

Third, *how to test a network change plan involving devices (i.e., devices under test, namely DUT) that are not part of the above baseline topology?* We connect DUT to the baseline topology through controlled expansion to form a connected graph. And instead of rebuilding the testbed for the expanded network, we create the expanded nodes and wire them dynamically to the baseline topology which has been emulated in a pre-built testbed called *canary testbed*. Such an incremental emulation scheme improved the testbed ready time for testing a network change plan by 30×.

Last, emulation is only the first step; it reproduces the impact of a configuration change but still requires someone to analyze the updates' impact. Instead of asking users to regularly monitor and interact with the emulated network, *Crescent* takes a more proactive approach. It tightly integrates with a diverse range of efficient verification tools to proactively detect errors and unexpected behaviors that arise due to changes in real-time.

The rest of the paper is structured as follows: §2 outlines the motivation and describes the challenges in our network. The design of *Crescent* to address these challenges is proposed in §3, while scalability is tackled with a novel graph partitioning algorithm in §4. The incremental emulation approach is detailed in §5, along with various verification and monitoring tools integrated into *Crescent* to automatically detect issues in an emulated network in §6. Evaluation results for *Crescent* are presented in §7, with other use cases described in §8. Lessons learned from building and running *Crescent* for 3.5 years are shared in §9, with related works discussed in §10. Finally, §11 concludes the work.

2 Background and Motivation

2.1 ByteDance's Network

Our network mainly consists of three components: (1) a set of data center networks (DCNs) at different scales, (2) a global

WAN and several regional WANs that connect all DCs and PoPs, and (3) vDCs from various third-party cloud providers that are connected to our major DCs. Figure 2 gives a simplified overview of our network. All these components together form one of the world's largest networks, serving as one of the largest online services [7].

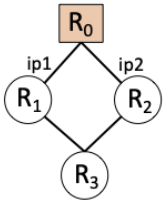
ByteDance's DCN is a variant of traditional multi-stage Clos network that can be described in the following simplified model. The bottom layer consists of ToR (Top of Rack) switches, each of which directly connects a rack of servers. These ToRs also connect to the middle layer aggregating connections from the bottom layer. These middle layer switches connect to the top layer, that comprises the core switches that aggregate middle layer fabrics and connect to other DCNs, WANs, and clouds. To be consistent with terminologies used in other works, e.g., [32], we call the middle layer "aggregation layer", and the top layer "spine layer". We define the *core devices* in our network as the WAN and spine layer devices, and the other devices as non-core devices [13, 42].

For the third-party clouds, we have no direct control over their physical devices. But we can configure certain routing policies in their management portals to control the peering and the traffic between our DC's border devices and the clouds. These policies are applied based on different demands and thus sometimes are not standardized, i.e., different regions may have different policies. Even in the same region, there can be different numbers of spine devices connected to a same cloud. As a result, the configurations on core devices are complicated and non-standardized.

Different from those core devices, the configuration of non-core devices in DCN is usually highly standardized. It entails that the devices on the same level tend to exhibit the same routing behaviors (e.g., receive the same routes from higher level devices, and announce the same granularity of routes to higher level devices). The high standardization for the non-core device configurations is ensured by (1) generating configuration for all non-core devices using automatic script and config templates; and (2) checking the running configurations periodically to detect violations against DCN design rules [26, 38]. Many other hyperscalers [13, 15, 32, 41, 42, 48, 51] also adopt the practice of standardizing and simplifying the configurations of non-core devices in DCN. Leveraging symmetry and synergy created by such standardization and simplification to streamline network analysis has been studied by many works in network verification [15, 16, 30, 43]. To the best of our knowledge, we are the first to extend this idea to the area of network emulation.

2.2 Incidents Analysis

In this section, we show two examples of our past network incidents. Then, we give a statistical analysis of all change-induced incidents in our network over the past years. These incidents motivate our features, design choices, and use cases.



R_0 Vendor	Other Vendors
bgp X	bgp X
group G1	group G1
peer ip1 group G1	peer ip1 group G1
peer ip2 group G1	peer ip2 group G1
ipv4-family unicast	ipv4-family unicast
peer G1 enable	peer G1 enable
peer ip1 enable	
peer ip1 group G1	
peer ip2 enable	
peer ip2 group G1	

Figure 3: An illustration of incident B.

2.2.1 Incidents Examples

Incident A. In one incident, a config change was made on a global WAN device to prepend an AS number to the AS-PATH of a DC route. However, the same AS number of global WAN was used for prepending, which triggered an unexpected VSB: while some vendors prevent AS loop only for eBGP, others consider iBGP too. Each of our global WAN devices peers with a route reflector (RR) through iBGP. The vendor of RR did not consider the route with prepended AS number forming an AS loop and propagated the route to the rest of global WAN. But the vendor of the rest of global WAN treated this as an AS loop and dropped the route. To prevent such an incident, we need to emulate at least the global WAN device under change, the RR and another global WAN device. To detect this issue via end-to-end connectivity test, we need to emulate more devices, i.e., from the origin DC of the affected route and from the DC connected to the other global WAN.

Incident B. In another incident, a network operator drained the traffic on a switch by disabling the peers in a BGP peer group and then re-enabling the peers to undrain the traffic. However, for this vendor, disabling the BGP peer group operation will automatically delete all statements that associate peers to a peer group under IPv4-family section (highlighted in Figure 3). Later re-enabling the peers will not automatically add those statements back. Figure 3 illustrates this incident: when doing this operation on R_0, the highlighted statement disappeared after the traffic undrain operation. The BGP peer group $G1$ has a route policy to extend `AS-PATH` length to 2 from R_1 or R_2 to R_0. However, the newly enabled peer $ip1$ is not associated with $G1$, thus has a shorter `AS-PATH` compared to the other peers, e.g., $ip2$, leading to all traffic going through the link from R_1 to R_0, which caused severe link congestion. To capture this issue, emulation must include all the core devices shown in the Figure 3, emulate R_0 with its respective vendor image, and compare the routes before and after the change.

We can draw 2 quick takeaways from the above incidents:

1. VSBs are notoriously hard to prevent because people are often unaware of VSBs, most of which are not well documented. These VSBs could lead to all kinds of incidents, also observed in prior works [41, 55]. High-fidelity emu-

lation using switch vendor images is the effective option to capture the unknown VSBs.
2. Emulating only the devices under test is insufficient to catch the impact of a change. It is difficult to identify the minimum scope of the network to be emulated, especially when the blast radius of the impact of a network change is hard to predict [32, 42]. More discussions are in §2.3.

2.2.2 Statistical Analysis

We conduct a statistical analysis over the O(100) incidents that happened in our networks in the last 3 years. Out of them, about 1/3 network incidents were caused by network changes and thus can be potentially captured by emulation. The others are caused by issues such as traffic bursts, capacity losses, device failures, etc. Among these change-induced incidents, **30% of them involved VSBs**. The impact of these incidents spans all 5 severity levels, among which level 1 means the most severe incident, while level 5 means the least. Approximately 80% of change-induced incidents are identified as level 3 or higher, resulting in substantial business losses. By further examining the change-induced network incidents, we found that **over 90% of these incidents happened on our core devices.** This is likely because the configurations on core devices are more complex than non-core devices, i.e., more routing protocols besides BGP (e.g., ISIS, OSPF, SRTE, etc) are used, more route policies are applied, and more complex topologies are formed. Such complexity is more likely to trigger various unknown VSBs.

2.3 Cost v.s. Coverage

To prevent the above incidents, an intuitive solution is to emulate the entire network, but it comes with a high cost (with respect to resource usage). As the number of devices to be emulated increases, so does the cost. We list all the vendor's images, for one or more versions, and the corresponding resource requirements in Table 1. Note that different images have different resource requirements , e.g., memory usage, the primary resource bottleneck, ranges from 1 to 16 GB.

An alternative is to find a tighter safe boundary as defined in CrystalNet [41]. After analyzing the incidents that happened in our network from the past 3 years, we find that their strategy to find a safe boundary does not apply to other networks like ours. First, it assumes that AS-PATH removal and rewrite are rare, leading to shortened boundaries. However, in our network, these actions are frequently utilized, particularly on core devices, and have caused **15%** of our incidents. Second, we find the expansion algorithm proposed by CrystalNet is primarily about expanding the network to higher layers. This is not enough for us to find a safe boundary to prevent **55%** inter-DC incidents in our network. For instance, to prevent Incident A, we need to include our global WAN and two DCNs to detect the issue. But CrystalNet can only expand until the global WAN that contains all DUTs of this change. On the other hand, it's crucial to expand the network downwards and include both DCs. Similarly, in Incident B, it's important to emulate all lower-level devices and observe that after the change, R3 will only forward traffic to R1 and not both R1 and R2. Overall, CrystalNet's expansion algorithm can potentially miss approximately **60%** of the incidents, among which 10% involve both the limitations. Moreover, we find that in some scenarios, even when their assumption holds, e.g., only `AS-PATH` prepend policy exists, an incident may still happen due to other issues, e.g., ECMP reduction in Incident B.

Vendor	Image type	vCPU	RAM(GB)	Size (GB)	Port limit
v1	container	1	2	1.60	unlimited
	VM	2	4	2.08	64
v2	VM	2	8	1.47	64
v3	VM	2	8	4.31	128
v4	VM	2	4	1.57	10
	VM	8	16	4.14	100
v5	container	1	2	0.43	64
v6	VM	2	4	1.34	10
v7	VM	4	5	1.48	96

Table 1: Specifications of various vendor images (*v1-7*).

Based on our analysis result, it is essential to always emulate the core devices in our network to avoid those 90% incidents that are caused by network changes on our core devices. Besides, incidents that happen on core devices tend to cause a larger blast radius than non-core ones. Meanwhile, we are not trading off safety for cost. By including the sampled non-core devices into the canary testbed (§3), the rest (less than 10%) of network incidents caused by changes on non-core devices can also be captured by *Crescent*, assuming their configurations are standardized. By doing this (i.e. including all the core devices and sampled non-core devices), we only need to emulate less than 2% of our entire network fleet.

2.4 Scalability over Multiple Hosts

Even though we choose to emulate a small fraction of our entire network, it still contains thousands of devices that we cannot emulate on a single host due to the high resource requirement of emulating each device (Table 1). A natural idea is to employ multiple hosts, as CrystalNet [41] did. But, it is unclear how CrystalNet maps thousands of nodes onto multiple hosts. In other words, how to partition a given network to let each host emulate a subset of the network? While similar problems have been studied in other contexts e.g. [52], none of them applies to the emulators like CrystalNet and *Crescent*. We observe that a multihost testbed using different node-to-host assignment schemes can yield significant performance differences. For example, a simple random partitioning may not scale at all (§7.2). We study this challenge by uncovering the system bottleneck, formulating it as a graph partitioning problem, and solving it with a variant of the community

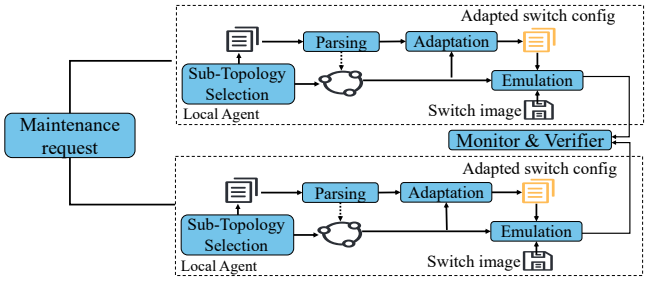


Figure 4: An illustration of *Crescent* workflow to create a new multihost canary testbed.

detection algorithm (§4.3).

3 *Crescent* Design Overview

We build a canary testbed to emulate the baseline topology that includes all the core devices and sampled non-core devices. We sample the non-core devices by selecting one plane out of multiple planes, and selecting one device per vendor on each level of a plane. To keep it simple and consistent, our strategy is to always pick devices from the 1st plane on every level, ensuring that aggregate routes can be activated, the same as in our production network. Also we integrate each testbed with proactive monitoring and verification tools to automate the detection of impact of changes.

3.1 Multi-Host Canary Testbed

We run the large-scale canary testbed by distributing thousands of emulated nodes across a cluster of baremetal servers using a multi-host setup (§4.1). We employ a novel partitioning algorithm to minimize the number of cross-host links, which significantly reduces the canary testbed bootup time (§4.3) as well as the time to connect DUTs to a canary testbed.

Figure 4 shows the workflow in a multi-host setup. Each host runs a local agent that manages emulation on that host. When the agent receives input for a maintenance request, it first identifies which new nodes and links need to be emulated on that host. Next, it fetches the configurations of new nodes from production networks and parses and adapts them to be compatible with vendor images. After this, the agent launches and connects each node's containers with the appropriate configurations and vendor images. Finally, when required, it sends the emulation data to the monitoring and verification tools for proactive analysis.

3.2 Connect DUTs to Canary Testbed

Instead of creating a large-scale testbed from scratch for every network change request, we run multiple instances of nonstop canary testbeds. For each request, we connect its DUTs to one idle canary testbed.

If DUTs are not in the baseline topology, *Crescent* needs to emulate these DUTs in new containers and connect them to the canary testbed, along with the intermediate nodes between them. *Crescent* achieves this using a novel expansion algorithm that ensures the heterogeneity of the devices along the paths from DUTs to canary testbed (§5.1). The expansion algorithm considers different factors (e.g., vendors, planes, levels) when exploring the paths from DUTs to the running canary testbed. To allow DUTs and the intermediate devices to connect to and disconnect from the canary testbed on the fly, *Crescent* supports dynamic link addition/removal (§5.2).

It is more cost-effective to run a nonstop canary testbed than rebuilding a new one. Also based on our historical data, over 50% of our network changes happened on core devices. It suggests that there is no need to create any new containers for most of the requests, as we already include all core devices in canary testbed. Besides, the time to connect DUTs to canary testbed is also much lower than the time to rebuild a new canary testbed (§7.1).

3.3 Proactive Verification and Monitoring

Emulation alone can not detect network issues (e.g., loop and blackhole) at scale. It must be combined with various verification and monitoring tools (§6) to automatically analyze the impact of change on an emulated network. After automatically applying each command of the MOP to DUTs, we take a snapshot of the converged network and proactively run the verification and monitoring tasks to detect potential issues caused by each command.

4 Building Canary Testbed

A large-scale high-fidelity canary testbed containing all core devices in our network is the key to the design of *Crescent* as described in §3. In this section, we discuss our approach to address the scalability challenge in running a large-scale canary testbed. We formalize the scalability challenge as a graph partitioning problem and then propose a heuristic algorithm to solve the problem.

4.1 Network Emulation

Crescent's network emulation platform shares similarities with most of the other container-based network emulators [3, 8–10,41,53]. Each node is wrapped in an individual container with a switch OS running inside. A link on the same host is emulated by a veth pair, while a link across hosts is emulated by OVS bridge. Figure 5 shows an example of such a container network implemented in *Crescent*. We defer the implementation details to §A.

We run the canary testbed on a cluster of baremetal hosts. On each host, *Crescent* runs a local agent that interacts with OS for node and link creation. Running a multi-host canary

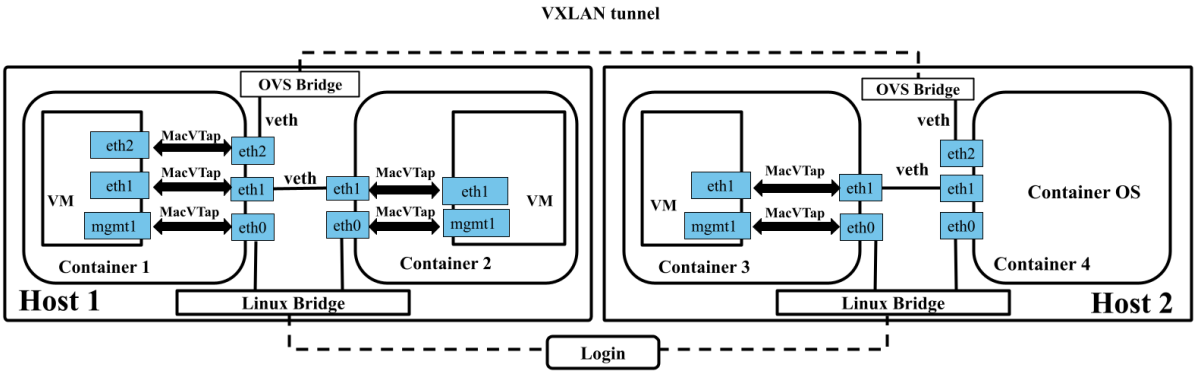


Figure 5: An illustration of *Crescent* implementation with virtual nodes (container-based and VM-based) and links.

testbed is straightforward with this setup. When booting up a testbed, we send the same topology input to all the host agents. The input topology specifies the host each node is running on. Then, each local agent boots up the local portion of the testbed based on the input topology, including the nodes and links. The agent creates local veth pairs between nodes on the same host and OVS bridges for cross-host links.

4.2 Multi-Vendor Config Adaptation

Upon the testbed bootup, *Crescent* immediately loads the configuration file into the emulated nodes. However, we find the production configuration cannot be directly applied to the emulated nodes due to the limitations of the vendor's software switch images. To that end, *Crescent* has to adapt the production configuration into an image-compatible form before loading it into the emulated nodes. Figure 6 shows an example where *Crescent* adapts an interface's configuration to ensure ISIS [1] is activated on this interface. First, it is important to note that the interface names in emulation may not precisely match those used in production. Therefore, it is necessary to rename these interfaces and maintain a mapping between the emulated interface names and their counterparts in the production environment. Second, certain security-enhanced commands, like "macsec," can prevent ISIS from being activated and thus must be filtered. Last, the MTU (Maximum Transmission Unit) value in the production environment is typically greater than the default MTU value supported by our host. Thus, we must reduce the MTU value in the adapted config to a value lower than the default MTU value on our host. Config adaptation, which involves all vendors and image types, is critical to automating large-scale testbed creation. Whenever a new, previously unseen configuration template is introduced, extension of *Crescent* is necessary to support it.

4.3 Multi-Host Partitioning

A large-scale canary testbed with thousands of nodes must run in a distributed manner. Nonetheless, prior research, such

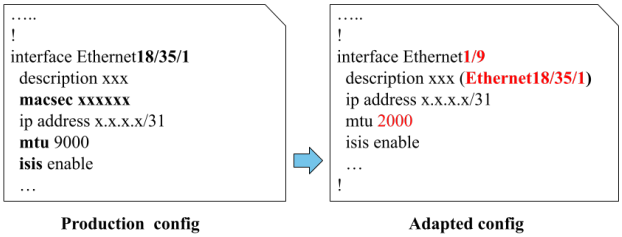


Figure 6: An example of configuration adaptation to a simple interface config to activate ISIS status on this interface.

as Crystalnet [41], does not delve into the detail on how to partition a large-scale testbed across a set of hosts (or VMs on cloud). Other works [56,60] focuses on minimizing cross-host communication overhead by reducing the number of cross-host links. However, we find that cross-host communication overhead is negligible (§7.3). *Crescent* also tries to reduce the number of cross-host links, but for a different reason, i.e., to reduce testbed creation and DUT connection time.

Cross-host Link Creation Overhead. A simple strawman approach is to partition all the emulated nodes across hosts randomly, following a uniform distribution. However, we find that it creates thousands of cross-host links, which incurs a significant overhead on creating a testbed and connecting DUTs. This phenomenon is because the overhead to create cross-host links increases linearly in the Linux kernel. It is not specific to the OVS bridge, but also applicable to the Linux bridge (used by CrystalNet [41]). As shown in Figure 7, both Linux bridge and OVS bridge creation incur a linear overhead (the x-axis is in log scale), i.e., the more links there are on the host, the longer it takes to create a new link[1]. This overhead is inevitable even if all bridges are created in parallel.

Consequently, we must minimize the number of cross-host links. The benefit has two folds. First, connecting DUTs usually needs to create O(100) new links (§5.2), thus, by reducing

[1]Besides merely creating the bridge, the bridge creation operation we consider here also includes the operations of creating veth pair, binding VXLAN port to the bridge, and bringing up all these virtual devices

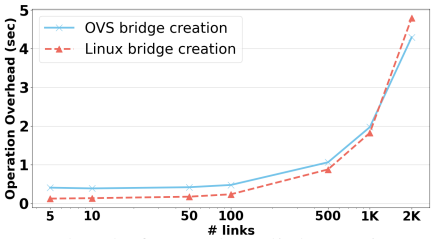


Figure 7: Overhead of cross-host link creation (OVS bridge and Linux bridge).

the number of cross-host links, we can substantially expedite the process of connecting DUTs. Second, when booting up the multihost testbed, with fewer cross-host links to create, the overall bootup time can also be reduced.

Problem Model. We model the cross-host link minimization problem as follows. Given a graph $G = (V, E, W_V, W_E)$, where V is the set of vertices, E is the set of edges, $w_v \in W_V$ is the weight of node v and $w_e \in W_E$ is the number of links between the vertices connected by e. In practice, we find that memory usage is usually the bottleneck. Thus the node's weight is the memory a node needs, which is constrained by the server's total amount of available memory C. Given the server capacity C, the goal is to find a partition P for V with n disjoint sets $V_1, V_2, ..., V_n$.

$$\begin{aligned} \min \quad & \sum_{i,j} \sum_{e \in E_{ij}} w_e \\ \text{s.t.} \quad & 1 \le i, j \le n \\ & E_{ij} = E \cap V_i \times V_j \\ & V = V_1 \cup V_2 \cup ... V_n \\ & V_i \cap V_j = \emptyset \\ & \sum_{v \in V_k} w_v \le C, 1 \le k \le n \end{aligned} \tag{1}$$

Partitioning Algorithm. This is a graph partitioning problem, which is NP-complete [56]. It has also been widely studied in previous literatures [21, 56, 60]. In the field of social networks, the problem is called community detection [18, 22, 31]. Namely, detecting closely connected vertices in a graph merely based on the links rather than other attributes, which is different from clustering.

To solve the problem, we explore the traditional community detection algorithm [31]. The traditional approach begins with a set of all n vertices in the graph, with no edges between them. Then, it adds edges between pairs one by one in order of their weights. As edges are added, the resulting graph shows a nested set of increasingly large components (connected subsets of vertices), eventually forming a partitioning scheme.

We made a few modifications to the traditional community detection algorithm to solve the optimization problem (1). First, we define the edge weight when merging two components with newly added edges as the geometric mean of the two components' weights. Second, we add a capacity constraint to the traditional algorithm to ensure the size of each partition does not exceed a single host memory capacity. Last, to avoid being trapped in a local optimum partitioning scheme, we follow an exponential stochastic process to randomly choose the next edge to add to the graph. These modifications ensure that the algorithm generates a scheme with well-balanced partitions, all within the server's capacity. We defer the detailed algorithm pseudocode to §B.

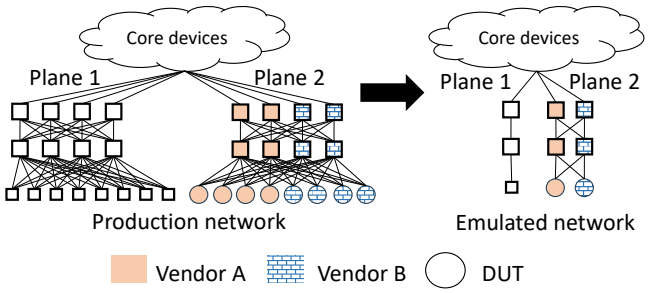


Figure 8: DUT sampling and topology expansion.

5 Connecting DUTs to Canary

For a maintenance request, if DUTs are not in the canary testbed, *Crescent* needs to connect these DUTs to the canary testbed, along with the intermediate nodes between them.

5.1 Topology Expansion

As mentioned in §2.1, the configuration on non-core devices is auto-generated, and DCN topology is highly standardized. We take advantage of this simplicity and standardization to do a more controlled expansion. While finding a generic algorithm that works for all networks may be challenging, we have successfully tested this strategy in hundreds of maintenance requests without causing any issues in production.

When expanding from DUTs, the algorithm ensures that each node has at least one neighbor for each vendor, level, and plane. The number of devices included in the expansion process grows exponentially. However, given that there are at most 3 to 4 levels from the canary testbed to the lowest non-core device (i.e., ToR), at most 3 different vendors on the same level in our DCN, and tens of planes, each DUT will only expand to tens of devices to establish paths up to the canary testbed at worst.

If DUTs are on the lower level, they also exhibit similar routing behavior per level. And we can apply the same expansion rules to select DUTs for emulation during maintenance.

Figure 8 presents an example of applying the algorithm on a change with 8 DUTs in a plane. It only shows the result for two planes. In this example, all 8 ToR DUTs have a similar configuration, and the devices belong to two vendors. After sampling, we reduce the emulated DUTs to 2. And besides

the 2 DUTs, we only emulate 7 instead of 24 additional nodes. We explain the expansion algorithm in §C.

5.2 Dynamic Connection

To connect the DUTs to a canary, *Crescent* sends a new topology input to all the host agents of the canary testbed. Each agent compares its local topology information with the newly received topology to find the new links it needs to create. Similarly, when removing links, the local agent must determine the existing nodes as well as the links it needs to destroy based on the topology information.

CrystalNet [41] does not support dynamic link addition and removal; thus, we extend its techniques to enable this functionality. Link addition/removal requires rebooting nodes on both ends of the link because no vendor image supports dynamic interface addition/removal, i.e., software switch OS can only detect interfaces during bootup. To solve this issue, we decouple the network stack from the actual DUT container's runtime. For each actual DUT container, we create a container to hold the network stack for the DUT container. This way, links can be added after stopping the DUT container, and we can restart the DUT after adding all the new links.

6 Proactive Verification and Monitoring

We run various verification and monitoring tasks to test, analyze, and troubleshoot the changes made in the emulated network before pushing it to production. Motivated by our incidents (§2.2), our tools are tailored to catch the following behaviors: routing loop, blackholes, unexpected ECMP reduction, and unexpected route withdrawal/change/churn. We run four major tasks to cover these issues.

Config Checker. A static configuration analysis tool similar to RCC [26] analyzes the routing-related segments of the parsed switch configurations to identify basic syntactical and semantical errors. We first convert vendor-specific configurations into a vendor-independent format. Then our config checker analyses them to identify common errors like undefined variables [26] (referring variables without defining them), configuration dissimilarity (configs deviating from the template [38]), etc. This tool's primary goal is to detect non-standard configurations on non-core devices and to identify any new configuration errors on DUTs.

Route Differ. Our route-differ aims to detect local forwarding changes at each node. It periodically captures the emulated nodes' latest FIB (Forwarding Information Base) and compares it to the previous snapshot to detect route changes. A centralized full-scale implementation may not perform well in emulation due to resource scarcity. As a runtime optimization, we developed a lightweight distributed version to run the diff locally within each container and only send the result to the central process for aggregation. The route differ plays a crucial role in identifying incidents that lead to reduced ECMP next hop count, resulting in congestion (e.g., Incident B as discussed in §2.2). Additionally, this helps in detecting unexpected additions or removal of routes.

Pingmesh. We run end-to-end ping connectivity tests between all the emulated servers of a network. It is much simpler than the original Pingmesh [33]. We run a script inside each emulated server to ping the other IPs and to only send the ping failure messages to the central process for aggregation. The main objective of this task is to identify connectivity-related incidents, e.g., Incident A in §2.2

Data Plane Verifier. Data plane verifiers (DPV) model how packets will be sent in the network across a set of forwarding tables and verify if they satisfy specific policies, like detecting routing loops, blackholes, waypoint violations, etc. We built our DPV based on APKeep [58] with customized optimizations. The occurrences of blackhole and routing loop incidents are primarily detected by DPV.

7 Evaluation

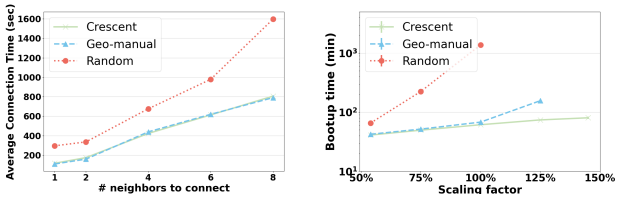
In this section, we evaluate *Crescent* performance for (1) the time to connect different numbers of DUTs to the canary testbed, (2) canary testbed bootup time at different scales, (3) the time to run the most commonly used MOP commands for a canary testbed, and (4) the time to run various verification and monitoring tasks.

We show the impact of three different partitioning schemes on the performance metrics. The first scheme is random partitioning that assigns all n emulated nodes to k hosts randomly, so that each host has $\frac{n}{k}$ nodes to run. In the experiment, we find that the random partitioning scheme over the original testbed creates too many cross-host links, which prevents the testbed from booting up successfully. The second scheme is done by our proposed algorithm (§4.3). We call it *Crescent* scheme. The last scheme is done by manually partitioning the canary testbed into k parts, with each part containing nodes within a specific geographic affinity region This scheme is referred to as the *Geo-manual*. The *Geo-manual* scheme is provided by one of our network experts who is familiar with our network architecture.

We do all the experiments on a dedicated cluster of bare-metal servers, each equipped with a 2.10GHz 128-core CPU and 500GB DDR4 RAM. Currently, a canary testbed without any connected DUTs takes $k = 4$ servers to run. Unless otherwise specified, all results shown in this section are the median over 10 runs.

7.1 Connection Time

The *Crescent* partitioning scheme (§4.3) minimizes the number of cross-host links. Thus it is expected to outperform random partitioned testbeds when connecting DUTs. The connection time is measured as the time to establish cross-



(a) Connection time. (b) Bootup time.

Figure 9: Connection time and testbed bootup time.

host links between two pre-existing testbeds (i.e., canary and a small testbed with DUTs).

Figure 9a shows the testbed connection time for different numbers of neighbors (i.e., aggregation level devices) to connect to the canary testbed. The geo-manual scheme almost yields the same connection time as the *Crescent* scheme, while it takes 2-3 times longer to connect the same testbed to the canary for the random scheme. This is because the random scheme has the most cross-host links. Note that connecting DUTs to the canary testbed yields a much better performance than creating a new large-scale canary testbed. For example, it only takes about 2 minutes to connect a DUT to the canary, while it takes about an hour on our platform to create a new canary testbed, resulting in a 30× performance improvement.

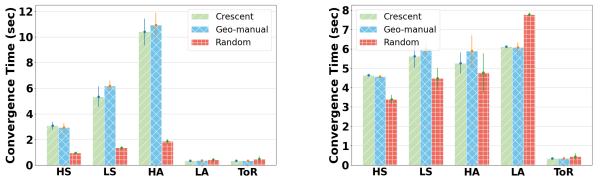
7.2 Bootup Time

We show *Crescent*'s scalability by adjusting the size (i.e. the number of nodes) of the canary testbed. We define *scaling factor* as the ratio of the size of a scaled canary testbed over the size of the original canary testbed. For example, if the original canary testbed contains N nodes, and the scaled canary testbed contains $0.75N$ nodes, then the corresponding scaling factor is 75%. When scaling down the canary testbed, we sample core devices on the spine level to ensure that the canary remains completely interconnected. When scaling up, we add more non-core devices to the canary testbed. In this experiment, we set the number of hosts proportional to the size of the scaled testbed. For example, given that each host runs about 25% portion of the original canary testbed, we use 5 hosts to run a 125% scaled testbed.

Figure 9b shows the system bootup time with different scaling factors for the three partitioning schemes. It shows only *Crescent* scheme is scaling, while both the other two stop scaling at a certain point, i.e., 100% for Random and 125% for Geo-manual. A scheme is considered not scalable when it takes too much time to boot up a new testbed. In this experiment, we set the limit to 2 hours. The reason that the Geo-manual strategy fails to scale is because there are a few DCNs whose size are much larger than the others, thus it ends up with an imbalanced partitioning scheme when adding non-core devices. *Crescent* partitioning scheme also outperforms the other two in terms of bootup time. For example, it is 20× faster to boot up the original canary testbed (i.e., 100%) with *Crescent* partitioning scheme than the random partitioning.

We need to reboot canary testbeds occasionally. For example, when there is a significant change to our production network (e.g., a new DC or a new backbone dataplane is added), we need to reshard the testbed to rebalance all the partitions. To that end, we use the proposed algorithm (§4.3) to generate a new partitioning scheme, then reboot canary testbeds in production with the new partitioning scheme one by one. During the service reboot, we want to introduce minimum impact to the online service, i.e., DUTs of planned maintenances can still connect to the running canary testbeds. In that case, *Crescent* scheme exhibits a better bootup time, about 10% faster than Geo-manual scheme. Moreover, the Geo-manual strategy stops scaling when running 125% size of the original canary testbed. This is because our global network is not geo-graphically balanced, i.e., certain regions may have more devices than others. Consequently, we may need to manually divide these regions to rebalance the network. Nevertheless, as our network expands, manually rebalancing the graph becomes increasingly challenging, even infeasible.

After booting up a canary, it's necessary to periodically update its configuration to align with the latest changes in our production network. Although more than 50% network changes occur on core devices (3), typically only a small fraction (less than 10%) of the network undergoes changes in canary. In the worst-case scenario, where the entire canary testbed needs updating, it takes an average of 208.4s to complete the process, which includes fetching, parsing, uploading, and reloading configuration in the emulated node. Given that we update the canary on a daily basis, the 3min update time is almost negligible for our platform in practice.



(a) Traffic drain. (b) Traffic undrain.

Figure 10: Convergence time with different operations.

7.3 Network Convergence Time

In this experiment, we show that the emulated network can converge quickly after applying MOP commands to different levels of devices. We test the traffic drain operation, which is one of the most commonly used operations in network maintenance. We test the operation in the canary testbed across various levels of devices, including ToR, lower level aggregation (LA), higher level aggregation (HA), lower level spine (LS), and higher level spine (HS) devices. We use a BGP listener to collect all BGP update messages from all the emulated nodes in the canary testbed. The convergence

Task Runtime (s)	Partitioning		
	Random	Geo-manual	*Crescent*
route-diff	2.31±0.2	1.58±0.11	1.57±0.19
pingmesh	2±0.5	1±0.3	1±0.5
DPV loop	52±6		
DPV reach-diff	55±4		
config check	21±1.2		

Table 2: Evaluation of proactive task performance.

time is measured as the duration from the time an operation is issued to the time the last BGP update message is received.

Figure 10a shows the convergence time for traffic drain operations on different levels of devices. Surprisingly, the random partitioning scheme converges faster than the other two schemes. The reason is that the overall convergence time largely depends on the route update processing time instead of the route propagation time. Since both *Crescent* and geo-manual scheme tend to put the geographically close nodes on the same host, and most route updates happen within the same DCN, the host that runs the nodes of the DCN becomes the bottleneck for route update processing. Whereas, in the random scheme, the nodes that get affected by this operation are spread across all hosts, thus can process the route updates much faster with more resources. Figure 10b also shows the convergence time for undraining traffic after maintenance is done on the device. Overall, as depicted in both figures, the emulated network can converge very quickly, i.e., within 10s. Compared to other overhead, such as testbed connection time, the network convergence overhead is negligible.

7.4 Verification Time

We conducted four experiments, each with eight DUTs. After connecting the DUTs and network convergence, we execute multiple verification and monitoring tasks. Table 2 displays the average runtime for all experiments across all partitioning schemes. For DPV, we list the result of loop check and reachability difference (reach-diff) tasks. The reach-diff tasks compute end-to-end reachability differences for all network traffic classes. Since DPV tasks model and analyze O(100K) routes for O(1K) devices, they have a longer runtime than other tasks. The config checker has the next longest runtime. These centralized tasks fetch and send route/configs to a verification service and remain agnostic to the partitioning scheme. Route diff and pingmesh are lightweight, distributed tasks executed inside the containers, resulting in a faster runtime. Both the *Crescent* and geo-manual schemes exhibit similar runtime results for these tasks. And the random scheme shows a slight increase in runtime. Overall, it takes approximately a minute to verify the network for all tests.

8 Beyond Network Change

Crescent's primary mission was to prevent change-induced network incidents. Meanwhile, we have found *Crescent* helpful in many other use cases.

8.1 Catching Regression Under Failures

Some network changes may not trigger immediate impact until certain failures (e.g., link down) events occur. This has led to a few outages in the past. To expose such *deferred* impact from a network change, we run periodic workflows on *Crescent* to catch regressions in the network for various failure scenarios. We import the latest configurations from production networks and inject failure events such as a node or link down for critical parts of the network. Some significant errors we have identified so far include:

1. A border device missing a critical configuration that could lead to routing loops under specific link failures.
2. A global WAN device missing routes for specific regions due to misconfiguration. This could cause blackholes in case of another WAN device failure.
3. A newly-added configuration having incorrect community commands that could redirect a large volume of traffic to the management device, leading to congestion.

8.2 Self-Service Platform

Crescent provides rich user interfaces, including a web portal and chatbot, to assist users in creating emulation environments for any purpose.

Network Design Evaluation. When building a new network or evolving an existing one, our planners need to evaluate multiple design choices and conduct feasibility studies before coming up with the final MOPs. They leverage *Crescent* to create large testbeds, which would cost much higher if using physical devices. In one case, they found tens of bugs in their configuration templates.

Device Behavior Comparison. Cautious about VSB, our operators use *Crescent* to test the vendor software for non-standardized behaviors. One of these tests helped us discover VSB regarding BGP route aggregation. We observed that the inherited attributes for the aggregate route differed among the vendors. For example, some did not inherit the AS-PATH at all, while some inherited the common AS numbers.

Incident Reproduction. *Crescent* serves as a learning platform to reproduce past incidents, which is used to refresh our memory and educate new hires. Such training helps us avoid repeating similar mistakes from the past.

8.3 SDN Testing

SDN-based traffic engineering (TE) [36,39,45,54] has played an important role in our networks. A high-fidelity testbed is needed to test the correctness of the TE software, including the controllers and their dependent components. Small-scale physical testbeds are helpful but cannot scale to the same size as the production network. Besides, multiple TE projects may

compete for limited physical testbeds, with high overhead to reset the environment during handover. Instead of waiting for days to use physical testbeds, it takes minutes for TE developers to create a production-like environment using *Crescent* for both component and integration tests of TE software.

9 Discussion

Crescent has been deployed in ByteDance since 2020. It has been serving tens of network change requests per week. This section shares our lessons learnt from its deployment and our thoughts for future work.

9.1 Lessons Learned

Road To Enforcement. *Crescent* has been integrated with our production network change workflow since its early days. But as shown in Figure 1b, it took almost two years from the initial rollout of *Crescent* to enforce it as a mandatory step for changing critical devices in production networks. Other than resource constraints, the major blockers were the lack of vendor images and the unfriendliness of user interfaces. We should partner with switch vendors earlier to develop and qualify the images as a collaborative effort. And we should involve our users more closely when designing user interfaces.

Unknown Image Limitations. Switch vendor images are blackboxes to us. Many of their limitations are unknown or undocumented. Some of these limitations may even manifest in a surprising way. For example, when we created a certain amount of ports in a container running a vendor image, its LLDP process started crashing while its layer 3 and above protocols, including BGP, worked fine. We did not notice the issue until the host ran out of memory. It turned out that the vendor used memory to store all the logs, and the huge-sized LLDP crash logs exhausted the host memory.

Fidelity Can Backfire. *Crescent*'s mission is to prevent network incidents. But as it is deployed in production hosts, failure to isolate from the production network could trigger unexpected consequences. For example, each emulated device connects to the host via a management or CPN interface through the docker bridge. Docker would add a default route automatically for such interfaces. In one case, we did not remove such a default route, and the emulated device sent syslog messages to the production collector, which could not distinguish it from the production device. This caused false alarms as the syslog reported many down links. It cost our network operators tremendous efforts to figure it out. We added the fix to remove the default route from each container and disable the syslog configuration during adaptation.

Kernel Tuning Can Help. In addition to employing multiple hosts, we explored increasing the number of containers for each host to support emulating larger networks. But, we failed to create a large number of containers while both CPU and memory usage were well below the host limits. After digging into the container logs, we spotted an error message and realized certain kernel settings for *inotify* should be relaxed.

9.2 Incidents Missed

The deployment of *Crescent* has significantly reduced the number of incidents, as demonstrated in Figure 1b. Emulation and verification using *Crescent* has helped detect and prevent various errors from occurring in production, e.g., configuration syntax errors and routing loops. However, there are still some network incidents that cannot be captured adequately with *Crescent*. For instance, we cannot identify issues related to traffic dependencies. One such incident was caused by hash polarization or imbalance, emerging from an unevenly balanced traffic load after being hashed twice or more. Additionally, high-level Quality of Service issues such as congestion, latency, and jitter can cause traffic-related complications. Emulation also lacks the potential to capture hardware-related issues. These include common problems like device failures, link flapping, etc, and rare issues like reduced Optical Transport Network (OTN) capacity leading to packet losses, etc. Moreover, in some cases, incidents may result from other components interacting with the network and its configuration, such as malfunctioning management software like a Netconf controller. While we have some regression tests with failures, these still need to be exhaustive, as certain failures could lead to never-emulated scenarios. Nevertheless, we constantly look for potential problems and enhance our emulation testing pipeline.

9.3 Future Work

Crescent does not attempt to search the tight safe boundary, as we think the correct boundary, if it exists, depends on certain assumptions of the routing policies. One idea to explore in future is to derive the routing policies from production network configurations, confirm them with network operators, and then add invariant checks for these policies. The derived routing policies can help guide the search of the boundary. Selecting the appropriate boundary is one method of reducing emulation resources without impacting fidelity. Another option is to investigate reducing the size of the emulation image by customizing it to include only relevant routing features.

Crescent presents network emulation as an alternative to CPV, but both can be used in combination to achieve high fidelity and performance. CPV can model specific aspects of the network, while emulation can emulate the remaining components to incur minimal performance overhead.

The baseline topology may evolve, e.g., adding more core devices to the production network. Today *Crescent* has to recreate the canary testbed with re-partitioning. We are exploring the possibility of updating the canary testbed dynamically with minimum overhead.

We plan to quantify *Crescent*'s fidelity by comparing the data plane of the emulated network with the one in production. The key challenge is that the data plane in production is more dynamic due to link flaps, controller programming, etc.

Finally, we anticipate that the number of emulation jobs will increase over time on multiple canaries. The scheduler plays a critical role in determining which job to schedule first. We are exploring using different scheduling strategies to handle incoming requests so that we can handle those requests not merely based on the request order, but also based on other factors, e.g., request emergency level.

10 Related Work

Network Emulation. Network emulation allows network operators to evaluate and design networking solutions. In today's landscape, commercial and open source network emulators, such as EVE-NG [4], GNS3 [8], DockerTopo [3], Mininet [40], Maxinet [53], vrnetlab [10], and SEED [23], are widely adopted by network operators for small scale network tests. Nevertheless, these emulators prove inadequate when it comes to large-scale network emulation involving thousands of nodes, primarily due to their lack of features such as dynamic lab formation (e.g., automatic node and link creation) and automatic configuration adaptation. CrystalNet [41] is the closest work to ours, which enables a large scale emulation for DC networks. Our work is inspired by CrystalNet, but with several enhancements on top. First, we find the algorithm to identify a safe boundary proposed in [41] is not safe for us. As a result, we choose to always include core devices in emulation. Second, we propose a greedy algorithm to address the graph partitioning problem when running a large-scale testbed using a multihost setup. Note that running a large-scale testbed with a better partitioning scheme can benefit any multi-host setup, no matter with VMs on a cloud infrastructure or with baremetal servers. In another word, the deployment strategy (where to run) addressed by CrystalNet is orthogonal to the partitioning scheme (how to run). Last, we made other efforts to enhance system automation for *Crescent*, e.g., combining emulation with automatic verification, config adaptation, etc. In summary, all of these emulators lack the features required for automatic timely verification through high-fidelity large-scale emulation, such as cross-vendor adaptation, dynamic link/node alterations for connecting DUT, etc.

Network Verification. Over the years, researchers have developed many DPVs and CPVs. DPVs usually analyze devices' forwarding rules and detect reachability policy violations. Recently researchers have added optimizations like making them incremental based on data plane updates [14,34,58], partitioning packets into equivalence classes to reduce the input search space [35, 58], parallelizing verification by dividing global properties to local checks [37], etc. CPVs usually model the network control plane to detect similar reachability violations. Similarly, researchers have added optimizations like making them incremental based on configuration updates [57], modeling policies as graph properties [12], using Packet and Failure Equivalence Class [59] and abstract interpretation [15, 16, 30] to reduce the input search space, etc. Hoyan [55] is a verifier that aims to model vendor software behavior. It uses black-box testing to detect modeling deficiency but requires manual intervention to correct the model. We can integrate *Crescent* with any of these verifiers.

Configuration Management. Operators have made many attempts to automate configuration management. Network synthesizers use high-level policy intents to produce policy-compliant configurations. Some [17, 24] create brand-new configurations from scratch. Whereas others [11, 25, 46] incrementally update the existing configurations. Some are very platform-specific, like Facebook's configuration management and syntax generation tool called Robotron [50]. NAssim [20] creates configuration models from device user manuals using NLP and deep-learning techniques. However, these are still limited in terms of feature coverage. *Crescent* can operate orthogonally to assist these tools' development for troubleshooting, debugging, etc.

11 Conclusion

Network changes are a major source of incidents. As a solution, we have developed *Crescent*, a large-scale high-fidelity emulation platform coupled with timely verification and monitoring tools. Instead of emulating the entire network, we exploit the symmetry and modularity of data center networks and build canary testbeds that emulate all core devices and sampled non-core devices. To achieve high scalability, we use a multi-host setup and propose a graph partitioning algorithm for a scalable node-to-host assignment to reduce the overhead due to cross-host links. We support dynamic link addition/removal to allow expansion/modification to the canary testbed on the fly. Our experience running *Crescent* has demonstrated its effectiveness in detecting problematic network changes, especially those resulting from undocumented vendor-specific behaviors.

Acknowledgements

We would like to thank many ByteDance colleagues who have contributed to this work. These include Jianghang Ning, Puyi Dang, Xiang Zhao, Yanlin Qiao, Feiyu Yue, Yuchen Shi, Hui Liu, Yifeng Zhou, Beixin Huang, Anjian Chen, Changqing Zhu, Henry Han, Weibin Wang, Amandeep Singh, Jianfei Cheng, Yanwen Chen, Jianwei Hu, Dong Zhao, Weibin Shen, Huang Zhang, Jintao Liu, Juntao Zhong, Gen Li, Bo Cao, Rohit Khairnar, Navneet Nagori, Wenhao Wang, Sisi Wen, Rui Du, Haotian Deng, Yiyang Chang, Yantong Meng, Heng Qiu, Yihan Zou, Yibo Zhu, Chuanxiong Guo, and many others. We would also like to thank Peng Zhang at XJTU, our reviewers, and our shepherd Guyue Liu, for their helpful comments.

References

[1] Intermediate System to Intermediate System (IS-IS) Extensions for Traffic Engineering (TE). https://datatracker.ietf.org/doc/html/rfc3784, June 2004.

[2] QEMU Documentation/Networking. https://wiki.qemu.org/Documentation/Networking, 2019. Last accessed on February 13, 2023.

[3] docker-topo. https://github.com/networkop/docker-topo, 2020. Last accessed on February 13, 2023.

[4] EVE-NG. https://www.eve-ng.net/, 2021. Last accessed on Sep 13, 2023.

[5] Hyperscale Data Center Count Grows to 659 – ByteDance Joins the Leading Group. https://www.srgresearch.com/articles/hyperscale-data-center-count-grows-to-659-bytedance-joins-the-leading-group, 2021. Last accessed on Sep 14, 2023.

[6] KVM guest virtual network configuration using MacVTap. https://www.ibm.com/docs/en/linux-on-systems?topic=configurations-kvm-guest-virtual-network-configuration-using-macvtap, 2021. Last accessed on Sep 13, 2023.

[7] Cloudflare Radar. https://radar.cloudflare.com, 2023. Last accessed on Sep 14, 2023.

[8] GNS3. https://www.gns3.com/, 2023. Last accessed on Sep 13, 2023.

[9] MaxiNet. https://maxinet.github.io/, 2023. Last accessed on Sep 4, 2023.

[10] vrnetlab. https://github.com/vrnetlab/vrnetlab, 2023. Last accessed on Sep 13, 2023.

[11] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Aed: Incrementally synthesizing policy-compliant and manageable configurations. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 482–495, 2020.

[12] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 201–219, 2020.

[13] Anubhavnidhi Abhashkumar, Kausik Subramanian, Alexey Andreyev, Hyojeong Kim, Nanda Kishore Salem, Jingyi Yang, Petr Lapukhov, Aditya Akella, and Hongyi Zeng. Running bgp in data centers at scale. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 65–81, 2021.

[14] Ryan Beckett and Aarti Gupta. Katra: Realtime verification for multilayer networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 617–634, 2022.

[15] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 476–489, 2018.

[16] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Abstract interpretation of distributed network control planes. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–27, 2019.

[17] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication*, pages 328–341, 2016.

[18] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[19] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd Millstein. Lessons from the evolution of the batfish configuration analysis tool. In *Proceedings of the 2023 Conference of the ACM Special Interest Group on Data Communication*, pages 122–135, 2023.

[20] Huangxun Chen, Yukai Miao, Li Chen, Haifeng Sun, Hong Xu, Libin Liu, Gong Zhang, and Wei Wang. Software-defined network assimilation: bridging the last mile towards centralized network configuration management with nassim. In *Proceedings of the 2022 Conference of the ACM Special Interest Group on Data Communication*, pages 281–297, 2022.

[21] Patrick Ciarlet Jr and Françoise Lamour. On the validity of a front-oriented approach to partitioning large sparse graphs with a connectivity constraint. *Numerical Algorithms*, 12(1):193–214, 1996.

[22] Marek Ciglan and Kjetil Nørvåg. Fast detection of size-constrained communities in large networks. In *International Conference on Web Information Systems Engineering*, pages 91–104. Springer, 2010.

[23] Wenliang Du, Honghao Zeng, and Kyungrok Won. Seed emulator: an internet emulator for research and education. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 101–107, 2022.

[24] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Network-wide configuration synthesis. In *International Conference on Computer Aided Verification*, pages 261–281. Springer, 2017.

[25] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 579–594, 2018.

[26] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 05)*, pages 43–56, 2005.

[27] Andrew D Ferguson, Steve D Gribble, Chi-Yao Hong, Charles Edwin Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, et al. Orion: Google's software-defined networking control plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 83–98, 2021.

[28] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Bruno Decraene, Stephane Litkowski, and Rob Shakir. Segment routing architecture. Technical report, 2018.

[29] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, May 2015.

[30] Nick Giannarakis, Ryan Beckett, Ratul Mahajan, and David Walker. Efficient verification of network fault tolerance via counterexample-guided refinement. In *International Conference on Computer Aided Verification*, pages 305–323. Springer, 2019.

[31] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002.

[32] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication*, pages 58–72, 2016.

[33] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 Conference of the ACM Special Interest Group on Data Communication*, pages 139–152, 2015.

[34] Dong Guo, Shenshen Chen, Kai Gao, Qiao Xiang, Ying Zhang, and Y Richard Yang. Flash: fast, consistent data plane verification for large-scale network settings. In *Proceedings of the 2022 Conference of the ACM Special Interest Group on Data Communication*, pages 314–335, 2022.

[35] Alex Horn, Ali Kheradmand, and Mukul Prasad. Delta-net: Real-time network verification using atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 735–749, 2017.

[36] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[37] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, et al. Validating datacenters at scale. In *Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication*, pages 200–213. 2019.

[38] Siva Kesava Reddy Kakarla, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir, and George Varghese. Finding network misconfigurations by automatic template inference. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 999–1013, 2020.

[39] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. Decentralized cloud wide-area network traffic engineering with blastshield. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 325–338, 2022.

[40] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6, 2010.

[41] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 599–613, 2017.

[42] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference 2018*, pages 393–407, 2018.

[43] Gordon D Plotkin, Nikolaj Bjørner, Nuno P Lopes, Andrey Rybalchenko, and George Varghese. Scaling network verification using symmetry and surgery. *ACM SIGPLAN Notices*, 51(1):69–83, 2016.

[44] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[45] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication*, pages 418–431, 2017.

[46] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. Snowcap: synthesizing network-wide configuration updates. In *Proceedings of the 2021 Conference of the ACM Special Interest Group on Data Communication*, pages 33–49, 2021.

[47] John Scudder, Rex Fernando, and Stephen Stuart. Bgp monitoring protocol (bmp). Technical report, 2016.

[48] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 45(4):183–197, 2015.

[49] Kotikalapudi Sriram, Doug Montgomery, D McPherson, Eric Osterweil, and Brian Dickson. Problem definition and classification of bgp route leaks. Technical report, 2016.

[50] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication*, pages 426–439, 2016.

[51] Alan Tang, Ryan Beckett, Steven Benaloh, Karthick Jayaraman, Tejas Patil, Todd Millstein, and George Varghese. Lightyear: Using modularity to scale bgp control plane verification. In *Proceedings of the 2023 Conference of the ACM Special Interest Group on Data Communication*, pages 94–107, 2023.

[52] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev.*, 36(SI):271–284, dec 2003.

[53] Philip Wette, Martin Dräxler, Arne Schwabe, Felix Wallaschek, Mohammad Hassan Zahraee, and Holger Karl. Maxinet: Distributed emulation of software-defined networks. In *2014 IFIP Networking Conference*, pages 1–9. IEEE, 2014.

[54] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication*, pages 432–445, 2017.

[55] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, et al. Accuracy, scalability, coverage: A practical configuration verifier on a global wan. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 599–614, 2020.

[56] Ken Yocum, Ethan Eade, Julius Degesys, David Becker, Jeff Chase, and Amin Vahdat. Toward scaling network emulation using topology partitioning. In *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003.*, pages 242–245. IEEE, 2003.

[57] Peng Zhang, Aaron Gember-Jacobson, Yueshang Zuo, Yuhao Huang, Xu Liu, and Hao Li. Differential network analysis. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 601–615, 2022.

[58] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. Apkeep: Realtime verification for real networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 241–255, 2020.

[59] Peng Zhang, Dan Wang, and Aaron Gember-Jacobson. Symbolic router execution. In *Proceedings of the 2022 Conference of the ACM Special Interest Group on Data Communication*, pages 336–349, 2022.

[60] Huaiyi Zhao, Xinyi Zhang, Yang Wang, Zulong Diao, Yanbiao Li, and Gaogang Xie. Improving the scalability of distributed network emulations: an algorithmic perspective. *IEEE Transactions on Network and Service Management*, 2023.

A Network Emulation

In this part, we describe the detials about how we emulate network devices, how we wire these emulated devices via various types of virtual interfaces (§A.2), and how we emulate edges (§A.3) and control plane network (§A.4) with *Crescent*.

A.1 Emulating Network Devices

Most of our switch vendors only provide VM images, while a couple also provide container images. Table 1 outlines all the vendors' images supported by *Crescent*, along with the least resources (e.g., CPU, memory) required to boot up. For vendors that support both VM and container images, we use container images as default because they are more lightweight than VM ones. However, if a device uses functions not supported by container images but supported by VM images, e.g., Segment Routing [28], we will emulate it with the VM image. Some vendors provide multiple VM images of different specs. For example, *v4* provides two VM images: one supporting more ports while consuming more resources. Thus before emulating a device of *v4*, we count the number of ports the device needs to emulate. If it is below the limit, we use the image with a lower resource requirement.

Native Vendor Image Support. *Crescent* uses containers as the basic units to mock up network devices [3,4,41]. *Crescent* treats VM images the same as container images by wrapping the VM image, a KVM hypervisor, and other libraries into a container image. We host a centralized image registry for hosts to pull the images from. By making this implementation choice, we are able to manage all emulated devices using a single underlying runtime system. Specifically, *Crescent* uses the Docker engine to manage all images and containers.

A virtual network interface attached to a container with a VM running inside are mapped to an interface of the VM via a MacVTap interface [6], which forwards ingress traffic to or egress traffic from the guest OS's corresponding virtual interface without traversing guest OS's kernel (e.g., container 1,2, and 3 in Figure 5). Inside the container, we use virtio [44] as the default network interface adapter attached to the guest VM to achieve a better performance. However, due to compatibility issue, some vendor's image only supports obsolete network adapters such as e1000 [2].

While there may be some additional overhead in wrapping a VM image into a container and routing traffic through virtual interfaces compared to running guest VMs directly on the host, this overhead is negligible in comparison to the time it takes for network convergence. For example, our evaluation using vendor *v3* shows that the additional overhead resulted in about 2×10^{-4}s of end-to-end ping latency, while network convergence could take several seconds.

A.2 Emulating Network Links

We use different virtual network interfaces to wire the emulated devices. We emulate a link on the same host by a veth pair directly connecting two virtual interfaces attached to two emulated devices. We must put a virtual network interface attached to a container running with a container image into the container's namespace before its containerized switch OS boots up [41]. Otherwise, the switch OS cannot recognize the virtual network interfaces. To deal with this issue, *Crescent* pauses a container immediately after the container starts running so that its network namespace is ready to attach virtual interfaces before starting the OS.

We setup a virtual link across two hosts with a VXLAN tunnel via the OVS bridge. One end of a veth pair is attached to the emulated device container network namespace, and the other end is attached to the OVS bridge. OVS is responsible for encapsulating and decapsulating packets across hosts. We always reserve the first virtual interface (i.e., eth0 in Figure 5) for the management port of the emulated device. The management port's virtual interface is then attached to the host's default Docker bridge, which is exposed for users to log into the emulated device via SSH or Web UI.

Figure 5 illustrates how we wire devices on the same host and across hosts. Containers running a guest switch VM inside (e.g., containers 1-3) use MacVTap to pipe traffic from the virtual interfaces attached to the VM to the virtual interface attached to the container or vice versa. For a container booted up from a container-based image, e.g., container 4, an end of the veth pair is attached to the container directly.

A.3 Emulating Edges

We consider three types of network elements at the edge: ISP, cloud, and servers under ToR, over which we have no, partial, and complete control in reality, respectively.

ISPs and clouds peer with our border devices via BGP. For the minimum requirements, *Crescent* must be able to inject the routes received from the edges into our network to ensure some properties, e.g., the traffic to certain IP prefixes must be routed through the gateway between our network and a public cloud rather than the gateway to an ISP's network. We have a partial control over the cloud peers (such as the ability to specify route policies), while no control over the ISP peers. As mentioned in [41], one way to replay the routes received from cloud peers is to inject these routes directly into our border devices using BGP speakers. However, the simplicity of this approach comes at a cost, as it can only be used to replay the routes announced from the edge to our network, but not vice versa. For example, if we want to observe the routes announced from our network to cloud peers (e.g., to avoid route leaks [49] from us), we would have to employ another component to collect the routes announced to the cloud peers, then analyze the collected routes later.

Crescent emulates these edge peers using a lightweight switch container image, as opposed to the proposed approach in [41]. *Crescent* adds loopback and static prefixes to an emulated device (which need not be identical to the real-world cloud peers) to establish BGP connections with our border devices and then configures these devices to announce the routes. By doing this, we are also emulating the cloud peer's behavior in addition to merely injecting routes to our border devices, which allows us to capture not only the changes in FIB of our border devices but also those of the emulated cloud peers with our proactive monitoring tools (§6). Besides, this approach allows us to emulate and validate route policies over which we have control on some public cloud peers in our emulation environment.

In a production DC, there are tens of servers under each ToR, where we have full control of their network stack. For ToR level devices, *Crescent* connects at least one virtual server to it to run proactive monitoring such as pingmesh (§6). These virtual servers are emulated using lightweight Linux images by default. We also support running switch images in these virtual servers, to help emulate the scenarios where advanced users establish BGP sessions between a server and the ToR.

A.4 Emulating Control Plane Network

Our networks are impacted not only by device configuration changes, but also by traffic engineering (TE) controllers that install BGP routes to override the default routing behaviors. It is critical to ensure the reliability of the software implemented for the TE controllers and their dependent components, e.g., BGP speakers for route announcement and BGP Monitoring Protocol (BMP [47]) collector for route collection. Instead of competing for limited physical testbeds, we utilize *Crescent* to provide a testing environment for these SDN software. One way is to containerize these SDN software and then let them establish connections to emulated devices through the emulated network. Nonetheless, this in-band approach requires TE developers to wrap their software into container images, which takes extra efforts if the software was not originally developed for a containerized deployment. Also it is not resource-efficient to emulate the in-band network while the controller manages only a small portition of the network. *Crescent* provides a more flexible option by using a dedicated Linux bridge to emulate Control Plane Network (CPN) [27]. CPN can be used to establish out-of-band connections between emulated devices and software components in a way that is transparent to the software under test. This method eliminates the need for the SDN software to be enclosed in containers, thereby reducing the amount of time required for our software developers to set up the test.

B Multi-Host Partitioning Algorithm

Algorithm 1 shows the pseudocode of the heuristic multi-host partitioning algorithm proposed in §4.3.

C Topology Expansion

Algorithm 2 presents our topology expansion algorithm. This algorithm primarily takes the devices under change and *expansion_width* (w) as input and returns the emulated nodes as output. It uses a variable N to track all traversed nodes and map them to their traversed neighbors.

The algorithm first expands the topology upwards (line 5) and downwards (line 6) to add upstream and downstream neighbors. This expansion uses a modified depth-first search (DFS) algorithm called MV_DFS (line 11). MV_DFS stands for multi-vendor DFS. It is a directional DFS to expand the network by w and add all required vendors per level. For each node, MV_DFS does not explore all the node's neighbors. It terminates when it has added at least w neighbors and ensures they represent all the unique vendors attached to this node (line 28). The algorithm customizes the for-loop traversal in line 21 in two ways. It prioritizes nodes already selected for emulation. This strategy ensures we reduce the number of expanded nodes. And it also prioritizes nodes belonging to vendors that it has yet to explore in that loop. This strategy aims to satisfy the condition in line 28 as soon as possible. During MV_DFS, it adds nodes (line 22) representing either the leaf or top node or those representing nodes with newly explored neighbors (line 26).

Next, the algorithm expands the network horizontally (line 7). It ensures that for each device under change, we add other nodes from all vendors that belong to the same level (lines 33-34). Again, the algorithm customizes this to prioritize nodes already selected for emulation.

Algorithm 1: Randomized Multi-Host Partitioning

Preprocess: Generate an undirected weighted graph (V, E). For every partition p, we have a corresponding node p' in V. And for every pair of partitions p_1, p_2, we have $e = edge(p'_1, p'_2)$ in E and $e.weight$ = # of links between p_1 and p_2. We also define the cost of a node p': $p'.cost$ = sum(memory_cost of each node in p).
Input: $V, E, partition_memory_limit$
Output: map($v \to n$) for each v in V `// map each node to a partition number`

$uf \leftarrow$ union_find(V); `// assign each` v `in` V `to a new set`
$non_zero_edges \leftarrow [e$ for e in E if $e.weight \neq 0]$;
$zero_edges \leftarrow [e$ for e in E if $e.weight = 0]$;
`// Instead of always choosing the edge with the largest weight,`
`// we sample(without replacement) from a distribution based on the edge weight.`
$non_zero_edge_distribution \leftarrow$ distribution over non_zero_edges, $p(e_i) = e_i.weight/sum$;
$zero_edge_distribution \leftarrow$ distribution over $zero_edges$, $p(e_i) = \sqrt{e_i.src.cost \times e_i.dst.cost}/sum$;
while $non_zero_edge_distribution$ *is not empty or* $zero_edge_distribution$ *is not empty* **do**
  `// Merge the edges with non-zero weight first. If all remaining edges has zero weight,`
  `// use the geometric mean of` $src.cost$ `and` $dst.cost$ `as weight.`
  `//` $sample_without_replacement$`: with replacement, a value can be selected multiple times.`
  `// For each edge we only visit it once, so we use` $without_replacement$ `here.`
  if $non_zero_edge_distribution$ *is not empty* **then**
    $e \leftarrow$ $non_zero_edge_distribution$.sample_without_replacement();
  else
    $e \leftarrow$ $zero_edge_distribution$.sample_without_replacement();
  $src_root \leftarrow uf$.get_root($e.src$);
  $dst_root \leftarrow uf$.get_root($e.dst$);
  if $src_root = dst_root$ **then**
    continue;
  if $src_root.cost + dst_root.cost > partition_memory_limit$ **then**
    continue;
  $new_root \leftarrow uf$.union($e.src$, $e.dst$);
  $new_root.cost \leftarrow src_root.cost + dst_root.cost$
$m \leftarrow$map();
for v *in* V **do**
  $m[v] \leftarrow$ indexof(uf.get_root(v));
return m

Algorithm 2: Topology Expansion

Global inputs: S: devices under change, w: expansion width;
Global output: T: emulated devices, initialized to S;
Global variables: N: map node to a set of its neighbors;
procedure *TOPO_Expansion()*
  EXPAND_VERTICAL(up);
  EXPAND_VERTICAL(down);
  EXPAND_HORIZONTAL();
Input: dir: direction (up or down);
procedure *EXPAND_VERTICAL(dir)*
  for $s \in S$ **do**
    MV_DFS(s, dir);
    if N *has changed* **then**
      for n *in new nodes in* N **do**
        T.add(n);
Input: s: visiting node;
procedure *MV_DFS(s, dir)*
  if $s \in N$ **then**
    return
  $newN \leftarrow \{\}$;
  $D \leftarrow s$.neighbors(dir);
  for $n \in D$ **do**
    `// The for loop will prioritize nodes already in` T `and belonging to untraversed vendors.`
    if n *is top or leaf node* **then**
      $newN$.add(n);
    else
      MV_DFS(n, dir);
      if N *has changed* **then**
        $newN$.add(n);
    if $size(newN) \geq w$ ***and*** $vendors(newN) = vendors(D)$ **then**
      break;
  $N[s] \leftarrow newN$;
procedure *EXPAND_HORIZONTAL()*
  for $s \in S$ **do**
    `// Prefer selecting` n `already in` $T - D$
    if $L(s) \neq \emptyset$ *s.t. L represents other nodes at same level* **then**
      $\forall v \in vendors, \exists n \in L(s, v)$ s.t. T.add(n);

Reasoning about Network Traffic Load Property at Production Scale

Ruihan Li[ℙ𝔸], *Fangdan Ye*[𝔸], *Yifei Yuan*[𝔸], *Ruizhen Yang*[𝔸], *Bingchuan Tian*[𝔸], *Tianchen Guo*[𝔸], *Hao Wu*[𝔸], *Xiaobo Zhu*[𝔸], *Zhongyu Guan*[𝔸], *Qing Ma*[𝔸], *Xianlong Zeng*[𝔸], *Chenren Xu*[ℙ], *Dennis Cai*[𝔸], *Ennan Zhai*[𝔸]
[ℙ]*Peking University* [𝔸]*Alibaba Cloud*

Abstract

This paper presents JINGUBANG, the first reported system for checking network traffic load properties (*e.g.*, if any link's utilization would exceed 80% during a network change) in a production Wide Area Network (WAN). Motivated by our network operators, JINGUBANG should meet three important requirements: (**R1**) comprehensive support for complex traffic behavior under BGP, IS-IS, policy-based routes (PBR), and segment routes (SR), (**R2**) reasoning on traffic load of billions of flows across a period of time, (**R3**) real-time failure-tolerance analysis. These requirements pose challenges in modeling the complex traffic behavior and maintaining the checking efficiency. JINGUBANG has successfully addressed these challenges. First, we propose the traffic distribution graph (or TDG), capable of modeling equal-cost multi-path (ECMP), packet rewriting, and tunneling, introduced by BGP/IS-IS, PBR, and SR, respectively. Second, we design an algorithm based on TDG to simulate traffic distribution for billions of flows across a time period both efficiently and accurately. Third, JINGUBANG proposes an incremental traffic simulation approach that first computes an incremental TDG and then simulates only the *differential* traffic distribution, avoiding the need to simulate the entire network traffic distribution from scratch. JINGUBANG has been used in the daily checking of our WAN for more than one year and prevented service downtime resulting from traffic load violations.

1 Introduction

Alibaba Cloud serves over one billion customers with its services including cloud computing, search, and video. To support these services, we operate a global Wide Area Network (WAN) infrastructure to interconnect tens of data centers. Our WAN is a traditional, distributed network, rather than an SDN network. This WAN maintains hundreds of routers and forwards a huge amount of service traffic by combining diverse protocols including BGP, IS-IS, policy-based routes (PBR), and segment routes (SR). According to our recent three-year records, more than 90% of outages caused by misconfigurations on our WAN were related to traffic load violations.[1]

Checking whether network traffic load meets our variety of specifications, therefore, is vital to the availability and reliability of our WAN. For example, in the scenario of planned network changes (*e.g.*, updating configurations and upgrading network routers), our network operators need to ensure that no link would be overloaded at any point during the entire planned change time window. As another example, our operators need to check whether any traffic load property (*e.g.*, no drastic traffic load increase on any link) may be violated if given links fail. Conventional techniques (*e.g.*, traffic engineering [4, 11, 17, 19, 25, 26]) do not help since they mainly focus on optimizing and controlling traffic load; however, what we need is a system capable of answering our queries about specified traffic load properties.

Requirements. We decided to build a system to model the traffic behavior and *proactively* check traffic load violations in our WAN. By surveying our operators, a practical checking system must meet the following requirements.

R1: Comprehensive protocol support. The checker should support comprehensive traffic behavior under BGP, IS-IS, PBR, SR, and static routes. Our WAN uses BGP and IS-IS for routing; our traffic scheduling and engineering heavily rely on PBR and SR, due to their precise traffic control and low operation cost. For example, SR can specify forwarding paths on a per-hop basis, while PBR can identify QoS and classify traffic into different classes.

R2: Reasoning on traffic load for billions of flows in a period of time. The checker should support the traffic load reasoning not only at a single time point, but also during a period of time. In a network change scenario, the change time window may last multiple hours; thus, our operators need to check whether the intended traffic load properties can hold during the entire change window. In addition, billions of flows can appear during the time period; thus, the checker must be scalable to reason on the traffic load of billions of flows.

R3: Real-time failure-tolerance analysis. Our operations frequently run failure-tolerance analysis to check if a set of failed routers and links would cause traffic load violations. The checker should offer an efficient what-if analysis to meet the real-time requirement of our operators.

State of the arts. No prior work, nevertheless, can simultaneously satisfy all the above requirements. Specifically, current network verification systems have been focused on checking reachability properties (*e.g.*, if packets/route advertisements sent from a router A can reach another router B) in terms of control plane [1, 2, 10, 12, 14, 15, 22, 31, 32, 34, 39–42] and data plane [3, 18, 20, 21, 23, 24, 28, 30, 35, 37]; thus, they are unable to reason about traffic load properties in the network.

A recent effort, QARC [36], closest to our goal, proposes a verification approach to checking whether links may be overloaded under any failure up to a given degree (*i.e.*, *k*-failure

[1] Since Hoyan [41] was used three years ago, Hoyan has prevented most misconfigurations causing routing reachability errors in our network. The remaining misconfiguration-related outages, in recent three years, happened tens of times a year and accounted for about 30% of all our network outages.

tolerance reasoning for network traffic load). QARC, however, cannot meet our requirements. First, QARC's encoding is based on the shortest path with flow quantities. It is non-trivial to extend the encoding to support PBR and SR, which introduces traffic behavior beyond shortest-path-forwarding, *e.g.*, packet rewriting and tunneling. Second, the verification algorithm of QARC mainly focuses on traffic load reasoning at a single time point. Extending the algorithm to *efficient* verification of traffic load in a period of time is not straightforward. Finally, QARC's main focus is k-failure tolerance typically with small k, it is reported that it cannot scale to a larger number of failed links/routers (*e.g.*, $k > 5$) [36].

Our approach: JINGUBANG.[2] In this paper, we present JINGUBANG, the first system for checking network traffic load properties for WANs. JINGUBANG simultaneously achieves the above-mentioned requirements proposed by our operators. In essence, JINGUBANG is a system that can simulate traffic distribution (*i.e.*, traffic load on each link) for traffic load property reasoning. Specifically, JINGUBANG takes as inputs (i) the network topology and routers' configurations, (ii) IP prefixes advertised into the network, and (iii) multiple traffic snapshots each recording the traffic information of a set of flows entering the network at a time point, then accurately simulates the traffic distribution, and finally checks whether it meets the given traffic load property.

Building JINGUBANG requires us to address three challenges (corresponding to the three requirements, respectively). First, the traffic behavior under BGP, IS-IS, PBR, and SR is complex. In particular, SR relies on *tunneling* and PBR offers fine-grained flow processing such as *packet rewriting*. In addition, routing protocols (*e.g.*, BGP, IS-IS) enforce protocol-based *equal-cost multipath* (ECMP), where traffic is load-balanced first between BGP next hops and then between IS-IS next hops. To comprehensively model the traffic behavior under those protocols, we introduce the *traffic distribution graph* (or TDG), translating the traffic behavior of different protocols into a uniform traffic forwarding representation. TDG can represent not only destination-based forwarding behavior (like BGP and IS-IS) with ECMP but also tunneling and packet rewriting used in SR and PBR. (§4.1)

The second challenge is efficiently reasoning on traffic load for billions of flows in a period of time. In our operation, the time period to be checked typically lasts several hours, involving a huge number of time points (1 minute for each in our settings). A strawman solution is to generate many TDGs, each corresponding to a time point, and then run traffic simulation on each of all these TDGs. Such a solution is inefficient. We observe that traffic flows across different time points highly overlap. Driven by this insight, we propose an approach that constructs only one TDG with the union of all the traffic flows, enabling us to simulate traffic distribution across a period of time within one run. To further improve the efficiency of handling billions of flows, we propose two optimizations based on sampling and equivalence classes which significantly reduce the number of flows to be considered while still maintaining accuracy. (§4.2-§4.4)

Third, daily operation requires a large number of what-if analysis on network link (or router) failures. Straightforwardly simulating traffic for each link (or router) failure request cannot meet the real-time requirement of our operators. To support real-time failure-tolerance analysis, JINGUBANG proposes an incremental traffic simulation approach that first computes an incremental TDG and then simulates only the *differential* traffic distribution, avoiding the need to simulate the entire network traffic distribution from scratch. (§5)

Scope. JINGUBANG operates under two key settings: (i) a time point is defined as a minute for the purpose of checking traffic load properties, (ii) traffic is assumed to be distributed evenly or based on pre-configured weights across ECMP paths. Thus, the examination of micro-burst congestion [45] and ECMP unfairness falls outside the scope of this paper, as these considerations are orthogonal to our operators' requirements. [3] In addition, JINGUBANG takes as input flows entering our WAN and external routes advertised into our WAN, and will suffer from inaccuracy if traffic measurement is imprecise [5, 38] or route monitoring is incomplete (§7).

Real-world deployment. JINGUBANG has been used to check network traffic load in our WAN and successfully prevented service downtime in operations such as network changes and failure-tolerance analysis. We share our experience with JINGUBANG in §6 and evaluate JINGUBANG in §7.

Ethics. This work does not raise any ethical issues.

2 Background and Motivation

This section starts by describing the background of our WAN (§2.1), and then presents our motivation (§2.2).

2.1 Background: Our Production WAN

Alibaba Cloud operates a private WAN infrastructure that interconnects all its data centers and peers with external ISPs, serving both traffic internal to Alibaba Cloud and traffic between Alibaba Cloud and external networks. Our WAN is a distributed setting without a centralized SDN controller. By Jan 2023, this WAN has tens of autonomous systems (ASes), nearly a thousand routers, and tens of thousands of links, where each router has a forwarding table with millions of entries. [4] Our operators conduct hundreds of network changes per week, with frequent changes such as link capacity expansions and IP prefix publications happening a dozen times per day. Our WAN uses BGP (including eBGP and iBGP), IS-IS, static routes, SR, and PBR to route our network traffic.

[2] In ancient Chinese mythology, Jingu Bang is a tool used by Yu the Great to measure water levels during his efforts to control floods. Therefore, we named our network traffic load checking system JINGUBANG.

[3] In >99% of our WAN links, the top flow consumes <1% bandwidth. With a large number of flows and no elephant flows, flow-level ECMP can be approximated by evenly distributing flow volumes across ECMP paths.

[4] We omit absolute numbers for the confidentiality reason.

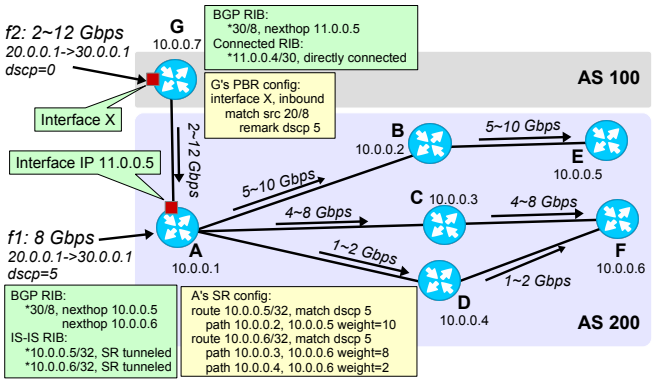


Figure 1: Motivating example. The green boxes represent RIBs for router A and G, respectively. Two yellow boxes are G's PBR configuration and A's SR configuration, respectively. In the given period of time (say one hour), the traffic rate of f_2 fluctuates between 2 Gbps and 12 Gbps, while f_1 keeps 8 Gbps. G belongs to AS 100, and A~F belong to AS 200. This example shows how BGP, IS-IS, PBR, and SR work together to distribute the traffic in our WAN.

PBR and SR. Unlike routing protocols (*e.g.*, IS-IS and BGP), PBR and SR are non-destination-based forwarding techniques. A PBR policy can match packets by their destination addresses, source addresses, and DSCP values; for matched packets, it can set next hops or modify packet fields (*e.g.*, DSCP). SR forwards the packets along one or more explicit paths built on Multiprotocol Label Switching (MPLS) or IPv6 stacks (*i.e.*, SRv6); different paths may have different routing weights. The mainstream SR used in our WAN is SRv6.

Traffic monitoring. Our WAN employs a large-scale traffic monitoring system that collects detailed traffic information from the network via Netflow [8] and sFlow [29] at a sampling rate of 8192. The monitoring system records the information of traffic flows entering each interface of routers, including the value of each field (*e.g.*, the IP-port 5-tuple), the timestamp of report time, and the total traffic volume of each flow sent since the last report time. The system generates a *traffic snapshot* per minute, recording the traffic flows (entering our WAN) and their rates in that minute. JINGUBANG receives 60 one-minute traffic snapshots every hour.

2.2 Motivations and Goals

Motivating example. Figure 1 shows a small but illustrative example of a traffic distribution situation in our operation. Suppose two traffic flows f_1 and f_2 enter the network by router A and G, respectively. In the given period of time, the traffic rate of f_1 keeps 8 Gbps, while f_2's rate fluctuates between 2~12 Gbps. When receiving f_2, G sets f_2's DSCP to 5 according to G's PBR configuration; then, G routes f_2 to A based on its eBGP route. By receiving f_1 and f_2 (both with DSCP=5), A performs the longest prefix matching for the flow based on A's routing information base (RIB), identifying two iBGP next hops E and F. The traffic rate from A to E and F is equal, *i.e.*, 10~20 Gbps/2=5~10 Gbps, because of the ECMP of iBGP. Router A looks up the direct next hop to E and F by checking A's SR configuration. According to A's SR configuration in Figure 1, A distributes the traffic into three paths ($A \to B$, $A \to C$, and $A \to D$) in traffic rates 5~10 Gbps, 4~8 Gbps, and 1~2 Gbps, respectively. Note that $A \to C$ and $A \to D$ forward traffic rates in a ratio of 4:1 due to weights 8 and 2 specified in A's SR configuration.

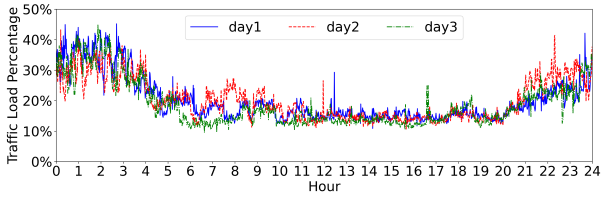


Figure 2: Traffic load fluctuation on one of the links in our WAN during randomly sampled three days.

Requirements from our operators. It is hard to ensure whether the traffic meets the intended properties (say "whether all traffic on $C \to F$ keeps the rate lower than 8 Gbps even if $B \to E$ fails") in a real WAN which is much larger than the above-mentioned motivating example. By surveying our operators, Table 1 lists representative scenarios where our operators need to check traffic load properties. There are mainly two types of analysis: what-if analysis and auditing. The former mainly focuses on network changes and specified failure scenarios, while the latter checks a network without any change. As shown in Table 1, our operators want to check traffic not only at a single time point but also over a period of time, because the network traffic fluctuates significantly during a time period. Figure 2 shows the traffic fluctuation occurred on one link in our WAN in randomly sampled three days. We can observe that the network traffic load fluctuates between 10% and 50% of this link bandwidth during these three days. Furthermore, as shown in Table 1, we should support protocols including BGP, IS-IS, SR, and PBR, since our WAN employs these types of protocols to forward traffic.

Traffic load properties of interest. The network traffic load property our operators mainly focus on is whether the traffic load/rate (in absolute number or percentage) or the change of it on a given set of links is above or below the specified threshold, such as "the traffic rate on the given set of links should be lower than 3 Gbps" and "given a network change, the traffic load on a specified link should not increase by 40% during the change". The *prop* in Figure 4 shows the formal definition of properties of interest (detailed in §3.3).

Representative scenarios for what-if analysis. We now describe two representative scenarios (network change analysis and failure tolerance analysis) to illustrate why we need what-if analysis in our WAN.

Network change analysis. Network changes, *e.g.*, changing configuration and updating routers, are one of the most important operations. In a typical network change, the operator makes a step-by-step change plan including each atomic step

Table 1: The main specifications our operators express in their operations.

Type	Name	Specification	Protocols
What-if Analysis	Change analysis	Whether the traffic load property holds in the network change time window	BGP/IS-IS/SR/PBR
	Change safety	Whether the network change would cause a traffic violation for a specified period of time	BGP/IS-IS/SR/PBR
	Failure tolerance	Whether the traffic load property holds if a specified set of routers or links becomes unavailable	BGP/IS-IS/SR/PBR
Auditing	Daily auditing	Whether the traffic load property holds for a given period of time	BGP/IS-IS/SR/PBR

to be executed (*e.g.*, changing routing policy, turning off interfaces, and cutting off the connection with neighbor routers), and reserves a network change time window (*e.g.*, 1am-7am on the upcoming Thursday). Because network changes are prone to failures [27,41], our operators want to check whether the network change would cause traffic load violations at any point in the change time window (note that network traffic fluctuates significantly during a period of time as shown in Figure 2) based on the traffic monitored in similar time periods (*e.g.*, 1am-7am on last Thursday).

Failure tolerance analysis. Another key type of operations is to check whether the load property holds if a specified set of links (or routers) becomes unavailable, due to the following scenarios: (i) router upgrading/replacement, and (ii) link maintenance. In the above cases, the router or link should be specified as unavailable, and the operators want to know whether the "failure" may cause any property violation.

Non-goal. JINGUBANG can check failure tolerance under a given set of failed routers or links, but cannot check arbitrary k-failure tolerance (*i.e.*, whether the property holds if arbitrary k links fail). While QARC [36] presented a good first step toward checking arbitrary k-failure tolerance, our operation experience found that designing a production-scale k-failure-tolerance checking for quantitative properties is extremely hard. This is because such a checker needs to support more detailed protocol properties, *e.g.*, BGP add-path and SR tunnels, which significantly expands the encoding space. We leave scalable k-failure traffic load checking to future work.

2.3 Related Work

For the traffic load property reasoning, why the state-of-the-art efforts do not help? Existing verification systems [2, 3, 10, 12, 14, 15, 18, 20, 21, 23, 24, 28, 30–32, 34, 35, 37, 39–43] mainly focus on checking qualitative properties such as packet reachability (*e.g.*, "whether a given route sent from router A can reach another router B"). To the best of our knowledge, except QARC [36], none of the prior work is able to check quantitative properties related to the traffic load.

Traffic engineering. The major goal of traffic engineering (TE) is to optimize network traffic and resource usage [4, 11, 17, 19, 25, 26]. Our goal, on the contrary, is to check whether the network implementation (*e.g.*, network topology, router configurations, and injected routes) satisfies the high-level traffic load properties. In particular, Alibaba Cloud uses a hybrid approach to realizing the TE optimization objectives via reconfiguring routers (*e.g.*, setting up SR tunnels) and injecting routes (*e.g.*, advertising BGP routes). However, they are unable to answer what-if or auditing queries about traffic properties as shown in Table 1. In addition, our work can also be used in non-TE scenarios. For example, when a router needs to be updated from vendor A to B, JINGUBANG can check that the new configuration under vendor B is correct in that no traffic load properties are violated.

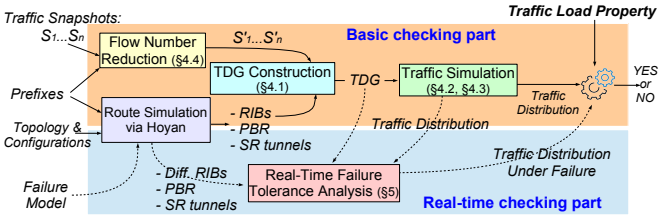


Figure 3: JINGUBANG's workflow overview. Solid arrows denote the basic traffic property checking process; dashed arrows mean the failure-tolerance checking process.

Network emulation. Network emulators, *e.g.*, CrystalNet [27], run the real control plane software in emulated environments and generate the corresponding forwarding behaviors for validation purposes. We do not choose emulation due to two issues. First, the emulator needs vendors to provide their router firmware within virtual machines or containers. However, we found it practically hard to get such support from all vendors for all router models in our WAN. Second, running real router software requires a large number of computing resources (*e.g.*, $100 per hour for emulating just one data center [27]). As routers in WAN typically have much more sophisticated firmware than data center switches, emulating a WAN can be even more expensive. Therefore, emulation is useful but not a good choice for our purpose.

Quantitative property analysis. Our focus on traffic properties is also different from the work on probabilistic analysis of the network control plane [16, 34, 44] and data plane [13, 33]. We check traffic at the network implementation level, which is different from the work [6, 46] that validates network designs.

3 JINGUBANG Overview

We built JINGUBANG, the first system for checking traffic load properties in production. JINGUBANG meets the requirements specified by our operators in §1.

3.1 JINGUBANG Workflow

Figure 3 shows the workflow of JINGUBANG. At a high level, JINGUBANG takes as inputs (i) the network topology and routers' configurations, (ii) IP prefixes advertised into the network, and (iii) multiple traffic snapshots each recording a set of traffic flow information at a time point (say one minute), then accurately simulates the *traffic distribution* (defined later), and finally checks whether the distribution meets

Table 2: Important terminologies.

Terms	Meaning
flow	A traffic flow, defined as ⟨*dst*, *src*, *dscp*⟩
located flow	A traffic flow at a particular interface defined as ⟨*dst*, *src*, *dscp*, *interface*, *traffic volume*⟩
traffic snapshot	A set recording all located flows entering the network at the given time point
traffic distribution	A vector recording the traffic load of each link in the network at the given time point

the specified traffic load property. For failure-tolerance analysis, JINGUBANG takes an additional failure model as input, which specifies a collection of unavailable routers or links.

As shown in Figure 3, we have two parts in JINGUBANG: the basic checking part (*i.e.*, the solid arrows) and the real-time checking part (*i.e.*, the dashed arrows). The real-time part is used for failure-tolerance analysis, *i.e.*, checking "whether the property holds if a specified set of routers or links becomes unavailable", while the basic part is responsible for the regular checking, such as network change analysis and daily auditing.

Basic checking part. In the basic part, we use Hoyan [41] to generate the RIBs, PBR, and SR tunnels for each virtual routing and forwarding (VRF) of all routers in the network.

Then, JINGUBANG reads multiple *traffic snapshots* $\mathcal{S}_1, \cdots, \mathcal{S}_n$. Each traffic snapshot records a set of *located flows* entering the network at the corresponding time point (*e.g.*, all traffic flow information from 12:30am to 12:31am on Aug 20, 2022), where a located flow is represented as ⟨*dst*, *src*, *dscp*, *interface*, *traffic volume*⟩. For example, in Figure 1, f_1 can be represented as ⟨*30.0.0.1*, *20.0.0.1*, *dscp=5*, *interfaceA*, *8 Gbps*⟩. By reading n traffic snapshots, $\mathcal{S}_1, \cdots, \mathcal{S}_n$, JINGUBANG's flow number reduction module (§4.4) generates n snapshots, $\mathcal{S}'_1, \cdots, \mathcal{S}'_n$, but each with a much smaller size due to our proposed sampling and equivalence class approaches.

By putting the above information together, JINGUBANG constructs a traffic distribution graph (or TDG) and simulates the traffic distribution (§4.2, §4.3). Finally, it checks whether the traffic distribution meets the specified property.

Real-time checking part. In the real-time part, JINGUBANG takes the failure model—containing a collection of specified failed routers or links—as input. We first use Hoyan to generate differential RIBs that record the difference of RIBs due to the failure. Then, JINGUBANG checks the TDG computed in the basic process and only updates the affected part. Finally, JINGUBANG computes traffic distribution under the given failure via the incremental traffic simulation module (§5).

3.2 Important Terminologies

Table 2 summarizes important terminologies, including flow, located flow, traffic snapshot, and traffic distribution. For the sake of simplicity, we may use the term "flow" for the located flow when the context is clear. Besides, the terms "traffic rate" and "traffic volume" of a flow may be used interchangeably at a given time point, since they are equal in unit time.

task	::=	*conf*; *flow*; [*fail*;] *prop*;	Checking task
		...	...
prop	::=	(*link l* : *expr*)*	Traffic load property
expr	::=	(**load** | **diff**) (≥|≤) *n* (**Gbps** | %)	Per-link property

Figure 4: Specification of JINGUBANG checking tasks.

3.3 JINGUBANG Checking Task's Specification

The specification of an JINGUBANG's checking task is similar to previous verification systems [2,3,9,37] with the additional traffic load properties. As shown in Figure 4, our operators will use a domain-specific language (DSL) to specify four parts: *conf*, *flow*, *fail* (optional), and *prop*: (i) *conf* specifies the topologies and configurations of the network to check; (ii) *flow* specifies the traffic snapshots; (iii) *fail* is only used for real-time checking, and allows the operators to specify a failure model, *i.e.*, a collection of failed routers and links in this checking task; (iv) *prop* specifies the traffic load property, defined as a set of ⟨*link*, *per-link traffic load property*⟩ pairs, where a per-link traffic load property (defined in *expr*) specifies the condition that the traffic load or its difference (in the failure scenario) on a link should hold.

Given a task defined above, JINGUBANG simulates the traffic distribution and checks if the specified traffic load property holds. The checking part follows directly from the simulated traffic distribution. Below we discuss how to simulate the traffic distribution accurately and efficiently in detail.

4 Traffic Simulation Using TDG

Given the complex features used in our WAN, flows exhibit various behavior as described in §2.2. In order to accurately simulate the traffic load on each link, we need to correctly model the traffic behavior of each flow in the traffic snapshots, such as ECMP, packet rewriting, and tunneling.

In this section, we first introduce the traffic distribution graph (TDG), capable of modeling the above-mentioned traffic behavior of flows. We then describe how to simulate traffic distribution based on TDG. Finally, we propose additional optimizations to further improve the simulation efficiency.

4.1 Traffic Distribution Graph

There are two existing ways that may be used to model the network behavior: 1) the shortest-path-based approach [14, 36] and 2) the SMT-based approach [2]. Neither of them can model the traffic behavior in our WAN. First, the shortest-path-based approach uses a weighted graph to model the behavior of each traffic class, where the shortest paths correspond to the actual forwarding paths. This approach is fundamentally limited in expressiveness: it cannot model iBGP and local preference which are inevitable in WANs. Second, the SMT-based approach encodes the entire control plane into an SMT formula and employs a solver to solve the formula. While there are encodings [2] for BGP, it is nontrivial to extend them to model the complex features in our WAN. In addition, SMT solving often induces prohibitively high overhead in

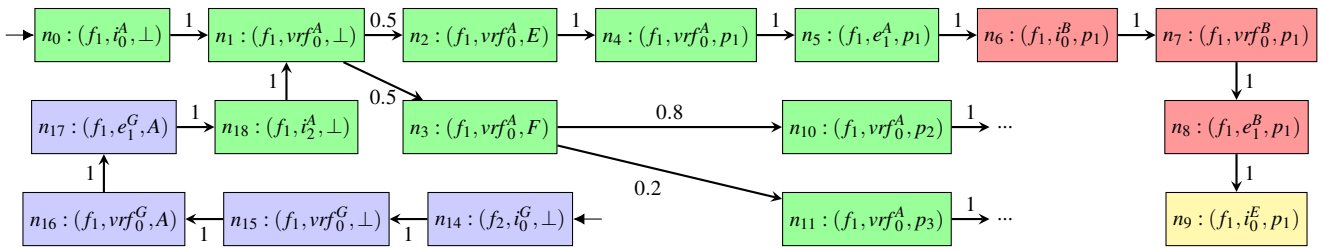


Figure 5: A (partial) TDG representing the example scenario in Figure 1. The green, red, yellow, and purple nodes correspond to router A, B, E, and G, respectively. Other nodes are omitted due to space limits.

runtime [36], which makes it impractical for our WAN with a large number of routers and prefixes.

To model the complex behavior under BGP, IS-IS, PBR, and SR, while still supporting efficient reasoning for traffic load properties, we introduce the *traffic distribution graph* (or TDG). Below, we first define the TDG and illustrate the TDG for the motivating example in Figure 1. The formal definition and construction of TDG can be found in Appendix B.

Definition. A TDG, corresponding to a traffic snapshot, models the entire traversal of *all* flows in this traffic snapshot in a fine-grained manner. Specifically, a TDG is a directed acyclic graph (JINGUBANG performs loop checking beforehand and will report errors if the TDG contains loops), where a *node* contains three elements: (i) the corresponding flow, *i.e.*, $\langle dst, src, dscp \rangle$, (ii) its location (*e.g.*, receiving at an interface or being processed at a VRF), and (iii) its next hop (*e.g.*, an IP address or an SR tunnel) based on the RIB lookup and policy matching results on the router. An *edge* between two nodes denotes a single processing step for this flow, such as forwarding the flow from an interface on router A to an interface on router B, looking up the RIB on a router's VRF, and so on. Each edge is associated with a non-negative weight representing the fraction of traffic volume distributed on that edge; thus, it is required that the sum of weights on all outgoing edges of a node equals 1. Finally, the sources in the TDG denote the initial state of flows entering the network.

Example. Figure 5 shows a (partial) TDG for the motivating example shown in Figure 1. For readability, nodes are colored based on the routers they correspond to; specially, green, red, yellow, and purple nodes correspond to routers A, B, E, and G, respectively. Each node is labeled with a name n_k. We use vrf_k^X to denote the k-th VRF on router X. We also use i_k^X (e_k^X, resp.) to denote the k-th interface labeled with the incoming (outgoing, resp.) flow direction on router X.

The node n_0 denotes the initial state of flow f_1 entering the network on router A's interface i_0^A, where $\perp$ means that the next hop is currently unknown. The edge (n_0, n_1) corresponds to the step that f_1 will be matched against the RIB on vrf_0^A. The two edges from n_1 correspond to the BGP lookup step, which finds the next hop E and F. Note that the weights on the two edges are both 0.5, reflecting the BGP ECMP mechanism implemented by router vendors. The edge (n_2, n_4) corresponds to the step that A resolves the next hop for E and finds the SR tunnel p_1; the two edges from n_3 denote that two tunnels are resolved for F and the weights reflect the ones set in the SR configurations. Then (n_4, n_5) and (n_5, n_6) denote that the flow is sent to the outgoing interface and forwarded to B's interface i_0^B; the meaning of other edges follows directly.

Similarly, the node n_{14} denotes the initial state of flow f_2 entering router G's interface i_0^G with no next hops. The edge (n_{14}, n_{15}) corresponds to the PBR step which modifies f_2 to f_1. The edges from n_{15} to n_{18} and finally to n_1 are similar to the ones described above. Note that after f_2's DSCP is modified, it undergoes processing identical to f_1. Therefore, the TDG does not have repetitive nodes for f_2 after n_{18}, showcasing its efficiency in WAN scenarios where it is common for flows to enter through different ingress points but later be processed identically. In contrast, the shortest-path-based modeling [14, 36] has to build complete graphs for both f_1 and f_2, even though the two graphs may share a large common subgraph.

4.2 Simulating A Single Traffic Snapshot

Given a TDG modeling the traffic behavior of all flows in a traffic snapshot, we can compute the traffic distribution of each flow on the TDG following its structure. By aggregating the traffic load of all flows traversing a link, we can obtain the traffic load of that link for the given traffic snapshot.

Algorithm 1 shows the traffic simulation algorithm for a single traffic snapshot $\mathcal{S}$, implementing the above idea. First, the function TRAFFICSIM constructs a TDG for all located flows in the traffic snapshot $\mathcal{S}$. After that, the function TRAFFICSIMONTDG takes the generated TDG and the traffic snapshot $\mathcal{S}$ as inputs to compute TL, *i.e.*, the traffic distribution at the given time point. Recall that the traffic distribution is a vector that records the traffic load of each link in the given snapshot (§3). Specifically, TRAFFICSIMONTDG calculates the traffic volume entering each node and distributes it on each edge of the TDG in topological order. For each node n in the TDG, we compute the traffic volume that reaches n (*i.e.*, $V[n]$) by summing up all volumes $V[e]$ on each incoming edge e of n (line 7). Then, the volume at n is distributed into each outgoing edge of n determined by their weights, *e.g.*, ECMP and SR weight (line 8). Finally, a link l's volume (or rate) $TL[l]$ is computed by aggregating $V[e]$ for each edge e where e denotes the forwarding step along the link l (line 10). Thus, we obtain a traffic distribution for a given time point.

Algorithm 1: Traffic simulation at a time point

```
1  Function TrafficSim(S):
2      Construct the TDG (N,E,w) for all (f_k, i_k, v_k) in S;
3      return TrafficSimOnTdg(S, TDG(N,E,w));
4  Function TrafficSimOnTdg(S, TDG(N,E,w)):
5      Add an incoming edge e'_k to source node n_k for each located
         flow in S and let V[e'_k] ← v_k;
6      forall n in N sorted in topological order do
7          V[n] ← sum{V[e]} for all n's incoming edge e;
8          V[e] ← V[n] × w(e) for all n's outgoing edge e;
9      forall link l in the network do
10         TL[l] ← sum{V[e]} for all e in E that share the same pair
             of interfaces with l;
11     return TL;
```

4.3 Simulating Multiple Traffic Snapshots

We now consider the case where the system accepts multiple traffic snapshots as input and needs to validate traffic load properties for a period of time. A naive solution may simply apply the TrafficSim function, which is designed for a single traffic snapshot, to each input traffic snapshot. However, it is inefficient for scenarios requiring a large number of traffic snapshots. For example, a planned change with a 2-hour time window requires 120 per-minute traffic snapshots.

To improve efficiency, we make the key observation that the input traffic snapshots are highly correlated. In the above example, all the traffic snapshots are extracted from the continuous 2-hour period. Thus, a located flow may appear across multiple snapshots which allows us to avoid repetitive computation for each appearance.

We then propose a global-construction separate-evaluation approach. As shown in Algorithm 2, JingubangG constructs a global TDG representing the union of all flows in all traffic snapshots (line 2) and calls TrafficSimOnTdg with the global TDG and each given traffic snapshot (line 3). Thus, the output of the algorithm *TLM* is a matrix recording the traffic distributions across multiple time points.

4.4 Flow Number Reduction Optimization

To further improve the efficiency of traffic simulation, we next propose two optimization techniques that can significantly reduce the number of flows needed to be simulated.

Traffic sampling optimization. First, we propose a sampling approach with *provable* guarantees on the accuracy loss to reduce the number of flows in the traffic snapshot. Our key insight is that a flow with a higher traffic volume should have a higher probability of being sampled, and the total traffic volume should be preserved. For example, if we sample only one flow from the traffic snapshot $\mathcal{S}$, the sampled flow should carry the total volume of all flows in $\mathcal{S}$.

Based on this insight, we propose the traffic sampling algorithm. This algorithm attempts to draw K located flows from the traffic snapshot $\mathcal{S}$. In each round, the algorithm draws a flow from $\mathcal{S}$ such that a flow with a volume v gets drawn with probability v/V, where V is the total traffic volume of all flows. Then, we assign V/K as the volume to the sampled flow and add it to the new traffic snapshot $\mathcal{S}'$. Repeating this process K times, we sample K located flows with a preserved total volume. We leave the algorithm's mathematical representation and pseudocode in Appendix A.1.

It is important to note that the number of *distinct* flows in the sampled traffic snapshot is likely to be much lower than the parameter K. This is because high-volume flows are likely to be sampled multiple times, and their volumes can be aggregated in the sampling process. We define the *flow sample ratio* as the ratio of the number of distinct flows in the sampled snapshot $\mathcal{S}'$ to the number of distinct flows in $\mathcal{S}$, which is related not only to the sample parameter K, but also to the distribution of volumes in $\mathcal{S}$.

Traffic sampling can introduce errors in estimating the traffic load on a link. Intuitively, for links with low/high traffic load, it is reasonable to require that the estimated traffic load does not deviate by an unacceptable absolute/relative value, respectively. We rigorously establish a lower bound of K such that traffic volumes of *all* links are guaranteed not to deviate by given absolute and relative values at a given level of confidence. The significance of K's proven bound lies in its tightness, making it practical for the use in our WAN, which is detailed in Appendix A.2. In the derivation of K's bound, we start from the law of large numbers and make no assumptions about the distribution of flows in $\mathcal{S}$. We also evaluate traffic sampling by experiments in §7.

Traffic equivalence class optimization. Besides, we observe that many flows exhibit the same behavior in the network despite that they differ in field values and incoming locations. For example, suppose in the motivating example there is another flow f_3 entering the network at some interface of A and matches the same BGP route and SR policy as f_1 does. Instead of performing the computation for both f_1 and f_3, we may only perform the computation for one of them, assuming it carries the volume of both flows.

Leveraging this insight, we propose the *traffic equivalence class* (TEC). Two flows are equivalent if they have the same traffic distribution over all links of the network; a TEC is a set of located flows that are equivalent to one another. See Appendix C for the formal definition and generation of TECs.

Among multiple flows in a TEC, we only need to consider one for traffic simulation. Therefore, given a traffic snapshot $\mathcal{S}$, we can aggregate flows in the same TEC by replacing them with a single flow that has the combined volume of all flows in the TEC. As a result, we generate a new traffic snapshot $\mathcal{S}'$ that has a much smaller number of flows.

5 Real-Time Failure-Tolerance Analysis

Checking whether the network satisfies the intended traffic load properties under a variety of failure scenarios is a key task in our daily operation (see Table 1). In such a task, our op-

Algorithm 2: Traffic simulation for a period of time

```
1 Function TrafficSimMul({S_k}_{k=1}^m):
2     Construct the TDG (N,E,w) for all (f_k,i_k,v_k) in ⋃_{k=1}^m S_k;
3     TLM[S_k] ← TrafficSimOnTdg(S_k, (N,E,w)) for all
        k = 1,···,m;
4     return TLM;
```

erators submit a large number of checking requests for failure-tolerance analysis, each specifying a set of failed routers or links, and expect to obtain results within *seconds*.

To meet the high-efficiency challenge, we observe that a typical checking request contains a small number (*i.e.*, $O(10)$) of link and router failures. Thus, the new TDG for the failure scenario may only differ slightly from the one without failure. For example in Figure 1, if $B \to E$ fails, the new TDG has all the nodes and edges except the edges $n_1 \to n_2 \to \cdots \to n_9$ in Figure 5. This observation allows us to adopt a two-phase design, where we first compute the TDG and store the necessary information for the network (without failure) in the basic checking part (see Figure 3), and then perform an *incremental* computation in the real-time checking part (see Figure 3) for each received failure-tolerance checking request.

There are two key steps to be conducted incrementally: TDG construction (§5.1) and traffic simulation (§5.2).

5.1 Incremental TDG Construction

When a failure happens, RIBs on routers converge to a new state (including the new SR tunnels); thus, the routes/tunnels may change, and the nodes in the old TDG corresponding to changed routes/tunnels may need to be updated (*e.g.*, connecting to different nodes or changing the weights of outgoing edges). In addition, the nodes corresponding to the failed links or routers should also be updated by removing their associated edges. All other nodes irrelevant to the failure (*e.g.*, receiving a flow at an active interface) remain unchanged. Therefore, to construct the new TDG, we keep most nodes and edges in the old TDG, and only re-run the construction for changed nodes.

Based on this observation, we construct the TDG for a given failure model incrementally. The algorithm takes the old TDG, the set of changed routes (*e.g.*, next hops changed) and tunnels (*e.g.*, tunnel removed) ΔRib, and the new RIBs (denote Rib') for the failure as input, and generates the new TDG incrementally. The algorithm also outputs the set of changed nodes *chg* for the post incremental traffic simulation (§5.2). Note that simulating Rib' and ΔRib is easy; many existing systems [12, 41] can extract the information.

The pseudocode and detailed description of the algorithm can be found in Appendix D.1. We illustrate how the algorithm works using the following example.

Example. Suppose $B \to E$ in Figure 1 fails. Due to the failure, the SR tunnel $A \to B \to E$ becomes invalid and thus the set *chg* contains the nodes for matching tunnel p_1 (*i.e.*, n_2 and n_4-n_9 in Figure 5). Furthermore, the BGP next hop E is also inactive, so *chg* also contains n_1 (in Figure 5) representing the match of the BGP route that is changed by the failure. Then the algorithm removes all outgoing edges for the nodes in *chg*. Using BFS exploration, the algorithm only re-constructs the edge from n_1 to n_3 with weight 1. The constructed TDG is shown in Figure 6.

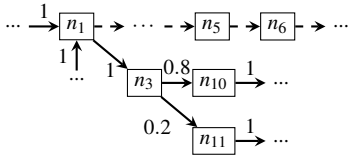


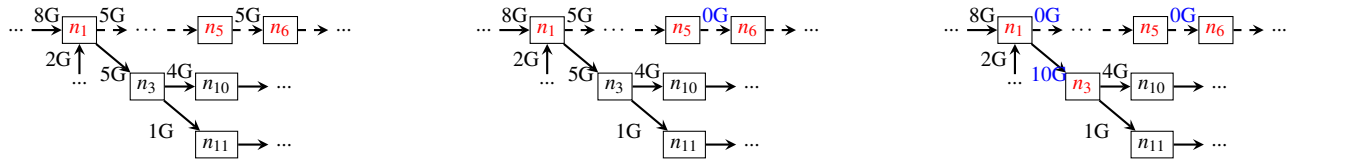
Figure 6: The (partial) TDG representing the example in Figure 1 when link $B \to E$ fails. Solid lines represent edges in both the old (*i.e.*, before failure) and new (*i.e.*, after failure) TDGs; dashed lines represent edges only in the old TDG.

5.2 Incremental Traffic Simulation

With the new TDG, one may re-run the traffic simulation algorithm on it to compute the volume on each node and edge as in Algorithm 1. However, since the new TDG largely overlaps with the old TDG, this straightforward approach leads to numerous redundant computations. In particular, if a node's volume remains unchanged and its outgoing edges/weights are also unchanged, the traffic distribution on the outgoing edges of that node should remain the same as that in the old TDG. We therefore only need to run the simulation for the *affected* nodes where either the volume or the outgoing edges/weights change. Clearly, all nodes in *chg* are affected nodes since their outgoing edges are changed; all nodes with incoming edges (in the old TDG) from any node in *chg* may also be affected, since their volumes may be changed. In addition, when updating the volume on the new TDG, other nodes may also be affected due to the change in their volume; thus, we also need to update the set of affected nodes along with the execution of traffic simulation.

Algorithm 3 shows the incremental traffic simulation for a single traffic snapshot; it can be naturally generalized to support multiple snapshots similar to the approach described in §4.3. The detailed description is in Appendix D.2.

Example. We illustrate the incremental computation of the example in §5.1. Suppose without failure, the volume of n_1's incoming edges is 8 and 2 Gbps respectively as shown in Figure 7. As shown in Figure 7a, the affected nodes α initially contain n_1, n_2, and all nodes from n_4 to n_9, since all of them are in *chg* as described in §5.1. Thus, the first node that needs to be updated is n_2 or anyone from n_4 to n_9, as none of them have any incoming edges in the new TDG G'. Suppose n_5 gets updated first. Since n_5 has no incoming edges in G', its volume is updated to 0; since the outgoing edge $n_5 \to n_6$ only appears in the old TDG G, the algorithm also decreases the traffic distribution on the corresponding link from $A \to B$, as shown in Figure 7b. The update of n_2 and others from n_4 to n_9 is similar and we omit the detailed description due



(a) Initially, affected nodes α contains n_1, n_2, and all nodes from n_4 to n_9 as described in §5.1.

(b) After updating n_5, the volume on $n_5 \rightarrow n_6$ is decreased to 0 Gbps. Since $n_5 \rightarrow n_6$ corresponds to link $l = A \rightarrow B$ in Figure 1, $TL[l]$ is updated to 0.

(c) After updating n_1, the volume on $n_1 \rightarrow n_3$ is increased to 10 Gbps and n_3 is added to α.

Figure 7: Representative steps of Algorithm 3 running on the TDG in Figure 6. The black (blue, resp.) number on an edge e denotes the value of v_e (v'_e, resp.), and the nodes in red are in α, after the corresponding step.

Algorithm 3: Incremental traffic simulation

```
1  Function IncTrafficSim(G, V, TL, G', chg):
2      Let G = (N,E,w), G' = (N',E',w');
3      V'[n] ← V[n] for all n ∈ N';
4      V'[e] ← V[e] for all n ∈ E';
5      α ← chg ∪ {n | ∃n' ∈ chg s.t. (n',n) ∈ E};
6      forall n in N' sorted in topological order of G' do
7          if n is not in α then
8              continue;
9          V'[n] ← sum{V'[e]} for all n's incoming edge e ∈ E';
10         forall e = (n,n') ∈ E' ∪ E do
11             V'[e] ← V'[n] × w'(e) if e ∈ E';
12             Let v_e = V[e] if e ∈ E else 0;
13             Let v'_e = V'[e] if e ∈ E' else 0;
14             if v'_e ≠ v_e then
15                 Add n' to α ;
16                 TL[l] ← TL[l] + v'_e − v_e if e corresponds to link l;
17     return TL;
```

to space limits. After those nodes are updated, the traffic distribution on link $B \rightarrow E$ is correctly decreased to 0 as expected. The next node of interest to be updated is n_1. As shown in Figure 7c, since the volume on the only outgoing edge $n_1 \rightarrow n_3$ increases, the algorithm adds n_3 to α. In the following iterations, the algorithm updates n_3 and all other nodes afterward, which results in the correct increase of traffic distribution for all links on the paths from A to F.

6 Deployment and Use Cases

JINGUBANG has been used in our WAN for more than one year and covers operation scenarios in Table 1. During this time, no outage resulting from traffic load violations occurred, since JINGUBANG successfully detected many violations ahead of time; before JINGUBANG, tens of outages caused by load violations happened every year. As a result, JINGUBANG prevents millions of dollars in losses. We next present real cases for two representative operation scenarios, *i.e.*, validating network changes and validating failure tolerance.

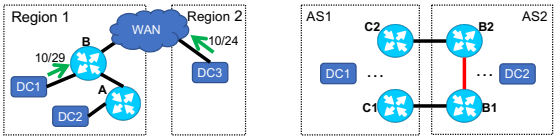
6.1 Validating Network Changes

JINGUBANG has been regularly used to check whether network change plans are correctly designed. JINGUBANG successfully detected several severe network change risks. Table 3 shows the statistics of root causes of network changes that were detected by JINGUBANG in the past one year. We classify the root causes into three types: change plan errors, unexpected routes, and existing misconfiguration. Specifically, change plan errors refer to errors in the change plan itself; unexpected routes mean that the change triggers issues because certain routes were not considered during the change; existing misconfiguration refers to hidden misconfigurations in the network that were not triggered because there was no traffic, but this change triggered the error configuration. We now detail several real, tricky cases as examples for demonstrating the effectiveness of JINGUBANG.

Table 3: Network change risks detected by JINGUBANG: root causes and their frequency of occurrence.

Root causes	Change plan errors	Unexpected routes	Existing misconfiguration
Percentage	44%	33%	23%

Change plan errors. Given the complexity and large size of our WAN, it is nontrivial to change the network correctly even for simple configuration-changing tasks. Our operators use JINGUBANG to check a change plan against intended link load properties during the time window of network changes. In one of our real network changes, our operators planned an important topology architecture upgrade for one business region in our WAN. The original topology in that region has many edge routers and border routers. Each edge router A_i is directly connected with all the border routers B_j with 100 Gbps links, while all the edge routers are connected in full mesh with 10 Gbps links. All routers run IS-IS as underlay and SR as overlay; the IS-IS cost for the links between the edge and border routers is 10000, and the cost between the edge routers is 5. For the new topology, our operators planned to add a layer of core routers C_k between the edge layer and border layer connected using 100 Gbps links. The IS-IS cost of links between the core routers and the border routers should be 10, while the cost of the links between core routers and edge routers is 10000. After the upgrade, traffic from border routers to edge routers should pass the core routers. In this network change, our operators first added all the core routers and set up the links without disabling old links between the border routers and the edge routers. Then, our operators disabled



(a) Unexpected routes. (b) Existing misconfiguration.

Figure 8: Severe risks detected by JINGUBANG.

IS-IS on the links between edge and border layers one by one via changing the interfaces on the edge routers to passive.

Given the change plan, JINGUBANG checked link overload for all links in the WAN using traffic snapshots in a similar time window. JINGUBANG detected that just after disabling the first edge router's (A_1) interfaces to the border routers, the utilization of links between A_1 and other edge routers suddenly increases to up to 150% during the time period, indicating severe link overload that would result in service downtime. The reason for the link overload is that: after disabling A_1's link to the border routers, the IS-IS shortest path from the border router B_j to A_1 becomes B_j-A_i-A_1 instead of B_j-C_k-A_1 as in the desired topology; meanwhile, the links between A_1 and other edge routers as backup links have a capacity of only 10 Gbps, significantly smaller than the 100 Gbps of other links. As a result, the traffic from border routers to A_1 caused links between A_1 and other edge routers to overload.

This situation is very hard to be detected without JINGUBANG. First, without JINGUBANG, our operators have to manually check link overload for each step in the change plan, which is error-prone. Second, what we describe is just a subset topology of our WAN. Our operators have to analyze IS-IS shortest paths for the entire scope, which should cover at least nearby regions. Third, the backup links between edge routers are designed to carry a small amount of traffic; thus, in addition to identifying that IS-IS shortest paths pass through those links, our operators have to manually analyze the traffic rate passing those links. As a result, our operators cannot detect such complex outages without JINGUBANG. Before JINGUBANG, similar problems occurred multiple times.

Unexpected routes. Our WAN carries millions of routes; some of them are well understood while some of them are unexpected. It is impossible for the operators to examine the effect of a change on all of the routes, especially for the unexpected routes. JINGUBANG offers an intuitive approach to checking the correctness of a change plan under the existence of unexpected routes. Figure 8(a) shows such a case. In this scenario, router A was configured with a default route (*i.e.*, 0/0) to forward DC2's traffic to router B while router B did not advertise any routes to router A. Due to business reasons, our operators needed to block the traffic from DC2 going to Region 2. After carefully checking that no traffic flowing from DC2 to Region 2, our operators planned to update the policy on router B such that it can advertise routes from Region 2 (those routes have pre-defined communities) to router A, and also add policies on router A to explicitly drop traffic matching those routes. Our operators used JINGUBANG to check that the traffic load should remain the same during the change. However, JINGUBANG identified load decreases on links in Region 1, violating the intended traffic load property. The root cause is that, DC3 in Region 2 was advertising a 10/24 route, while a service in DC1 in Region 1 unexpectedly used and advertised 10/29. Before the change, traffic from DC2 can reach the service in DC1 by matching the default route on router A and the 10/29 route on router B. After the change on routers A and B, the traffic would match the 10/24 route on router A and thus get dropped, which may cause a service outage. Without JINGUBANG, such risk is very hard to detect due to the unawareness of those routes; but fortunately, JINGUBANG successfully prevented it by identifying the violation to the specified traffic load property.

Existing misconfiguration. A typical network change may only involve a small set of routers. However, given the existing misconfiguration on other routers in the network, such changes can introduce severe risks. Figure 8(b) shows a real case in our WAN. In this scenario, traffic from DC1 to DC2 used to exit AS1 via router C2. Our operator planned to shift the exit point from C2 to C1. To achieve this, our operators first changed the policies on the routers that act as BGP route reflectors [7] (not shown in the figure) to propagate routes that would be advertised by C1 to DC2; then changed the policies on C1 to advertise these routes while assigning them a higher local preference to make them effective. Our operators use JINGUBANG to check that no links in the network would be overloaded during the change. JINGUBANG successfully detected overloaded links between B1 and B2 (highlighted in red). The root cause of this risk was that, while the change plan could successfully steer the traffic from C2 to C1, however, due to the incorrect IS-IS cost setup in AS2, traffic transited from B1 to B2 instead of flowing directed to DC2, causing the low-capacity links between B1 and B2 to overload. This risk is extremely hard for our operators to identify manually because (i) the change plan correctly altered the exit point on AS1 as intended and (ii) the misconfigured IS-IS costs in AS2 were not a problem before the change because there was no large volume of traffic going through those links. Fortunately, JINGUBANG successfully prevented the severe risk by detecting the overloaded links.

6.2 Validating Failure Tolerance

Our WAN is designed to tolerate failures up to a certain level by configuring redundant links and backup routers, which we call the redundancy property. However, due to the high-churn network changes and carelessness of our operators, redundancy property is violated in many regions in our WAN. Our operators cannot know the violations without a checker. Once the bottlenecked links or routers fail, a cascading failure would span across the entire WAN.

Our operators systematically inject router and link failures and use JINGUBANG to check two properties: (i) no link overload and (ii) no drastic drop in link load (which is typically

caused by reachability issues). JINGUBANG has successfully detected tens of failure risks. One of them happened on a remote site of our WAN, where we deployed two border routers to provide service to our customers abroad. After our operators injected a failure model that made one of the border routers unavailable, JINGUBANG detected that the traffic load on several links close to that site dropped hugely. Further analyzing each flow using JINGUBANG's capability of flow-level traffic simulation, JINGUBANG successfully identified a prefix where flows with destinations in that prefix got dropped on the other router. We examined this situation with our operators, and confirmed that the prefix belongs to a new service and one of the routers is configured incorrectly such that the prefix is only advertised out of another router.

7 Evaluation

We evaluate the accuracy and efficiency of JINGUBANG using real-world data obtained from our production WAN.

Experiment setup. Besides the entire WAN, we select two sub-networks, N1 and N2, to represent diverse scales of network traffic. N1 and N2 are chosen as they respectively account for 7% and 85% of the total traffic in our WAN. In each hour, there are billions of flows entering N2 and hundreds of millions of flows entering N1. There are hundreds of routers in both N1 and N2. All networks are configured with BGP, IS-IS, SR, and PBR. Unless otherwise specified, we use JINGUBANG to check traffic load across one hour (*i.e.*, 60 time points). All the experiments are conducted on a server with 768 GB RAM and a 2.50 GHz 104-core processor.

Performance. We first use JINGUBANG to check traffic load for each hour across one randomly selected day. As shown in Figure 9, the time cost for checking these hours is within an order of magnitude, ranging from 19.6 s to 25.6 s for N1, 168.2 s to 272.4 s for N2, and 225.5 s to 308.4 s for the WAN.

Accuracy. We inject failures that actually occurred in the selected hours. We define the accuracy for each link as $\frac{\min(r_{\text{sim}}, r_{\text{real}})}{\max(r_{\text{sim}}, r_{\text{real}})} \times 100\%$ where r_{sim} represents the traffic rate computed by JINGUBANG and r_{real} represents the rate measured by the router. Based on this, we further determine the overall accuracy of JINGUBANG by averaging accuracy of all links with weights proportional to the link's traffic load. Figure 9 depicts such accuracy of each hour. The overall accuracy ranges from 88.2% to 94.6% for N1, 87.4% to 92.5% for N2, and 87.8% to 92.1% for the WAN, which is stable and meets our operators' requirements. Nevertheless, the accuracy is not 100%, mostly because our route simulation takes external routes advertised into our WAN as input and they are provided by our internal route monitoring system, which occasionally misses certain routes due to coverage issues.

We further demonstrate the variation of r_{sim} and r_{real} during one hour using an example link. The link connects two service areas of our WAN and its link load varies between 40% and 95% within a single hour. As shown in Figure 10, JINGUBANG can accurately track these fluctuations. Over the 60 time points, the median, 90th-percentile, and maximum error rate is 4.0%, 8.7%, and 19.0%, respectively, indicating the high accuracy of JINGUBANG.

Multiple traffic snapshots. [5] We evaluate the checking time of JINGUBANG for different numbers of time points. As shown in Figure 11, JINGUBANG's checking speed at 60 time points is only 4.2× (from 4.5s to 18.9s) and 5.2× (from 67.5s to 353.9s) slower than at one time point for networks N1 and N2, respectively. The reason behind this is that the sets of flows at consecutive time points have a significant amount of overlap with each other. By constructing a single TDG for all 60 time points, JINGUBANG avoids duplicate processing and speeds up the checking by 14.1× (11.4×) for N1 (N2), compared to construct separate TDGs for each time point.

Traffic equivalence classes. In Figure 12, we show the impact of applying TEC on the performance of JINGUBANG for different flow sample ratios in network N1. When traffic sampling is disabled, the use of TEC significantly accelerates the checking, from 601.8s to 18.4s, resulting in a 32.7× speedup. This improvement is substantial due to the large number of flows that can be grouped into a single class. However, when the flow sample ratio is 11%, the speedup is only 3.8× (from 33.4s to 8.8s) as many low-rate flows are not sampled, but their equivalence class still exists because of either the large flow number in the class or the presence of an elephant flow. We do not evaluate the use of TEC in N2 as it would result in memory exhaustion without TEC.

Traffic sampling. Figure 13 illustrates the relationship between the flow sample ratio and the time cost of JINGUBANG in network N2. By setting the sample ratio to 11%, the checking time can be significantly reduced from 358.3s to 113.7s, a 3.2× speedup. To assess its impact on accuracy, we define the maximum accuracy loss as the maximum reduction in link load accuracy for all links that use ≥5% of their bandwidth. As also shown in Figure 13, the accuracy loss does not exceed 1.1% across 10 runs. Besides, for links with bandwidth utilization <5%, the absolute changes in their traffic rates due to traffic sampling never exceed 60 Mbps. Therefore, for most links, the accuracy loss is not substantial at all. As an example, in Figure 10, we additionally plot r_{sim} after sampling at the flow ratio of 11%, and the difference is barely noticeable.

Incremental simulation. The daily operation of our WAN involves handling up to $O(100)$ checking requests per hour, each of which asks us to validate a few traffic load properties under up to $O(10)$ failed links. To compare the checking time between incremental and full simulation, we randomly selected $O(100)$ checking requests and depict the CDF of time cost using two methods in Figure 14. We apply the traffic sampling with a flow ratio of 11% in N2. As demonstrated in

[5] Since our operation suggests that JINGUBANG can check N2 and the WAN at a similar time cost (as also shown in Figure 9), and for simplicity, we do not evaluate JINGUBANG on the WAN in the following experiments.

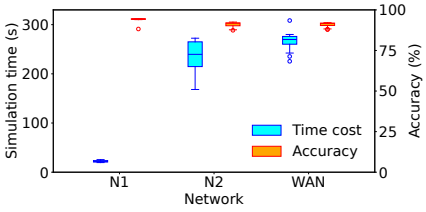


Figure 9: Performance and accuracy.

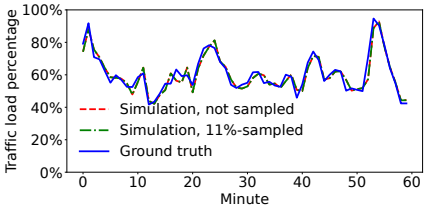


Figure 10: Per-minute accuracy.

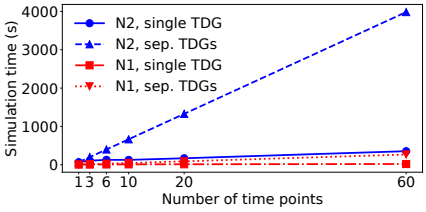


Figure 11: Effects of the single TDG.

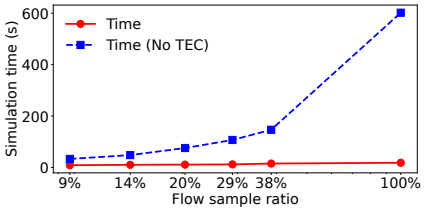


Figure 12: Effects of TEC.

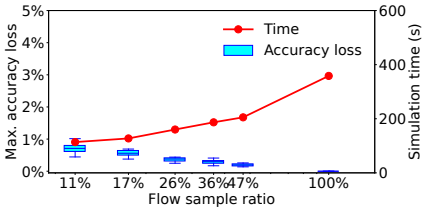


Figure 13: Effects of traffic sampling.

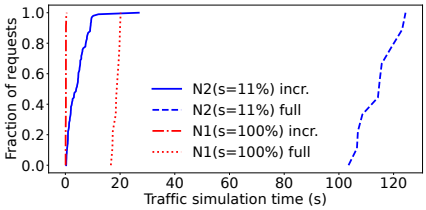


Figure 14: Failure-tolerance check time.

the figure, the average time cost for incremental simulation is 4.3s, which is a significant improvement over the average time cost of 113.7s for full simulation. This 26.2× speedup meets our deployment requirements and shows the effectiveness of using incremental simulation in our WAN.

JINGUBANG is also capable of handling router failures in an incremental manner. However, such requests are relatively infrequent, occurring dozens of times per month. Nevertheless, with ∼100 checking requests that involve supported kinds of router failures, we find that incremental simulation in N1 (N2) costs 2.3s (14.5s) on average, which is still 8.0× (7.8×) faster than full simulation.

8 Discussions and Lessons

We now present our lessons and discuss limitations.

Verifying arbitrary *k*-failure tolerance. The *k*-failure problem lies beyond the scope of this paper, as discussed in §2.2. Meanwhile, we argue that JINGUBANG provides a solid foundation by efficiently modeling comprehensive traffic behaviors at a production scale. We plan to extend the model to support arbitrary *k*-failure verification in future research.

Addressing vendor-specific behaviors. Improving JINGUBANG's precision is often hindered by vendor-specific traffic behaviors and unexpected corner cases. For instance, SR tunnels are configured with specific next-hop IPs and DSCP values. If traffic flows match the next-hop IP but exhibit different DSCP values, routers from one vendor attempt alternative SR tunnels configured for other DSCP values, while other routers default to the IS-IS shortest path. [6] We refine JINGUBANG's accuracy over years of operation and defer the proactive identification of such behaviors to future work.

Enhancing user experience. JINGUBANG can indicate if traffic properties remain intact under various conditions but falls short in diagnosing reasons for property violations. Even with additional traffic flow visualization utilities, pinpointing the root issues remains time-intensive for both system developers and network operators. We envision an automated system that can find victim routes, identify configuration errors, and even offer potential repair solutions as the next step.

Handling altered traffic snapshots. JINGUBANG cannot reason about shifts in traffic snapshots. Failures, for example, can prompt routers to announce altered routes to ISPs, leading to changes in the traffic entering our WAN. Since ISP routers fall outside our simulation domain and are not covered by our monitoring system, we currently rely on heuristic rules defined by our operators to adapt to such changes in traffic snapshots. A rigorous methodology to understand variations in traffic snapshots is left for subsequent research.

9 Conclusion

This paper presents JINGUBANG, the first traffic load property checking system for production WAN. JINGUBANG makes three contributions: (i) a new model, named the traffic distribution graph, capable of encoding complex traffic behavior under BGP, IS-IS, SR, and PBR; (ii) an efficient traffic simulation approach, which can handle billions of flows in a period of time; (iii) an incremental simulation approach, enabling real-time failure-tolerance checking. JINGUBANG has been used to check our WAN for more than one year and prevented many potential failures resulting from traffic load violations.

Acknowledgments

We thank our shepherd, Stefano Vissicchio, and anonymous reviewers for their insightful comments. This work is supported in part by National Key Research and Development Plan, China (Grant No. 2023YFB2903902), National Natural Science Foundation of China (Grant No. 62022005, 62272010, and 62061146001), and Alibaba Cloud through Alibaba Research Intern Program and Alibaba Innovative Research Program. Ennan Zhai and Chenren Xu are co-corresponding authors. Ruihan Li and Chenren Xu are affiliated with School of Computer Science at Peking University, Zhongguancun Laboratory, and Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

[6] We use real examples, but omit the vendor names.

References

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 201–219, Santa Clara, CA, February 2020. USENIX Association.

[2] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 155–168, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 328–341, New York, NY, USA, 2016. Association for Computing Machinery.

[4] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. Teavar: striking the right utilization-availability balance in wan traffic engineering. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 29–43, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Tobias Bühler, Romain Jacob, Ingmar Poese, and Laurent Vanbever. Enhancing global network monitoring with magnifier. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1521–1539, Boston, MA, April 2023. USENIX Association.

[6] Yiyang Chang, Sanjay Rao, and Mohit Tawarmalani. Robust validation of network designs under uncertain demands and failures. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 347–362, Boston, MA, March 2017. USENIX Association.

[7] Enke Chen, Tony J. Bates, and Ravi Chandra. BGP Route Reflection: An Alternative to Full Mesh Internal BGP (IBGP). RFC 4456, April 2006.

[8] Benoît Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954, October 2004.

[9] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. NetComplete: Practical Network-Wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 579–594, Renton, WA, April 2018. USENIX Association.

[10] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient network reachability analysis using a succinct control plane representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 217–232, Savannah, GA, November 2016. USENIX Association.

[11] Mikel Jimenez Fernandez and Henry Kwok. Building express backbone: Facebook's new long-haul network, 2017.

[12] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, Oakland, CA, May 2015. USENIX Association.

[13] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. Probabilistic netkat. In Peter Thiemann, editor, *Programming Languages and Systems*, pages 282–309, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[14] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 300–313, New York, NY, USA, 2016. Association for Computing Machinery.

[15] Nick Giannarakis, Ryan Beckett, Ratul Mahajan, and David Walker. Efficient verification of network fault tolerance via counterexample-guided refinement. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 305–323, Cham, 2019. Springer International Publishing.

[16] Nick Giannarakis, Alexandra Silva, and David Walker. Probnv: probabilistic verification of network control planes. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021.

[17] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 15–26, New York, NY, USA, 2013. Association for Computing Machinery.

[18] Alex Horn, Ali Kheradmand, and Mukul Prasad. Delta-net: Real-time network verification using atoms. In

14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 735–749, Boston, MA, March 2017. USENIX Association.

[19] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 3–14, New York, NY, USA, 2013. Association for Computing Machinery.

[20] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 200–213, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. Automated analysis and debugging of network connectivity policies. Technical Report MSR-TR-2014-102, Microsoft, July 2014.

[22] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. Groot: Proactive verification of dns configurations. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 310–328, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, April 2012. USENIX Association.

[24] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying Network-Wide invariants in real time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27, Lombard, IL, April 2013. USENIX Association.

[25] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 1–14, New York, NY, USA, 2015. Association for Computing Machinery.

[26] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 527–538, New York, NY, USA, 2014. Association for Computing Machinery.

[27] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 599–613, New York, NY, USA, 2017. Association for Computing Machinery.

[28] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 499–512, Oakland, CA, May 2015. USENIX Association.

[29] Sonia Panchen, Neil McKee, and Peter Phaal. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176, September 2001.

[30] Aurojit Panda, Katerina Argyraki, Mooly Sagiv, Michael Schapira, and Scott Shenker. New Directions for Network Verification. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morriset, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 209–220, Dagstuhl, Germany, 2015. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[31] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 699–718, Boston, MA, March 2017. USENIX Association.

[32] B. Quoitin and S. Uhlig. Modeling the routing of an autonomous system with c-bgp. *IEEE Network*, 19(6):12–19, 2005.

[33] Steffen Smolka, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. Cantor meets scott: se-

mantic foundations for probabilistic networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 557–571, New York, NY, USA, 2017. Association for Computing Machinery.

[34] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Probabilistic verification of network configurations. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 750–764, New York, NY, USA, 2020. Association for Computing Machinery.

[35] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 314–327, New York, NY, USA, 2016. Association for Computing Machinery.

[36] Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, and Aditya Akella. Detecting network load violations for distributed control planes. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 974–988, New York, NY, USA, 2020. Association for Computing Machinery.

[37] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, and Ben Y. Zhao. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 214–226, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. Stroboscope: Declarative network monitoring on a budget. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 467–482, Renton, WA, April 2018. USENIX Association.

[39] Yaron Velner, Kalev Alpernas, Aurojit Panda, Alexander Rabinovich, Mooly Sagiv, Scott Shenker, and Sharon Shoham. Some complexity results for stateful network verification. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 811–830, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[40] Anduo Wang, Limin Jia, Wenchao Zhou, Yiqing Ren, Boon Thau Loo, Jennifer Rexford, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. Fsr: Formal analysis and implementation toolkit for safe interdomain routing. *IEEE/ACM Transactions on Networking*, 20(6):1814–1827, 2012.

[41] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. Accuracy, scalability, coverage: A practical configuration verifier on a global wan. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 599–614, New York, NY, USA, 2020. Association for Computing Machinery.

[42] Ennan Zhai, Ang Chen, Ruzica Piskac, Mahesh Balakrishnan, Bingchuan Tian, Bo Song, and Haoliang Zhang. Check before you change: Preventing correlated failures in service updates. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 575–589, Santa Clara, CA, February 2020. USENIX Association.

[43] Peng Zhang, Aaron Gember-Jacobson, Yueshang Zuo, Yuhao Huang, Xu Liu, and Hao Li. Differential network analysis. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 601–615, Renton, WA, April 2022. USENIX Association.

[44] Peng Zhang, Dan Wang, and Aaron Gember-Jacobson. Symbolic router execution. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 336–349, New York, NY, USA, 2022. Association for Computing Machinery.

[45] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, page 78–85, New York, NY, USA, 2017. Association for Computing Machinery.

[46] Yunmo Zhang, Hong Xu, Chun Jason Xue, and Tei-Wei Kuo. Probabilistic analysis of network availability. In *2022 IEEE 30th International Conference on Network Protocols (ICNP)*, pages 1–11, 2022.

APPENDIX

A Traffic Sampling

In this appendix, we will formally define the traffic sampling process (first presented in §4.4), then prove a practical bound for the error rate, and demonstrate its use in our WAN.

A.1 Algorithm

Initially, we consider a single network and a single traffic snapshot at a specific time point. We will extend our analysis to encompass multiple networks (*e.g.*, networks under different failure scenarios) and multiple time points later.

Suppose that there are N located flows $f_1, \cdots, f_N$ in the traffic snapshot $\mathcal{S}$, with volumes $v_1, \cdots, v_N$, respectively. We denote the set of flows as $\mathcal{F}$, *i.e.*, $\mathcal{F} := \{f_1, \cdots, f_N\}$ and the total volume as V, *i.e.*, $V := \sum_{i=1}^{N} v_i$. The network contains M links, where the volume of the i-th link is l_i, which is a function of the traffic snapshot.

Lemma 1. *Assuming the absence of traffic loops, the volume on each link in the network is a linear function of the volumes of the flows in the traffic snapshot. More specifically,*

$$l_i = \sum_{j=1}^{N} w_i(f_j) v_j,$$

where

$$w_i : \mathcal{F} \to [0,1]$$

is the flow weight function for the i-th link, for all $i = 1, \cdots, M$.

Proof. This lemma follows from the construction of TDGs during the traffic simulation process. The flow weight functions can be easily calculated according to these TDGs. □

Then we give a formal definition of traffic sampling, corresponding to Algorithm 4.

Definition 1. *In traffic sampling, a new traffic snapshot* $\mathcal{S}'$ *is generated by selecting K flows,* $F_1, \cdots, F_K$*, from the original traffic snapshot* $\mathcal{S}$*, each with a volume of* $\frac{V}{K}$*. These K flows are considered as K independent and identically distributed (i.i.d.) random variables, and the probability of selecting the i-th flow in the original traffic snapshot is defined as*

$$\mathbb{P}(F_1 = f_i) = \frac{v_i}{V}$$

for all $i = 1, \cdots, N$.

As noted previously in Appendix A.1, the number of distinct flows in the sampled traffic snapshot is typically much less than K, since a flow with a large volume can easily be sampled multiple times.

Algorithm 4: Traffic sampling algorithm

```
Function SAMPLE(S, K):
    Let V be the total volume of all flows in S;
    Let P be the probability distribution s.t.
      ∀(f_k, i_k, v_k) ∈ S, P(f_k, i_k) = v_k/V;
    S' ← ∅;
    repeat K times
        Draw a (f_k, i_k) from the probability distribution P;
        Add (f_k, i_k, V/K) to S';
    return S';
```

A.2 Error Bound

We next introduce the error bound for traffic sampling.

Theorem A.1. *Suppose that M links have volumes* $l_1, \cdots, l_M$ *before sampling. After sampling, these links have volumes represented by M random variables* $L_1, \cdots, L_M$*. Given a maximum relative error* μ*, a maximum absolute error* Δ*, and a confidence parameter* δ*, the following holds with probability at least* $1 - \delta$*:*

$$\left|L_i - l_i\right| \le \max\left\{\Delta, \mu l_i\right\} \qquad \forall i \in \{1, \cdots, M\}$$

if the number of samples K satisfies

$$K \ge \frac{2V}{\Delta}\left(\frac{1}{\mu} + \frac{1}{3}\right)\ln\left(\frac{2M}{\delta}\right).$$

Proof. For the i-th link, we define K random variables

$$X_j^{(i)} = w_i(F_j)\frac{V}{K} \qquad \forall j \in \{1, \cdots, K\}.$$

Since $F_1, \cdots, F_K$ are i.i.d., $X_1^{(i)}, \cdots, X_K^{(i)}$ are i.i.d., too. Besides, we have

$$\begin{aligned}\mathbb{E}\left[X_1^{(i)}\right] &= \sum_{r=1}^{N} w_i(f_r)\frac{V}{K}\mathbb{P}(F_1 = f_r) \\ &= \sum_{r=1}^{N} w_i(f_r)\frac{V}{K}\frac{v_r}{V} \\ &= \frac{1}{K}\sum_{r=1}^{N} w_i(f_r) v_r = \frac{l_i}{K}\end{aligned}$$

and

$$\begin{aligned}\mathrm{Var}\left[X_1^{(i)}\right] &= \mathbb{E}\left[\left(X_1^{(i)}\right)^2\right] - \left(\mathbb{E}\left[X_1^{(i)}\right]\right)^2 \\ &\le \sum_{r=1}^{N}\left(w_i(f_r)\frac{V}{K}\right)^2 \mathbb{P}(F_1 = f_r) \\ &\le \sum_{r=1}^{N} w_i(f_r)\frac{V^2}{K^2}\frac{v_r}{V} = \frac{V}{K^2} l_i.\end{aligned}$$

According to Lemma 1, we know

$$L_i = \sum_{i=1}^{K} w_i(F_k)\frac{V}{K} = \sum_{i=1}^{K} X_i^{(i)},$$

which implies $\mathbb{E}[L_i] = K\mathbb{E}\left[X_1^{(i)}\right] = l_i$ as well as $\mathrm{Var}[L_i] = K\,\mathrm{Var}\left[X_1^{(i)}\right] \le \frac{V}{K} l_i$.

As $0 \le X_1^{(i)} \le \frac{V}{K}$ holds almost surely, the use of Bernstein's equalities on $\left\{X_j^{(i)} - \frac{l_i}{K}\right\}_{j=1}^{K}$ yields

$$\begin{aligned}\mathbb{P}\left[\left|L_i - l_i\right| \ge t\right] &\le 2\exp\left(-\frac{\frac{1}{2}t^2}{\mathrm{Var}[L_i] + \frac{1}{3}\frac{V}{K}t}\right) \\ &\le 2\exp\left(-\frac{K}{V}\frac{3t^2}{6l_i + 2t}\right)\end{aligned}$$

for any $t > 0$.

Taking t as $\max\{\Delta, \mu l_i\}$, we have

$$\begin{aligned}\mathbb{P}\left[\left|L_i - l_i\right| \ge \max\left\{\Delta, \mu l_i\right\}\right] &\le 2\exp\left(-\frac{K}{V}\left(\frac{2l_i}{t^2} + \frac{2}{3t}\right)^{-1}\right) \\ &\le 2\exp\left(-\frac{K}{V}\left(\frac{1}{\mu}\frac{2}{\Delta} + \frac{1}{3}\frac{2}{\Delta}\right)^{-1}\right).\end{aligned}$$

So

$$\mathbb{P}\left[\left|L_i - l_i\right| \ge \max\left\{\Delta, \mu l_i\right\}\right] \le \frac{\delta}{M}$$

as given

$$K \ge \frac{2V}{\Delta}\left(\frac{1}{\mu} + \frac{1}{3}\right)\ln\left(\frac{2M}{\delta}\right).$$

Finally, by the union bound,

$$\begin{aligned}&\mathbb{P}\left[\forall i \in \{1, \cdots, M\}\ \left|L_i - l_i\right| < \max\left\{\Delta, \mu l_i\right\}\right] \\ &= 1 - \mathbb{P}\left[\exists i \in \{1, \cdots, M\}\ \left|L_i - l_i\right| \ge \max\left\{\Delta, \mu l_i\right\}\right] \\ &\ge 1 - \sum_{i=1}^{M} \mathbb{P}\left[\left|L_i - l_i\right| \ge \max\left\{\Delta, \mu l_i\right\}\right] \\ &\ge 1 - M \times \frac{\delta}{M} = 1 - \delta,\end{aligned}$$

which proves the theorem. □

This theorem bounds either the relative error or absolute error. As a corollary, we can bound only the relative error for links whose volumes are greater than a threshold value.

Corollary A.1. *Given a maximum relative error μ, a threshold volume V_T, and a confidence parameter δ, the inequality $\left|L_i - l_i\right| \le \mu l_i$ holds for all links with volumes above the threshold (*i.e.*, $l_i \ge V_T$) with probability at least $1 - \delta$, if the number of samples K satisfies*

$$K \ge \frac{2V}{V_T}\left(\frac{1}{\mu^2} + \frac{1}{3\mu}\right)\ln\left(\frac{2M}{\delta}\right).$$

Proof. Setting $\Delta = \mu V_T$, we have

$$\left|L_i - l_i\right| \le \max\left\{\Delta, \mu l_i\right\} = \mu l_i$$

for all links satisfying $l_i \ge V_T$. Therefore, applying Theorem A.1 directly proves this corollary. □

Thus far, we have demonstrated the theorem for a single failure model and a single traffic snapshot. However, it is straightforward to extend these results to accommodate multiple models and snapshots. This can be achieved by treating links associated with different failure models or traffic snapshots as distinct entities. As a result, instead of examining M interfaces, we consider $M \times A \times B$ interface-failure-snapshot combinations, where A is the number of failure models and B is the number of snapshots. The proof of the theorem remains valid. It is worth noting that the parameter M appears within the logarithmic function in the inequality. Thus, its increase has a minimal impact on the overall result.

Evaluation. The proven bound is practical enough for use in our WAN. In network N2 (as defined in §7), we employ a flow sample ratio of 11% to expedite the checking process by a factor of 3.2. The application of the theorem ensures that the error is limited to either 150 Mbps in absolute terms or 2.4% in relative terms for all links, with probability at least 99%. In our experiments (as detailed in §7), across 10 trials, either the relative error does not exceed 1.1% or the absolute error does not exceed 60 Mbps for all links in network N2.

B Traffic Distribution Graph

In this appendix, we will formally define the traffic distribution graph (*i.e.*, TDG, first presented in §4.1), and then detail our TDG construction algorithm.

B.1 Formal Definition

Formally, let $\mathcal{F}$ denote the space of all flows, $\mathcal{V}$ denote the set of all VRFs in the network, I denote the set of all interfaces, $\mathcal{IP}$ denote the set of all IP addresses, and $\mathcal{T}$ denote the set of tunnels (*e.g.*, established by SR and MPLS) in the network. To differentiate the direction of a flow at an interface, we use $I^* = I \times \{in, out\}$ to denote the set of interfaces labeled with flow directions. A traffic distribution graph is defined as a triple (N, E, w), where $N \subset \mathcal{F} \times (I^* \cup \mathcal{V}) \times (\mathcal{IP} \cup \mathcal{T} \cup \{\perp\})$ is the set of nodes in the graph, $E \subset N \times N$ is the set of edges, $w : E \to R^{[0,1]}$ is the weight function where $R^{[0,1]}$ denotes the set of real numbers ranging from 0 to 1 (inclusive).

B.2 TDG Construction

Algorithm 5 shows the overall construction algorithm. Function ConstructTDG takes a set of located flows as input and constructs the TDG in a standard BFS fashion as shown in function BfsExplore. Function ExploreNode takes a newly generated node and explores new edges and nodes according to the network's behavior on that node. Specially, we currently support the following five types of flow processing used in our production network.

• (Line 12-14) When a flow is received at an interface, a PBR policy defined on that interface may be applied to the flow. In this case, the algorithm needs to generate a node corresponding to the PBR's modification to the flow and its

Algorithm 5: Construction of TDG

```
1  Function ConstructTDG({(f_k, i_k)}_{k=1}^m):
2      S ← {(f_k, i_k, ⊥) : k = 1,..,m}; N ← ∅; E ← ∅; w is undefined;
3      BfsExplore(S, N, E, w);
4      return (N, E, w);
5  Function BfsExplore(S, N, E, w, Rib):
6      while S is not empty do
7          Pop a node n from S and add it to N;
8          ExploreNode(n, N, E, w, Rib);
9  Function ExploreNode(n, N, E, w, Rib):
10     Let n = (f, x, h);
11     switch (f, x, h) do
12         case x is an interface and f is incoming to x do
13             Generate a node n' = (f', y, h) if the PBR policy defined on x
               modifies f to f' and change its location to y;
14             Add e = ((f, x, h), (f', y, h)) to E and set w(e) to 1;
15         case x is an interface and f is outgoing do
16             Generate a node n' = (f, y, h') where y is the interface
               connected with x, h' is h if h is a tunnel otherwise ⊥;
17             Add e = ((f, x, h), n') to E and set w(e) to 1;
18         case x is a VRF and h is an SR tunnel do
19             If the router is the endpoint of h, then generate a node
               n' = (f, x, ⊥); otherwise generate a node n' = (f, y, h)
               where y is the outgoing interface of h on the corresponding
               router;
20             Add e = ((f, x, h), n') to E and set w(e) to 1;
21         case x is a VRF and performs BGP matching for f do
22             Generate nodes n_k = (f, x, h_k) for all BGP next hops h_k using
               Rib;
23             Add e_k = ((f, x, h), n_k) to E and set w(e_k) to 1/m for all k
               where m is the total number of next hops;
24         case x is a VRF and h is an indirect next hop do
25             Generate nodes n_k = (f, x, h_k) for all resolved next hops h_k
               (can be IP addresses or tunnels) using Rib;
26             Add e_k = ((f, x, h), n_k) to E and set w(e_k) according to the
               weight defined on corresponding the SR policy or IS-IS;
27     Add new generated nodes into S if they are not in N;
```

location (*e.g.*, redirect the flow to a VRF), and add an edge with weight 1 to the node.

• (Line 15-17) When a flow is ready to be forwarded out to another router, the algorithm generates a node for the receiving interface and adds an edge with weight 1.

• (Line 18-20) When a flow is forwarded in an SR tunnel, the algorithm needs to generate nodes based on whether the flow reaches the endpoint. If so, the algorithm generates a node with next hop $\perp$ indicating that the flow reaches the endpoint of the tunnel and needs further lookup for the next hops; otherwise, the algorithm generates a node with the same next hop and the corresponding outgoing interface. The weight of the edge is set to 1 in both cases.

• (Line 21-23) When a flow needs to look up the BGP RIB on a VRF, the algorithm needs to generate nodes for all next hops. To reflect the BGP ECMP mechanism implemented by router vendors, the weights of the edges to those nodes should be set equally.

• (Line 24-26) If an indirect next hop needs to be resolved by IS-IS or SR, the algorithm needs to generate nodes for each resolved next hop. The weights on the edges to the nodes should be correctly set based on the SR configuration or the IS-IS protocol (*e.g.*, normal ECMP).

C Traffic Equivalence Class

In this appendix, we will formally define the traffic equivalence class (*i.e.*, TEC, first presented in §4.4), and then detail our TEC construction algorithm.

C.1 Formal Definition

We define traffic equivalence class (TEC) using the notion of TDG. Given a TDG G, we use $G[f_1 \mapsto f_2, i_1 \mapsto i_2]$ to denote the TDG obtained by changing f_1 to f_2 and i_1 to i_2 for all nodes in G.

We first define the notion of *flow equivalence*.

Definition C.1. *Two located flows* $(f_k, i_k), k = 1, 2$ *are equivalent, denoted as* $(f_1, i_1) \equiv (f_2, i_2)$*, iff* $G_1 = G_2[f_2 \mapsto f_1, i_2 \mapsto i_1)]$ *and* $G_2 = G_1[f_1 \mapsto f_2, i_1 \mapsto i_2)]$*, where* $G_k =$ ConstructTDG$(\{f_k, i_k\})$ *is the TDG constructed for* (f_k, i_k)*.*

Now we define the notion of traffic equivalence class below.

Definition C.2. *A set of located flows* $\{(f_k, i_k), k = 1, \cdots, m\}$ *forms a traffic equivalence class, if* $(f_j, i_j) \equiv (f_k, i_k)$ *for all* $j, k \in \{1, \cdots, m\}$*.*

C.2 TEC Generation

Ideally, to achieve optimal efficiency we may want each TEC to contain as many flows as possible. However, this task is computationally hard due to the large number of flows. To balance computational efficiency with the optimality of TECs, we adopt a *compositional* approach. We establish equivalence classes for the value space of each field based on a global view of the network and then compose these classes together.

Let D and S be the sets of all prefixes (which can be efficiently collected from RIBs and router configurations) used to match destination and source IPs, respectively. Two flows $((s_1, d_1, v), i_1)$ and $((s_2, d_2, v), i_2)$ are considered equivalent if (i) the longest prefix match in S (D, resp.) for s_1 (d_1, resp.) is the same as that for s_2 (d_2, resp.) and (ii) two interfaces i_1 and i_2 sit in the same VRF and are configured with the same PBR policy. The correctness relies on the fact that at any router, there must be a subset of prefixes $D' \subset D$ ($S' \subset S$) used to match destination (source) IPs; therefore, d_1 and d_2 (s_1 and s_2) still share the same longest prefix match, leading to the same forwarding behavior and traffic distribution. While establishing equivalence for DSCPs is feasible, we do not further consider such equivalence, since our WAN only uses a small number of them.

D Incremental TDG Construction and Traffic Simulation

In this appendix, we provide the detailed description of our incremental TDG construction and traffic simulation algorithms, in §D.1 and §D.2, respectively.

Algorithm 6: Incremental TDG construction

```
1  Function IncUpdateTDG((N,E,w), ΔRib, Rib'):
2      N' ← N; E' ← E; w' ← w;
3      let chg be the set of nodes n ∈ N s.t. n representing the
         matching of some r ∈ ΔRib or a failed router/link;
4      forall n ∈ chg do
5          remove e from E' and w'(e) ←undefined for all n's
             outgoing edge e;
6      BfsExplore(chg, N', E', w', Rib');
7      return G' = (N',E',w'), chg
```

D.1 Algorithm for Incremental TDG Construction

Algorithm 6 presents how to incrementally construct a TDG for a given failure model. The algorithm takes the old TDG, the set of changed routes (*e.g.*, next hops changed) and tunnels (*e.g.*, tunnel removed) ΔRib, and the new RIBs (denote Rib') for the failure as input, and generates the new TDG incrementally. Note that simulating Rib' and ΔRib is easy, since many existing systems [12, 41] can extract the information.

In the first step, we find the nodes that may change in the given TDG. As described above, a node changes if (i) it represents route/tunnel matching for some r in ΔRib; or (ii) its location field is associated with a failed link or router (*e.g.*, the location field is an interface of some failed link). To efficiently identify those nodes, we maintain a mapping from routers, links, and routes/tunnels respectively to nodes in the TDG, when constructing a TDG (in the basic part).

Second, for those changed nodes *chg*, the algorithm removes all their outgoing edges and invalids all associated weights, since they may be inconsistent with the failure case. To reflect the new behavior under the failure, the algorithm performs the standard BFS algorithm from those changed nodes to construct the new edges and nodes using the new RIBs Rib'. Finally, the algorithm returns a new TDG with the set of changed nodes *chg* for the post processing (§5.2).

D.2 Detailed Description of Incremental Traffic Simulation Algorithm

Algorithm 3 shows the incremental traffic simulation for a single traffic snapshot. Using the approach described in §4.3, it can be naturally generalized to support multiple snapshots.

Algorithm 3 takes two parts of input: (i) the information computed in the basic part including the old TDG G, the volume V for nodes and edges in G, and the traffic distribution TL for all links, and (ii) the new TDG G' and the changed nodes *chg* computed by the incremental TDG construction (as detailed in §5.1).

The algorithm then computes the volume V' on nodes and edges incrementally based on the old volume V and also updates the traffic distribution TL incrementally based on the change of volume on edges. Similar to the traffic simulation algorithm in the basic part, the incremental algorithm computes the new volume of nodes and edges following the topological order of the new TDG G' but only for the affected nodes (line 6-16). Initially, the affected nodes α contains all nodes in *chg* and those having incoming edges from *chg* (line 5) as described above, and keeps adding potentially affected nodes to α. For an affected node n, the algorithm updates its volume (line 9) and computes the volume of all its outgoing edges in G' (line 11). Then the algorithm checks if any outgoing edges' volumes get changed for all the edges in both G and G'. If some edge's volume has changed, the node on the other end of the edge may become affected and so the algorithm adds that node to α (line 15). Furthermore, the algorithm also updates the traffic distribution by adding the changed volume to the corresponding link if needed. Finally, the updated TL contains the traffic distribution for all links under the failure and is returned.

POSEIDON: A Consolidated Virtual Network Controller that Manages Millions of Tenants via Config Tree

Biao Lyu[*†], *Enge Song*[†], *Tian Pan*[†], *Jianyuan Lu*[†], *Shize Zhang*[†], *Xiaoqing Sun*[†], *Lei Gao*[†], *Chenxiao Wang*[†], *Han Xiao*[†], *Yong Pan*[†], *Xiuheng Chen*[†], *Yandong Duan*[†], *Weisheng Wang*[†], *Jinpeng Long*[†], *Yanfeng Wang*[†], *Kunpeng Zhou*[†], *Zhigang Zong*[†], *Xing Li*[*†], *Guangwang Li*[†], *Pengyu Zhang*[†], *Peng Cheng*[*□], *Jiming Chen*[*], *Shunmin Zhu*[¶†□]

[*]*Zhejiang University* [†]*Alibaba Cloud* [¶]*Tsinghua University*

Abstract

As the cloud rapidly expands in scale, the virtual network controller must manage an increasing number of devices with higher update frequencies. Furthermore, the emergence of cloud-native services has substantially intensified program-triggered updates, leading to more frequent API invocations. To enhance performance and extensibility, we propose POSEIDON, a novel virtual network control framework. Specifically, to reduce operational expenses (OpEx), we have consolidated the common functions of multiple service controllers into a single controller. To manage heterogeneous devices and eliminate the multi-table lookup complexity due to config dependencies, we introduce *Trident*, a tree-based service- and device-independent abstraction, so that config dependency calculation can be replaced by more efficient tree traversal. After deploying POSEIDON on Alibaba Cloud, we observed a 21x increase in the throughput of virtual network configuration tasks, along with a 4.4x decrease in the P99 API processing latency. POSEIDON completes the task of enabling hundreds of Elastic IP addresses (EIPs) 1.8 to 55 times faster than Vendors A and B, both of which are among the top 5 providers [6], for identical network configuration jobs.

1 Introduction

Today's cloud virtual networks are managed by multiple separate controllers [14], which are responsible for configuring, monitoring, and recovering individual service [14, 30, 31] in the cloud virtual networks, such as virtual private cloud (VPC), VMs, *etc*. Managing and configuring large-scale cloud virtual networks are extremely challenging due to the following reasons:

- **Rapid growth of northbound API calls and southbound devices:** The controller needs to process massive number of network configuration API calls which are generated by cloud users, network operators, *etc*. In 2022, our cloud controller processed ∼17 trillion API calls per day. To make it worse, the number of devices to be managed also increases dramatically. In our cloud, a single ACL rule change might lead to config updates across ∼100,000 servers.
- **Long search chains and large table size:** A virtual network service, such as VPC, usually depends on other services, such as ACL and routing. When configuring virtual networks, the controller needs to query databases associated with the targeted service and its dependent services. Since the number of services increase dramatically over years, the dependencies become so complex such that we need to spend lots of engineering effort to process each API call. In addition, the size of tables in each database increases dramatically as well, which makes the table lookups for API parsing more time-consuming. We observed that due to these two factors, the 90% (P90) tail completion time for handling an API call has nearly doubled over the past year.
- **Cloud-native applications intensify the controller performance requirements:** Cloud-native applications [4] require extremely high throughput for concurrent resource creation/deletion, which demands much higher controller performance compared to traditional console-based network configuration. For instance, to handle surges in user access during peak events, social media applications require the creation of tens of thousands of backends to be completed within an exceptionally short time.
- **High cost of managing separate controllers:** To ensure the agility and iterative development of various services, we build and maintain separate controllers, one for each service (same as Andromeda [14]). In the past few years, with the growth of services, the number of controllers has boomed to over 50, leading to a substantial increase in the costs of managing numerous controllers by separate teams.

Some existing works focus on improving control efficiency and minimizing direct human interaction with network devices [13], with device-independent unified config abstraction/model [10, 16, 20–22], or even automating control decision-making with user intent [17–19, 27]. Andromeda [14] introduces how to improve the performance and scalability of pushing the configurations to Google's virtual network devices. However, they do not touch how to compute

□ Co-corresponding author

the updates of configurations (abbreviated as configs) needed on a physical server based on the tenant's intent, which is the actual performance bottleneck of a virtual network controller.

To address the above challenges, we propose POSEIDON, a virtual network controller consisting of a unified virtual network control abstraction over heterogeneous devices and services. POSEIDON introduces innovations in three aspects.

1) At the architecture level, we find that among several modules in a controller, only the module that parses user intents changes frequently due to the addition of new APIs, devices and dependencies. Other parts do not change much. Therefore, we consolidate and unify the more stable modules of different controllers into one, which greatly reduces the development and operational costs of the controller when introducing new services and new devices.

2) At the abstraction level, in order to unify the management of heterogeneous devices and diverse services, we abstract them into generic objects. Then, to eliminate the table lookup complexity due to dependencies between configurations, we propose *Trident*, a novel abstraction for describing configuration dependencies. We find that the dependency between configs is transitive, so we can associate multiple services to generate a config tree. Based on this tree, we can easily find the configs that a particular config depends on, as well as the devices they configure, by traversing the nodes, thus eliminating traditional complex database table lookups.

3) At the implementation level, to address the issue of large config tree traversal time in production, we deploy a traversal cache to bypass the time-consuming reverse tree traversal for top tenants and design a hierarchical storage structure for achieving both high I/O performance and strong data stability.

We deploy POSEIDON on Alibaba Cloud and keep it running for more than 3 years. We obtain the following results.

- With POSEIDON, virtual network controller's latency is significantly reduced and throughput is increased. When enabling hundreds of EIPs (Elastic IP address) [8] in a same VPC, POSEIDON's latency is 1.8x~55x and 2.6x~4.8x lower than cloud vendor A (Top 5 [6]) and cloud vendor B (Top 5). Compared with the previous controller, its throughput has increased from 160 TPS to more than 3400 TPS (21x).
- The costs of developing and maintaining lots of controllers has been significantly mitigated with the consolidated controller in POSEIDON. The controllers' lines of codes has been significantly reduced by 22%~41%. Beneath the consolidated controller, the human efforts of developing a controller for a new service is reduced by 50%.
- The consolidated controller has taken over lots of the workload from each service controller, reducing the CPU and memory consumption by 50%.

2 Background and Challenges

In this section, we present the virtual network configuration workflow and discuss the challenges encountered during years of production deployment of our virtual network controller.

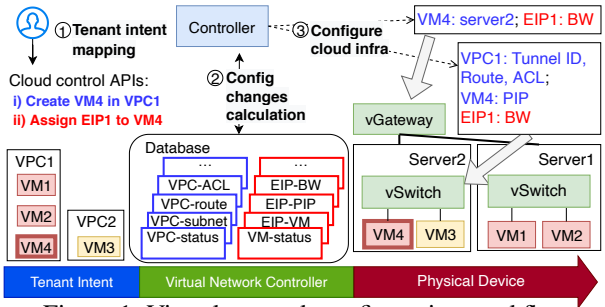


Figure 1: Virtual network configuration workflow.

2.1 Background

As Fig. 1 shows, the workflow consists of three steps:

① **Tenant intents mapping to cloud control APIs.** When tenants attempt to configure their virtual networks (*e.g.*, VM create/deletion), they use the cloud control APIs [1–3, 7] provided by cloud vendors to describe their intents. The cloud control APIs are standard interfaces for tenants to create, read, update and delete their virtual resources (*e.g.*, VMs). Different vendors offer different suites of cloud control APIs [1–3, 7]. Taking VPC configuration as an example, in Fig. 1, a tenant intent of "create VM4 in VPC1 with a public address EIP1" can be implemented with two cloud control APIs as "create VM4 in VPC1" and "assign EIP1 to VM4", which are fed to the virtual network controller for further processing.

②**Device config changes calculation.** After receiving tenants' API calls, the controller needs to compute the necessary updates to be configured on the physical devices. This is done with the following steps.

- Step 1: Identify all the dependent configs. Let us use the VM4 creation in VPC1 as an example (see Fig. 1). Because VM4 belongs to a specific VPC, to create VM4, we need to identify the VPC VM4 belongs to, which is VPC1. In addition, we also need to identify the dependencies of VPC1, such as ACL rules and routing tables. This step is done by doing SQL-based database queries. In this process, many tables, such as VPC-ACL, VPC-VM, *etc*, are queried.
- Step 2: Identifying the physical devices onto which the configs must be installed, as any network config is implemented on the physical device. As shown in Fig. 1, Server2 and vGateway need to be configured with the dependent configs.
- Step 3: Config changes calculation. After executing step 1, we know the configs needed for a API call. With step 2, we know the targeted devices and their existing configs. If the configs in step 1 already exist on the device, no config change is applied on the physical server. In contrast, the difference between the two is pushed to the physical server.

In our initial controller implementation, for each cloud control API, we write SQL code to manipulate databases and use if-else control block to arrange the SQL query order based on our service logic and the value acquired by table lookups. As device number, table size and if-else logic grow, the config changes calculation time is increasing rapidly.

③**Physical device configuration.** In the last step, the calculated config changes are pushed to the physical device. For

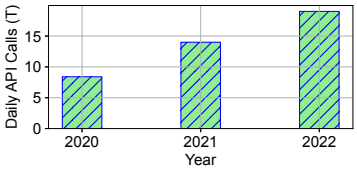


Figure 2: Daily API calls grow year by year in our cloud.

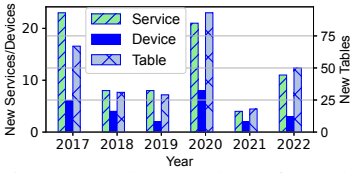


Figure 3: The number of yearly added services, devices and tables.

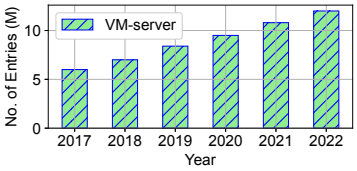


Figure 4: Number of entries in VM-server table of a region.

example, in Fig. 1, the intent of "create VM4 in VPC1" is eventually translated into the following device configs: the private IP (PIP) and VPC ID are configured on vSwitch for VM communication and VXLAN tunnel encapsulation, the routes and ACLs of the VPC are also added to vSwitch to regulate VM traffic, the VM-server mapping is installed to vGateway to route traffic from remote VMs to the VMs on the local host. Apart from that, in production, we need to deal with devices with different hardware forms, *e.g.*, x86- or Tofino-based vGateway [25]. Therefore, the controller needs to ensure that the translated configs in installed on these heterogeneous devices both timely and correctly.

2.2 Challenges

In this section, we describe two challenges of building a virtual network controller that manages millions of tenants.

2.2.1 Insufficient Performance for Growing Workloads

The scale of networks and the number of services grow dramatically over the past decades. We find that previous controller has degrading performance when handling the growing workloads due to the following three reasons.

Rapid growth of northbound API calls and southbound devices. Over the years, the usage of cloud resources by tenants has significantly increased. For example, in the past two years, the average number of PIPs within a VPC has increased from hundreds of thousands to millions. This causes a rapid growth in the number of VMs in VPCs. When a cloud user creates, deletes or updates of a VM, the user has to call the cloud controller API to take the corresponding actions. As the number of VMs increases, the number of cloud control API calls increases dramatically as well. Meanwhile, the growing number of tenants further exacerbates the frequencies of API calls that the controller needs to handle. As shown in Fig. 2, the API calling frequencies have doubled in two years, reaching tens of trillions of calls within a single day.

Furthermore, as the scale of an individual VPC increases, the physical devices the VPC covers also expand. This leads to a sharp rise in the number of southbound devices that a single API call needs to configure, imposing additional performance overhead on the controller. For instance, in extreme cases, a change of a single ACL rule in a large VPC may require a batch configuration to more than 100,000 servers.

Long search chains and large table size. With expansion of our cloud scale, the number of services and their dependencies increase, leading to a growing number of tables that need to be queried sequentially to finish the process of a single API call (sometimes dozens of tables are queried). Fig. 3 shows the recent growth of tables in our cloud. When a new table is added to record the relationship between services, the number of tables involved in the SQL query logic of some APIs may increase. Since the cloud controller needs to handle trillions of API calls each day, the additional table queries in a single API causes a significant burden on our database. Simultaneously, with the increase in the number and scale of virtual networks, the number of entries in major tables also grows, resulting in a longer database search/update latency. Using the VM-server table as an example, Fig. 4 shows its recent growth.

Due to the significant growth in table size and the number of tables being queried, over the span of one year, the P99 latency of calculating config changes for one API in our cloud (configure the vGateway) has increased from 860ms to 1615ms. The growth in latency has far exceeded the performance improvements of our controller that employs nested table searches, resulting in a lack of scalability in performance.

We follow a distinct cyclical pattern to iterate our systems in Alibaba, typically spanning several years. In the beginning of a span, a significant number of new services/devices/tables will be added to our cloud (*e.g.*, 2017 and 2020 in Fig. 3). After that, a considerable amount of time is required for debugging and large-scale deployments. In this period, new devices/services/tables (*e.g.*, 2018, 2019, 2021 and 2022 in the figure) are added only for reliability or security reasons, hence the number will be very limited. After that, we will kick-off a new cycle with a batch of new services/devices/tables.

Cloud-native applications intensify the controller performance requirements. The emerging cloud-native applications [4] require that the controller allocates and releases resources in a short time to adapt to the elastic changes in workloads. Such program-triggered API calling approach results in a significantly higher frequency than tenants manually configuring the network, posing challenges to the controller's performance on throughput and latency. For example, in e-commerce business, the time of peak traffic is known, and to save costs, resources will be massively scaled up (*e.g.*, tens of thousands of containers) just before the peak arrives, which needs a high-throughput controller. Similarly, cloud-native based social media applications need to handle surges in user access during hot events. To cope with the sudden increase in traffic, thousands of backends need to be elastically scaled in a short interval (*e.g.*, 500ms). This poses challenges to both the controller's throughput and latency. Insufficient controller performance will lead to backlog of tasks and delayed configurations. In the past, this will only impact the experience

of network operators as they need to wait a bit longer before configurations take effect. However, for cloud-native applications, when the controller cannot support rapid network configurations for a large number of new instances, the unexpected surge in user traffic cannot be handled, resulting in service interruptions and user experience degradation.

2.2.2 High Cost of Managing Separate Controllers

Service-dedicated controllers for rapid iteration. In the beginning, we had only a small number of services, and different service teams developed and maintained service-dedicated controllers independently to ensure rapid iteration and isolated deployment. If we develop a monolithic controller that covers all services, huge continuous code integration efforts will be needed for such a complicated system, hindering iteration pace. Google's Andromeda [14] also holds a similar viewpoint. They believe that different services and their associated devices need to be managed by separate controllers [14].

However, as our services expand, the number of controllers has increased to dozens (*e.g.*, LB controllers, VPC controllers, cross-region controllers), and the development and maintenance costs of these controllers gradually become unsustainable. Fig. 3 shows the annual additions of services, devices, and tables between services in our cloud. When a new service is added, we need to not only develop new APIs, but also deal with possible dependencies with existing services. When a new device is added, it requires extensive development of device-specific tasks in the controller, such as life cycle management, consistency checks, device-dependent rule translations. Furthermore, during rapid iteration, each controller requires dedicated personnel from the service team for maintenance. The maintenance costs also increase linearly with the number of controllers.

Code redundancy and logic dependencies between controllers. In the service-dedicated controller architecture, we discover that the code redundancy for the southbound interfaces of different controllers is relatively high, as these interfaces are service-agnostic and primarily responsible for device management and configuration. However, each service team independently optimizes the performance of their interfaces, leading to significant waste of human resources.

In addition, as the business grows, the dependencies between services have become increasingly complex. A cloud control API processing may require multiple controllers to collaborate. For example, the Internet gateway is managed by both the VPC controller and the cross-region controller, so any modifications to the VPC controller must also consider the potential impact on the cross-region controller. The code redundancy and logic dependencies diminish the benefits of managing separate controllers.

3 Design

3.1 Design Overview

To address these challenges, we propose POSEIDON, a virtual network controller that can manage a large-scale cloud with millions of tenants. It contains the following 5 key designs:

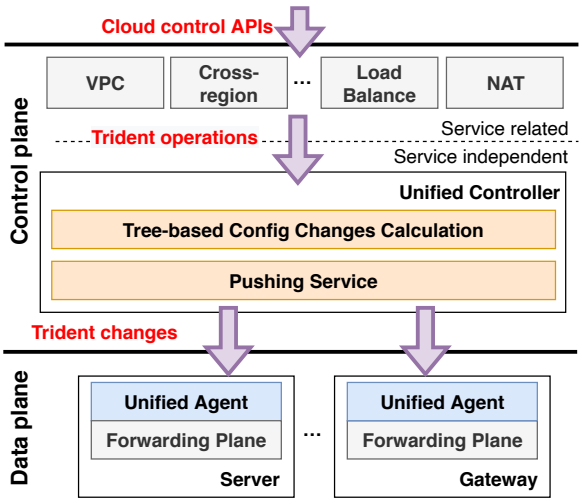


Figure 5: Layered controller architecture of POSEIDON.

Partial consolidation architecture. To reduce the costs of developing and maintaining multiple service-dedicated controllers, POSEIDON introduces a layered design that incorporates the consolidation of common logic among various services into a unified controller (§3.2), as shown in Fig. 5.

Trident: Service-independent abstraction. To unify the management of heterogeneous devices and diverse services, we design *Trident*, a service- and device-independent cloud control abstraction for the unified controller. Trident exposes 5 operations over 3 objects, allowing for flexible programming of existing cloud control APIs. Specifically, in order to achieve equivalence with SQL-based config changes calculation, Trident offers Relate/Unrelate, a pair of unique operations to represent the relations between objects (§3.3).

Config representaion through Trident tree. By continuously recording Trident operations, we can build a tree-like data structure named as Trident tree. Trident tree represents configurations, devices and relations between them (§3.4).

Tree-based config changes calculation. With Trident tree, the service-related table lookup logic (SQL+if-else) is replaced by equivalent tree traversal which decoupled config changes calculation from specific service logic. To accelerate the config calculation process, we take advantage of the observation that the descendant relation is transitive in the Trident tree and propose function-based descendant relation calculation with $O(1)$ computational complexity (§3.5).

Cloud-scale performance optimizations. For production deployment, we design a hierarchical storage structure for POSEIDON and deploy a traversal cache to bypass the time-consuming reverse Trident tree traversal for top tenants who have substantial number of virtual network elements (§3.6).

Fig. 6 shows the workflow of device configuration of POSEIDON. In POSEIDON, after tenant intents are mapped to cloud control APIs, instead of writing (SQL + if-else) query logic, the service-related controllers maintained by individual service teams translate the APIs into Trident operations.

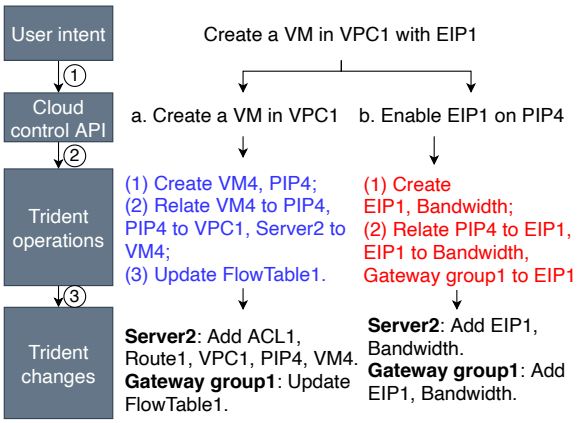


Figure 6: Device configuration workflow of POSEIDON.

Then, the unified controller changes the config tree topology according to the Trident operations and conducts tree-based config changes calculation to generate the device-independent Trident changes, which will finally be configured into devices.

3.2 Consolidation of Separate Controllers

In this section, we discuss the problems of consolidating all the functions of multiple controllers into a big one. Then, we introduce the partial consolidation adopted by POSEIDON.

3.2.1 Problems with full consolidation of controllers

To reduce the overhead of multiple controllers, it is natural to consider consolidating their functions into one big controller. However, full consolidation incurs significant costs. In our clouds, each service-dedicated controller is rapidly iterated to satisfy changing service requirements. Before each iteration goes online, it is necessary to conduct comprehensive testing of the controller on all relevant APIs and their parameters. Given there are x controllers (in our cloud x is 50~60), and each is iterated $Iter_i$ times a month and has $Test_i$ test cases to cover before put online, the total test cost is $O(\Sigma_{i=1}^{x} Iter_i * Test_i)$. Comparatively, after full controller consolidation, the iteration frequency will be the sum of the iteration frequencies of each controller. Moreover, on each iteration, all the test cases need to be examined and the total test cost grows to $O(\Sigma_{i=1}^{x} Iter_i * \Sigma_{i=1}^{x} Test_i)$, which is likely to impede the rapid iteration of cloud services.

3.2.2 POSEIDON's choice: partial consolidation

Despite the significant cost of full consolidation, we have observed that some parts of the controller are service-independent, which can be consolidated without high cost. For example, at the northbound of the controller, the API parsing logic is service-related and constantly undergoes rapid iterations. In the middle, although config changes calculation for different services may involve orchestration of different table query sequences, the underlying table query mechanism is quite similar. At the southbound, the controller needs to interact with devices. The code for interacting with different devices often shares a significant amount of common logic such as life cycle management which transitions between device states. Additionally, the device update frequencies are usually much lower compared to the service iteration frequencies (Fig. 3), making southbound logic more stable.

Based on the above observations, to maintain the flexible iterative capability of cloud service at the northbound and reduce the OpEx of maintaining multiple controllers, we carry out partial consolidation of common parts of each controller. Specifically, we build a service-independent abstraction layer in the middle, offering a set of atomic operations, enabling flexible orchestration of diverse northbound service requirements using these atomic operations. Past SQL + if-else approach poses a high demand for developers' understanding of mechanisms of cloud networks. With the new abstraction, developers only need to parse APIs into these atomic operations that are independent of both services and devices. Beneath this abstraction layer, we consolidate the implementations of different controllers for config changes calculation and device configuration to reduce redundant development and maintenance costs, as shown in Fig. 5. As mentioned in §2.1, config changes calculation is the most time-consuming step. Consolidation helps us concentrate on its optimization collectively, rather than conducted by individual teams.

Furthermore, for installing configs on heterogeneous device, we deployed a unified agent on each device for receiving Trident changes, as shown in Fig. 5. There is a device-specific program for translating unified Trident changes into device-specific primitives on the agent. The program is developed by the device management team, who is more familiar with the command-line interface of the device.

3.3 Service-independent Abstraction

We first introduce three goals of designing abstractions for managing virtual network, then elaborate on how POSEIDON abstracts the API parsing logic and various services/devices.

3.3.1 Goals of designing abstraction

- **Service-independent:** The processing logic is agnostic to service, namely, all APIs share a unified codebase.
- **Device-independent:** The objects should hide device heterogeneity to ease the programming with the abstraction.
- **Equivalence:** The results obtained with the abstraction should be the same as past SQL + if-else based approach.

3.3.2 POSEIDON's abstraction

Based on the design rationale, we propose *Trident*, the cloud control abstraction for the unified controller in POSEIDON. Trident exposes 3 objects (Conf, Device, Group) and 5 operations (Create, Update, Delete, Relate, Unrelate) upward, facilitating the interpretation of existing cloud control APIs.
Conf: During virtual network configuration, the manipulation of tenant resources through cloud control APIs will finally translate into manipulation of configs (*e.g.*, routes and ACLs) on the physical device. We use *Conf* to denote the manipulable configs (*e.g.*, VM, ACL in Fig. 7) on the device.

Device: In order to hide the differences of underlying devices in the abstraction, we use *Device* to abstract different kinds of devices (*e.g.*, server, gateway in Fig. 7) that carry configs.
Group: To manage increasing workloads, horizontally scaling devices into clusters is common in the cloud [15, 25, 26]. Within the same cluster, all device share same configs. To avoid the repetitive config changes calculation for devices within the same cluster, we introduce *Group* as the representative of the devices to carry Confs for the cluster. For consistency, we also create a Group with the same name for an isolated device that does not form a cluster (*e.g.*, Server1).
Create/Update/Delete: The CRUD (Create, Read, Update, Delete) of tenant resources through cloud control APIs internally turns into CRUD of related Confs/Devices/Groups by the unified controller. Since this paper focuses on virtual network configuration due to Trident tree changes, we do not include the *read* operation in this paper, which will only involve retrieving values from the tree. The Create/Delete Devices takes place when devices are allocated/deallocated for the scale-out or scale-in of cloud workloads.
Relate/Unrelate: There may be relations between Confs, Devices and Groups. Specifically, for relations between two Confs, we use Conf1→Conf2 to denote that the successful configuration of Conf1 depends on Conf2. For example, VPC→ACL meaning that each time configuring a VPC, we need to configure its ACL simultaneously. For relations between Devices and Groups, we use Device1→Group1 to denote that Device1 belongs to Group1 as Device1 has the same configurations with other devices (if any) in Group1. For relations between Groups and Confs, we use Group1→Conf1 to denote that Conf1 is configured to all the devices attached to Group1. As Group1 represents a cluster of devices, we no longer associate Conf1 with each device attached to Group1 to reduce redundant configurations. In other words, the relation between Conf and Device will not exist. In Trident, we introduce *Relate* to add → between two objects if they have a relation and *Unrelate* to remove → if the relation disappears.

3.4 POSEIDON's Trident tree

With Trident, the processing of cloud control API is abstracted into creating/deleting objects and relating/unrelating objects to other objects. After that, multiple "Device→Group→Conf" chains are formed. To enhance visualization and comprehension, we represent these interconnected chains as a tree structure, referred as "Trident tree", *e.g.*, the tree in Fig. 7.

3.4.1 Building Trident tree

Fig. 6 shows the equivalent Trident operations for the two example APIs. These operations create, delete various objects, and modify the relations between them. For example, the Trident operations of API *a* in Fig. 6 will create two Confs, VM4 and PIP4, as shown in Fig. 7. Then, it will build relations between VM4, PIP4 and the dependent Confs. Additionally, VM4 needs to be related with the corresponding Group, indicating its creation on the Devices represented by that Group (*i.e.*, Server2). These two API calls form a path (marked with the blue and red path in Fig. 7). With more API calls, finally, a Trident tree that records the dependencies between objects is created as shown in Fig. 7. Since the configs within the Group are identical, we consider the Group as the root of the Trident tree in the following config changes calculation procedure.

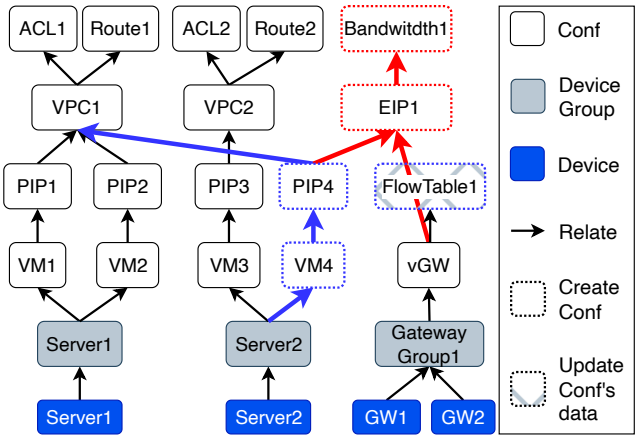


Figure 7: Trident abstraction and an example of Trident tree.

3.4.2 Handling API calls with Trident tree

Any API call can be represented as changes to the Trident tree, and based on the Confs changes of the Group, we can determine what updates should be made to the Devices within it. We use the API "create VM4 in VPC1" to illustrate this statement. To execute this API call, we have two steps:

1) We identify the Confs that VM4 depends on. This is achieved by traversing from VM4 to its descendants, which are PIP4, VPC1, ACL1, and Route1. Then, by traversing from these dependent Confs to their roots with a reverse tree traversal, we can locate the target Groups that carry these Confs (*i.e.*, Server1 and Server2).

2) We deduce the actual changes that need to be pushed to the devices based on the relations between dependent Confs and Groups. We can obtain the set of Confs carried by target Groups (*e.g.*, Server2) by traversing their descendants before and after changing the tree topology with Trident operations. By performing a diff operation on the two Conf sets, we can obtain config changes for target Groups. In Fig. 7, the added descendants of Server2 include VM4, PIP4, VPC1, ACL1, and Route1. Hence, the config changes of Server2 are "add VM4, PIP4, VPC1, ACL1, Route1". These changes are represented with Trident objects and device-independent.

3.5 Tree-based Config Changes Calculation

3.5.1 Basic procedure

Algo. 1 describes the solution adopted by POSEIDON. API calls change the tree topology by either adding/deleting tree nodes or modifying the relations between them (line 3). After the tree is updated, we can find all the dependent Confs of these Trident operations by traversing the tree (line 4). Then, by reversely traversing from these dependent Confs to their roots, we can obtain the target Devices. As Devices are represented by their Groups, finding Groups is enough for

subsequent calculation (line 5). For each Group, the Confs that need to be added/deleted/updated are obtained by detecting the changes of its descendants (line 6-13). For example, when a Group's descendants expand due to a new Conf being added, we need to add a corresponding configuration rule to the Devices represented by this Group (line 8-9).

Algorithm 1: Tree-based config changes calculation.

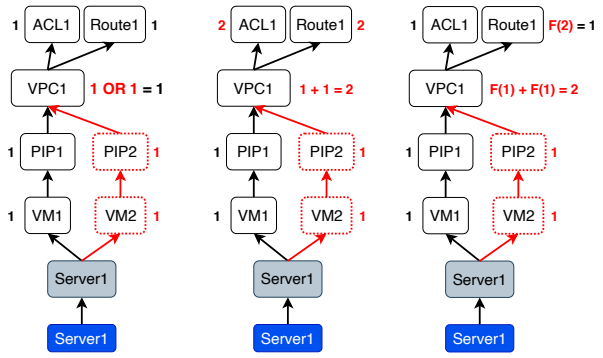
```
1  Function ChangesCalculate(Cloud control API):
2      Mapping API to Trident operations.
3      Change the Trident objects and their relations according to
         Trident operations.
4      conf_set = all the dependent Confs of Trident operations.
5      group_set = FindRoots(conf_set).
6      for group in group_set do
7          for c in conf_set do
8              if c will be added into descendants of group then
9                  Add c to group.
10             else if c will be removed from descendants of group
                 then
11                 Delete c from group.
12             else if c changes & c is in descendants of group then
13                 Update c in group.
```

3.5.2 Fast finding descendant changes for a Group

A critical step in this procedure is to verify whether a Conf resides in a Group's descendants (line 8, 10, 12).

Naive tree traversal is unscalable. The naive approach employs a widely used traversal algorithm, *e.g.*, depth-first search, which has a time complexity of $O(nodes)$, starting from the Group and checking each visited tree node. However, this solution is very time-consuming for large-scale cloud networks, as most devices accommodate a significant number of configs (*i.e.*, Confs in Trident tree), and the number of configs keeps growing as the device performance improves. For example, our gateway accommodates millions of VXLAN routing table entries [25]. It is inefficient if each API call requires a complete tree traversal, which will consumes hundreds of seconds for traversing the Trident tree with millions of nodes.

Descendant relation is transitive. We adopt descendant relation between Conf and Group to indicate that a Conf is within the descendants of a Group, that is, the Conf is configured to the Devices represented by the Group. We observe that in Trident tree, the descendant relation is transitive and can be propagated from parent nodes to their children. Specifically, if a Conf is configured on a Group, then all of its children must also be configured on the same Group. Because parent nodes need to be configured together with the dependent Confs (*i.e.*, their children) on the Group. Therefore, for a Conf, determining whether it has a descendant relation with a Group only requires examining its parents. If any parent of the Conf has a descendant relation with a Group, the Conf inherits the descendant relation with that Group. We employ *reference count* to record the descendant relation between Conf and Group, denoted as $ref(Group, Conf)$. For example, we can use $ref = 1$ to indicate there is a descendant relation while $ref = 0$ to indicate no relation.



(a) OR-based (b) SUM-based (c) Function-based

Figure 8: $ref(Server1, Conf_X)$ calculation. The changes and unchanges are marked by red and black, respectively.

There are multiple potential solutions to execute an API call on the Trident tree with transitive descendant relation. Here, we illustrate the pros and cons of two potential solutions.

Solution1: OR-based ref calculation. To figure out whether a Conf is a descendant of a Group, we can check if either of its parents is a descendant of the Group. This indicates that we can apply the *OR* operation on the reference count of all parents to obtain the reference count of the Conf itself. Thus, we have $ref(Group, Conf_A) = \vee_i ref(Group, P_i)$, where P_i is the ith parent of $Conf_A$. As shown in Fig. 8, $ref(Server1, VPC1) = ref(Server1, PIP1)$ OR $ref(Server1, PIP2) = 1$. The computational complexity is $O(P_i)$ (for a detailed analysis, please refer to §B.3).

Based on our experience, a Conf's parents can reach several million in our cloud. For example, during shopping festivals, a VPC can have millions of VMs. Therefore, the OR calculation due to a single API call (like VM create or delete) could take more than tens of seconds. In order to ensure accuracy of ref calculation, when a Conf's parent undergoes a change, the calculation triggered by changes of other parents will be blocked until the refresh of the ref for this round (*e.g.*, tens of seconds) is completed. This approach significantly limits the system's throughput and increases its latency, making it unsuitable for production deployment.

Solution2: SUM-based ref calculation. Another solution is asking a Conf to inherit all parents' ref by using a summation method, as defined by $ref(Group, Conf_A) = \Sigma_i ref(Group, P_i)$. Whenever the $ref(Group, Conf)$ becomes zero, we need to delete the Conf from the Group. Because the summation method adheres to the *reversibility property*, the computational complexity of this SUM-based calculation is $O(1)$ (for a detailed analysis, please refer to §B.4).

However, this solution also has a limitation, as any change in a node's ref propagates through all its descendants layer by layer. For example, creating VM2 of VPC1 on Server1 will cause $ref(Server1, VPC1)$ to increase from 1 to 2, as shown in Fig. 8. As ACL1 and Route1 are children of VPC1, this will result in synchronized changes for $ref(Server1, ACL1)$

Table 1: Changes of $ref(Group, Conf)$ when Trident tree changes. The blue and red text denote the changes due to the blue and red operations in Fig. 7, respectively. "0 → 1" represents the value changes from 0 to 1.

Group \ Conf	VM1, PIP1 VM2, PIP2	VM3, PIP3, VPC2 ACL2, Route2	VM4 PIP4	VPC1	ACL1 Route1	EIP1 Bandwidth1	FlowTable1
*Server*1	1	0	0	2	1	0	0
*Server*2	0	1	0 → 1	0 → 1	0 → 1	0 → 1	0
*Gateway Group*1	0	0	0	0	0	0 → 1	1

and $ref(Server1, Route1)$ as well. When the ref of a Conf changes only within the positive range, it implies that a descendant relation always exists and there is no need to trigger a configuration to the corresponding Group. For cloud networks, service dependencies are intricate and the tree depth can reach dozens of layers. While this propagation may trigger ref changes in many layers, it won't incur configuration updates, leading to a waste of computation resources.

Our solution: Function-based ref calculation. As discussed, the OR-based solution has high computational complexity, while the SUM-based solution has ineffective propagation. With lessons of both, we propose function-based ref calculation. Specifically, to reduce computational complexity, we retain the reversibility property advantage of the SUM-based solution, refreshing ref with $O(1)$ complexity. In addition, to address the ineffective propagation, we filter out irrelevant changes from a Conf's parents by counting the number of its parents with a non-zero ref. The ref calculation is defined by $ref(Group, Conf_A) = \Sigma_i F(ref(Group, P_i))$, where $F(x)$ is a piecewise function as $F(x) = 1, x > 0; F(x) = 0, x = 0$. With this function, any changes of x within the positive range will not alter its function value (remain 1), thereby preventing ineffective ref calculation propagation (hide children from irrelevant ref changes of parents).

Assuming we continuously create VMs of VPC1 on Server1, $ref(Server1, VPC1)$ will keep increasing. With our solution, $ref(Server1, ACL1)$ and $ref(Server1, Route1)$ will always remain 1, as shown in Fig. 8. This demonstrates that children won't inherit irrelevant ref changes from their parent. However, if there are no more VMs of VPC1 on Server1, $ref(Server1, VPC1)$ will be 0. With our solution, $ref(Server1, ACL1)$ and $ref(Server1, Route1)$ will become 0 in response because $F(0) = 0$. To sum up, our solution addresses the performance and overhead issues of previous two solutions, making it suitable for large-scale cloud networks.

3.5.3 Fast finding updates for a Conf.

When we need to update a Conf, we perform a reverse traversal on the tree from the Conf to locate the Groups that accommodate it. Then, we push the modification of the Conf to the Devices under these Groups. In production environment, there are always concurrent updates to the same Conf. To ensure the correctness of the order in which updates are pushed to devices, we introduce a *version* field to Conf. On each update, the version is incremented by 1. When multiple updates of a Conf are pushed to a device, the on-device unified agent maintains the update install order based on the version numbers and performs reordering for out-of-order updates.

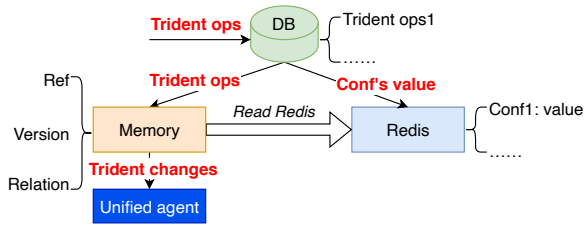


Figure 9: Hierarchical storage structure of POSEIDON.

3.5.4 A tree-based config changes calculation example

With the ability of ref and *version* to fast detect the changes of Group's descendants and updates of Conf, we can modify lines 8, 10, and 12 in Algo. 1. For example, we can replace the logic of line 8 by checking if ref transitions from 0 to a positive value. By observing changes in *version* and ensuring ref is not 0, we can determine whether an update needs to be pushed to a certain Group, which can replace the logic of line 12. With ref and *version*, unnecessary configurations can be efficiently filtered out. Finally, we read the value of the Confs that require deployment to generate the final Trident changes and push them to the unified agent. Table 1 illustrates the changes in ref resulting from Trident operations parsed from API a and API b in Fig. 6. Using these changes, we can efficiently compute config changes. For example, when $ref(Server2, ACL1)$ transitions from 0 to 1, it indicates that ACL1 needs to be configured on Server2. Similarly, an update to FlowTable1 will increase its *version* by 1, requiring this update to be pushed to the Gateway Group1, which has a descendant relation (with a non-zero ref) with FlowTable1.

3.6 Cloud-Scale Performance Optimizations

Hierarchical storage structure of POSEIDON. In order to achieve higher I/O performance while ensuring data reliability, we propose a hierarchical storage structure for POSEIDON, consisting of memory, Redis and database, as shown in Fig. 9. Among these, memory provides the highest performance but is vulnerable to data loss in case of power failure. Redis offers intermediate performance and a certain level of reliability, as it periodically batches data onto hard disk. The database is stored onto disk, offering the lowest performance but ensuring data persistence even during power loss.

POSEIDON needs to store various data objects, including Trident operations, *refs*, *versions*, Conf relation and Conf's value. Based on the data read frequency and reliability requirements, they are stored in three different storage mediums. Among them, Trident operations are derived from the interpretation of APIs. All other data structures are calculated based on them. Hence, Trident operations serve as the source of

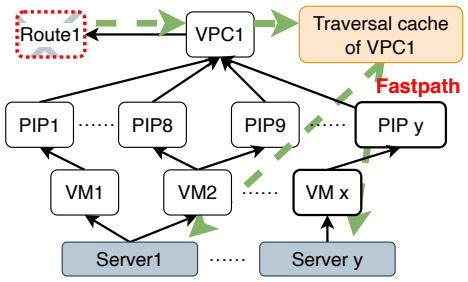


Figure 10: Traversal cache.

data for config changes calculation but will not be frequently accessed during the calculation. Therefore, they are suitable for storage in database as ground truth data. In the event of memory power loss, they can be used to restore other data. The *ref*, *version*, and relation involve high-frequency read and write requirements, thus they reside in memory. The entire computation process takes place in memory for ultimate performance. In addition, we store the Conf's value in Redis, as the Conf's value has a lower read frequency compared to *ref*/*version*/relation but higher than Trident operations. This is because the controller only needs to read the Conf's value in the final step of config changes calculation. To minimize the storage pressure on memory, we store it in Redis. In addition, in order to prevent data lost of Redis due to failover, we design a failover detection method in §B.6.

Traversal cache. Based on POSEIDON's deployment experiences, we have observed performance degradation when performing config changes calculation for some top tenants (*e.g.*, those with millions of VMs). In a real case, the reverse tree traversal time reaches a surprising 150s when a visited Conf has 10^5 parent nodes. To tackle this issue, we introduce *traversal cache*. The basic idea is to maintain an additional cache for nodes with an excessive number of parent nodes, which directly records the Groups obtained from the reverse traversal. As shown in Fig. 10, as the parents of VPC1 continue to increase, when reaching a certain threshold, we will attach a traversal cache to VPC1, which records the Groups holding descendant relation with VPC1. When Route1 changes, VPC1 will be visited during the reverse traversal to find the Groups. When VPC1 is visited, the Groups recorded in its traversal cache will be directly retrieved, eliminating the need for time-consuming traversal of the entire tree. This is shown in Fig. 10, where the green line with traversal cache bypasses the entire tree traversal. When Trident tree is updated by Trident operations, the traversal cache will be refreshed according to the function-based *ref* calculation.

4 Evaluation

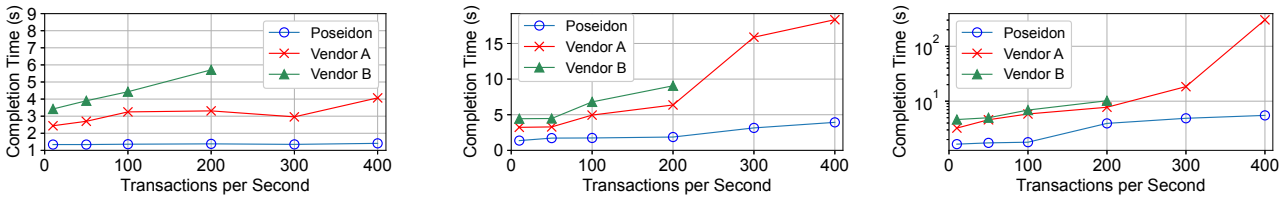
POSEIDON has been deployed in our production cloud for over three years and is responsible for handling the majority of API processing. The following data were collected from a production region containing hundreds of thousands of vSwitches, thousands of GWs, and tens of millions of VMs.

4.1 Performance improvement

End-to-end completion time under different TPS. *Experiment setting.* For a controller, the key performance metric is its concurrent processing capacity and the corresponding completion time. Since the concurrent capacity of unrelated API calls can be enhanced through horizontal scaling, we focus on the capacity for processing related API calls. Specifically, we conducted experiments to measure the concurrent capacity of Vendor A (Top 5 [6]), Vendor B (Top 5), and ours (Top 5) on two of the most widely used APIs in our public cloud services, that is, *Enable and Disable Elastic IP (EIP) [5] for a VM* in a same VPC. We have selected these two APIs for two specific reasons. Firstly, they enable us to precisely gauge the time it takes for the controller to process an API call, excluding any time spent on unrelated modules like the VM's startup time, which occurs when utilizing the VM creation API. Secondly, these two APIs represent a 10% of the 1200 APIs available to our customers. It should be noted that as POSEIDON utilizes the same logic and codes to handle all APIs, there is no specific optimization to these two APIs. We leverage the Cloud control API interfaces provided by the major cloud vendors to initiate API calls with different transactions per second (TPS). By recording the time taken for successful (for testing enabling EIP) or unsuccessful (for testing disabling EIP) pings to these EIPs, we can obtain the P50, P90, and P99 completion time. It is important to note that the timing starts from when the controllers of Vendor A/B and ours receive the API calls returning timestamp when the controller's acknowledgment of receiving the API calls), which effectively eliminates the impact of transmission latency on the measurements. All the experiments were conducted in the region of Jakarta[††] from August 20th to 29th, 2023.

Performance analysis. Fig. 11 depicts the P50, P90, and P99 completion time for enabling EIP. As shown in the figure, the completion time of Vendor A and Vendor B is 1.8x~55x and 2.6x~4.8x higher than that of POSEIDON. Moreover, our system demonstrates superior stability in high concurrent scenarios. Even with 400TPS, the P50 of POSEIDON remains consistently stable at 1.3s. The reason for testing Vendor B's performance only up to 200TPS is that when we attempted 300TPS, an unsupported error occurred. The specific error details can be found in the appendix (Fig. A1). Furthermore, it is worth noting that during the testing at 400 TPS, the majority of completion time for Vendor A are below 30s. However, there are 8 specific cases where the process of enabling EIP took over 300s. Thus, the P99 of Vendor A is much higher than that of ours (about 55x). This observation suggests that the Vendor A control plane encounters instability under high TPS scenarios. In the case of disabling EIP, the completion time for Vendor A and Vendor B are 1.6x~12x and 1.3x~2.5x compared to ours, respectively, which will be detailed in the appendix (Fig. A2) due to the space limits.

[††]We repeated the experiments across multiple regions and observed consistent results; for brevity, we illustrate only one set of results in this paper.



(a) P50 (b) P90 (c) P99

Figure 11: End-to-end completion time of API calling for enabling EIP.

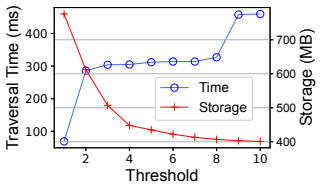


Figure 12: Threshold selection of traversal cache.

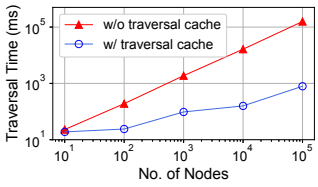


Figure 13: Traversal time optimization with traversal cache.

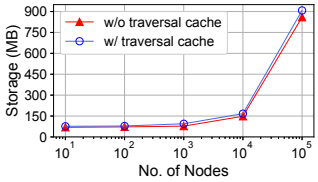


Figure 14: The storage usage (w/ vs. w/o traversal cache).

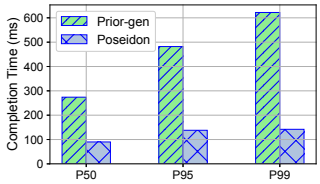


Figure 15: Completion time of config changes calculation.

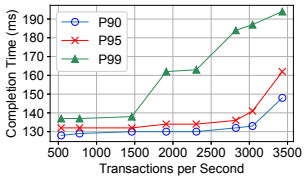


Figure 16: Completion time of config changes calculation.

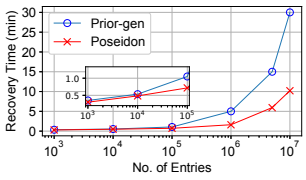


Figure 17: Recovery time after restarts.

Traversal cache. *Threshold selection.* The most important metric for the traversal cache is the threshold, which determines the number of parent nodes that trigger the attachment of a traversal cache for a Conf. We conducted experiments in a Trident tree containing hundreds of thousands of nodes. Fig. 12 shows the effects of the threshold on the P99 completion time of reverse traversal from all nodes to find their roots, as well as the storage consumed by the traversal cache. As the threshold increases from 2 to 8, the P99 completion time remains relatively stable, while the storage consumption decreases significantly. Once the threshold exceeds 8, there is a notable increase in the P99. Therefore, the threshold of 8 is a sweet trade-off point and is adopted in our production.

Traversal time optimization. Fig. 13 shows the P99 completion time for traversing (traversal time) different scales of Trident tree. As the number of nodes increases, the traversal time increases in both scenarios. When the Trident tree contains 100,000 nodes, the traversal time without traversal cache reaches 157s, which has a significant impact on the user experience due to a long configuring delay. When we attach a traversal cache for the node whose parents number exceeds the threshold, the traversal time is notably reduced. Even if the number of nodes reaches 100,000, the traversal time with traversal cache is less than 1s, that is, only 792ms.

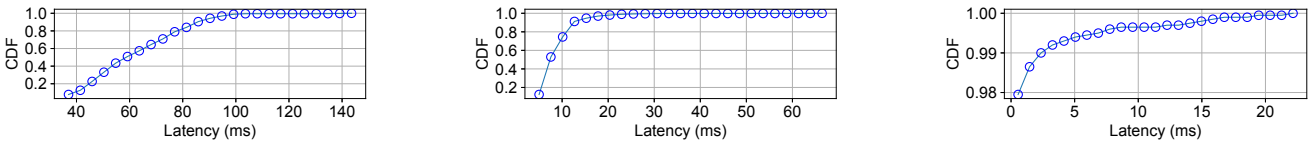
Storage usage. Fig. 14 illustrates the storage usage with/without traversal cache. As shown in the figure, traversal cache only causes a small piece of extra storage costs.

Config changes calculation. *Bottleneck of prior-gen.* After conducting a thorough analysis of the performance bottleneck of the prior generation controllers, we have figured that the main bottleneck resides in the table lookup I/O of database for calculating config changes. Based on our historical records, for the prior-gen, the P99 completion time for config changes calculation reached 9s at 160TPS. In order to overcome the I/O bottleneck in config changes calculation, POSEIDON caches the Trident tree in the memory and Redis, which significantly improves the the completion time and concurrent capacity of calculating config changes.

Completion time improvement. Fig. 15 depicts the config changes calculation completion time of the same API in the prior-gen and POSEIDON controller. The experiment is conducted by calling the API 100 times repeatedly. As illustrated in the figure, the completion time is notably improved compared to the prior-gen. The P50, P95 and P99 completion time of prior-gen are 3x, 3.5x and 4.4x larger than that of POSEIDON. In addition, compared to the prior-gen, a smaller difference between P99 and P50 of POSEIDON indicates a smaller variance and thus a more stable performance.

Concurrent capacity improvement. Fig. 16 shows the config changes calculation completion time of POSEIDON under different TPS. As illustrated in the figure, P99 experiences a drastic increases after 1500TPS. There is a sharp increase in P90 and P95 when the frequency of concurrent calls exceed 3000TPS. The P99 of POSEIDON is less than 200ms when the frequency of concurrent calls is 3432TPS. Compared to the prior-gen, where the performance sharply deteriorated to 9s after reaching 160TPS, the concurrent processing capability of the config changes calculation module in POSEIDON has been greatly enhanced.

Completion time of key steps in POSEIDON. Fig. 18 depicts the completion time distribution of three key steps in the POSEIDON workflow. The concurrent calling frequency is about 1,000TPS. As shown in the figure, config changes calculation is the most time-consuming work due to its high complexity and a multitude of I/O between memory and Redis. The P99 of config changes calculation, pushing and enabling is about



(a) Config changes calculation (b) Config changes pushing (c) Config changes enabling

Figure 18: Completion time of three key steps in the POSEIDON workflow.

100ms, 20ms and 3ms, respectively. It also can be seen from the figure that the CDF curve of config changes calculation (Fig. 18(a)) is the shallowest one since the completion time is greatly affected by the number of Confs and their dependencies in the Trident tree (as shown in Fig. 13), which varies dramatically. Besides, the tail in Fig. 18(b) and Fig. 18(c) is much longer than that of Fig. 18(a). The reason for the long tail in Fig. 18(b) is the extreme case for a small piece of customers. When pushing the updates of the routing table for a large VPC, the changes need to be pushed to hundreds of thousands of devices, resulting in a few outliers with extremely long completion time. The long tail in Fig. 18(c) is caused by resource constraints on the device when other applications exhaust the available resources.

4.2 OpEx and development cost optimization

OpEx. Since the common logic of controllers has been taken over by the POSEIDON unified controller, the service-related controllers focus on translating the API into Trident operations, leading to a notable reduction in the number of Lines of Codes (LOC) of controllers. As shown in Table 2, the reduction rate of LOC are 22%~41%, which demonstrates a significant mitigation in OpEx.

Development cost. Additionally, as the controller's development no longer needs to care about lots of works (*e.g.*, config changes calculation and pushing, consistency checking, *etc*), the development cost has also gained a substantial reduction. By comparing the development cost of two LB controllers with similar business logic, we found that the cost has been reduced by half. When developing a controller without POSEIDON, the human effort is 6 person-months and 66K lines of code are written. With POSEIDON, the human effort amounts to 3 person-months and we write 30K lines of code.

It is important to note that the reduction in OpEx and development cost has not been added to the POSEIDON platform, as the LOC of the POSEIDON is only around 150K, which is much lower than the total LOC reduction.

Table 2: Reduction in Lines of Code of controllers.

	LOC (prior-gen)	LOC (POSEIDON)	Reduction
LB1	167*K*	98*K*	41.3%
LB2	76.9*K*	46.9*K*	36.4%
VPC	873*K*	559*K*	36%
NAT	107*K*	65*K*	39.3%
VPN	97*K*	70*K*	27.8%
Private Link	31.8*K*	22.8*K*	28.3%
Accelerator	135*K*	105*K*	22.2%

Table 3: Resource consumption of unified agent.

	Physical device		Virtual element		
	SNA	Server	VM	LB	GW
CPU-Avg	0.12%	0.21%	0.81%	0.47%	0.16%
CPU-Max	0.94%	1.04%	1.88%	1.41%	0.99%
Memory	0.16%	1.30%	0.47%	0.31%	0.73%

4.3 Benefit and cost of unified agent

Recovery time after restarts. Fig. 17 illustrates the recovery time for devices carrying different numbers of entries. The more entries, the more reduction. The persistent storage of config changes by the unified on-device agent enables fast data recovery. The substantial optimization of recovery time mitigates the downtime of devices, which improves the overall utilization of devices and facilitates device iterations.

Resource consumption. The POSEIDON agent is deployed on the device, consuming computing and storage resources. We evaluate resources consumption for different physical devices (*e.g.*, Smart Network Appliance (SNA) [25]) and virtual elements, as shown in Table 3. For the CPU, the average usage remains below 1%, and the maximum usage is below 2%. As for memory usage, it does not exceed 1% for all cases except the server. This indicates that the agent only needs to consume a small piece of on-device resources.

5 Experiences of deploying POSEIDON

How to migrate to POSEIDON? For a large-scale public cloud provider, how to migrate from its old controller to POSEIDON without affecting the quality of ongoing services is a very challenging task. Initially, we leveraged lots of test cases to verify the consistency between the POSEIDON and the old controller. Only upon passing all test cases would we allow the replacement of the old controller with POSEIDON in the production environment. However, infrequently used services not covered by the test cases led to continuous online configuration errors. The practice of debugging after errors has seriously compromised the availability of our services, resulting in a substantial number of complaints. We gradually recognized that test cases could not guarantee comprehensive coverage of all usage of numerous services. Consequently, we opted for a dual-running test in production, leveraging massive real user API calls to eliminate uncovered corner cases. Specifically, we apply a dual-running test where both the old controller and POSEIDON are running to handle the same API calls, but deliver configs to production devices and test devices, respectively. For each physical device in production, POSEIDON will fully take over the configuration management only after it has accurately handled the API calls of over 90% of the tenants using the device, over 95% of the services im-

plemented by the device, and over 90% of the APIs related to the device for a specified period—this period being one week for smaller regions and one month for larger regions. Employing this strategy, the faults attributed to POSEIDON have been significantly mitigated throughout the year.

POSEIDON's performance in extreme situations: *Extensive route fluctuations.* For tenants with self-built Internet data center (IDC), their IDC's BGP routes must be learned by their VPCs in the cloud. Some tenants operate large-sized IDCs with thousands of BGP routes. When there are fluctuations in the links between IDCs and the cloud, it triggers massive BGP route convergence. In extreme cases, BGP route oscillation may also occur. All changes in BGP routes must be processed and configured by the controller. For prior-gen controller, it takes over five minutes to converge two thousand routes. With POSEIDON, the converge time is reduced to tens of seconds, significantly mitigating the impact on tenant services.

Large-scale device restart. Normally, network devices undergo scattered restarts due to upgrades or SW/HW issues. Extremely, there may be a widespread batch restart of devices, *e.g.*, during a power outage. Over the past years, we experienced three data center-level power outage incidents. The first two were handled by the prior-gen controllers. During restarts, a sudden surge in config retrieval requests from a vast number of devices directly saturated the database connections. Consequently, the recovery took hours. For the third incident with POSEIDON deployed, due to its trident tree algorithm and hierarchical storage, the recovery was finished in minutes.

Where to record the descendant relation between Conf and Group? How to record these relations in a Trident tree will affect the performance of configuration delivery, including the incremental configuration delivery (*i.e.*a tenant adding a new Conf) which ensures the timeliness of user configuration delivery, and the full configuration delivery (*i.e.*device restart) which is crucial to device upgrades and maintenance. Specifically, if each Conf records its targeted Device, it will be easy to calculate the incremental config changes, but the full configuration delivery will be very slow because all Confs must be traversed. In contrast, if each Group records the Conf it needs, the full configuration delivery will be fast because all required configs are already recorded in the Group, while the incremental configuration delivery will be slightly slower because the Trident tree needs to be reversely traversed to find the roots. Note that we cannot record the relation in both Conf and Group as this implementation contains too many locks, leading to terrible performance. Considering POSEIDON servers a large-scale public cloud, where device updates are extremely frequent, and stability maintenance is a critical task, we choose Group to record the descendant relation to achieve the best performance of full configuration delivery.

6 Related work

The previous abstraction works [9, 12, 16, 22–24, 27] provide a unified abstraction to shield the heterogeneity of devices, making the network configuration easier for network operators. For example, based on NETCONF [16], YANG [9] has been proposed to facilitate device-independent configuration. However, these abstractions only provide device-independent capabilities and do not support service-independent config changes calculation. Our proposed abstraction, Trident, eliminates the need to maintain service-specific config changes calculation logic for each controller, greatly reducing OpEx.

The architecture of network control systems has also been widely researched. [17, 21] manage various services and heterogeneous devices of physical network in unified manner, which could notably reduces the OpEx of managing networks consists of diverse devices. [17–19] introduce intent-driven network management, which takes network management a step further. Andromeda [14] and Achelous [30] design the mechanism of configuring the devices in their VPC with the config changes. However, all the above works do not touch the config changes calculation procedure in the virtual network, whose optimization is first discussed in this paper.

Additionally, there are several notable efforts [11, 28, 29] dedicated to achieving rapid, highly scalable, and automated database-based incremental view maintenance. We acknowledge that the adoption of these methods could indeed improve the controller performance with SQL-based optimizations. While further SQL-based optimization of the controller may meet current performance requirements, it sets us on a path where future performance improvements become heavily reliant on SQL-database enhancements—a challenge we have struggled with for over a decade. To have a better assurance of controller performance with predictable improvement, we choose to optimize the whole architecture, especially in virtual network abstraction and API processing logic. In this way, the performance enhancement is guaranteed irrespective of the SQL-database we employ. Nevertheless, combining SQL-based optimizations [11, 28, 29] and architecture optimization proposed in this paper might be an interesting exploration and the community could be inspired by POSEIDON.

7 Conclusion

This paper presents POSEIDON, a pioneering effort to build a unified virtual network controller capable of managing a virtual network at the scale of millions of tenants. With POSEIDON, we can save the OpEx of managing numerous controllers without sacrificing flexibility of services iterations. The innovative config dependency abstraction enables the calculation of config changes independent of both service logic and physical devices. We have been operating Alibaba Cloud with POSEIDON over 3 years, and it has substantially mitigated the workload of adding services/devices, lowered the configuration latency, and reduced network incidents.

Acknowledgements: The work was partially supported by National Science Foundation of China (NSFC) under Grants (62293511, 62372053). We thank our shepherd Soudeh Ghorbani and the anonymous reviewers for their helpful comments that greatly improved the paper.

References

[1] Alibaba Cloud API. https://api.alibabacloud.com/home, 2023.

[2] AWS Cloud Control API, manage cloud infrastructure with a consistent set of APIs. https://aws.amazon.com/cloudcontrolapi/, 2023.

[3] Azure REST API reference . https://learn.microsoft.com/en-us/rest/api/azure/, 2023.

[4] Build cloud-native applications in Azure. https://azure.microsoft.com/en-us/solutions/cloud-native-apps/, 2023.

[5] Elastic IP Address in Alibaba Cloud. https://www.alibabacloud.com/en/product/eip, 2023.

[6] Gartner Says Worldwide IaaS Public Cloud Services Revenue Grew 30% in 2022, Exceeding 100 Billion for the First Time. https://www.gartner.com/en/newsroom/press-releases/2023-07-18-gartner-says-worldwide-iaas-public-cloud-services-revenue-grew-30-percent-in-2022-exceeding-100-billion-for-the-first-time, 2023.

[7] Google Cloud Platform APIs and references . https://cloud.google.com/compute/docs/apis, 2023.

[8] What is an EIP? https://www.alibabacloud.com/help/en/eip/product-overview/what-is-eip, 2024.

[9] M Bjorklund. RFC 6020: YANG-A Data Modeling Language for the Network Configuration Protocol. *Tail-f Systems*, 2010.

[10] M Bjorklund. RFC 7950: The YANG 1.1 Data Modeling Language, 2016.

[11] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *Proceedings of the VLDB Endowment*, 16(7):1601–1614, 2023.

[12] Eric Hayden Campbell, William T Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. Avenir: Managing Data Plane Diversity with Control Plane Synthesis. In *USENIX NSDI 21*, pages 133–153, 2021.

[13] Huangxun Chen, Yukai Miao, Li Chen, Haifeng Sun, Hong Xu, Libin Liu, Gong Zhang, and Wei Wang. Software-Defined Network Assimilation: Bridging the Last Mile Towards Centralized Network Configuration Management with NAssim. In *Proceedings of the ACM SIGCOMM 2022*, pages 281–297, 2022.

[14] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *USENIX NSDI 18*, pages 373–387, 2018.

[15] Danielle E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *USENIX NSDI 16*, pages 523–535, 2016.

[16] R Enns, M Bjorklund, J Schoenwaelder, and A Bierman. RFC 6241: Network configuration protocol (NETCONF), 2011.

[17] Andrew D Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, et al. Orion: Google's Software-Defined Networking Control Plane. In *USENIX NSDI 21*, pages 83–98, 2021.

[18] Victor Heorhiadi, Sanjay Chandrasekaran, Michael K Reiter, and Vyas Sekar. Intent-Driven Composition of Resource-Management SDN Applications. In *Proceedings of the ACM CoNEXT 2018*, pages 86–97, 2018.

[19] Arthur S Jacobs, Ricardo J Pfitscher, Rafael H Ribeiro, Ronaldo A Ferreira, Lisandro Z Granville, and Sanjay G Rao. Deploying Natural Language Intents with Lumi. In *Proceedings of the ACM SIGCOMM 2019 Posters and Demos*, pages 82–84, 2019.

[20] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A Distributed Control Platform for Large-scale Production Networks. In *USENIX OSDI 10*, 2010.

[21] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. Automatic Life Cycle Management of Network Configurations. In *Proceedings of the ACM SIGCOMM 2018 Workshop on SelfDN*, pages 29–35, 2018.

[22] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[23] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. OpenDaylight: Towards a Model-Driven SDN Controller Architecture. In *Proceeding of the IEEE WoWMoM 2014*, pages 1–6. IEEE, 2014.

[24] Jeffrey C Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with Modeling Network Topologies at Multiple Levels of Abstraction. In *USENIX NSDI 20*, pages 403–418, 2020.

[25] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, et al. Sailfish: Accelerating Cloud-Scale Multi-Tenant Multi-Service Gateways with Programmable Switches. In *Proceedings of the ACM SIGCOMM 2021*, pages 194–206, 2021.

[26] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud Scale Load Balancing. *ACM SIGCOMM Computer Communication Review*, 43(4):207–218, 2013.

[27] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. Robotron: Top-down Network Management at Facebook Scale. In *Proceedings of the ACM SIGCOMM 2016*, pages 426–439, 2016.

[28] Debnil Sur, Ben Pfaff, Leonid Ryzhyk, and Mihai Budiu. Full-stack SDN. In *Proceedings of the ACM HotNets 2022*, pages 130–137, 2022.

[29] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. Building Scalable and Flexible Cluster Managers Using Declarative Programming. In *USENIX OSDI 20*, pages 827–844, 2020.

[30] Chengkun Wei, Xing Li, Ye Yang, Xiaochong Jiang, Tianyu Xu, Bowen Yang, Taotao Wu, Chao Xu, Yilong Lv, Haifeng Gao, Zhentao Zhang, Zikang Chen, Zeke Wang, Zihui Zhang, Shunmin Zhu, and Wenzhi Chen. Achelous: Enabling Programmability, Elasticity, and Reliability in Hyperscale Cloud Networks. In *Proceedings of the ACM SIGCOMM 2023*, pages 769–782, 2023.

[31] Shunmin Zhu, Jianyuan Lu, Biao Lyu, Tian Pan, Chenhao Jia, Xin Cheng, Daxiang Kang, Yilong Lv, Fukun Yang, Xiaobo Xue, et al. Zoonet: A Proactive Telemetry System for Large-Scale Cloud Networks. In *Proceedings of the ACM CoNEXT 2022*, pages 321–336, 2022.

Appendices

A Additional Figures

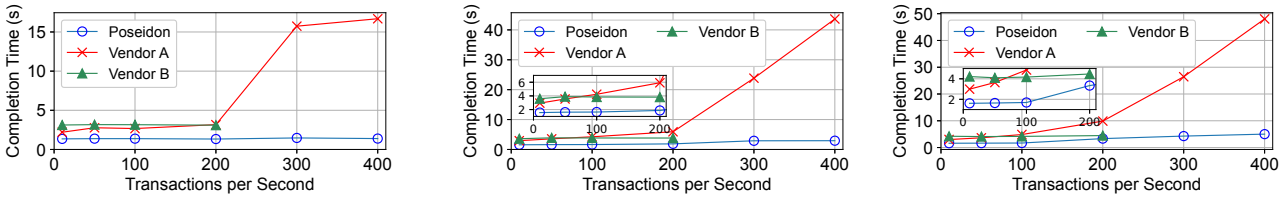
End-to-end completion time for disabling EIP. Fig. A2 illustrates the completion time of disabling EIP under different TPS. Compared to Vendor A, our cloud is less affected by TPS. The P50, P90 and P99 of Vendor A increase dramatically (about 8x, 15x and 16x compared to their lowest value, respectively). Meanwhile, the P50, P90 and P99 of ours only experience a slight increase, that is, the increase rate is 1.02x, 1.9x and 3.1x. respectively. The P50, P90 and P99 of Vendor B remain stable throughout all the tests, but it still cannot measure data above 200TPS. Moreover, Vendor A performs better than Vendor B at lower TPS.

For Vendor A, the completion time of enabling EIP (as shown in Fig. 11) is higher than that of disabling EIP, whereas for us and Vendor B, it is the other way around. When conducting enabling and disabling EIP in our cloud, both server and gateway require configuration. For the case of enabling EIP, both devices must be configured successfully for a successful ping. While for the case of disabling EIP, a unsuccessful ping is achieved when either device has been configured. Therefore, the completion time of disabling EIP is less than that of enabling EIP in our cloud.

Resource usage analysis. While theoretically we can use a single POSEIDON controller to serve all service-related controllers, it may lead to an explosion radius affecting all services. To minimize the explosion radius, we deploy independent POSEIDON controller for each service-related controller. However, the development and maintenance of these POSEIDON controllers are managed by a single team to reduce the OpEx. This low-sharing deployment may not effectively improve the overall resource consumption and, in some cases, may even increase it. We conducted a comprehensive analysis of CPU and memory consumption of the VPC controller with and without POSEIDON, as well as the resource usage of POSEIDON for VPC controller. Fig. A3 shows the total resource usage for processing all the API of VPC. The POSEIDON in the figure represents the total resource usage of VPC controller and POSEIDON controller. As shown in the figure, the CPU and memory usage of POSEIDON is 2.5x and 1.75x than that of prior-gen VPC controller. Based on our observations, the resource usage of the VPC controller halved because lots of works have been taken over by the POSEIDON controller. However, the works taken over by POSEIDON involve are the CPU- and memory-intensive tasks, *e.g.*, config changes calculation. In order to provide significantly improved config changes calculation performance compared to the prior-gen, the resource consumption of the POSEIDON's config changes calculation component is slightly higher than that of prior-gen VPC controller. In summary, the total resource usage of POSEIDON and VPC controller is higher than that of prior-gen

```
most one access config currently supported.
    raise core_exceptions.from_http_response(response)
        api_core.exceptions.BadRequest: 400 POST
                                                                                    At most one access config currently
supported.
```

Figure A1: Error of testing Vendor B for 400TPS.



(a) P50 (b) P90 (c) P99

Figure A2: End-to-end completion time of API calling for disabling EIP.

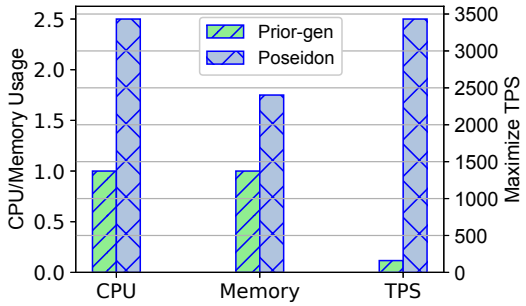


Figure A3: Resource usage increase and concurrent processing capacity improvement.

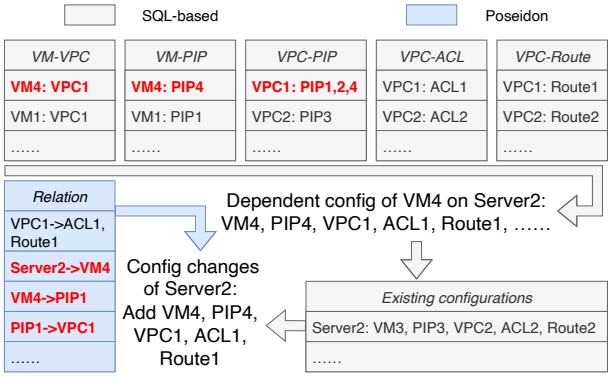


Figure A4: Config changes calculation (SQL-based vs. POSEIDON).

VPC controller. However, we believe that this trade-off is well worth it because the concurrent processing capacity of config changes calculation has increase by about 21x, as shown in Fig. A3.

B Supplementary Materials

B.1 Design goal achievement analysis of abstraction

Why Trident-based config changes calculation is service-independent. According to the Algo. 1, it can be observed that our code for config changes calculation is the same for any API, meaning it is independent of the service.

Why Trident-based config changes calculation is device-independent. Since we abstract all heterogeneous devices into a standardized object type, namely "Device", our implementation of config changes calculation and pushing is agnostic to the specific devices. The device-specific translation is done by the unified agent.

Why SQL-based and Trident-based config changes calculation is equivalent. To guarantee the equivalence, we analyze the essence of the cloud control APIs and their execution. The majority of cloud control APIs can be regarded as the CRUD (create/read/update/delete) of tenant resources (see §B.2). The execution of cloud control APIs is to translate tenant intents for resource manipulation into configurations on the physical device. As mentioned in §2.1, the controller will make config changes calculation for each cloud control API, which can further be divided into two steps: (1) find all the dependent configs of the target tenant resource; (2) search a table recording all the existing configs on each device to calculate the necessary changes to install. The original API execution approach is service-dependent since the key/value of each table and the table query sequence need prior knowledge of the related service. For example, in Fig. A4, for the API "create VM4 in VPC1", by sequentially querying five tables, its dependent configs of "VM4, PIP4, VPC1, ACL1, Route1" can be obtained. By querying the existing configs of the device, the necessary changes can be calculated as "Add VM4, PIP4, VPC1, ACL1, Route1 on Server2". Parsing the API with SQL needs a deep understanding of the service logic. To guarantee the equivalence, the information of dependent configs and existing configs can also be obtained through the implementation of Trident. Specifically, we can get the needed information by searching for the relation table, as shown in Fig. A4. This procedure is service-independent since the query table is same for all APIs (*i.e.*, Relation table).

B.2 Trident operations of cloud control API

Based on our analysis of the cloud control APIs provided by major cloud vendors (including ourselves), most of them

Table A1: API of configuring Public (Elastic) IP.

	Create	Delete	Update	Read
AWS	1	3	2	5
GCP	1	1	1	2
Azure	4	1	1	3
Ali Cloud	5	4	3	8

Algorithm 2: Abstraction of API: Create Resource1

```
Create Conf_R1 of Resource1
for C in Conf_R1 's dependant Conf do
    Relate Conf_R1 to C
for G in Conf_R1 's target Group that needs to be configured do
    Relate G to Conf_R1
```

are Create/Delete/Update/Read, as shown in Table A1. In addition, there are a few other APIs such as Associate and Disassociate. Algo. 2,3,4,5 detail how to abstract the main cloud control API into Trident operations.

B.3 Computational complexity analysis of OR-based calculation

The OR-based ref calculation has excessively high computational complexity. When $ref(Group, P_i)$ changes, recalculation is needed to refresh $ref(Group, Conf_A)$. The change of $ref(Group, P_i)$ can have two cases: *i.e.*, $ref(Group, P_i)$ turns from 0 to 1 or 1 to 0. For the 0 to 1 case, according to the OR calculation rule, as long as at least one $ref(Group, P_i)$ is 1, $ref(Group, Conf_A)$ is 1. Hence, we can directly infer that $ref(Group, Conf_A)$ is 1 without complex computation. By contrast, for the 1 to 0 case, we cannot assert that $ref(Group, Conf_A)$ becomes 0 unless we perform the OR calculation to each $ref(Group, P_i)$. In another case, if one parent node is unrelated due to tree changes, because the OR operation does not satisfy reversibility, once a bitwise OR operation is performed, it cannot simply be reversed to obtain the value before this parent node was included. In other words, it's not possible to determine the updated OR value based on the change in a parent node's ref and the current OR value. Because it is necessary to perform a bitwise OR on the ref values of all parent nodes to obtain the OR result, the complexity is proportional to the number of parents as $O(P_i)$.

B.4 Computational complexity analysis of SUM-based calculation

To address the performance issue of OR-based ref calculation, we can adopt SUM-based solution to inherit Conf's ref from its parents. The advantage of SUM is that during calculation, we can refresh Conf's current ref based on the change of its parent's ref and Conf's past ref. In SUM-based solution, if $ref = 0$, it indicates no descendant relation, while in other cases, it indicates the presence of a descendant relation. At the start, we initialize $ref(Group, Group.children)$ to 1 and continuously propagate this value to the remaining Confs according to the SUM rule, as defined by $ref(Group, Conf_A) = \Sigma_i ref(Group, P_i)$.

Algorithm 3: Abstraction of API: Delete Resource1

```
for C1 in Conf_R1 's children do
    Unrelate Conf_R1 to C2
for C2 in Conf_R1 's parents Conf do
    Unrelate C2 to Conf_R1
Delete Conf_R1
```

Algorithm 4: Abstraction of API: Update Resource1

```
Update Conf_R1 of Resource1
```

Since SUM adheres to the reversibility property, for scenarios where a parent node is removed, the new result can be obtained simply by subtracting that parent node's ref. Similarly, for scenarios where a new parent node is added, the result can be updated by simply adding the ref of that new parent node. Moreover, for scenarios where the ref of a parent node changes, the $ref(Group, Conf_A)$ can also be expressed as $ref(Group, P_x) + \Sigma_{i \neq x} ref(Group, P_i)$. When only $ref(Group, P_x)$ undergoes a change, we can deduce that $\Delta ref(Group, Conf_A) = \Delta ref(Group, P_x)$. It implies that when a $ref(Group, P_x)$ undergoes a change, $ref(Group, Conf_A)$ will experience a similar change. For instance, when a $ref(Group, P_x)$ transitions from 0 to 3, $ref(Group, Conf_A)$ will also increase by 3 accordingly. Since the refreshing mechanism focuses solely on the change of a specific parent, the computational complexity is only $O(1)$.

B.5 Case study of POSEIDON

To clearly illustrate the workflow of POSEIDON, we present a step-by-step introduction using the API call of "create VM4 in VPC1" as an example.

Step 1: After the user issues this API call, service-related controllers (*e.g.*, VPC controller in Fig. 5 in this example) will map the API into Trident operations. This step is service-related and generates different Trident operations based on the processing logic specific to each service API. The Trident operations corresponding to this API are shown as the blue color text in Fig. 6.

Step 2: Subsequently, upon receiving the Trident operations, the service-independent unified controller will add/delete/update Confs or modify their relations according to the operations. The changes incurred by the above operations are depicted in blue in Fig. 7.

Step 3: Then, based on the nodes/topology changes in the Trident tree, we calculate the appropriate Trident changes for Groups. §3.4.2 details the procedure for this example.

Algorithm 5: Abstraction of API: Associate Resource1 to Resource2

1 **Relate** $Conf_{R1}$ of Resource1 to $Conf_{R2}$ of Resource2

Step 4: Finally, the unified controller will push the Trident changes to the unified agent residing on the devices within Groups. Specifically, this example involves pushing "Add ACL1, Route1, VPC1, PIP4, VM4" to Server2 and "Update FlowTable1" to GW1 and GW2, according to Fig. 6 and Fig. 7. Upon receiving the Trident changes, a program on the agent translates them into device-specific primitives to enable the user's intent.

B.6 Redis failover detection and data recovery

Unlike the real-time data storage in the database, Redis caches the latest data in memory and performs batch storage. When failover occurs, the cached data in memory will be lost. Therefore, we need to detect the failover and conduct data recovery. We inject lots of probes and accumulate them at the frequency of batch storage. When probes' data meet expectations, this batch of data are stored in Redis. Otherwise, this batch of data must be lost due to failover. We can locate the batch of data affected by failover according to the timestamp of the probe. Then, the data obtained from the database will be re-pushed.

C Additional Experiences

C.1 How to choose pushing and pulling when configuring devices?

Most traditional controllers deliver configurations to the physical devices in a *pushing* mode, namely, the controller actively pushes config changes to devices so that user's intent can take effect in a timely manner. However, in large-scale public clouds, the pushing method will cause large pressure on systematical performance (*e.g.*, TPS, southbound bandwidth), as there are enormous underlying devices. What's worse, users may frequently modify their network, and each modification may consist of thousands of underlying config changes (*e.g.*, a newly added route entry may involve thousands of VM host servers). Another way to deliver configurations is the *pulling* method, where devices pull configs from the controller on-demand or periodically [14, 30]. This method eliminates the above performance bottlenecks at the cost of adding configuration delay. POSEIDON employs the pushing method for most configuration delivery, and utilizes the on-demand pulling method for FlowTable. This is because communications between VMs usually occur within a small subset, so most information in the FlowTable will not be queried by the device. Besides, in our production enviroment, the additional latency caused by FlowTable configuration delay is too short to be perceived by users.

C.2 How to deploy POSEIDON to a small-scale virtual network?

POSEIDON is an SDN architecture designed for high-performance control of large-scale virtual networks. Its Trident abstraction is implemented through a Trient Tree structure maintained in memory, cache, and database. However, for a small-scale virtual network with medium performance requirements, POSEIDON may be over-engineered and cost-prohibitive. For example, a standard deployment of POSEIDON requires at least 8 high-performance physical servers to support virtual network control in a data center of 10,000 servers. However, the number of servers may be less than 100 in some edge regions or small private cloud environments. (Even the number of servers in some cluster is less than 20 in some edge POP site.) In such a small-scale deployment, POSEIDON's architecture needs to be optimized to reduce resource usage, especially memory usage. One possible idea is to design a variant of Trident tree with high compression and efficient data exchange. The Tridient tree currently implemented by tree structure requires that all configuration relationships should be maintained in memory, which is a performance-first but cost-ignoring approach. Therefore, limited memory resources in small-scale scenarios can be solved by compressing Trident data, or even maintaining only part of the relationship in memory through data exchange.

OPPerTune: Post-Deployment Configuration Tuning of Services Made Easy

Gagan Somashekar* [1] Karan Tandon* [2] Anush Kini[2] Chieh-Chun Chang[4] Petr Husak[4]
Ranjita Bhagwan+ [5] Mayukh Das[3] Anshul Gandhi[1] Nagarajan Natarajan[2]

[1]*Stony Brook University* [2]*Microsoft Research* [3]*Microsoft365 Research* [4]*Microsoft* [5]*Google*

Abstract

Real-world application deployments have hundreds of inter-dependent configuration parameters, many of which significantly influence performance and efficiency. With today's complex and dynamic services, operators need to continuously monitor and set the right configuration values (*configuration tuning*) well after a service is widely deployed. This is challenging since experimenting with different configurations post-deployment may reduce application performance or cause disruptions. While state-of-the-art ML approaches do help to automate configuration tuning, they do not fully address the multiple challenges in end-to-end configuration tuning of deployed applications.

This paper presents OPPerTune, a service that enables configuration tuning of applications in deployment at Microsoft. OPPerTune reduces application interruptions while maximizing the performance of deployed applications as and when the workload or the underlying infrastructure changes. It automates three essential processes that facilitate post-deployment configuration tuning: (a) determining which configurations to tune, (b) automatically managing the scope at which to tune the configurations, and (c) using a novel reinforcement learning algorithm to simultaneously and quickly tune numerical and categorical configurations, thereby keeping the overhead of configuration tuning low. We deploy OPPerTune on two enterprise applications in Microsoft Azure's clusters. Our experiments show that OPPerTune reduces the end-to-end P95 latency of microservice applications by more than 50% over expert configuration choices made ahead of deployment. The code and datasets used are made available at https://aka.ms/OPPerTune.

1 Introduction

The performance and efficiency of large services and application deployments depend heavily on how they are configured. Configurations can be *system-level*, such as the read_ahead_kb parameter, which decides how much extra data to read from disk during I/O in Linux, and resources.limits.cpu that limits the amount of CPU a Kubernetes container uses. They can also be *application-level*, such as maxmemory, the memory usage limit at which Redis starts evicting keys. Any large application deployment invariably includes hundreds, if not thousands, of such configuration parameters at multiple layers and components [25, 35, 48, 50, 66, 75] .

Today, application operators determine the configuration values using domain-knowledge and canary testing on relatively small deployments before widely deploying the application. However, application behavior can change considerably with time and therefore the configuration values set before deployment may not work well in the longer term; e.g., developers continuously add features, the user population and their usage behavior varies [47], the underlying hardware hosting the applications also can also change [67, 75]. Consequently, to squeeze the most performance—say throughput or latency—or to make it run efficiently on as small a set of resources as possible without compromising performance, operators need to constantly monitor and modify these configuration parameters well after they have deployed the application.

Manually exploring and changing the configurations at regular time intervals can be tedious and risky, given that the number of parameters is large and, more often than not, the values of parameters can depend on each other and the deployment environment. Several recent efforts have proposed using machine learning (ML) based techniques [15, 25, 27, 40, 44, 46, 48, 61, 68, 69] to automate the process of configuration tuning. These efforts use online learning or reinforcement learning (RL) to set the configurations, observe the application state to determine how well it is doing, and then iteratively refine the configuration based on the observed states. This approach does reduce the burden on the operator, and yet, the problem is far from solved. The algorithm is only one necessary component of post-deployment configuration tuning. Through our experience deploying popular techniques (e.g., Bayesian Optimization (BO)) and state-of-the-art frameworks (e.g., SelfTune [40], Kubernetes Autoscaler [3]) for parameter tuning in application deployments at Microsoft, we have observed that there are significant gaps to be bridged in the *end-to-end process of configuration tuning*. Some of these gaps, discussed below, may have been overlooked by prior research, as they are more pronounced in actual production deployments.

*These authors contributed equally.
+Work done while employed at Microsoft Research.

First, since services can easily have hundreds of configuration parameters, it would be prohibitively expensive to automatically tune all of them simultaneously. Thus, an automated approach should determine which components or layers of the service to tune, and for each component or layer, which configuration parameters it should tune.

Second, when a service is running, all configuration parameters are not equally easy to tune. For instance, changing worker_process of Nginx (this sets the number of Nginx worker processes) requires only a service reload after changing a configuration file [2] with no downtime, but changing wiredTigerCollectionBlockCompressor in MongoDB requires a pod restart which leads to a downtime of close to 8 seconds when deployed with the recreate strategy on Kubernetes [6]. Changing isolcpus, the parameter that isolates CPUs from the kernel scheduler in Linux, will require an entire system reboot [63]. Previously proposed algorithms do not consider this varying difficulty of tuning different types of parameters. Importantly, to reduce potential disruptions for deployed services, the tuning approach should use a very small number of iterations to converge on the right values.

Third, the tuning system needs to determine the right granularity for each tuning instance. It could tune a single set of configuration values for the entire service, or it could tune different values for each geography, or perhaps for every machine type. We refer to this as determining the right tuning *scope*. Currently, the operator has to scope the tuning instances manually irrespective of the tuning algorithm used.

Finally, standard algorithms for tuning work only on numerical [30, 40] or only on categorical [19, 20, 38, 65] parameters; real-world services will almost always have a combination of categorical and numerical parameters. For instance, Redis's maxmemory and maxmemory-policy (sets the eviction policy) are related parameters that are of type numeric and categorical, respectively. The key difference between handling numerical and categorical parameters is the notion of continuity. Categorical parameters lack continuity so the performance can change drastically every time the value of the parameter is changed. There is no easy way to encode/decode categoricals as numerals as the usual approaches like 1-hot encoding [34] would increase the search space substantially. Popular techniques like BO [16, 17] do handle such hybrid parameter spaces, but they are not meant for scenarios where the environment of the service changes continually. Deep neural models for configuration tuning [45, 57], on the other hand, are usually trained offline rather than in deployment.

We have designed, developed, and deployed OPPerTune (**O**nline **P**ost-deployment **Per**formance **Tune**r), a configuration tuning service that addresses all the above challenges. Given a superset of configuration parameters[1] that can be potentially tuned, OPPerTune can automatically create, manage, and scope tuning instances for application operators of large-scale services. The key contributions we make are:
1. OPPerTune introduces a novel tuning algorithm, as part of its backend, that can tune categorical and numerical parameters simultaneously within the same instance.
2. OPPerTune uses a novel decision-tree based algorithm to automatically determine the right scope of tuning instances, taking into account the varying application context in production environments, such as workload types and volumes.
3. We have built and deployed OPPerTune as a cloud service at Microsoft. First-party applications that are already in production can be on-boarded to our service with minimal engineering overhead.

We have evaluated OPPerTune on two applications: (i) the social networking application from the DeathStarBench benchmark suite driven by workload traces collected from a large-scale service that is part of Microsoft Teams application, and (ii) a large-scale ML experimentation pipeline that uses Azure ML for model development and Apache Spark for data processing. Our evaluation and comparison with closely related works [17, 40, 45, 58, 65] show that by using OPPerTune to tune configuration parameters: (i) the tail latency of the application consisting of tens of microservices reduces by more than 50% while tuning in deployment, even under significant workload variations, compared to carefully-chosen pre-deployment configurations; (ii) service disruptions that may occur due to configuration tuning are reduced by nearly 30%; and (iii) workload completion times drop by 10%–50% on two Azure ML clusters, over two weeks of deployment;

2 OPPerTune **Overview**

The goal of OPPerTune is to continuously tune the configuration parameters of an application such that, over time, a given *reward* metric (e.g., daily P95 latency) is maximized, and the application sustains good performance through long-term and short-term hardware changes and workload fluctuations. Throughout, we use the term 'application' to refer to the system/service/application being tuned to avoid confusion with the OPPerTune service itself. OPPerTune works under the least knowledge of the application being tuned, i.e., black-box access. In particular, it does not have access to the code-base of the application, does not require any form of instrumentation or any knowledge of how its performance metrics are computed. OPPerTune relies only on the reward as feedback from the application (after a certain amount of time) for a set of configuration values it sets for the application. It uses this feedback to tune the configurations iteratively. OPPerTune's back-end supports algorithms that can function with or without context (e.g., workload characteristic, resource utilization, etc.). If context is available, the operators can use all of the algorithms in the back-end, but this is not necessary.

Consider the following example. A web application uses two containers on a single machine: one to host a front-end webserver, and the other to run a back-end database. While serving user requests, the application can enlist OPPerTune

[1] This work focuses on application- and infrastructure-layer parameters.

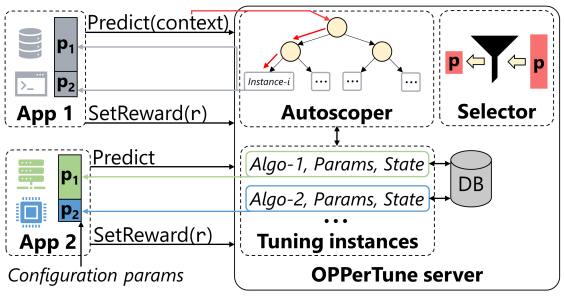


Figure 1: OPPerTune service architecture. Applications create *tuning instances* on the server to tune various configuration parameters. *Autoscoper* helps automatically create, manage, and scope tuning instances based on the application's dynamic context. *Selector* helps pick the most promising configuration parameters to tune.

to learn how to distribute the machine's memory and compute between the webserver and the database so as to minimize P95 request latency. OPPerTune consumes feedback (or reward) from the application in the form of observed hourly P95 latency, and uses multiple hours' feedback to converge on the right memory and compute distribution between the two containers. Request characteristics can vary with time; thus, OPPerTune may need to change the distribution of memory and compute frequently, *and* converge quickly to stable configuration values while continuing to minimize P95 latency.

OPPerTune **architecture:** Figure 1 presents the high-level architecture of the OPPerTune service. The basic unit of the service is a *tuning instance* that consists of (a) configuration parameters, their data types, enumerations of possible values/ranges for categorical/numerical configurations; and (b) a tuning algorithm for updating the instance. Applications can create one or more tuning instances to tune configuration parameters across various layers of the application stack, based on its requirements (as shown for 'App 2' in the figure). Alternatively, OPPerTune provides an automatic scoping component (*autoscoper*) to help applications create, manage, and scope the tuning instances in deployment based on dynamic context information they provide (as shown for 'App 1'). OPPerTune can also aid applications to pre-determine (using an offline step) which configuration parameters to tune in deployment via the *selector* module.

Creating, fetching, and updating tuning instances: An application intending to use OPPerTune makes an API call to create a *tuning instance* along with a list of parameters and their meta-data (e.g., range of parameters). This step can be automated, as in prior works [62, 72], or can involve the developer to convey a super set of parameters along with their meta-data. The operators can use their domain-knowledge to select a subset of parameters to pass to OPPerTune, but this is optional. The parameter list can be arbitrarily large, and OPPerTune's *selector* module will pick the performance-critical parameters as discussed later in this section. OPPerTune persists tuning instances on a database for the application to access at any point in time, possibly from multiple machines. For each tunable configuration parameter, the operator can optionally supply the cost (e.g., is a system restart required) associated with changing it. When such cost information is available, OPPerTune uses it to decide how often to tune the configuration parameters. OPPerTune implements the standard fetch and update client-API paradigm of existing work [14, 40] for online tuning. The application invokes (a) Predict to fetch the recommended configuration values from a tuning instance, and (b) SetReward, at some point in time after (one or more) Predict calls, with a reward value. The SetReward call updates the associated tuning instance, as per the tuning algorithm it uses. Any delay in sending back the rewards will only delay the model update/convergence but not affect the application performance.

OPPerTune supports parallel exploration when enough servers/resources are available to deploy multiple instances of the same application. If the servers in the cluster are identical, a single OPPerTune instance can be employed to which all servers issue Predict and SetReward calls. OPPerTune would use all deployments to explore and converge towards a single optimal configuration for the application. For heterogeneous servers, AutoScope (Section 4) can enlist contextual attributes (e.g. region, hardware type) to cluster the servers, and tune parameters for different deployments.

Autoscoper: Applications may have different performance characteristics on machines with different CPUs or memory sizes, and hence may consider using a different configuration tuning instance for each machine type. Similarly, applications could behave differently for light versus heavy workloads, and for different API call types. For instance, if the application runs independently on the cluster machines with varying hardware and workloads, then it could create one tuning instance per machine (as is done in [40] for tuning configuration parameters of a workload scheduler). Thus, tuning instances for the application could be "scoped" along (at least) three dimensions: *infrastructure* (e.g., machine type), *functionality* (e.g., API call), and *workload* (e.g., requests per second). Currently, determining the "right" scope for tuning instances is usually done, if at all, by domain experts [40], and periodically revisited. As an alternative, in Section 4, we present AutoScope, an automatic, efficient, and interpretable way of scoping tuning instances. A Predict call to an AutoScope instance returns configuration values corresponding to the context presented.

Configuration selector: For applications that have several hundreds or thousands of configuration parameters to tune across various layers of the application stack, OPPerTune employs a selector module to pick the most *promising* configuration parameters to tune. The selector module uses a simple and effective *microbenchmarking* strategy to identify such configurations, as discussed in Section 5. The module provides a list of parameters, ordered in decreasing order of their importance, along with a score that quantifies the importance.

The operator can then choose the top-n parameters from this list, based on their importance score.

Rounds and Sample complexity: Online tuning algorithms, implemented in OPPerTune's back-end, iteratively tune the configuration parameters. Each iteration is called a *round*. At each round, the tuning algorithm (i) determines the next set of parameter values for the application, (ii) observes a reward computed by the application over a predetermined period (1 hour, 24 hours, etc.), and (iii) updates its "policy" (which prescribes how to choose parameters) based on the reward. Changes made by the algorithm to configuration values can cause disruptions, e.g., may necessitate application restarts or even cause downtime. Thus, a desired property of a tuning service is that it require only a few rounds to learn suitable configuration values. This quantity, proportional to the number of rewards measured, is called *sample complexity*. OPPerTune achieves low sample complexity for tuning in real deployments, as we demonstrate in Section 7, using multiple techniques including a novel tuning algorithm (Section 3), automatic scoping, and microbenchmarking.

3 Configuration Tuning in Hybrid Spaces

In this section, we present a novel algorithm for the post-deployment configuration tuning problem (Section 2) on a hybrid space wherein the configuration parameters are a mix of numerical and categorical ones. We consider the basic setting when OPPerTune has no additional knowledge ("context") about the system being tuned; the setting where some context may be available is considered in the next section.

3.1 Problem Definition and Terminology

We pose the problem of configuration tuning for an application post deployment as that of online learning with bandit feedback. That is, we want to tune iteratively, only interfacing with the application for setting new parameter values (e.g., # CPU cores or memory size for containers), and for obtaining feedback in response to the set parameter values, as an observed *reward* value that is to be optimized (e.g., latency or throughput of the application).

A key aspect of bandit formulation is the *explore-exploit tradeoff*. We want to *exploit* the "best" parameters as per policy learned so far to ensure that the application is functioning well without disruption; at the same time, we want to *explore* potential parameter choices that might yield better rewards. This tradeoff is especially important in practical scenarios where the reward function itself changes with time—the same parameter choices could have different effects on the service at different time points. For instance, diurnal workload fluctuations can induce very different reward values for the same setting of memory requirements for a container, depending on how and when the reward is computed, e.g., hourly P95 latency can vary significantly between peak and off-peak hours.

Bandit learning techniques that can handle time-varying rewards, therefore, are more appropriate to our problem than popular alternatives such Bayesian Optimization (BO) [16, 17], heuristic search and global optimization [52], and genetic algorithms [32]. For instance, BO needs to evaluate the *same* reward function at multiple parameter values by design. This is infeasible for post-deployment tuning because we cannot evaluate a deployed application multiple times. Most systems research that leverage BO typically use it in offline scenarios (i.e., pre-deployment tuning in controlled settings) [17, 22], in contrast to our post-deployment tuning scenario.

3.2 Hybrid Configuration Space

In practical scenarios, the space of configuration parameters can be complex: (i) it can be very large; if there are n parameters to tune, with, say, s possible values each, we have s^n choices, and (ii) they may be discrete (e.g., number of CPU cores), real-valued (e.g., CPU utilization threshold), or categorical (e.g., cache eviction policy). Some state-of-the-art techniques for bandit learning/RL work for categorical spaces [19, 20, 38, 65] or numerical spaces [30, 40], but not both. Others have high sample complexity for tuning in deployment [42, 57]. To address this gap, we design a novel learning algorithm to handle hybrid configurations efficiently.

Formally, the "hybrid configuration space" comprises: (a) categorical space $\mathcal{C}$ over k categorical parameters, and (b) numerical space $\mathcal{W}$ over m numerical parameters which is a bounded subspace of $\mathbb{R}^m$. In our formulation, we treat discrete parameters as numerical rather than categorical to exploit the fact that they are ordered spaces.

3.3 Proposed Algorithm: HybridBandits

Our configuration tuning HybridBandits algorithm is presented in Algorithm 1. It leverages two simple but key ideas. At each round, **(1)** it maintains different *types* of policies for sampling categorical and numerical actions; in particular: (i) ε-greedy policy for the categorical configuration space, standard in multi-arm bandit algorithms, where with probability ε a random arm is explored, and with probability $1-\varepsilon$, high-reward arms are exploited; and (ii) a "perturbation" policy for numerical configurations, where the algorithm samples numerical configurations from an ε-radius ball centered around the "current best" configuration vector, and **(2)** it uses a single reward that the system provides as feedback to update both the policies simultaneously. In particular, it applies sample-efficient gradient-descent update [30, 40] for the numerical parameters, and the exponential weights update [41, Chap. 11] for the categorical parameters.

Algorithm description: Algorithm 1 maintains a multinomial distribution $\mathbf{p}^{(t)}$ over categorical actions $\mathcal{C}$, i.e., there is a probability associated with each possible k-tuple of categorical parameter choices at every round t. For the numerical actions, it maintains a vector $\mathbf{w}^{(t)} \in \mathbb{R}^m$.

Initialization: The weights for the numerical parameters $\mathbf{w}$ are initialized to default choices that the application provides. The multinomial $\mathbf{p}$ is initialized to the uniform distribution,

Algorithm 1 HybridBandits: Post-Deployment Configuration Tuning for Hybrid Spaces

1: **Input:** exploration parameter $\varepsilon \in (0,1)$, learning rate $\eta > 0$, space $\mathcal{C}$ of k categorical parameters, space $\mathcal{W}$ of m numerical parameters.
2: **Initialize**: categorical space weights $p_i^{(0)} = 1/|\mathcal{C}|$, for $1 \le i \le |\mathcal{C}|$ // uniform distribution, and numerical parameters $\mathbf{w}^{(0)} \in \mathcal{W}$ // default
3: **for** $t = 0, 1, 2, \ldots$ **do**
4: Let $\widetilde{p}_i := (1-\varepsilon)p_i^{(t)} + \varepsilon\frac{1}{|\mathcal{C}|}$ // Define explore-exploit multinomial distribution over the categorical space
(1) Sample categorical and numerical actions to deploy
5: Sample $c \sim \widetilde{\mathbf{p}}$ from the multinomial and let $\mathbf{a}_c^{(t)}$ be the corresponding k-tuple of categorical parameters
6: Sample numerical parameters from a ball centered at $\mathbf{w}^{(t)}$, radius ε; i.e., $\widetilde{\mathbf{w}}^{(t)} := \mathbf{w}^{(t)} + \varepsilon\mathbf{u}$, where $\mathbf{u} \in \mathbb{R}^m$ is sampled from $\{\mathbf{u} : \|\mathbf{u}\|_2 = 1\}$ // Identical to Bluefin [40]
(2) Deploy the actions and measure reward
7: Deploy numerical $\mathbf{a}_r^{(t)} := \Pi_{\mathcal{W}}(\widetilde{\mathbf{w}}^{(t)})$ // appropriately scaled and categorical actions $\mathbf{a}_c^{(t)}$ in the application
8: Receive reward $r^{(t)} := r_t\left(\mathbf{a}_c^{(t)}, \mathbf{a}_r^{(t)}\right) \in \mathbb{R}$ // black-box access to a metric, e.g., hourly P95 latency, computed by the application
(3) Perform updates based on the reward received
9: Update numerical parameters center: $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} + \frac{1}{\varepsilon} \cdot \eta \cdot r^{(t)} \cdot \mathbf{u}$, where $\mathbf{u}$ is the sample obtained in Step 6.
10: Define scaled reward: $\widetilde{r}^{(t)} = r^{(t)}/\widetilde{p}_c$, where c is the sample obtained in Step 5
11: Update categorical distribution: $p_c^{(t+1)} \leftarrow p_c^{(t)} e^{\eta\widetilde{r}^{(t)}}$, and for $i \neq c$, $p_i^{(t+1)} \leftarrow p_i^{(t)}$; Renormalize $\mathbf{p}^{(t+1)}$ to sum to 1

i.e., $p_i = \frac{1}{|\mathcal{C}|}$ for $i \in \mathcal{C}$. At each round, the algorithm performs:
Sampling actions (Steps 5-6): For the categorical actions, following the standard exponential weights algorithm (EXP3, [41, Chap. 11]), it samples a k-tuple from the distribution $\mathbf{p}$ (*exploit*) with probability $1-\varepsilon$, and from the uniform distribution (*explore*) with probability ε. For the numerical actions, it samples a m-dimensional vector from a ball centered at the current $\mathbf{w}$, with radius ε.
Deploy actions and receive reward (Steps 7-8): The sampled numerical (scaled appropriately) and categorical configurations are then deployed in the application, and (after a certain amount of time) the algorithm receives a reward value from the application (implementation details in Section 7).
Update policies (Steps 9-11): For the numerical parameter weights, the algorithm follows the gradient estimation scheme studied in the optimization literature [30], as well as applied in the context of online system parameter tuning [40]. For the categorical parameters, it: (a) computes an unbiased estimate of the reward for the sampled choices, and (b) scales the probability of the sampled choices using a factor that is exponential in the reward estimate.

The algorithm has hyperparameters ε and η, and we set these to default values in all of our experiments (following SelfTune [40]). In practice, $\mathcal{C}$ can be very large; the microbenchmarking strategy (Section 5) can be used to restrict $\mathcal{C}$ to the most impactful categorical parameters, and to ensure that the algorithm has low sample complexity. We conjecture that Algorithm HybridBandits has convergence guarantees for certain classes of reward functions (for instance, if the reward functions r_t are convex, for any fixed combination of the categorical parameters in $\mathcal{C}$). Empirically (in Section 7, and in our synthetic problem setup in Appendix B), we find that the algorithm convergences well in practice; obtaining a formal proof of convergence is an exciting open problem. We also note that when the configuration space contains only numerical parameters, HybridBandits is the same as SelfTune's Bluefin algorithm.

4 Automatic Scoping of Tuning Instances

Consider an operator who wants to tune the parameters of a distributed application that is I/O-bound. There are two extreme options available to the operator in terms of how they can set up tuning instances on the OPPerTune server (Figure 1): (1) set up one "global" instance to tune all the application parameters across all machines/workloads, that, say, uses HybridBandits presented in Section 3 for tuning; or (2) set up multiple "local" instances based on the domain expertise that the workloads are I/O-bound; e.g., one instance per disk type or one instance per spindle speed, where each instance independently tunes parameters using HybridBandits. The latter option is more appealing as the application performance, and therefore the optimal parameter choices will likely vary with the disk type the workloads are accessing.

In this section, we consider the setting when OPPerTune is provided some context of the application (i.e., disk type and spindle speed in the above example) being tuned at every round. OPPerTune can exploit the observed context to *simultaneously* do scoping and configuration tuning.

4.1 Joint Scoping and Configuration Tuning

To perform joint scoping and configuration tuning, at each round, along with the reward, the application must provide additional context information such as machine type, disk type, spindle speed, workload volume, etc. Using this additional context, OPPerTune determines a lightweight and interpretable scoping policy that the operators can understand. For instance, given job type `jobtype` and requests per second (`rps`) as context, and `numcores` and `mem` as the configuration

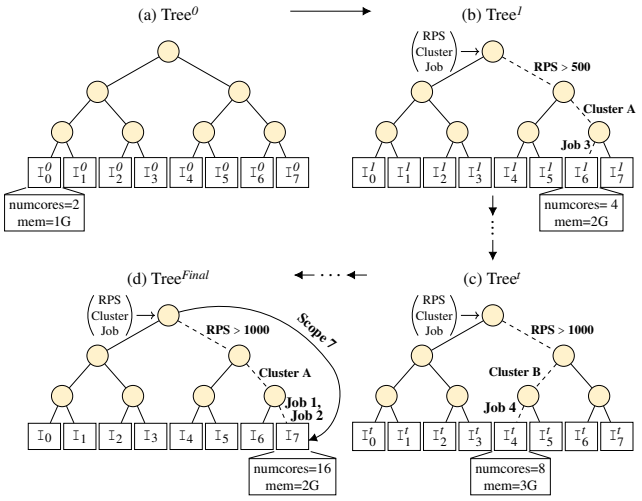


Figure 2: AutoScope: Iteratively learning to scope tuning instances via decision trees. At each round, the observed context (rps, cluster, job) is used to update the tree model and the leaf instance (numcores, mem configurations) it lands in.

values to tune, it learns rules of the form `if (jobtype == 'cpu_bound') and (rps > 1000) then` numcores=16, mem=2G `else` numcores=4, mem=2G.

These kinds of scoping rules can be captured by decision tree models illustrated in Figure 2(d). Each root-to-leaf path in the tree constitutes a scope, and each leaf maintains a tuning instance for the scope. In the Figure, 'Scope 7' is interpreted as application *running in Cluster A, when its workload volume (RPS) > 1000, involving jobs of type 1 or 2*. Its leaf node maintains a tuning instance, i.e., values for the two parameters numcores and mem. These values will be returned for Predict requests satisfying this scope, and will be updated when a reward arrives for these requests from the application.

4.2 Proposed Algorithm: AutoScope

Learning decision trees in the bandit setting is a challenging problem, and popular tree learning algorithms do not apply (see Section 8). We extend a state-of-the-art technique proposed in [39] (for trees with only one parameter in leaf nodes) to our general setting where each leaf node is a tuning instance with several (hybrid) parameters.

We start by giving the key intuitions, before giving a detailed technical description of the tree learning algorithm.

Algorithm outline: AutoScope maintains a binary decision tree f_T of max specified height h ($h = 3$ suffices for scenarios evaluated in Section 7), as illustrated in Figure 2. At first, the tree $f_T^{(0)}$ effectively behaves like a single tuning instance, initialized identical to Algorithm 1, i.e., all the leaf instances I_k^0 in Figure 2 (a), for $0 \le k \le 7$, are initialized identically. At round t, the algorithm observes a context vector, denoted as $\mathbf{c}_t$. When the current tree model $f_T^{(t)}(\mathbf{c}_t)$ is applied to $\mathbf{c}_t$, it will land in a unique leaf node containing a tuning instance. That is, the context vector is first applied to a linear model in the root node (whose weights are initialized to default value in the beginning). Depending on the sign of the resulting value, $\mathbf{c}_t$ traverses left or right, and continues in this fashion making branching decisions at every intermediate node until it reaches a leaf. The root-to-leaf path $\mathbf{c}_t$ traverses is its "current scope", and AutoScope will invoke the leaf's tuning instance. Figure 2 (b) shows the attributes $(RPS > 500, cluster = A, Job = 3)$ getting resolved to the scope 6 (I_6^1) in the first round.

This amounts to doing one round of Algorithm 1 on the leaf's tuning instance, thereupon updating it. Now the technical challenge is updating the tree model $f_T^{(t)}$ parameters (i.e., the internal node weights for making branching or scoping decisions), besides the configuration parameter values in each of the leaf nodes. We describe the algorithm in detail next.

Algorithm description: We present the procedure for jointly learning the scoping and the tuning instances, i.e., parameters corresponding to each scope, formally in Algorithm 2 in Appendix A. The algorithm maintains a decision tree model denoted f_T; each internal node j in the tree has a linear model $\mathbf{z}_j \in \mathbb{R}^d$ (where d is the number of context variables) that makes routing decisions of the form $\langle \mathbf{c}^{(t)}, \mathbf{z}_j \rangle > 0$. Each root-to-leaf path in the tree corresponds to a scope. Each scope ends at a leaf node ℓ which holds a tuning instance, i.e., $\mathbf{p}_\ell \in [0,1]^{|\mathcal{C}|}$ (for categorical actions) and $\mathbf{w}_\ell \in \mathbb{R}^m$ (for numerical actions). We work with binary trees of height h with 2^h scopes (and tuning instances).

Learning a tree policy entails learning the internal node parameters $\mathbf{z}_j$, for $0 \le j \le 2^h - 2$ and the leaf instances $\mathbf{p}_\ell, \mathbf{w}_\ell$, for $0 \le \ell \le 2^h - 1$. Widely-used tree learning algorithms like CART [23], C4.5 [56], and their variants do not apply to the bandit feedback setting because they need access to labeled training data (which in our scenario means optimal parameters for different context vectors, which is difficult to obtain in practice). There has been recent work on learning trees using bandit feedback [28, 29], but they work only for categorical spaces. The closest to our setting is the technique proposed in [39], but it handles only a single parameter (either numerical or categorical) in the leaf nodes. We extend their technique to handle multi-dimensional, hybrid tuning instances in the leaf nodes.

Algorithm 2 follows the structure of Algorithm 1 closely. Each leaf node is initialized (Step 2) identical to Algorithm 1, Step 2. The first key difference is that, at each round t, the observed context $\mathbf{c}_t$ determines which tuning instance, i.e., leaf node is selected (Steps 4–5). Once the leaf node is selected, in Step 6, we use the corresponding tuning instance to obtain the configuration and deploy the system with this configuration (Step 7) to observe the reward just as in Algorithm 1. The second key difference is how the tree is updated (Steps 8–9), as discussed below.

The main challenge in learning decision trees in general, not just in the bandit setting, is that the tree function $f_T : \mathbf{c}_t \mapsto$ leaf ℓ is highly discontinuous and non-differentiable. If

we can approximate f_T with a differentiable function, then we will be able to jointly learn f_T model as well as the leaf node parameters using online gradient descent techniques in a sample-efficient manner. We leverage the relaxation in [39] that, in particular, replaces discontinuous branching decisions ("go left or right") at the internal nodes with differentiable sigmoid functions ("go left with probability 0.9 and right with probability 0.1").

Using chain rule of differentiation, we can show that the gradient of the reward function r_t with respect to the tree model parameters (both internal node and leaf) can be written as a tensor product of the gradient of r_T with respect to the leaf node parameters and the gradient of f_T with respect to the tree model parameters.

Scalability: AutoScope scales well with arbitrary state spaces. Its model size scales linearly in the size of context vector **c** and the tree's height is typically very small. Though there are far too many ways of jointly slicing the context dimensions, AutoScope tries to automatically find scopes that are most beneficial in improving the reward metric without other hints.

5 Configuration Selection

For applications that have several hundreds or thousands of configuration parameters to tune across various layers of the application stack, OPPerTune employs a configuration selector module (Figure 1) that leverages a simple and effective *microbenchmarking* strategy to identify the most *promising* configuration parameters. while the techniques outlined here are heuristic, they are inspired by optimization theory [51,53].

The role of the selector module is two-fold: (a) it prunes the size of the configuration space, which in turn helps reduce the algorithm's sample complexity; and (b) it helps minimize the number of disruptions (e.g., container restarts) in the application while tuning. If (b) is the only goal, selecting just the configuration parameters that do *not* require restarts, may suffice. However, that might compromise on application performance by ignoring configurations that could significantly impact the performance (as illustrated in Section 7.3).

OPPerTune uses a *microbenchmarking* strategy to assess the effect of changing *each* configuration parameter on the application's performance (i.e., the reward value), while keeping the others fixed. Let us consider numerical configurations for the moment. The strategy is inspired by how coordinate descent algorithms [51], which are rigorously studied in the optimization community, work. These algorithms pick one coordinate (i.e., configuration parameter) at a time and compute the gradient of the reward function with respect to only that parameter. They iteratively pick coordinates (cyclically or randomly) to optimize the reward function.

We do not know of any variants of these algorithms that provably work in our online bandit setting. But, we find that the basic idea is empirically effective for the goal of selecting candidate parameters to tune. We use the same gradient estimation technique employed in Step 9 of Algorithm 1 for each configuration parameter while holding all other parameters fixed (to the default choices, for example). OPPerTune accomplishes this by simply creating microbenchmarking instances, each with just one configuration parameter, and performing one round of the HybridBandits algorithm. The *magnitude* of the (scalar) gradients computed at the instances tells us the impact of each configuration parameter. In practice, this idea can also be extended to categorical spaces—perform one round of the algorithm on each categorical parameter, and compute the magnitude of change in reward for a randomly chosen value vs. the default value for the parameter.

The configuration selector module then picks top-n parameters, sorted by decreasing magnitudes of these "gradients" where n is customizable by the application. This greedy selection strategy, i.e., picking the coordinate (or parameter) yielding maximum absolute gradient, has been shown to be provably better than other heuristics for selecting coordinates, for some classes of reward functions [53].

Microbenchmarking can be done in canary/test deployments of the application. The tuning instances for the application can work with the selected configuration parameters in deployment. OPPerTune provides a flag to periodically revisit microbenchmarking and re-assess the top-n parameter selections. Operators can turn this flag on when there are long-term changes (e.g., hardware, new workloads).

6 OPPerTune Implementation

OPPerTune's implementation has three major components: the server, client, and the algorithm backend. We have implemented the server in Go using Fiber [10], and the client in Python (for ease of integration with applications which are often written in Python). We have implemented our proposed algorithms in Python, and have integrated the server with existing Python implementations of other algorithms. We now describe each component in some detail.

1. OPPerTune Server: The server implements three key interfaces for the applications (clients) submitting requests via REST API calls: a) creation of tuning instances, b) fetching the values from the instances, and c) updating the instances using the the reward values sent back to the server. The server persists configuration tuning instances (consisting of the list of parameters to tune and their constraints, and the model for tuning) in a database. Persisting instances enables resuming from the saved model state at a later point of time, and freeing the memory taken up by instances that are not needed. For each fetch call from the client, the server responds with the configuration values along with a `requestId`. The client is expected to pass the reward value along with the associated `requestId`, for the server to be able to correctly issue an update to the corresponding tuning instance. System changes that are bursty and short-lived can potentially impact the observed reward. OPPerTune mitigates this by eliciting an optional reward measurement period from developers that

sustains such spikes (e.g., aggregation of the metric over a time period). Furthermore, our bandit algorithms (Section 3 and 4) accommodate such *adversarial* changes. We host the server on Microsoft Azure that provides persistent storage, high availability, and wide accessibility.

2. OPPerTune **Client:** The client is a library which implements easy-to-use REST API calls; these calls provide abstraction over raw HTTP requests, and applications use them to create, fetch (i.e., Predict), or update (i.e., SetReward) instances at the OPPerTune server. The library also manages mapping client requests to API endpoints, payload preparation, and error-checking. We provide an installable package of OPPerTune client for applications to use.

3. Service Backend/Tuning Algorithms: The backend consists of implementations of various tuning algorithms, and *autoscoper* and *configuration selector* components. Any tuning algorithm is expected to implement Predict and SetReward interfaces. We have implemented (i) the proposed algorithms HybridBandits and AutoScope; (ii) state-of-the-art online parameter tuning algorithm Bluefin [30, 40] and deep reinforcement learning (RL) algorithm DDPG [45, 55, 57]; and (iii) with minimal effort, we have integrated the contextual bandits-based algorithm Slates [65], and BayesianOptimization from the popular Python libraries VowpalWabbit [12] and scikit-optimize [11], respectively. The challenge in using deep RL techniques such as DDPG in deployment, typically, is their prohibitive model sizes and sample complexity. They use context differently than AutoScope to learn policies that require large complex models (the notion of scoping is implicit in DDPG). Hence, we have implemented a custom version of DDPG with light-weight models, similar to AutoScope, to be of use in post-deployment tuning.

7 Evaluation

To evaluate OPPerTune, we use a combination of microbenchmarking and real deployments of two applications. Our evaluation focuses on the following aspects: 1) *How does application performance improve using* OPPerTune?, 2) *How does* OPPerTune *reduce the cost of tuning (e.g., system restarts)*? 3) *How effectively does automatic scoping accelerate the tuning process in real deployments by reducing sample complexity*?, and 4) *How scalable is our service implementation (*OPPerTune *server and backend algorithms)?*

7.1 Evaluated Applications

To evaluate OPPerTune, we use two applications, each serving certain classes of workloads.

7.1.1 Social Networking Application

We use the SocialNetwork application from the DeathStarBench [31] benchmarking suite which mimics a stack consisting of a gateway server (Nginx), database engine (MongoDB), caches (Redis), and application logic. The application creates a network of users, and supports API calls to create and read messages from the users' home pages. We use wrk2 [9] to emulate two workloads: (a) **constRPS**: a mix of 90% GET (read timeline) and 10% POST (create posts) requests (this mix has been used in previous work [40]), with the requests generated at a constant rate (requests-per-second), and (b) **AMStraces**: real access traces collected from AMS, a large-scale asynchronous media-sharing service that is part of the Microsoft Teams application, on 3 production clusters over 4 weeks from Sep 14, 2022 to Oct 10, 2022 (see Appendix C.1 for details on the traces and experiments).

Microservice type	Number of parameters					
	Categorical		Continuous		Discrete	
	MS	RS	MS	RS	MS	RS
memcached	0	0	4	8	16	8
MongoDB	12	0	6	12	12	12
Nginx	0	0	0	4	8	4
RabbitMQ	0	0	0	2	0	2
Redis	4	0	0	8	16	8
App logic	0	0	0	24	23	24

MS=Microservices, RS=Rightsizing (Kubernetes)

Table 1: SocialNetwork application configuration parameters (217 in total) used for Figures 4, 5, and Tables 3, 6.

For this application, the performance metric of interest is **P95 latency of requests** submitted. This metric is critical to consumer-facing services [26, 31]. For the constRPS workload, we measure the P95 latency for each 10-minute period, and for AMStraces, we measure it for each hour. This is fed as the reward value to the OPPerTune service.

Table 1 outlines the list of microservices that SocialNetwork uses. Here, "rightsizing" layer refers to configuration parameters in Kubernetes that are used to determine the compute and memory limits for containers running the microservices. For each of the microservices, we tune *a mix of real-valued, discrete, and categorical parameters* picked from prior works and product documents [1, 4, 5, 7, 8, 13, 40, 61]. We note that operators can use such prior knowledge to reduce the number of parameters. However, the design of OPPerTune does not enforce this approach and the operators can pass arbitrarily large number of parameters. OPPerTune can automatically select a subset of performance-critical parameters if needed, as discussed in Section 7.3. The optimal values of these parameters depend on the workload characteristics; e.g., to support the same P95 latency, the MongoDB microservice will require higher resource limits for a higher request volume, and a larger number of clients would require higher concurrency setting for Nginx, etc. Additionally, the cost of tuning these parameters is passed to OPPerTune so it can decide how often to tune (as noted in Section 2). For example, some of these parameters require a container restart. OPPerTune piggybacks on scheduled maintenance (once daily for AMS) to tune such parameters. We defer the evaluation of other types of tuning costs (e.g., performance degradation without impacting availability and revenue cost associated with changing resource parameters) to future

research. We use the default values of all the parameters as the starting configuration for all the algorithms.

We deploy the SocialNetwork application on a cluster with 7 virtual machines (VMs) provisioned on Azure. Each VM hosts a copy of the application stack, with the component microservices running on individual containers on the same VM, managed by Kubernetes. Thus, by tuning the parameters in Table 1 appropriately, OPPerTune allocates each VM's resources in the right proportions to the various microservices so as to minimize P95 latency.

We use separate, dedicated VMs (7 more) in the same cluster as Kubernetes master nodes, to generate the workloads. Each VM has Intel Xeon Platinum 8272CL processor (32 vCPUs), 64 GiB RAM and 250 GiB storage (large enough to support the entire application stack).

7.1.2 ML-Experimentation Pipeline

We deploy OPPerTune in one of Microsoft's ML experimentation pipelines called MLExp (real name withheld) that uses Apache Spark to prepare data before training various ML models. This is an actual production deployment consisting of *workloads* and *jobs*. Each workload is an experiment, consisting of a collection of jobs arranged in a directed acyclic graph (DAG). A job loads, selects, filters, or processes data in different ways. MLExp supports 11 job types.

Table 2 shows the configuration parameters that MLExp developers manually set for each job before submitting the workload to the cluster. We use OPPerTune to tune the same set of parameters for each job, initialized to modest initialization values elicited from the developers for our and the baseline algorithms.

Developers wish to minimize **average job completion time** over all jobs that comprise a workload. Lower job completion times imply lower workload completion time, which further implies faster iterations of building ML models. When a workload completes, the platform provides the individual job completion times, which OPPerTune uses as the reward for updating tuning instances.

One could argue that setting the parameters in Table 2 to maximum values (such as setting spark.driver.memory to 25GB, spark.driver.cores to 4, etc.) would minimize average job completion time. However this does not work in practice because MLExp's scheduler will enqueue this workload until all the specified resources are available, thereby *increasing* the completion time. Hence, OPPerTune attempts to find the right balance between increasing these values and decreasing the workload's wait time in the MLExp scheduler's queue.

We have integrated OPPerTune in two production compute clusters: Cluster1, with 120 nodes, 1800 cores, and 7.03 TB of memory, and Cluster2 with 100 nodes, 1500 cores and 5.86 TB of memory. Each of these clusters serves 24 workloads on average every day, submitted by the ML pipeline developers. Figure 3 shows the spread of completion times for the 11 job types in the two clusters over a period of 1 week before integrating with OPPerTune.

Parameter	Range	
	Min	Max
`spark.driver.memory`	4GB	25GB
`spark.driver.cores`	1	4
`spark.executor.memory`	4GB	24GB
`spark.executor.cores`	1	4
`spark.executor.instances`	24	384
`spark.default.parallelism`	100	10000
`spark.sql.shuffle_partitions`	100	10000

Table 2: Job configuration parameters used for Tables 4, 5.

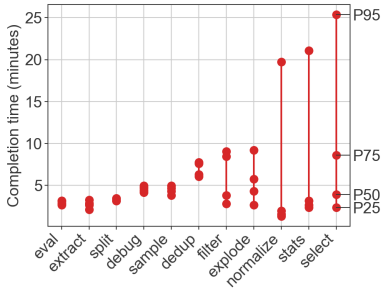


Figure 3: Apache Spark job completion times sorted by the P95 percentile in the third week of November 2022 in two production clusters of MLExp application.

7.2 Improving Application Performance

A) Effectiveness of HybridBandits: We first look at how effective the proposed algorithm HybridBandits is for performance tuning of applications in deployment, and how it fares relative to various state-of-the-art tuning techniques that we implement as part of OPPerTune backend (as discussed in Section 6). For this evaluation, we use the SocialNetwork application, as it has a mix of real-valued, discrete, and categorical parameters (Table 1). For each rps (constRPS workload), we run each tuning algorithm for a maximum of 50 rounds. The results are presented in Figure 4. We report the mean and standard deviation of P95 latencies, for the converged configuration values, over 5 trials (10-minute windows each).

"Predeployment" in Figure 4 refers to the baseline performance against manually chosen configurations that optimized the performance for a 3500 rps workload (which is close to the peak capacity supported by our cluster), and keeping them fixed for the rest of the rps. "Kubernetes Autoscaler" refers to the Vertical Pod Autoscaler (VPA) [3] for determining the rightsizing parameters. VPA performs poorly in general, as rightsizing decisions are solely based on container utilization, and not P95 latency of the application. We also baseline against several existing approaches, i.e. Bayesian Optimization (BO)[2], state-of-the-art RL techniques Slates and DDPG (we set episode length to 1 for the constRPS experiments, so DDPG is effectively standard contextual bandits [74]), and SelfTune's Bluefin with HybridBandits.

[2]Our implementation of BO differs from CherryPick [17] in the choice of acquisition function. We use GP-UCB instead of EI, motivated by the superior performance of GP-UCB as demonstrated by Hoffman et al. [36].

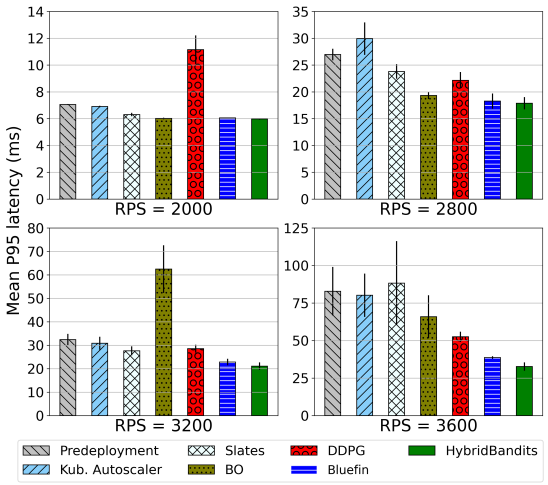


Figure 4: Comparison of various techniques for post-deployment configuration tuning of the SocialNetwork application using constRPS workloads. 'Predeployment' is the baseline performance achieved with configuration choices we manually chose based on 3500-rps workload.

First, the benefit of using OPPerTune service post-deployment is clear: every algorithm, almost for every rps, finds better configuration choices to adapt to changing workload volumes. Second, almost for every rps, HybridBandits significantly outperforms BO, Slates, and DDPG, and achieves the best P95 latency. For instance, at the peak workload of 3600 rps, HybridBandits achieves nearly 2x reduction in P95 latency compared to the best configuration predicted by BO that has a large variance. Third, the utility of tuning categorical parameters together with the numerical parameters, using HybridBandits, is clear at high workloads, compared to Bluefin algorithm in the SelfTune framework [40] that supports only numerical parameters. In particular, HybridBandits (32.6ms) achieves about 15% reduction in P95 latency relative to Bluefin (38.6ms) for the 3600-rps workload. DDPG performs reasonably well in high workloads, despite the absence of informative context. We see similar results when the initial configuration to our algorithm is bad (i.e., yields very high latency), indicating HybridBandits's ability to converge to near-optimal configurations quickly even if the configurations are poor in the initial few rounds.

Takeaway 1. OPPerTune *with* HybridBandits *achieves the best performance, especially at peak workloads, among the state-of-the-art ML techniques used in systems performance optimization.*

B) Effectiveness of AutoScope**:** We now evaluate the benefits of automatically scoping tuning instances using AutoScope, in terms of the application performance as well as sample complexity. We consider SocialNetwork with AMStraces and MLExp for this evaluation.

1. SocialNetwork + **AMStraces:** We compare AutoScope with a domain-expertise based scoping strategy, informed by the diurnal patterns of workloads in AMStraces. We use one tuning instance for every 2 consecutive hours in a cluster-day. Each tuning instance runs HybridBandits independently to learn suitable configuration parameters for its 2-hour scope. We refer to this as $\text{HybridBandits}_{\text{cluster,hour}}$. For AutoScope, we use average rps over every 2 consecutive hours in a day as context. We build a simple estimator for rps values using the first week traces, and use them for all the weeks. This is because we can not know the true rps values (in the future) at the time of Predict calls.

We let all the methods use the first 3 weeks' traces to tune the configuration parameters for SocialNetwork in deployment. We then evaluate the converged parameters on the last week's trace. In Table 3, we present, for each technique and for each cluster, (a) P50, P95, and maximum value of hourly P95 latency, computed over the last week, i.e., over 168 hours, and (b) sample complexity of the technique (i.e., # rewards used while tuning). We compare AutoScope with (i) the baseline of using "Pre-deployment choices" of configuration parameters, (ii) domain-expertise based scoping $\text{HybridBandits}_{\text{cluster,hour}}$, (iii) deep-RL based DDPG that uses rps, and CPU and memory utilization of microservices and VMs as features ("states") for implicit scoping, and (iv) $\text{AutoScope}_{\text{cluster}}$ where a separate AutoScope instance is created for each cluster with rps as the scoping attribute. We see that max P95 latencies for Clusters 2 and 3 are in the order of seconds with the Pre-deployment choices. OPPerTune, using each of the three algorithms, significantly reduces the worst-case P95 latencies. Importantly, AutoScope achieves significantly better performance in general compared to $\text{HybridBandits}_{\text{cluster,hour}}$, and DDPG in Clusters 2 and 3 especially.

Remarkably, AutoScope *achieves this performance using one-third of samples (i.e., # rewards) as that of other techniques*. AutoScope exploits the overlap of diurnal patterns and workload volumes (Appendix C.1) across clusters to improve overall performance, using as few as 8 tuning instances (a height-3 tree), compared to manual scoping using 36 (3 clusters × 12 time-windows in a day) tuning instances. Moreover, $\text{AutoScope}_{\text{cluster}}$ (one AutoScope instance per cluster) performs similarly (except for max latencies) to AutoScope; a single instance of AutoScope can adapt to application dynamics across deployments when the presented context (here, workload volumes) adequately captures the dynamics.

Takeaway 2. OPPerTune *with* AutoScope *is able to significantly improve the application performance, using 3*x *fewer samples needed by manual scoping strategies.*

2. MLExp: For this application, we have 3 types of context information available at the job submission time, namely, job type (11 possible values), dataset size ('large' or 'small'), and cluster used (1 or 2). So, as a baseline, we use the following domain-expertise based scoping strategy. We create one tuning instance per combination, yielding $11 \times 2 \times 2 = 44$

Method	(P50, P95, max) of P95 latency of each hour over the 4th week (ms)									Sample Complexity (#rewards for tuning)
	Cluster 1			Cluster 2			Cluster 3			
	P50	P95	max	P50	P95	max	P50	P95	max	
Pre-deployment choices	12.7	19.4	23.8	12.4	18.2	1959.1	12.3	44.1	4018.4	-
HybridBandits$_{\text{cluster, hour}}$	10.6	17.3	19.3	9.5	19.6	21.0	10.9	17.4	36.7	756
DDPG	10.4	17.0	23.2	8.3	14.2	18.4	9.2	18.1	32.6	756
AutoScope$_{\text{cluster}}$	10.6	15.8	17.0	8.7	15.8	16.8	7.1	15.2	17.6	756
AutoScope	10.1	15.8	23.7	8.5	13.5	17.1	7.9	15.7	33.4	252

Table 3: Comparison of various techniques for post-deployment configuration tuning of the SocialNetwork application using real workloads (AMStraces). Algorithms in the last three rows, implemented in OPPerTune, use the first 3 week-traces for tuning. The first row is the baseline performance achieved with manually-chosen configuration choices.

instances. Each instance runs Bluefinindependently to tune Apache Spark configuration parameters (Table 2) for the jobs in the scope. We refer to this as Bluefin$_{\text{cluster, type, size}}$. AutoScope uses the 3 context values for scoping via height-3 trees, i.e., at most 8 tuning instances (in the leaves). We initialized all the instances using the default choices for job parameters that the developers provided.

We use workloads submitted to the 2 clusters in the Nov 20-Dec 03, 2022 period to tune job parameters using Bluefin$_{\text{cluster, type, size}}$ (suffix indicates the manual scoping strategy) and AutoScope. We ensured that each of the 44 instances using Bluefin$_{\text{cluster, type, size}}$, as well as the 8 instances of AutoScope get at least 5 reward values during the period (to make meaningful updates). We then evaluate the converged instances on the workloads submitted to the 2 clusters between Dec 04, 2022 and Dec 10, 2022. Whenever a developer submits an ML workload, a Predict call is made which decides the scope. Rewards (completion times) are sent back when workloads are complete.

The mean and standard deviation of the workload (i.e., experiment) completion times over one week for different techniques are presented in Table 4. We compare AutoScope with (i) Pre-deployment choices, (ii) Expert choices, which are job-specific configuration choices we elicited from MLExp developers; they have implemented hand-crafted heuristics (refined over several months) to improve the job completion times on the clusters, based on individual job characteristics such as the number of data records processed in the pipeline, type of the job, repartitioning costs, etc., and (iii) Bluefin$_{\text{cluster, type, size}}$.

We see that Bluefin$_{\text{cluster, type, size}}$ and AutoScope have reduced the mean workload completion times by more than 50% that of the Pre-deployment choices in Cluster 1; and by about 10% in Cluster 2. Also, they perform as well as the Expert choices in Cluster 1, and better (significance determined using standard t-test at p-value of 0.05) in Cluster 2. For 8 out of 11 job types, AutoScope achieves up to 2× smaller P95 completion times than expert choices (see Appendix C.2).

A highlight of this deployment study is that AutoScope, using only 90 samples (rewards) for 8 instances, is able to achieve performance competitive to Bluefin$_{\text{cluster, type, size}}$ that uses 407 samples for 44 instances. The manual clustering of Bluefin$_{\text{cluster, type, size}}$ indeed is marginally better than

Method	Experiment Completion Time (in minutes)		Sample Complexity (#rewards)	
	Cluster 1	Cluster 2	Cluster 1	Cluster 2
Pre-deployment choices	105.85 ± 16.75	36.66 ± 1.60	-	-
Expert choices	42.41 ± 5.28	34.46 ± 4.72	-	-
Bluefin$_{\text{cluster, type, size}}$	38.56 ± 6.55	30.79 ± 0.52	94	313
AutoScope	38.98 ± 5.90	32.71 ± 0.26	29	61

Table 4: Comparison of techniques for post-deployment configuration tuning of Apache Spark parameters in MLExp application, on 2 production clusters, over 1 week of evaluation. AutoScope is competitive to domain-expertise based scoping strategy (Bluefin$_{\text{cluster, type, size}}$) using far fewer samples.

AutoScope. This is expected because Bluefin$_{\text{cluster, type, size}}$ fits optimal parameters for individual partitions of the deployment, i.e., job type, size, cluster. This is sample-inefficient in practice, as we can share parameters when partitions are similar (say two job types with similar resource requirements). AutoScope does this and can achieve similar results with a third of the samples. We also present individual job completion time statistics for the 11 different job types in Table 5. The observations are similar to Table 4.

7.3 Mitigating Cost Of Tuning In-Deployment

So far, we have focused on the impact of tuning on the application performance. We now turn to the cost of tuning in deployment—every change to configuration parameters in production introduces potential risk. This section also demonstrates how operators can use the *selector* (Section 5) module of OPPerTune to select a subset of parameters that improve another metric of interest along with performance. In this case, we use availability as the metric of interest. As we discussed in Section 2, tuning certain configuration parameters necessitates microservice/pod restarts, causing downtimes. Improving latency of the application at the expense of throughput, or service reliability, may not be acceptable.

We evaluate OPPerTune, in terms of how it trades off improving performance and mitigating restarts while tuning, on the SocialNetwork application and constRPS workloads. The results are summarized in Table 6 and in Figure 5. We consider various strategies for picking configuration parameters, followed by tuning the selected parameters with HybridBandits, to mitigate the number of pod restarts. The first row of the

Method	Job Completion Time in minutes (mean ± std. dev)										
	Select	Filter	Explode	Normalize	Stats	Extract	Dedup	Split	Sample	Debug	Eval
Default choices	14.92 ± 19.80	18.43 ± 7.53	21.70 ± 6.64	3.12 ± 2.26	4.17 ± 1.41	2.04 ± 1.01	4.86 ± 1.28	2.24 ± 0.45	6.49 ± 2.50	6.13 ± 2.88	4.09 ± 2.24
Expert choices	6.20 ± 6.87	5.58 ± 1.29	7.16 ± 2.56	1.96 ± 0.45	3.73 ± 1.17	1.43 ± 0.20	5.59 ± 2.31	2.52 ± 0.44	3.06 ± 0.78	3.31 ± 0.98	2.26 ± 0.88
$\text{Bluefin}_{\text{cluster, type, size}}$	4.84 ± 4.71	5.21 ± 0.87	7.42 ± 4.01	1.88 ± 0.38	2.51 ± 0.66	2.14 ± 0.09	3.18 ± 0.25	3.08 ± 0.39	2.78 ± 0.32	2.96 ± 0.19	1.85 ± 0.21
AutoScope	4.92 ± 4.23	6.45 ± 0.46	6.69 ± 3.12	1.66 ± 0.19	2.36 ± 0.42	2.22 ± 0.13	4.77 ± 0.35	2.97 ± 0.12	2.62 ± 0.3	3.88 ± 0.1	1.78 ± 0.23

Table 5: Job completion times for various job types submitted to 2 MLExp clusters over 1 week of evaluation.

Parameters/ Layers tuned	P95 Latency (ms) (mean ± std. dev)			
	RPS = 2000	RPS = 2800	RPS = 3200	RPS = 3600
MS ∪ RS	5.973 ± 0.046	17.864 ± 1.150	21.068 ± 1.641	32.656 ± 2.798
NR	6.916 ± 0.076	24.545 ± 1.184	31.281 ± 1.958	70.542 ± 22.485
MS	6.405 ± 0.039	24.209 ± 1.594	26.764 ± 2.184	38.853 ± 2.500
RS	6.828 ± 0.022	25.373 ± 0.607	26.774 ± 2.168	70.175 ± 8.267
MBT-25	6.820 ± 0.060	23.697 ± 1.802	27.022 ± 1.278	34.503 ± 2.642
MBT-50	6.094 ± 0.075	19.248 ± 1.362	21.888 ± 0.853	37.821 ± 1.489

MS=Microservices, RS=Rightsizing, NR=NoRestarts, MBT=Microbenchmark-Top

Table 6: Comparison of various ways of selecting parameters to tune (here, via OPPerTune-HybridBandits) in the SocialNetwork application stack using constRPS workloads. Microbenchmarking strategy (last row) yields performance nearly as good as tuning all the parameters (first row).

table ("Microservices ∪ Rightsizing") corresponds to tuning all the parameters listed in Table 1. The second row of the table ("NoRestarts") corresponds to tuning only the parameters that do *not* require any restarts. As expected, they achieve the best and the worst P95 latency values, respectively.

In Section 5, we introduced the microbenchmarking strategy in OPPerTune for picking the most promising configurations ahead of tuning in deployment. The last row of Table 6 shows the performance achieved using HybridBandits on the top-50 parameters (See Appendix D for the list of parameters): (i) in 3 out of 4 workload rates, we see that the strategy achieves statistically similar performance as the best ("Microservices ∪ Rightsizing"); (ii) with the reduced configuration space, HybridBandits converges within 30 rounds, compared to the 50 rounds needed by the best method (not indicated in the table); and (iii) HybridBandits is superior to tuning only the microservices layer parameters (third row) or rightsizing layer parameters (fourth row). We also included the performance using top-25 in the fifth row for comparison.

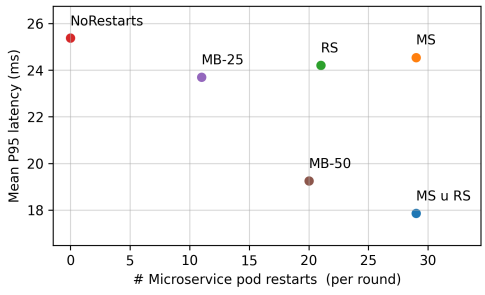


Figure 5: Number of microservice pod restarts (per round) and mean P95 latency (RPS=2800) while tuning different parameters/layers of SocialNetwork app using HybridBandits.

Figure 5 shows the relationship between average P95 latency (measured in milliseconds) and the number of pod restarts per round for each of the five strategies shown in Table 6. We see that our microbenchmarking strategy achieves a good trade-off between cost and performance, using 2800-rps workload as example (though the findings are consistent across all rps). The best average P95 latency (17.9ms) of "Microservices ∪ Rightsizing" or "MS ∪ RS" comes at the expense of 29 pod restarts *per round* as seen from Figure 5. The microbenchmarking strategy ("MB-50" in 5) nearly matches the best method's P95 latency (19.2ms), with nearly 30% fewer pod restarts *per round* (20 restarts).

Takeaway 3. HybridBandits + *microbenchmarking strategy of* OPPerTune *reduces the cost of tuning in terms of service disruptions in deployment significantly, while achieving competitive performance.*

7.4 Scalability

We evaluate the scalability of OPPerTune service with respect to various tuning algorithms that we implemented (or integrated) as part of the backend (as discussed in Section 6). We perform this study on the same VM type as in Section 7.1.1.

We focus on the throughput of OPPerTune service, i.e., number of Predict and SetReward requests served, with various back-end algorithms; so, we use a simple synthetic application to stress-test the service and the algorithms.

Using wrk2, we simulate a scenario where there is one client that (a) creates a new instance, specifying 30 configuration parameters (25 numerical and 5 categorical for algorithms that allow hybrid configurations) and the algorithm to use, and (b) then repeatedly sends Predict and SetReward requests to the created instance on the OPPerTune server, for 120 seconds. For algorithms which need context (AutoScope, DDPG), we use random context vectors of size $d = 8$. For rewards, we use a random value between 0 and 1 (actual reward value does not matter for this study). We maintain the instance state in memory for all the tuning algorithms for this study.

In this setting, we first verified that our Go-based server can handle 4096 rps without dropping any request, bypassing the tuning algorithm. Table 7 shows the OPPerTune service throughput for various backend algorithms. First, we find that the ordering of Bluefin, HybridBandits, AutoScope, and DDPG (our implementations) is as expected. The latency of the requests is proportional to the size of the model—HybridBandits achieves lower throughput than Bluefin, because it needs to sample from a probability distribution over all possible categorical choices ($|\mathcal{C}| = 720$ in this study).

Algorithm	Bluefin	HybridBandits	AutoScope	DDPG	Slates	BO
Throughput (RPS)	2226	1540	825	594	2.2	1.5

Table 7: Maximum throughput of OPPerTune service with various back-end algorithms.

Among the algorithms that use context, i.e., AutoScope and DDPG, AutoScope is faster because its inference time is proportional to $hd + m + |\mathcal{C}|$, where $h = 3$ is the tree height for experiments in this paper, $d = 8$, $m = 25$, for this study (see notation in Section 3.2); whereas, DDPG has to perform inference using a neural network of size $(m+k+d) \times$ #hidden layers, where $m+k = 30$ parameters in this study, and #hidden layers = 32 for experiments in this paper. Popular implementations of BO and Slates algorithms, that we integrated in our back-end, do not scale at all.

8 Related Work

Configuration Tuning: Performance optimization of systems through configuration tuning is a long-studied problem [37, 59, 60] that continues to garner interest from the systems research community. Prior works on configuration tuning mainly focus on parameters of specific subsystems of applications [16, 17, 24, 46, 54, 69, 69–71] such as database and storage or of the hosting infrastructure [21, 22, 55, 58, 63]. In such works, the configuration search space is relatively small, and the advantages of jointly tuning parameters across the software stack are not considered. Moreover, the approaches are tailored to the specific subsystem being tuned, sometimes requiring domain expertise [16, 24]. Recently, jointly tuning parameters across the software stack [25, 48] and across multiple components of an application [61, 64] is gaining attention. Such works either ignore the cost of reconfiguration [61] or require an expensive offline training [25, 48, 64].

RL/Bandit Algorithms: While there are several RL-based configuration tuning approaches [55, 57, 73], they are either limited in the type of parameters being tuned or are inefficient for online post-deployment tuning scenarios. Some approaches do handle hybrid parameter spaces [43, 49] via cascaded optimization which are effective only when trained offline. Deploying parameters, observing a reward and making updates in real systems fits the bandits paradigm (sample; reward; update) better than long-horizon RL paradigm (sample; reward; sample; reward; ...; update). Long-horizon/episodic RL has higher sample complexity and needs hand-crafted state information (e.g, utilization metrics) to learn effective policies. While this allows robustness in dynamic environments, it poses additional engineering overhead, and hand-designing state information is challenging in enterprise scenarios. In our experience working with developers, determining a useful reward cycle (time horizon) is fairly easy with some domain expertise making our bandits approach effective compared to RL and our proposed HybridBandits can tune all types of parameters in deployment without such overhead.

Recent work on learning trees using bandit feedback [28, 29] are designed for categorical spaces. Popular tree learning algorithms like CART [23] and C4.5 [56] do not apply to the bandit feedback setting because they need access to labeled training data (which in our scenario means optimal parameters for different context vectors, which we do *not* have).

Tuning Frameworks: Recent works have addressed the need for a generic configuration tuning framework for production systems [33, 40, 58, 67, 75]. KEA [75], Microsoft's internal tuning framework for cluster-wide configurations, uses historical data to make decisions on parameters in the pre-deployment stages. The most recent SelfTune [40] framework from Microsoft for tuning cluster managers supports post-deployment tuning but lacks support for tuning categorical parameters and requires domain expertise for setting up tuning instances for complex, distributed applications. Google's Vizier [33] is their internal service for hyper-parameter tuning of ML workloads in the offline setting. Twine [67] is Meta's cluster management system for workload-specific customizations such as tuning of hardware and OS settings. OpenTuner [18] provides a framework to build domain-specific tuners.

9 Conclusion

We have designed, built, and deployed the OPPerTune configuration tuning service at Microsoft. Our work differs from related work on configuration tuning in many ways: 1) we tackle challenges arising in post-deployment tuning, 2) we focus on sample complexity of algorithms as well as the cost of tuning, unlike systems tuning efforts that rely on offline training or controlled settings, 3) we give an end-to-end solution for configuration tuning that is fairly general and readily applicable. We demonstrate through two real-world deployments that our techniques yield state-of-the-art performance, are sample-efficient, and reduce the tuning cost.

This work addresses many challenges in post-deployment configuration tuning through OPPerTune. However, future work can address the following challenges and also improve the framework for easier adoption. Firstly, OPPerTune's scalability can be evaluated by using it to tune applications with a very large (say, thousands) number of parameters. Secondly, a fine-grained analysis of different costs (e.g., performance and revenue) associated with tuning parameters can be conducted. Thirdly, tuning parameters from different layers of the software stack (e.g., OS and hardware), along with an analysis of their interrelationships, would be an exciting direction. Lastly, OPPerTune can be implemented as a Kubernetes operator for seamless integration with the application ecosystem.

10 Acknowledgment

We are grateful to Sayak Ray Chowdhury for discussions and feedback on the tuning algorithms. We would like to thank our shepherd Arpit Gupta and the anonymous reviewers for their valuable feedback. This work was supported by NSF grants CNS 2324859, 2214980, 2106434, 1909356, and 1750109.

References

[1] Beginner's guide. https://nginx.org/en/docs/beginners_guide.html.

[2] Controlling nginx. https://nginx.org/en/docs/control.html.

[3] Kubernetes Vertical Pod Autoscaler. https://github.com/kubernetes/autoscaler/blob/master/vertical-pod-autoscaler/.

[4] memcached(1). https://linux.die.net/man/1/memcached.

[5] MongoDB. https://docs.mongodb.com/manual/reference/parameters/.

[6] Recreate Deployment. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#strategy.

[7] Redis configuration. https://redis.io/topics/config.

[8] Tuning nginx for performance. https://www.nginx.com/blog/tuning-nginx/.

[9] wrk2 HTTP Workload Generator. https://github.com/giltene/wrk2.

[10] Fiber. https://github.com/gofiber/fiber, 2022.

[11] Scikit-optimize. https://github.com/scikit-optimize/scikit-optimize, 2022.

[12] The vowpal wabbit library. https://github.com/VowpalWabbit/vowpal_wabbit, 2022.

[13] Randy Abernethy. *The Programmer's Guide to Apache Thrift*. 2018.

[14] Alekh Agarwal, Sarah Bird, Markus Cozowicz, Luong Hoang, John Langford, Stephen Lee, Jiaji Li, Dan Melamed, Gal Oshri, Oswaldo Ribas, et al. A multiworld testing decision service. *arXiv preprint arXiv:1606.03966*, 7, 2016.

[15] Ibrahim Umit Akgun, Ali Selman Aydin, Andrew Burford, Michael McNeill, Michael Arkhangelskiy, and Erez Zadok. Improving storage systems using machine learning. *ACM Trans. Storage*, nov 2022.

[16] Sami Alabed and Eiko Yoneki. High-dimensional bayesian optimization with multi-task learning for rocksdb. EuroMLSys '21, 2021.

[17] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, 2017.

[18] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014.

[19] Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E Schapire. The nonstochastic multiarmed bandit problem. *SIAM journal on computing*, 32(1):48–77, 2002.

[20] Alberto Bietti, Alekh Agarwal, and John Langford. A contextual bandit bake-off. *Journal of Machine Learning Research*, 22(133):1–49, 2021.

[21] Muhammad Bilal, Marco Canini, and Rodrigo Rodrigues. Finding the right cloud configuration for analytics clusters. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20.

[22] Muhammad Bilal, Marco Serafini, Marco Canini, and Rodrigo Rodrigues. Do the best cloud configurations grow on trees? an experimental evaluation of black box algorithms for optimizing cloud workloads. *Proc. VLDB Endow.*, 2020.

[23] L Breiman, JH Friedman, R Olshen, and CJ Stone. Classification and regression trees. 1984.

[24] Zhen Cao, Geoff Kuenning, and Erez Zadok. Carver: Finding important parameters for storage system tuning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*.

[25] Stefano Cereda, Stefano Valladares, Paolo Cremonesi, and Stefano Doni. Cgptuner: A contextual gaussian process bandit approach for the automatic tuning of it configurations under varying workload conditions. *Proc. VLDB Endow.*, 14(8):1401–1413, oct 2021.

[26] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[27] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proc. VLDB Endow.*, 2(1):1246–1257, aug 2009.

[28] Adam N Elmachtoub, Ryan McNellis, Sechan Oh, and Marek Petrik. A practical method for solving contextual bandit problems using decision trees. *Uncertainty in Artificial Intelligence*, 2017.

[29] Raphaël Féraud, Robin Allesiardo, Tanguy Urvoy, and Fabrice Clérot. Random forest for the contextual bandit

problem. In *Artificial intelligence and statistics*, pages 93–101. PMLR, 2016.

[30] Abraham D Flaxman, Adam Tauman Kalai, and H Brendan McMahan. Online convex optimization in the bandit setting: gradient descent without a gradient. In *Proceedings of the sixteenth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 385–394, 2005.

[31] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayantara Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Yuan He, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[32] Mitsuo Gen and Runwei Cheng. *Genetic algorithms and engineering optimization*, volume 7. John Wiley & Sons, 1999.

[33] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Elliot Karro, and D. Sculley, editors. *Google Vizier: A Service for Black-Box Optimization*, 2017.

[34] John Hancock and Taghi Khoshgoftaar. Survey on categorical data for neural networks. *Journal of Big Data*, 7, 04 2020.

[35] Herodotos Herodotou, Yuxing Chen, and Jiaheng Lu. A survey on automatic parameter tuning for big data processing systems. *ACM Comput. Surv.*, 53(2), apr 2020.

[36] Matthew Hoffman, Eric Brochu, and Nando de Freitas. Portfolio allocation for bayesian optimization. In *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence*, UAI'11, page 327–336, Arlington, Virginia, USA, 2011. AUAI Press.

[37] Chiaki Ishikawa, Ken Sakamura, and Mamoru Maekawa. Dynamic tuning of operating systems. In Mamoru Maekawa and Laszio A. Belady, editors, *Operating Systems Engineering*, pages 119–142, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[38] Satyen Kale, Lev Reyzin, and Robert E Schapire. Nonstochastic bandit slate problems. *Advances in Neural Information Processing Systems*, 23, 2010.

[39] Ajaykrishna Karthikeyan, Naman Jain, Nagarajan Natarajan, and Prateek Jain. Learning accurate decision trees with bandit feedback via quantized gradient descent. *Transactions on Machine Learning Research*, Sep 2022.

[40] Ajaykrishna Karthikeyan, Nagarajan Natarajan, Gagan Somashekar, Lei Zhao, Ranjita Bhagwan, Rodrigo Fonseca, Tatiana Racheva, and Yogesh Bansal. SelfTune: Tuning cluster managers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1097–1114, Boston, MA, April 2023. USENIX Association.

[41] Tor Lattimore and Csaba Szepesvári. *Bandit algorithms*. Cambridge University Press, 2020.

[42] Boyan Li, Hongyao Tang, YAN ZHENG, HAO Jianye, Pengyi Li, Zhen Wang, Zhaopeng Meng, and LI Wang. Hyar: Addressing discrete-continuous action reinforcement learning via hybrid action representation. In *International Conference on Learning Representations*, 2021.

[43] Boyan Li, Hongyao Tang, YAN ZHENG, HAO Jianye, Pengyi Li, Zhen Wang, Zhaopeng Meng, and LI Wang. Hyar: Addressing discrete-continuous action reinforcement learning via hybrid action representation. In *International Conference on Learning Representations*, 2021.

[44] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. Metis: Robustly tuning tail latencies of cloud systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 981–992, Boston, MA, July 2018. USENIX Association.

[45] Timothy Lillicrap, Jonathan Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, 09 2015.

[46] Chen Lin, Junqing Zhuang, Jiadong Feng, Hui Li, Xuanhe Zhou, and Guoliang Li. Adaptive code learning for spark configuration tuning. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1995–2007, 2022.

[47] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 412–426, New York, NY, USA, 2021. Association for Computing Machinery.

[48] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[49] Warwick Masson, Pravesh Ranchod, and George Konidaris. Reinforcement learning with parameterized actions. In *AAAI*, 2016.

[50] Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Maddila, Balasubramanyan Ashok, Sumit Asthana, Christian Bird, and Aditya Kumar. Rex: Preventing bugs and misconfiguration in large services using correlated change analysis. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 435–448, 2020.

[51] Yu Nesterov. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM Journal on Optimization*, 22(2):341–362, 2012.

[52] Vladimir I Norkin, Georg Ch Pflug, and Andrzej Ruszczyński. A branch and bound method for stochastic global optimization. *Mathematical programming*, 83(1):425–450, 1998.

[53] Julie Nutini, Mark Schmidt, Issam Laradji, Michael Friedlander, and Hoyt Koepke. Coordinate descent converges faster with the gauss-southwell rule than random selection. In *International Conference on Machine Learning*, pages 1632–1641. PMLR, 2015.

[54] David Buchaca Prats, Felipe Albuquerque Portella, Carlos H. A. Costa, and Josep Lluis Berral. You only run once: Spark auto-tuning from a single run. *IEEE Transactions on Network and Service Management*, 17(4):2039–2051, 2020.

[55] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[56] J Ross Quinlan. Program for machine learning. *C4. 5*, 1993.

[57] Fabiana Rossi, Valeria Cardellini, Francesco Lo PRESTI, and Matteo Nardelli. Dynamic multi-metric thresholds for scaling applications using reinforcement learning. *IEEE Transactions on Cloud Computing*, pages 1–1, 2022.

[58] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[59] Jeffrey Semke, Jamshid Mahdavi, and Matthew Mathis. Automatic tcp buffer tuning. *SIGCOMM Comput. Commun. Rev.*, 28(4):315–323, oct 1998.

[60] Dennis Shasha. Lessons from wall street: Case studies in configuration, tuning, and distribution. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, page 498–501, New York, NY, USA, 1997. Association for Computing Machinery.

[61] G. Somashekar, A. Suresh, S. Tyagi, V. Dhyani, K. Donkada, A. Pradhan, and A. Gandhi. Reducing the tail latency of microservices applications via optimal configuration tuning. In *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 111–120, Los Alamitos, CA, USA, sep 2022. IEEE Computer Society.

[62] Gagan Somashekar and Rajat Kumar. Enhancing the configuration tuning pipeline of large-scale distributed applications using large language models (idea paper). In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, ICPE '23 Companion, page 39–44, New York, NY, USA, 2023. Association for Computing Machinery.

[63] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F. Wenisch. Softsku: Optimizing server architectures for microservice diversity @scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 513–526, New York, NY, USA, 2019. Association for Computing Machinery.

[64] Akshitha Sriraman and Thomas F. Wenisch. Mtune: AutoTuned Threading for OLDI microservices. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI 2018, page 177–194, USA, 2018. USENIX Association.

[65] Adith Swaminathan, Akshay Krishnamurthy, Alekh Agarwal, Miro Dudik, John Langford, Damien Jose, and Imed Zitouni. Off-policy evaluation for slate recommendation. *Advances in Neural Information Processing Systems*, 30, 2017.

[66] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind

Narayanan, Patrick Dowell, and Robert Karl. Holistic configuration management at facebook. In *Proceedings of the 25th symposium on operating systems principles*, pages 328–343, 2015.

[67] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 787–803. USENIX Association, November 2020.

[68] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1009–1024, New York, NY, USA, 2017. Association for Computing Machinery.

[69] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, pages 1009–1024, 2017.

[70] Muhammad Wajahat, Salman Masood, Abhinav Sau, and Anshul Gandhi. Lessons learnt from software tuning of a memcached-backed, multi-tier, web cloud application. In *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, pages 1–6, 2017.

[71] Jinhan Xin, Kai Hwang, and Zhibin Yu. Locat: Low-overhead online configuration auto-tuning of spark sql applications. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 674–684, New York, NY, USA, 2022. Association for Computing Machinery.

[72] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 244–259, New York, NY, USA, 2013. Association for Computing Machinery.

[73] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 415–432, New York, NY, USA, 2019. Association for Computing Machinery.

[74] Zihan Zhang, Xiangyang Ji, and Simon Du. Is reinforcement learning more difficult than bandits? a near-optimal algorithm escaping the curse of horizon. In *Proceedings of Thirty Fourth Conference on Learning Theory*, pages 4528–4531, 2021.

[75] Yiwen Zhu, Subru Krishnan, Konstantinos Karanasos, Isha Tarte, Conor Power, Abhishek Modi, Manoj Kumar, Deli Zhang, Kartheek Muthyala, Nick Jurgens, Sarvesh Sakalanaga, Sudhir Darbha, Minu Iyer, Ankita Agarwal, and Carlo Curino. Kea: Tuning an exabyte-scale data infrastructure. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, 2021.

A AutoScope Algorithm

Algorithm 2 AutoScope: Automatic Scoping of Configuration Tuning via Decision Trees

Input: tree height h, learning rate $\eta > 0$, categorical parameter space $\mathcal{C} := \mathcal{C}_1 \times \mathcal{C}_2 \times \cdots \times \mathcal{C}_k$, numerical parameter space $\mathcal{W} = \mathcal{W}_1 \times \mathcal{W}_2 \times \cdots \times \mathcal{W}_m$

Initialize: Tree node weights $\mathbf{z}_j^{(0)} \in \mathbb{R}^d$, for each internal node $0 \le j \le 2^h - 2$, ℓth leaf node categorical space weights $p_{i,\ell}^{(0)} = 1/|\mathcal{C}|$, for $1 \le i \le |\mathcal{C}|$ // uniform distribution, and ℓth leaf node numerical parameters $w_{i,\ell}^{(0)} \in \mathcal{W}_i$, for $1 \le i \le m$, $0 \le \ell \le 2^h - 1$ // default choices

(☆) Each leaf node := a "tuning instance", and is initialized identical to Algorithm HybridBandits Step 2.

for $t = 0, 1, 2, \ldots$ **do**

(1) Determine the scope using the tree guided by the observed context, and use the appropriate tuning instance

Observe context $\mathbf{c}^{(t)} \in \mathbb{R}^d$

Get the instance at the leaf node ℓ by navigating the tree $f_T(\mathbf{c}^{(t)}; \{\mathbf{z}_j^{(t)}\})$

Sample categorical and numerical actions following Steps 4–6 of Algorithm HybridBandits on the ℓth instance

(2) Deploy the actions and measure reward

Follow Steps 7–8 of Algorithm HybridBandits and observe the reward $r^{(t)} = r_t(\cdot)$

(3) Perform updates on the tree

Update the ℓth instance using Steps 9–11 of Algorithm HybridBandits // Other tuning instances remain the same

Update tree node weights, for $0 \le j \le 2^h - 2$: $\mathbf{z}_j^{(t+1)} \leftarrow \mathbf{z}_j^{(t)} + \eta \widetilde{\nabla}_{\mathbf{z}_j} r_t(\cdot)$ // $\widetilde{\nabla}$ is the estimated gradient as in [38]

B HybridBandits Convergence

We design a simple synthetic function characterized by five continuous parameters and one categorical parameter. The categorical parameter offers a choice between two options {f1, f2} while all continuous parameters are bounded within the range $[0, 1]$. The blackbox function, $f: [\mathcal{C}, \mathcal{W}] \to \mathbb{R}$, calculates the root mean squared error of the continuous parameters. Additionally, it incorporates a bias term that depends on the categorical parameter's choice (bias=1 if f2 else bias=0). We compare the convergence of our proposed HybridBandits algorithm, with Hyperopt's Tree of Parzen Estimators (TPE) and the random search baseline in Figure 6. The metric plotted here is the mean cumulative "regret" (i.e., the difference between the estimates of the parameters and their optimal values, accumulated over rounds) with shaded 95% Confidence Interval (CI). We can see that random search does not converge as expected, while HybridBandits demonstrates significantly faster convergence compared to TPE.

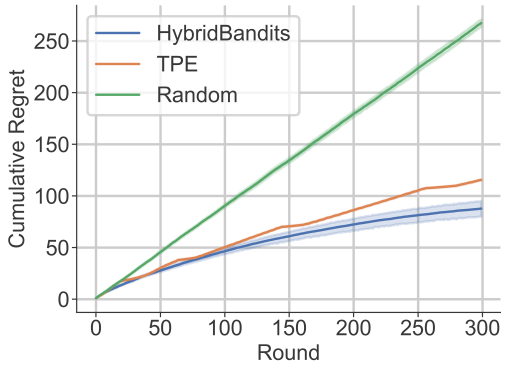


Figure 6: Mean ± 95% CI cumulative regret over 25 runs.

C Experiment Details

In this section, we provide additional experimental details.

C.1 SocialNetwork + AMStraces

The AMStraces consist of traces collected from 3 different data center regions across two continents. The traces consist of the number of GET and PUT requests that arrive each minute over four weeks along with additional details like payload size, latency, etc. The peak requests per minute (RPM) across the 3 clusters are around 30000, 40000, and 140000. As seen in Figure 7, the traces show a diurnal pattern with a reduced load over the weekend. We sample from these traces such that a day's original trace can be replayed in 1.2 hours. We started with a higher sampling rate and arrived at this as it succinctly captures the patterns in the original traces yet makes the experiment iteration feasible.

The PUT and GET requests in the AMStraces are mapped to the PUT (compose-post) and GET (read-user-timeline and read-home-timeline) requests of the social networking application. The payload size of the PUT requests is also driven by the payload sizes from the traces. The call graphs of the GET and PUT requests in the social networking application capture the complexities of the GET and PUT requests in the AMS service.

C.2 MLExp

In Table 8, we provide 95th percentile of the job completion time in minutes. For 8 out of 11 job types, AutoScope achieves up to smaller P95 completion times than expert choices.

D Microbenchmarking: Top-50 Parameters

Figure 8 gives the top-50 parameters used in Table 6 and Figure 5. The figure also indicates the importance of tuning the application (30 out of top-50) and Kubernetes parameters

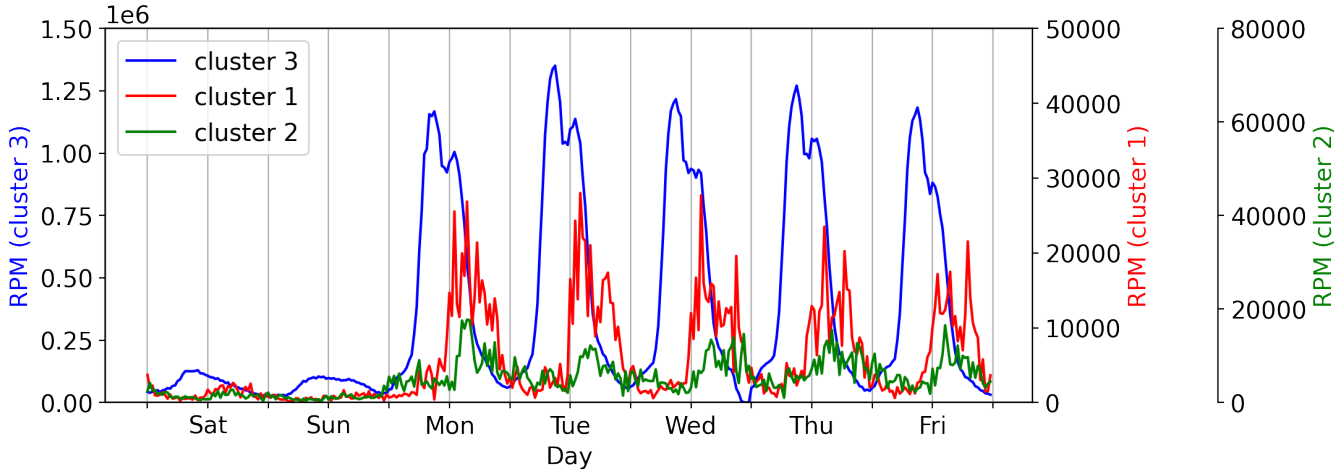


Figure 7: Requests Per Minute (RPM) across the three clusters over a week. The figure shows a diurnal pattern with a significant reduction in traffic over the weekends.

Method	P95 Job Completion Time in minutes										
	Select	Filter	Explode	Normalize	Stats	Extract	Dedup	Split	Sample	Debug	Eval
Default choices	58.39	29.16	31.78	6.72	5.96	3.71	6.96	2.96	11.56	10.76	7.64
Expert choices	17.36	7.65	9.9	2.68	5.48	1.75	9.04	3.13	4.38	4.60	3.55
$\text{Bluefin}_{\text{cluster, type, size}}$	13.47	6.33	12.38	2.49	3.92	2.29	3.56	3.69	3.43	3.13	2.11
AutoScope	11.45	7.05	10.56	1.85	3.12	2.35	5.14	3.14	3.15	3.94	2.09

Table 8: P95 Job completion times for various job types submitted to 2 MLExp clusters over 1 week of evaluation.

(20 out of top-50) jointly as they are both critical to the application's performance. We can also see a mixture of numerical and categorical parameters among the top 50 parameters.

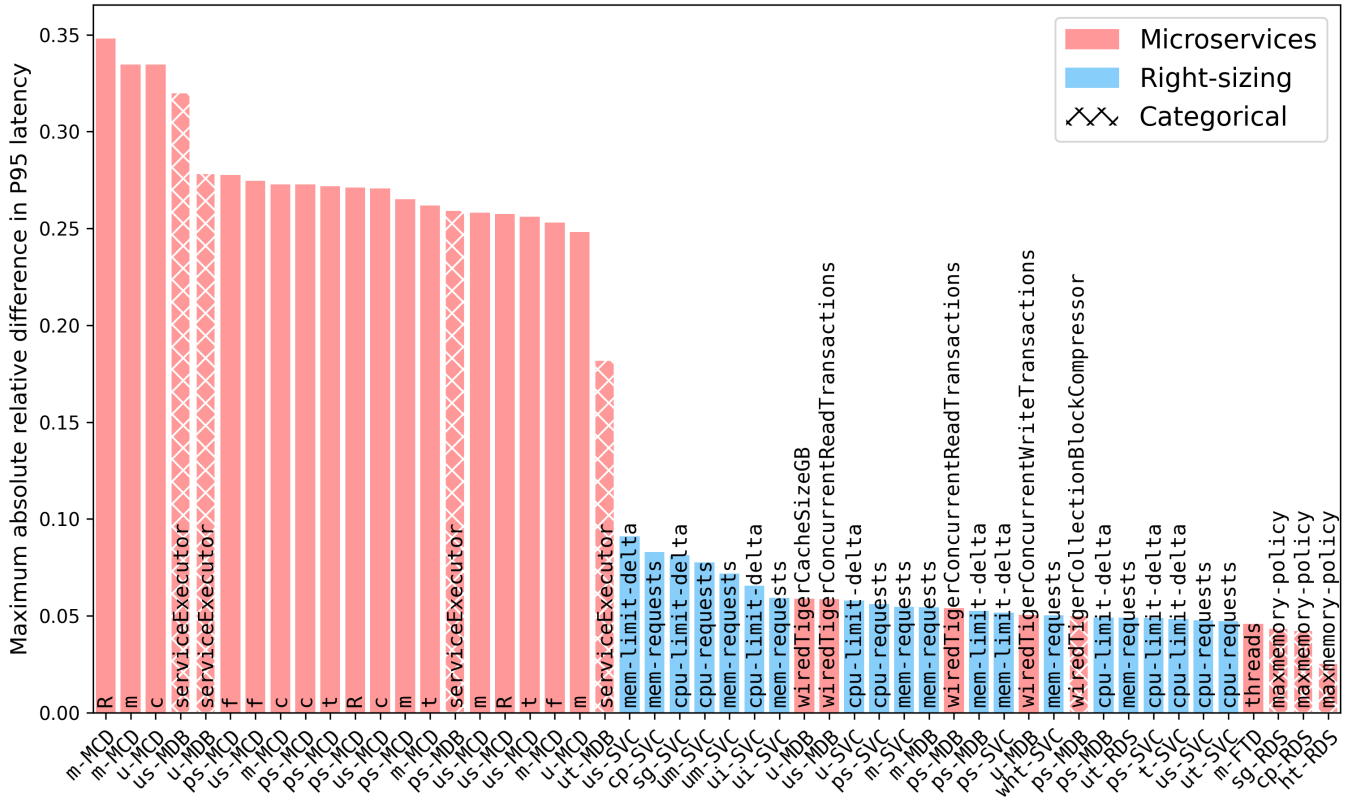


Figure 8: Top 50 parameters selected by the micro-benchmarking strategy in Section 6, and the maximum absolute relative difference in P95 latencies observed by perturbing each of the 50 parameters one at a time w.r.t. a fixed baseline.

Parcae: Proactive, Liveput-Optimized DNN Training on Preemptible Instances

Jiangfei Duan[‡♠] Ziang Song[§♠] Xupeng Miao[†♠] Xiaoli Xi[†]
Dahua Lin[‡] Harry Xu[♯] Minjia Zhang[◇] Zhihao Jia[†]

Carnegie Mellon University[†] *The Chinese University of Hong Kong*[‡]
ByteDance[§] *University of California, Los Angeles*[♯] *Microsoft*[◇]

Abstract

Deep neural networks (DNNs) are becoming progressively large and costly to train. This paper aims to reduce DNN training costs by leveraging preemptible instances on modern clouds, which can be allocated at a much lower price when idle but may be preempted by the cloud provider at any time. Prior work that supports DNN training on preemptive instances employs a *reactive* approach to handling instance preemptions and allocations after their occurrence, which only achieves limited performance and scalability.

We present Parcae, a system that enables cheap, fast, and scalable DNN training on preemptible instances by *proactively* adjusting the parallelization strategy of a DNN training job to adapt to predicted resource changes before instance preemptions and allocations really happen, which significantly reduces the cost of handling these events. Parcae optimizes *liveput*, a novel metric that measures the *expected* training throughput of a DNN job under various possible preemption scenarios. Compared to existing reactive, throughput-optimized systems, Parcae's proactive, live-optimized solution considers both the throughput of a job and its robustness under preemptions. To optimize liveput, Parcae supports lightweight instance migration and uses an availability predictor to forecast future preemptions. It then uses a liveput optimizer to discover an optimal strategy to parallelize DNN training under predicted preemptions. We evaluate Parcae on a variety of DNNs and preemption traces and show that Parcae outperforms existing spot-instance DNN training systems by up to 10×. More importantly, Parcae achieves near-optimal performance for training large DNNs under frequent preemptions, in which case existing approaches cannot make any progress.

1 Introduction

Deep neural networks (DNNs) have surpassed human predictive performance on a spectrum of tasks, including computer vision [18], natural language progressing [14], game playing [44], and content generation [46]. The success of DNNs is associated with progressively increasing energy and financial costs. For example, a single training run of GPT-3 [12], a language model with 175 billion parameters, requires more than 1.5 million GPU hours and costs $4.6 million to train on AWS even with the lowest priced GPUs [37]. While pre-trained models are publicly available and can be fine-tuned for different downstream tasks, training new models is often required for emerging applications and datasets.

Modern cloud platforms provide a variety of cheap *preemptible instances*, which can be leveraged to minimize the monetary cost of DNN training. First, spot GPU instances allow users to take advantage of unused GPU capacity at a price up to 90% lower than on-demand counterparts [1]. Second, modern data centers generally reserve additional GPU capacity for urgent jobs, which can be allocated by other jobs in a preemptible manner [35]. Third, some ML systems [51] support opportunistically running training jobs on inference-dedicated GPUs to maximize resource utilization and preempt these training jobs when inference requests arrive. While this paper focuses on spot GPUs, our techniques can easily generalize to other preemptible resources.

Existing systems that support DNN training on spot instances use a *reactive* approach to handling instance preemption and allocation, and can be categorized into two classes: *checkpoint-* and *redundancy-based* systems. We introduce the two categories and identify the limitations of these reactive approaches in *performance* and *scalability* when applied to DNN training on preemptible instances.

The first line of work uses *checkpoints* to maintain model states during training. For example, Varuna [8] periodically saves model states to persistent storage and loads the latest checkpoint back after a preemption, as shown in Figure 1c. Although Varuna offers promising training throughput when spot instances have low preemption rates, it struggles to make progress when preemptions are frequent. This is due to two reasons: (1) saving and loading checkpoints incur significant IO overhead, particularly as model size increases, making frequent checkpointing costly, and (2) high preemption rates cause training to frequently roll back to the last saved check-

♠ Contributed equally. Work done during internships at CMU.

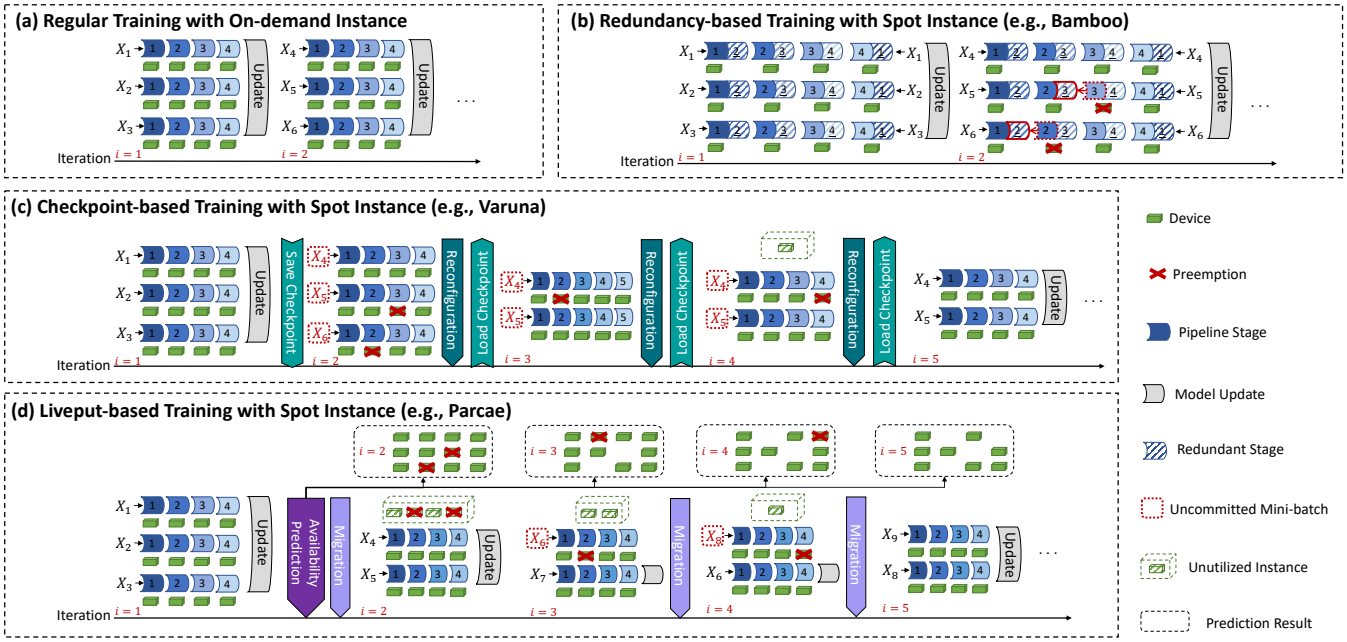


Figure 1: Illustration of pipelined data parallelism training over on-demand and spot instance respectively. Preempted spot instances are marked with red markers. X_j represents the j-th mini-batch of input data.

point, resulting in wasted computation as model updates made since the last checkpoint are lost.

The second line of work uses redundant computation to provide resilience in the presence of preemptions. For example, as shown in Figure 1b, Bamboo [47] replicates DNN computations across spot instances by letting each instance in a pipeline perform normal computations over assigned DNN layers (dark boxes) and redundant computations over its successor's layers (striped boxes). Upon an instance preemption, its predecessor has all the information (e.g., layers and activations) to continue DNN training. Although Bamboo achieves higher training throughput than pure checkpointing-based methods when preemption rates are high, its computation efficiency can still be limited. This is because it is difficult to completely hide the overhead of redundant computation through pipeline bubbles, especially for large-scale models (§10.2). Additionally, storing redundant model states increases per-GPU memory consumption. Existing redundancy-based methods such as Bamboo address this challenge by increasing pipeline depth, but this can lead to reduced computation efficiency and increased vulnerability to preemptions.

To address the performance and scalability limitations of existing approaches, this paper presents Parcae, a *proactive*, *liveput-optimized* system for DNN training on spot instances. Parcae combines data and pipeline parallelism for DNN training on spot instances, and maintains identical semantics as on-demand training. A key insight behind Parcae is that different strategies to parallelize DNN training exhibit *diverse robustness* under preemptions. For example, a strategy with long pipelines achieves higher throughput but is more vulnerable to preemptions than a strategy with shorter pipelines.

Parcae is designed to maximize *preemption-aware* throughput in a proactive way. We purpose a formulation of *liveput* for DNN training on preemptible instances, which is the *expected* training throughput of a DNN job under different preemption scenarios. A key advantage of liveput is that it considers both the throughput of a parallel configuration *and its robustness* under preemptions. Figure 1d illustrates how Parcae optimizes liveput. After observing two preemptions ($i = 2$ in the figure), Parcae anticipates that the cloud has reached its capacity limit and expects additional preemptions in the near future. Therefore, instead of maintaining two pipelines each with five instances, which maximizes throughput, Parcae keeps four instances on each pipeline, which is more robust under additional preemptions and maximizes liveput. This allows Parcae to cheaply handle future preemptions using lightweight live migrations ($i = 3, 4$ in the figure).

There are three key challenges Parcae must address to optimize liveput: (1) predicting liveput, (2) handling preemptions, and (3) discovering parallel configurations to maximize liveput. We elaborate these challenges and the main ideas Parcae uses to overcome them.

First, spot instances can be preempted and reallocated due to many reasons (e.g., market price changes, resource constraints) at any time. It is challenging to know ahead of time when and which specific instances will be preempted/allocated by the cloud provider; nor does the cloud provider provides any hints or auxiliary information on how

instance preemption and addition decisions are made. However, estimating the liveput of a parallel configuration requires considering a variety of preemption scenarios.

Instead of predicting preemptions and allocations for individual instances, Parcae uses a two-level approach to forecasting the availability of instances at a coarse granularity. First, the *availability predictor* takes the instance preemption and allocation history as input and only predicts the *number of available instances* in the near future. Second, the *Monte Carlo preemption sampler* uses the predicted instance availability to sample preemptions. This two-level approach allows Parcae to employ a lightweight predictor to forecast spot-instance availability and quickly estimate the liveput of different parallel configurations.

Second, existing checkpoint- and redundancy-based approaches to handling preemptions introduce significant memory and computation overheads. Checkpoint-based systems (e.g., Varuna [8]) omit all model updates since the last checkpoint after each preemption, and periodically saving and loading checkpoints introduce additional overheads. These overheads are substantial even by adopting fine-grained checkpointing mechanisms [32] for better overlapping (see §10.2). Meanwhile, redundancy-based systems (e.g., Bamboo [47]) require redundant computation on each instance even in the absence of preemptions, which decreases training throughput and increases monetary cost due to redundant computations.

To effectively handle preemptions, Parcae uses a lightweight live migration mechanism that allows DNN training to proceed despite losing instances and without introducing redundant computation as done by prior work. To achieve this goal, Parcae's live migration mechanism always uses the same number of samples to update model's parameters in each training iteration and opportunistically reorder samples to avoid redundant computation or restarting training. This approach preserves model accuracy by leveraging the stochastic nature of DNN training — all training samples are drawn independently from an intrinsic data distribution and *reordering samples does not affect model accuracy* [10].

Third, optimizing liveput requires reasoning about instance preemptions and allocations and quickly adapting to new resources allocations while minimizing transition cost. Recent work (e.g., PipeDream [34] and Alpa [55]) has proposed a variety of techniques to automatically discover throughput-optimized parallel configurations for DNN training. However, all these approaches assume a fixed set of GPUs and do not apply to spot-instance training.

To address this challenge, Parcae's *liveput optimizer* formulates the problem of maximizing liveput as an optimization task and uses a novel dynamic programming algorithm to explore the search space of parallel configurations that combine data and pipeline parallelism and discover an *optimal* parallel configuration in the search space.

The above techniques allow Parcae to significantly outperform prior work. Figure 2 compares Parcae against Bamboo and Varuna for training GPT-2 on 32 spot V100 GPU instances on AWS using a collected preemption trace. Parcae outperforms Bamboo and Varuna by 2.4× under the same preemptions. The grey curve shows an *ideal* case, where Parcae knows *all* future preemptions and allocations and maximizes liveput accordingly. Parcae achieves *89%* efficiency of the ideal case. We have evaluated Parcae on a variety of DNN models and preemption traces and shown that Parcae outperforms Varuna by up to 9.9× and Bamboo by up to 10.8×. Moreover, our evaluation shows that Bamboo and Varuna cannot scale to large models — for certain spot-instance traces, both of them cannot make *any* progress for training GPT-3 [12] with 6.7 billion parameters, while Parcae can achieve almost identical performance as its ideal case (i.e., knowing all future preemptions and allocations).

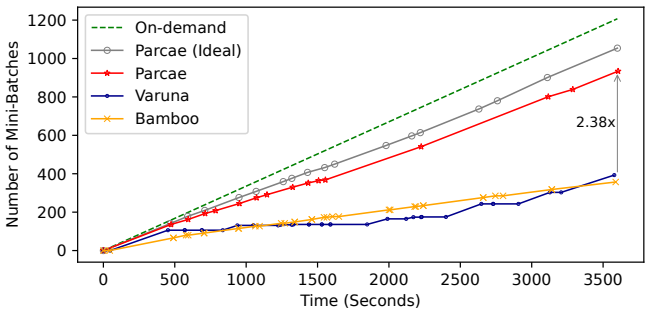


Figure 2: Comparing Parcae and prior work for training GPT-2 [38] on 32 spot GPU instances. Note that Parcae, Bamboo, and Varuna use an identical preemption trace.

This paper makes the following contributions:

- We propose liveput, a novel metric that simultaneously consider the performance and robustness of a parallelization strategy for DNN training on spot instances.
- We build Parcae, a liveput-optimized system for spot-instance training that accurately predicts instance availability, cheaply handles preemptions, and efficiently optimizes training performance under preemptions.
- We evaluate Parcae and show that it outperforms Varuna and Bamboo by up to 9.9× and 10.8×, and supports training large-scale models on spot instances.

2 Background

2.1 Distributed DNN Training

Data parallelism. Data parallelism [7, 36] is the most widely used parallelization strategy in distributed DNN training. Each GPU has a model replica and performs forward and backward computations for different batches of data samples independently. It requires to synchronize model gradients (e.g., All-Reduce [4]) before mode update.

Pipeline parallelism. Pipeline parallelism [19] partitions DNN model into different stages with data dependency. Each stage is trained on one GPU, and different GPUs communicate

activations and corresponding gradients, which are computed by forward and backward computation respectively, instead of parameter gradients. A mini-batch of training samples is split into multiple micro-batches in pipeline training and pipeline parallelism exploits the opportunity to parallelize the computations of different micro-batches.

Hybrid data and pipeline parallelism. Some studies [15, 33] combine data and pipeline parallelism to further accelerate the training of large models. Given a number of GPUs, the training throughput varies for different parallel configurations, which describes the number of stages and data-parallel pipelines it owns. Some recent systems (e.g., FlexFlow [48], Alpa [55], Galvatron [30]) further involve more complicated model parallelism to benefit distributed training of particular DNNs. However, they can not be applied on spot instance with dynamic device membership. Our approach considers hybrid data and pipeline parallelism, follows Varuna and Bamboo, and leaves the exploration of more fine-grained model parallelism as our future work.

2.2 Spot-Instance Training

Recent frameworks [6, 8, 47] exploit cheap but preemptible instances provided by clouds to train DNN models on. TorchElastic [6] focuses on elastic data parallelism training and cannot be adopted to large models, where pipeline parallelism is definitely needed. Since the availability of spot instances varies significantly and frequently, it is critical to decide the parallel configuration for a DNN model in response to preemptions and allocations. Bamboo [47] keeps the pipeline depth fixed and varies the number of pipelines according to the availability of spot instances. This mechanism makes it difficult for Bamboo to utilize spot instances, which have low availability, for large models that require a long pipeline. Varuna [8] introduces job morphing to dynamically change the parallel configuration and maximize throughput for a given number of spot instances. For instances with low preemption rate or models with negligible reconfiguration cost, switching to the optimal parallel configuration is definitely optimal. However, the current spot instance market and DNN models violate the two conditions, making it sub-optimal to always adopt the parallel configuration with the optimal throughput.

3 Liveput

This section introduces *liveput*, a new metric for DNN training that describes the *expected* training throughput of a parallelization strategy on spot instances by simultaneously considering its throughput and robustness under preemptions.

3.1 Definition of Liveput

To address the challenge mentioned above, we introduce liveput, a novel metric for distributed DNN training on spot instances that considers both the performance of a DNN system as well as potential preemptions.

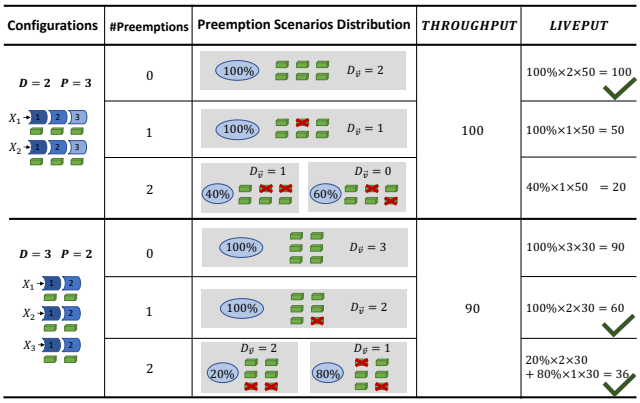


Figure 3: Comparing the liveput and throughput of different parallel configurations and preemption scenarios.

Definition 1 (Liveput)**.** *Let (D,P) denote the parallel configuration of a DNN training job, where P is the number of pipeline stages, and D is the number of data-parallel pipelines. The* ***liveput*** *of this training job is the expectation of its throughput under all possible preemption scenarios:*

$$\textsc{Liveput}(D,P,\mathcal{V}) = \mathop{\mathbb{E}}_{\vec{v}\sim\mathcal{V}}\left[\textsc{Throughput}(D_{\vec{v}},P_{\vec{v}})\right] \quad (1)$$

where $\mathcal{V}:\{0,1\}^{D\times P}\rightarrow[0,1]$ is the probability distribution of all preemption scenarios. Each $\vec{v}$ is an preemption indicator vector, $v_k=1$ if instance k will be preempted and $v_k=0$ otherwise. $\textsc{Throughput}(D_{\vec{v}},P_{\vec{v}})$ *is the throughput of the new parallel configuration $(D_{\vec{v}},P_{\vec{v}})$ after preemption $\vec{v}$.*

Note that we follow prior work [8, 47] and focus on data- and pipeline-parallel DNN training in this paper, while the liveput definition can easily generalize to other parallel configurations such as model [21] and reduction [48] parallelism.

3.2 Comparing Liveput and Throughput

A key advantage of liveput is that it considers how the performance of a parallel configuration changes under different preemption scenarios. Figure 3 demonstrates this advantage with a DNN training example on six spot instances with two possible parallel configurations: $\{D=2,P=3\}$ and $\{D=3,P=2\}$. For simplicity, we assume the throughput of a pipeline with three (or two) stages is 50 (or 30) samples/second and ignore the parameter synchronization cost. We compare the two parallel configurations under three preemption scenarios: (a) no preemption, (b) one preemption, and (c) two preemptions. We also assume that the preemption probabilities of all instances are the same.

Figure 3 compares the throughput and liveput of the two parallel configurations under the three preemption scenarios. Throughput is independent of instance preemptions; therefore, $\{D=2,P=3\}$ achieves a higher throughput than $\{D=3,P=2\}$ for all cases. On the other hand, liveput considers the amount of possible preemptions as well as the distri-

bution of these preemptions over spot instances. When there is no preemption (i.e., fixed resource allocation), liveput is equivalent to throughput. Once a concrete future preemption scenario is given as a prior condition, the corresponding liveput can be treated as the effective throughput after such a preemption. For example, under preemption of 1 or 2 instances, the configuration $\{D=3,P=2\}$ achieves higher effective throughput than $\{D=2,P=3\}$. Intuitively, due to data dependencies between the pipeline stages, longer pipelines are more vulnerable to preemptions, since a single preemption would invalidate an entire pipeline within a mini-batch, and shorter pipelines exhibit better elasticity and resilience under frequent preemptions. Existing throughput-optimized approaches fail to consider this trade-off when estimating training efficiency and may make suboptimal decisions.

4 Parcae Overview

Figure 4 shows an overview of Parcae, a liveput-optimized system for DNN training on spot instances. Computing liveput requires predicting instance preemptions and allocations. Since predicting instance-wise availability is infeasible (§5.1), Parcae uses a two-level approach to forecasting the availability of all instances at a coarse granularity, where an *availability predictor* takes the instance preemption/allocation history as input and only predicts the *number of available instances* in the future, and the Monte Carlo *preemption sampler* uses the predicted availability to sample preemptions and allocations.

Parcae's *liveput optimizer* takes the predicted instance availability as input and discovers a parallel configuration to maximize the liveput of the DNN model. The liveput optimizer formulates the problem of maximizing liveput as an optimization task and uses a dynamic programming algorithm to discover an optimal parallel configuration.

To migrate across different parallel configurations and handle potential preemptions, Parcae uses three *live migration* strategies. These migration strategies leverage statistical robustness of DNN training, allow Parcae to significantly reduce migration and preemption overheads compared to existing checkpoint- and redundancy-based systems.

For the rest of this paper, we introduce Parcae's availability predictor in §5, live migration strategies in §6, and liveput optimizer in §7. §8 describes how Parcae handles exceptional cases where actual preemptions mismatch Parcae's predictions. Finally, we discuss Parcae's design and implementation on modern clouds in §9 and evaluate its performance in §10.

5 Availability Predictor

5.1 Instance-wise Availability Unpredictability

There are several factors that affect spot-instance preemptions and allocations, including the types of the instances a user requires and their availability zones, the price of the current spot instance market, and competitions from other users. Most existing approaches to predicting the availability of spot instances focus on estimating their prices [16, 17], which cannot

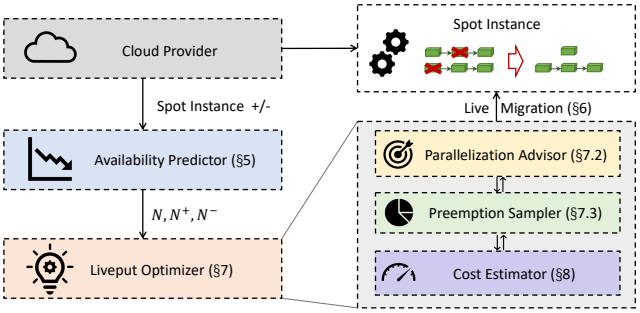


Figure 4: An overview of Parcae.

be used to estimate their lifetime. Prior work [16, 27, 31, 53] has also tried to predict the reliability of spot instances based on historical data collected from cloud providers. These attempts rely heavily on the cloud behaviour, which varies across cloud providers and availability zones within a cloud. Moreover, for a new cloud or zone, applying these data-driven approaches before running a job is expensive and time consuming. As a result, accurately forecasting individual instances' preemptions (i.e., $\vec{v}$ in Definition 1) is impractical since clouds currently do not support specifying preferences on the instance preemption order (i.e., which instances to preempt first), nor do they provide any auxiliary information that can help understand preemption and allocation decisions.

5.2 Statistical Availability Prediction

To make Parcae a general and practical DNN training system on spot instances, the only visible and reliable information is the past preemption/allocation records of the current user-submitted training job. Instead of forecasting when and which instance will be preempted in the future, Parcae uses a coarse-grained time-series forecasting approach. We observe that it is possible to predict the *total amount* of available spot instances for short time intervals in the future and benefit Parcae's proactive optimization performance.

Problem formulation. We split the timeline of a training job into equally sized intervals, where the length of an interval T is a hyper-parameter. For the i-th interval, we define a tupe (N_i, N_i^+, N_i^-) to represent the number of available instances, newly allocated instances, and preempted instances within the i-th interval, respectively. We assume that node preemptions and allocations only happen at the beginning of each time interval and that all available spot instances are stable within a time interval; this assumption is reasonable since each interval is small (e.g., 1 minute). Therefore, we have $N_i = N_{i-1} + N_i^+ - N_i^-$ $(i > 0)$. Instead of predicting N_i^+ and N_i^-, Parcae's availability predictor only forecasts a sequence of N_i (i.e., overall availability) and uses N_i to derive N_i^+ and N_i^-. This design is based on an important observation that a cloud does not preempt existing instances and allocate new instances at the same time, therefore $N_i^+ = \max(0, N_i - N_{i-1})$ and $N_i^- =$

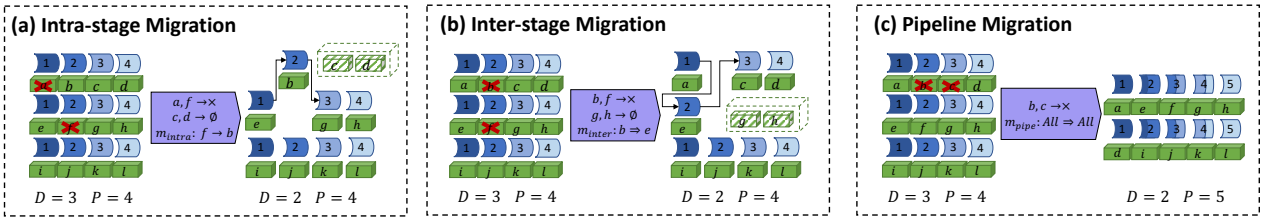
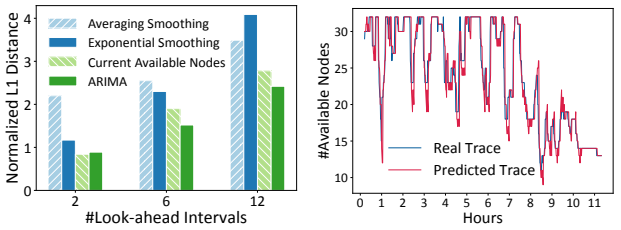


Figure 6: Illustrations of different migration strategies over a 3×4 parallel configuration facing 2 preempted instances.



(a) ARIMA vs. other models (b) ARIMA-predicted vs. real trace

Figure 5: (a) Comparison of normalized L1 distance of predictive performance for ARIMA and other solutions (H= 12, lower is better). (b) Comparison between ARIMA-predicted trace (H= 12, I= 4) and the ground truth.

$\max(0, N_{i-1} - N_i)$. Formally, in the time-series forecasting problem, an agent takes the instance availability trace in the past H intervals as input and forecasts the instance availability for the future I time intervals:

$$(N_i, \cdots, N_{i+I-1}) = \text{PREDICTION}(N_{i-H}, \cdots, N_{i-1}). \quad (2)$$

Note that $(N_i, \cdots, N_{i+I-1})$ can be used to derive the predicted instance preemptions and allocations for the next I intervals.

Limited input data prevents Parcae from using complex prediction models such as deep neural networks. Instead, we propose to leverage lightweight statistical algorithms (e.g., moving averaging, exponential smoothing, current available nodes) and empirically study their performance in Figure 5a (more details are in Appendix B). We select the auto-regressive integrated moving average (ARIMA) algorithm [11] as our availability predictor due to its superior performance. We observe that ARIMA can faithfully describe the tendency of instance availability, as shown in Figure 5b. Finally, our evaluation on collected real-world preemption/allocation traces further verifies that the ARIMA predictor can help Parcae achieve near-optimal liveput (§10.2).

6 Live Migration

This section describes the *proactive* migration mechanism of Parcae. Existing checkpoint- and redundancy-based approaches handle preemptions *reactively*, leading to significant overheads. Instead, we design several fine-grained *live migration* strategies to *proactively* handle different future preemption scenarios. Given the preemption prediction results, Parcae could schedule efficient adjustments in advance to adapt to the dynamic instance availability.

6.1 Pipeline-aware Preemption Mapping

Before introducing live migration, we first discuss the *preemption mapping* step in Parcae. Recall that the outputs of the availability predictor (§5) only include statistical information (i.e., the number of preemptions or allocations during a time interval). However, the impact of an instance preemption highly depends on the instance's position in the data- and pipeline-parallel topology. Therefore, instance-wise preemption predictions (i.e., $\vec{v}$ in Definition 1) is still necessary to make efficient live migration decisions.

To bridge this gap, Parcae uses a probabilistic model to reason about the mapping from preemption events to actual instances. This preemption mapping is essential for data- and pipeline-parallel training because of the unique dependencies between instances. In particular, instances in the same pipeline have sequential dependencies for both forward and backward computation, and instances in the same stage have synchronization dependencies for parameter synchronization. For each preemption event, Parcae assumes that all spot instances may be preempted with the same probability (see the example in Figure 3). Note that such an assumption can be replaced by more accurate estimations when additional preemption information is provided by cloud providers.

6.2 Migration Strategies

Parcae uses three migration strategies (Figure 6) to handle preemptions: *intra-stage*, *inter-stage*, and *pipeline* migration.

Intra-stage migration. In pipeline-parallel training, instances in the same stage maintain the same shard of model parameters. Therefore, when an instance is preempted, Parcae can opportunistically divert an available instance from the same stage in another broken pipeline. This intra-stage migration allows Parcae to re-establish a complete pipeline. As shown in Figure 6 (a), when instances a and f are preempted, Parcae can replace f by moving b to the second pipeline (e.g., $f \rightarrow b$), resulting in two complete pipelines. Intra-stage migration only requires updating the communication routing (e.g., $\rightarrow$) of a few instances and does not involve transferring parameters since b and the preempted instance f share the same model parameters and states.

Inter-stage migration. When intra-stage migration does not help recover broken pipelines, Parcae opportunistically performs inter-stage migrations, which moves intstances across stages. Figure 6 (b) shows an inter-stage migration,

where instances b and f are preempted, and Parcae moves e from the first stage to the second stage of the first pipeline (e.g., $b \Rightarrow e$), resulting in two complete pipelines. Inter-stage migration requires transferring model parameters (e.g., $\Rightarrow$) as the instances keep the model parameters and states of different stages. Both intra- and inter-stage migrations preserve pipeline depth and manage to recover as many data-parallel pipelines as possible.

Pipeline migration. Changing the pipeline depth is an important choice for maximizing training efficiency. Compared with the other two migration strategies, pipeline migration requires repartitioning the DNN model into a different number of pipeline stages, which involves significant migration overheads as instances need to broadcast their model parameters (e.g., $All \Rightarrow All$). Pipeline migration is similar to the reconfiguration mechanism in prior work (e.g., Varuna [8], Bamboo [47]) to handle instance preemptions.

Parcae makes migration decisions by considering the current parallel configuration, the new optimized parallel configuration and the actual preemptions. Given the probabilistic mapping of predicted preemptions, Parcae automatically renewals the optimal parallel configuration and the migration strategy (§7.2). Once the prediction mismatches with the actual availability, Parcae adjusts the parallel configuration as well as the corresponding migration strategies for adaptation (§8). The actual migration decisions are finalized when preemptions really happen, and Parcae leverages the grace period (e.g., 30s on Azure [2]) to perform these migrations.

7 Liveput Optimizer

This section describes Parcae's *liveput optimizer*, which determines the parallel configurations of training a DNN model on spot instances to maximize its liveput.

7.1 Problem Definition

We formulate liveput maximization as an optimization problem, where the objective is to discover a sequence of parallel configurations to maximize the committed training samples in expectation of spot instance availability. The sequence length is set to be consistent with the number of time intervals predicted by the availability predictor (Section 5). Formally, the objective function Φ is the accumulated number of committed training samples during the I time intervals:

$$\Phi(\mathbf{D},\mathbf{P} \mid \mathbf{N}) = \sum_{i=0}^{I-1} \phi(D_i,P_i,N_i \mid D_{i+1},P_{i+1},N_{i+1}), \quad (3)$$

where N_i is the predicted number of available instances (see Section 5) at the i-th time interval. Recall that Parcae derives N^-_{i+1} (i.e., the number of instances to be preempted) and N^+_{i+1} (i.e., the number of instances to be launched) from N_i and N_{i+1}. In addition, the preemption distribution $\vec{v}_{i+1}$ (Definition 1) is generated from N_i and N^-_{i+1} using the probabilistic preemption model developed in Section 6.1. Finally ϕ calculates the number of committed samples within a interval:

$$\begin{aligned} &\phi(D_i,P_i,N_i \mid D_{i+1},P_{i+1},N_{i+1}) \quad (4)\\ &\quad = \mathop{\mathbb{E}}_{\vec{v}_{i+1}} [\textsc{Liveput}(D_{i+1},P_{i+1} \mid \vec{v}_{i+1}) \times T_{\text{eff}}],\\ &T_{\text{eff}} = T - T_{\text{mig}}(D_i,P_i,D_{i+1},P_{i+1} \mid \vec{v}_{i+1}), \end{aligned}$$

where T and T_{eff} are the length of the time interval and effective training time after migrations, respectively, and T_{mig} is the migration overhead. Note that ϕ extends liveput by making the preemption distribution $\vec{v}_{i+1}$ a prior. With these definitions, the objective of the liveput optimizer is:

$$\arg\max_{\mathbf{D},\mathbf{P}} \Phi(\mathbf{D},\mathbf{P} \mid \mathbf{N}) \quad (5)$$

where $\mathbf{N} = \{N_1,N_2,\cdots,N_I\}$ is the output of the availability predictor, and Parcae discovers a sequence of parallel configurations $(\mathbf{D},\mathbf{P})$ to maximize liveput.

7.2 Parallelization Advisor

Parcae uses a dynamic programming algorithm to explore the optimization space and discovers an *optimal* sequence of parallel configurations. Specifically, let $F(i+1,D_{i+1},P_{i+1})$ represent the maximal number of committed training samples at the end of the i-th time interval, which uses parallel configuration (D_{i+1},P_{i+1}). We start from $F(0,D_0,P_0) = 0$ and have the following optimal substrates:

$$\begin{aligned} &F(i+1,D_{i+1},P_{i+1}) \quad (6)\\ &= \max_{\forall D_i \times P_i \le N_i} \left\{ \begin{array}{c} F(i,D_i,P_i) + \\ \phi(D_i,P_i,N_i \mid D_{i+1},P_{i+1},N_{i+1}) \end{array} \right\}, \end{aligned}$$

and figure out the final target as $\max_{\forall D_I \times P_I \le N_I} \{F(I,D_I,P_I)\}$.

The DP algorithm considers all possible parallel configurations that satisfy resource constraints (i.e., $D_i \times P_i \le N_i$), and $\phi(D_i,P_i,N_i \mid D_{i+1},P_{i+1},N_{i+1})$ is the product of two terms in Equation (4). Here the exploration adapts a similar search space as Varuna with a size of $O(N \log N)$, which is large enough for most recent large DNNs consisting of a stack of homogeneous layers. It is also possible to extend to a larger search space (e.g., Alpa) for more complicated workloads. The first term $\textsc{Liveput}$ can be replaced by $\textsc{Throughput}(D_{i+1},P_{i+1})$, where (D_{i+1},P_{i+1}) is the new parallel configuration after live migration. Note that (D_{i+1},P_{i+1}) should be a feasible model partition that satisfies the device memory capacity. For unfeasible cases that violate memory constraints, their $\textsc{Throughput}$ is set to be zero.

The second term T_{eff} depends on the preemption distribution, (D_i,P_i), (D_{i+1},P_{i+1}), and the migration strategy to transit from (D_i,P_i) to (D_{i+1},P_{i+1}). Given a pair of parallel configurations (D_i,P_i) and (D_{i+1},P_{i+1}), there may exist multiple migration strategies with different overheads T_{mig}, and the cost of each migration strategy also depends on the DNN workload. Parcae uses a *cost estimator* (Section 9.4) to estimate T_{mig} for different migration strategies. If the pipeline depth changes (i.e., $P_{i+1} \neq P_i$), Parcae infers that pipeline mi-

gration is performed. Otherwise, T_{mig} should be attributed to either inter- or intra-stage migrations. When both of them are applicable, Parcae selects the one with lower migration cost. In the absence of preemptions (i.e., $N_{i+1} = N_i$), there can be no migration cost if (D_{i+1}, P_{i+1}) equals to (D_i, P_i).

7.3 Preemption Mapping Sampler

As introduced in Section 6.1, preemption mapping is necessary to reason about live migration, since preemptions at different positions in the data- and pipeline-parallel topology require different migration strategies. Given N_i spot instances, among which N_{i+1}^- are to be preempted, the number of possible preemption mappings on a $D \times P$ topology grows exponentially in N_{i+1}^-. The large preemption mapping space makes it infeasible to explicitly consider all preemption scenarios or analyze the exact solutions mathematically.

To address this issue, Parcae uses sampling techniques to explore the mapping space and quickly discovers reasonable accurate approximations. Specifically, Parcae applies Monte Carlo (MC) sampling over the large space of all preemption scenarios and randomly samples $\vec{v}$ while preserving $N_{i+1}^- = \sum_{j=1}^{N_i} v_j$. For each sampled $\vec{v}$, Parcae identifies the corresponding migration costs. Parcae ensembles multiple trails of sampling to approximate the expectation in Equation (4). Note that this sampling step can be done offline in advance, therefore it does not block the dynamic programming optimization procedure. This allows parallelization advisor to quickly compute new parallel configurations and migration strategies during spot-instance training.

8 Exception Handling

This section describes how Parcae handles exceptional cases where actual spot-instance preemptions mismatch Parcae's predictions or the suggested parallel configuration is not compatible with the available spot instances.

Parallelization adaptation. Compared to prior work, Parcae proactively adjusts parallel configurations by predicting instances' availability and planning live migrations ahead. However, if actual preemptions rarely differ from predictions, the liveput optimizer may not work on available spot instances. To address this issue, Parcae includes a *configuration adaptation* step to adjust the target parallel configuration before live migration. Specifically, when the number of actual available spot instances is greater (or less) than the predicted N_i, Parcae adds (or drops) data-parallel pipelines while preserving the pipeline depth. When available spot instances cannot even formulate a single pipeline, it will try to re-partition the pipeline into fewer stages. This adaptation ensures a feasible configuration without significant migration overheads, performing at least as well as existing throughput-optimized approaches that reactively handle preemptions when predictions go wrong.

Fault tolerance. Even if the predictions align well with actual preemptions, there still exist rare cases where the migration strategies do not work. For example, if all instances in one stage are preempted, both inter- and intra-stage migration cannot recover this stage's status. Parcae uses a cheap, in-memory checkpointing mechanism (§9.3) to handle these cases. In addition, for the extreme cases where the number of available instances is less than the minimum feasible pipeline depth P, the training process has to be suspended until new spot instances are available.

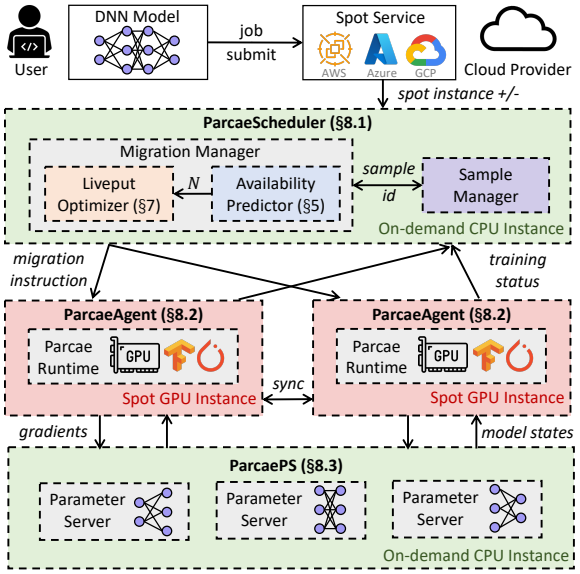


Figure 7: Overview of Parcae's design and implementation.

9 Parcae's Design and Implementation

Parcae consists of three main components as illustrated in Figure 7. First, ParcaeScheduler (§9.2) runs persistently on one on-demand CPU instance, determining the migration schedule based on our liveput optimizer and availability predictor. It also manages the training data samples to maintain the training semantics. Second, each spot GPU instance runs a ParcaeAgent (§9.2), which performs assigned training workload, monitors training progress, and executes the migration strategies issued by the ParcaeScheduler. Third, ParcaePS (§9.3) runs on several on-demand CPU instances to keep model checkpoints for rare rollback cases.

Parcae's implementation consists of $\sim 8K$ LoC in Python and takes PyTorch [25] as the default runtime. Communications between ParcaeScheduler and ParcaeAgent use `etcd` [5], a distributed key-value store. We implement live migration strategies by modifying DeepSpeed [40]. We show the workflow of ParcaeScheduler and ParcaeAgent in Algorithm 1 and introduce the detailed components as follows.

9.1 ParcaeScheduler

ParcaeScheduler has two major components: a *migration manager* and a *sample manager*. The former is responsible for parallelization and live migration, and the latter handles the data samples distribution.

Migration manager. As shown in Algorithm 1, the migration manager keeps receiving instance availability information (i.e., preemption or allocation interruptions) from the cloud provider and updating the current number of available instances (line 3). As discussed in §8, the parallel configuration (D_i, P_i) computed in the previous iteration using predicted availability may be incompatible with the current instances' availability. To handle this exception, ParcaeScheduler first adjusts the target parallel configuration (line 4) and then generates the required migration strategy S_i based on the current and target configurations (D_{i-1}, P_{i-1}) and (D_i, P_i). Note that the adaptation step (line 4) is performed before generating the migration strategy (line 5) so it will not involve re-adjustment overheads. Next, the availability predictor will forecast the number of available instances for a series of future intervals (i.e., $\mathbf{N}_{i+1}, \cdots, \mathbf{N}_{i+I}$) based on the historical information (line 7). Finally, the liveput optimizer makes parallelization suggestions for the following time interval using the prediction (line 8). The workflow continues until the training job is completed.

The handling of instance preemption and allocation interruptions are slightly different. Allocations are *controllable* as they only occur after we consciously send requests to the cloud, although they may not always succeed. We let a new instance join after its ParcaeAgent is successfully initialized. In contrast, preemptions are *passive* and may interrupt instances at any time, which requires additional mechanisms to handle various exceptions. Fortunately, the clouds usually provide a small grace period to inform the preemption before it happens. As the duration is usually enough to finish a mini-batch's training, we utilize the preemption notice to simplify the implementation and enforce instances to be preempted only at the mini-batches' boundaries. Parcae also handles rare failures that may interrupt training process, in which case ParcaeScheduler restarts training using the latest checkpoint in ParcaePS, avoiding losing model updates.

Sample manager. The training dataset is divided into mini-batches of fixed size and trained by DNNs iteratively. Each mini-batch of samples are "committed" after each iteration. However, preemptions may terminate training at any time, resulting in uncommitted mini-batches (Figure 1). To guarantee the same training semantics as on-demand instances, the sample manager tracks each data sample, records all uncommitted samples' indices, and makes them rejoin the training process later. This guarantees that all data samples are trained exactly once per epoch, preserving identical theoretical convergence property as the original data feeding order. We also provide a convergence experiment in Figure 16 to verify its training correctness.

9.2 ParcaeAgent

A ParcaeAgent runs on each spot GPU instance to interact with ParcaeScheduler as shown in Algorithm 1. It repeatedly receives a migration instruction from the ParcaeScheduler (line 13). If no migration is required, the ParcaeAgent requests a batch of training samples and starts model training (line 15-16). Otherwise, it performs the assigned migration instruction (line 14). ParcaeAgent manages to reuse the current model states to alleviate checkpoint overheads and rollbacks. For example, intra-stage migration is implemented by rebuilding communication groups and reusing previous model states on each GPU. For inter-stage and pipeline migration, additional costs are required for loading the latest model states from other instances via GPUs' peer-to-peer communications. Specially, if all instances of a stage are preempted, all the ParcaeAgents have to roll back to a previous checkpoint. In this way, ParcaeScheduler automatically generates the most efficient migration strategy and let the ParcaeAgents transit to the target parallel configuration. Note that, the ParcaeScheduler also notifies a ParcaeAgent if it will be preempted or stay idle (i.e., $N_i - D_i \times P_i$ instances will be idle) by sending a halt or termination instruction to the ParcaeAgent.

Algorithm 1 Workflow of Parcae components.

```
    ▷ ParcaeScheduler
 1: function MigrationManager(D_0, P_0)
 2:     for i in 1, 2, 3, ··· do
 3:         N_i ← Receive availability info from cloud provider
 4:         (D_i, P_i) ← AdjustParallelConfiguration(N_i)
 5:         S_i ← GetMigrationStrategy ((D_{i-1}, P_{i-1}), (D_i, P_i))
 6:         Send migration strategy S_i to all ParcaeAgents
 7:         N_{i+1}, ··· , N_{i+I} ← AvailPredictor(N_{i-H+1}, ..., N_i)
 8:         (D_{i+1}, P_{i+1}) ← LiveputOpt((D_i, P_i), N_i, ..., N_{i+I})
 9:         if job completes then
10:             break
    ▷ ParcaeAgent
11: function ParcaeRuntime(model, batch_size)
12:     while job does not complete do
13:         Receive migration instruction m from ParcaeScheduler
14:         Apply migration instruction m if m is not empty
15:         X, Y ← DataLoader(batch_size)
16:         Train(model, X, Y)
```

9.3 ParcaePS

Parcae needs checkpoints to handle rare cases as introduced in §8. Unlike prior checkpointing approaches relying on expensive cloud storage (e.g., S3 on AWS), Parcae employs several cheap on-demand CPU instances (e.g., `c5.4xlarge` instance, $0.68\$/hour$) to maintain the latest model states in their DRAM. Instead of directly communicating model states and weights as prior checkpointing approaches, the ParcaePS maintains an up-to-date checkpoint by iteratively synchronizing gradients with spot GPU instances to update the model states the on CPU side (e.g., parameters and optimizer states), which reduces communication by $5\times$ for stateful optimizers (e.g., Adam [23]) in the FP16 format [41]. Parcae also partitions gradients into small pieces for better overlapping and prevents bandwidth competition with cross-stage activation transfer.

Table 1: Overview of the four trace segments evaluated.

Trace	H_AD_P	H_AS_P	L_AD_P	L_AS_P
Availability	High	High	Low	Low
Preemption intensity	Dense	Sparse	Dense	Sparse
#avg instances	27.05	29.63	16.82	14.60
#preemption events	9	6	8	3
#allocation events	8	5	12	0
length	1h	1h	1h	1h

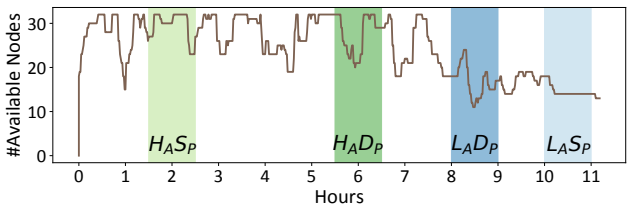


Figure 8: The complete trace and segments of four scenarios.

9.4 Cost Estimator

We develop a *cost estimator* to estimate migration cost by considering different preemption scenarios and parallel configurations. We conduct an empirical study to profile the migration cost and find that it varies across several factors (details in Appendix A). Some of these terms have relatively fixed overheads like CUDA context initialization (less than 10s). The communication group updating and model building costs are associated with the parallel configuration (less than 30s). The model transfer cost varies considerably according to the preemptions (up to 60s). We consider the instance network topology for each preemption scenario and adopt an $\alpha - \beta$ model [49] to accurately estimate the communication cost.

10 Evaluation

10.1 Experimental Setup

DNNs. We select five popular DNNs for various applications. ResNet-152 [18] and VGG-19 [45] are CV tasks, and we use CIFAR-100 [24] as the training dataset. BERT [14], GPT-2 [38], and GPT-3 [12] are popular model architectures for NLP tasks, and we evaluate them on WikiText-2 [28]. We use GPT-2 and GPT-3 including 1.5 and 6.7 billion model parameters respectively. More setting details are in Appendix C.

Traces. Due to the dynamic availability of spot instance, it is almost impossible to evaluate different systems on real spot instances multiple times and expect consistent dynamic environments. Instead, to make a fair comparison, we take the real spot instance availability traces and replay them on regular instances. Specifically, we collect a 12-hour trace on a 32-instance cluster with `p3.2xlarge` instances on AWS. Inspired by Bamboo [47], we extract representative segments from the whole trace for our evaluation. We design two new measurements for each segment, including the *availability* (i.e., the average number of instances) and the *preemption intensity* (i.e., the number of instance preemption and allocation events). Table 1 and Figure 8 show four extracted 1-hour trace segments based on different availability and preemption intensity. Traces with over 70% available instances are high availability (i.e., H_A) traces, otherwise have low availability (i.e., L_A). Dense preemption intensity traces (i.e., D_P) have around 20 instance preemption and allocation events, but sparse preemption intensity traces (i.e., S_P) only have few. We replay these four trace segments on 32 on-demand V100-16GB GPU instances to simulate spot-instance clusters.

Table 2: Comparison of monetary cost ($\times 1e^{-6}$USD) for different models and approaches. We report per-image cost for ResNet and VGG and per-token cost for BERT and GPT.

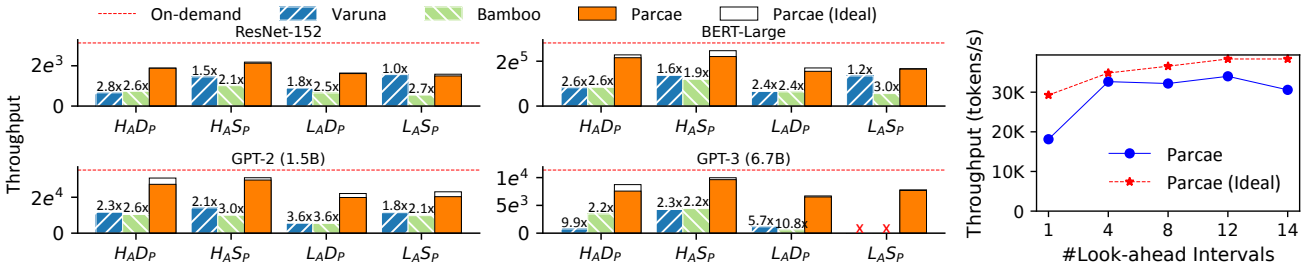
Model	Trace	On-Demand	Varuna	Bamboo	Parcae
ResNet	H_AD_P	8.68 (2.3×)	10.86 (2.8×)	9.77 (2.6×)	3.81 (1×)
	H_AS_P	8.68 (2.4×)	5.32 (1.5×)	7.61 (2.1×)	3.62 (1×)
	L_AD_P	8.68 (3.2×)	4.89 (1.8×)	6.72 (2.5×)	2.71 (1×)
	L_AS_P	8.68 (3.4×)	2.43 (1.0×)	6.96 (2.7×)	2.54 (1×)
VGG	H_AD_P	12.43 (2.7×)	12.10 (2.6×)	12.11 (2.6×)	4.62 (1×)
	H_AS_P	12.43 (2.7×)	6.52 (1.4×)	13.12 (2.8×)	4.66 (1×)
	L_AD_P	12.43 (3.4×)	5.43 (1.5×)	9.40 (2.6×)	3.66 (1×)
	L_AS_P	12.43 (4.0×)	3.37 (1.1×)	8.88 (2.9×)	3.11 (1×)
BERT	H_AD_P	0.10 (2.9×)	0.09 (2.6×)	0.09 (2.6×)	0.03 (1×)
	H_AS_P	0.10 (2.8×)	0.06 (1.6×)	0.06 (1.9×)	0.03 (1×)
	L_AD_P	0.10 (3.4×)	0.07 (2.4×)	0.07 (2.4×)	0.03 (1×)
	L_AS_P	0.10 (4.2×)	0.03 (1.2×)	0.07 (3.0×)	0.02 (1×)
GPT-2	H_AD_P	0.62 (2.9×)	0.49 (2.3×)	0.55 (2.6×)	0.21 (1×)
	H_AS_P	0.62 (3.0×)	0.44 (2.1×)	0.62 (3.0×)	0.21 (1×)
	L_AD_P	0.62 (3.5×)	0.63 (3.6×)	0.64 (3.6×)	0.18 (1×)
	L_AS_P	0.62 (4.1×)	0.27 (1.8×)	0.31 (2.1×)	0.15 (1×)
GPT-3	H_AD_P	2.39 (2.5×)	9.35 (9.9×)	2.07 (2.2×)	0.94 (1×)
	H_AS_P	2.39 (3.0×)	1.81 (2.3×)	1.74 (2.2×)	0.80 (1×)
	L_AD_P	2.39 (3.6×)	3.81 (5.7×)	7.28 (10.8×)	0.67 (1×)
	L_AS_P	2.39 (4.8×)	-	-	0.49 (1×)

10.2 End-to-End Evaluation

We first compare the end-to-end training performance between Parcae and existing SOTA spot-instance training systems including Bamboo [47] (redundancy-based) and Varuna [8] (checkpoint-based). We also compare with on-demand instances training approach. The results are displayed in Figure 9a and Table 2. In all experiments, Parcae looks ahead 12 intervals based on the availability predictor, while Parcae (Ideal) looks ahead 12 intervals based on truth traces.

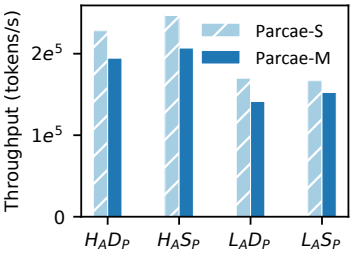
Parcae significantly outperforms both Bamboo and Varuna in terms of throughput for almost all the models and preemption traces. On average, Parcae delivers an overall of 2.59× higher throughput than Varuna and 3.0× than Bamboo. Apparently, Parcae is much more economical than Varuna and Bamboo as it completes more samples with the same monetary costs. Compared with on-demand instances, Parcae is 3.24× cheaper, and Parcae (Ideal) even achieves competitive throughput, e.g., only 14.2% lower for GPT-2 on high availability traces. The results also show that the performance of Parcae is quite close (i.e., up to 13.3%) to Parcae (ideal).

The performance improvement mainly comes from two aspects. First, Parcae's liveput optimized configurations balance the trade-off between throughput and available duration, in-



(a) Parcae throughput. (b) The effect of I in predictor.

Figure 9: (a) Training throughput comparison among existing frameworks and Parcae on four traces. The dotted on-demand line shows the best throughput with on-demand instances. The numbers over the bars represent the speedup of Parcae over Varuna and Bamboo respectively. (b) The GPT-2 training throughput for the H_AD_P trace with different look-ahead intervals.



Trace	Monetary Cost Parcae-S	Parcae-M
H_AD_P	**3.13**	3.78
H_AS_P	**3.12**	3.71
L_AD_P	**2.57**	3.28
L_AS_P	**2.27**	3.00

Figure 10: The comparison of throughput and monetary cost ($\times 1e^{-8}$USD/token) of BERT for Parcae on single-GPU instances (Parcae-S) and multi-GPU instances (Parcae-M).

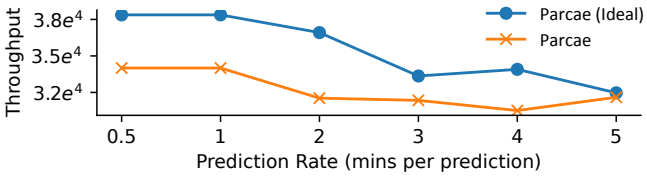


Figure 11: GPT-2 training throughput (tokens/s) using the H_AD_P trace with different prediction rates.

stead of greedily doing expensive reconfiguration like Varuna. While Bamboo maintains a fixed long pipeline depth (e.g., 16 for GPT-2), leading to many unutilized instances, especially for low availability traces. Second, the migration mechanism in Parcae is highly efficient to handle preemptions. Varuna is designed for low preemption environments and relies on shared storage (e.g. S3) to save and load checkpoints. Although Varuna overlaps checkpoint saving with training iterations, when preemptions happen, it requires rolling back to the last checkpoint and loses tens of seconds' (i.e., the duration of one complete checkpointing) training progress for large models. To recover from preemptions, Varuna needs to load the last checkpoint from persistent storage and restart training, which is also expensive. Bamboo is designed for high preemption environments based on redundant computation. It can efficiently handle preemptions, but the redundant computation is inefficient and brings additional synchronization overheads between redundant and normal modules.

Multi-GPU instances. To demonstrate the generality of Parcae, we also evaluate Parcae on multi-GPU instances. Unfortunately, we fail to collect meaningful multi-GPU spot instance traces on the cloud (e.g. `p3.8xlarge` with 4 V100 GPUs) as they show extremely low availability recently. Instead, we propose to generate the 4-GPU instance based on the single GPU trace by accumulating every four preemption or allocation events. Each 4-GPU instance is allocated at the first allocation event and preempted at the last preemption event. In this way, multi-GPU instance trace will have higher GPU hours than the single GPU trace in total. For multi-GPU instances, we follow prior work [39, 43] using pipeline parallelism only for inter-nodes. Figure 10 shows the training throughput and cost for different trace segments. Although our trace generation favors multi-GPU instances in theoretical availability, Parcae on single GPU instance still performs better in terms of both throughput and monetary cost. The major reason is that preempting one 4-GPU instance will interrupt 4 pipelines, significantly slowing down training. Besides, unutilized 4-GPU instances are also significant as it takes four times more GPUs to increase a new pipeline.

10.3 Breakdown Analysis

Look-ahead interval length. Figure 9b shows the results of training GPT-2 with different numbers of look-ahead intervals on the H_AD_P trace. Here Parcae looks back past 12 intervals and predicts the next $1, 4, 8, 12$, and 14 intervals respectively. The results show that Parcae (Ideal) keeps improving by considering longer futures and achieves the best performance when looking ahead 12 intervals. It shows the benefits of liveput-optimized configurations by considering future preemptions and allocations. On the other hand, Parcae exhibits a slightly different pattern, where its performance improves significantly by looking ahead 4 intervals compared with 1 interval ($1.8\times$). As Parcae looks ahead more intervals, the prediction error increases as we evaluated in Figure 5a. Figure 9b shows that Parcae can still yield significant improvement compared with looking ahead 1 interval, and achieves best performance by looking ahead 12 intervals. Overall, Parcae's throughput is 12.8% lower than that of the ideal case.

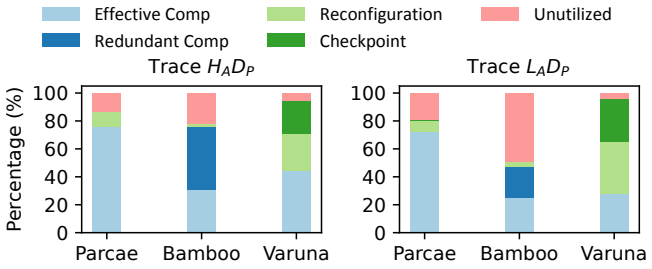


Figure 12: GPU hours breakdown of GPT-2 execution.

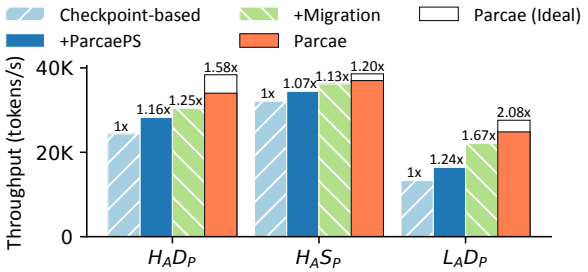


Figure 13: The decomposed throughput speedup on GPT-2.

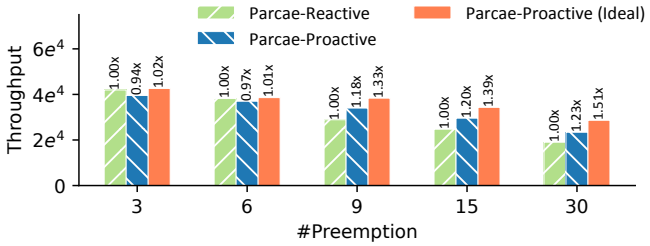


Figure 14: The throughput comparison between Parcae and Parcae-Reactive under different preemption intensity.

The result demonstrates that looking ahead longer can indeed help Parcae make more optimized decisions, and that there is still room to improve our availability predictor.

Prediction rate. Figure 11 shows the results of training GPT-2 with different prediction rates on the H_AD_P trace. As the prediction rate decreases, so do the training throughput achieved by Parcae and Parcae (Ideal). Fortunately, the execution time of the liveput optimizer is much less than one minute, which allows Parcae to use a high prediction rate and optimize frequently (i.e., per minute) for better performance.

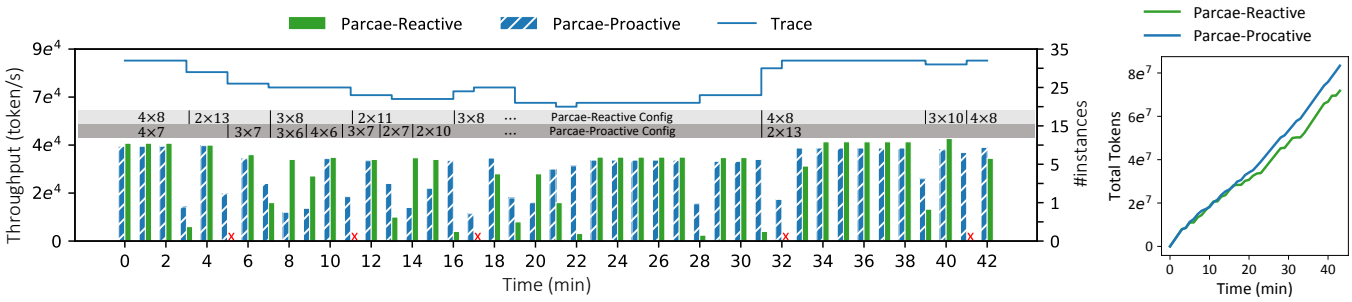
GPU hours breakdown. To further understand the performance and drawbacks of different approaches, we breakdown the GPU hours of GPT-2 training into five components (Figure 12). The results demonstrate that Parcae spends the majority of GPU hours performing effective computation (i.e., committed mini-batches). In contrast, Bamboo spends more than 40% GPU hours on redundant computation on H_AD_P, while wastes more than 50% GPU hours on L_AD_P. Similarly, Varuna takes a long time to handle preemptions, including checkpointing and reconfiguration. As a result, their unutilized parts are quite small compared with Parcae. The results also align with the disadvantages we mentioned in §10.2.

Parcae components analysis. Figure 13 shows how each component contributes to the performance improvements, using GPT-2 as an example. We start from a checkpoint-based approach with throughput-optimized execution plans. By adding ParcaePS and migration strategies, we improve the throughput by 13%-67%. Especially for trace L_AD_P with low availability, it leaves little room for parallel configuration variation. When there are frequent preemption and allocation events, the migration allows training to make more progress than frequently triggering the costly reconfiguration. Finally, adopting liveput optimized parallel configurations improves an additional 25.5% over migration mechanisms.

10.4 Proactive v.s. Reactive

Preemption Tolerance. We evaluate the performance of Parcae (i.e., Parcae-Proactive) and Parcae-Reactive with GPT-2 on a synthetic preemption trace Figure 14. The auxiliary baseline (i.e., Parcae-Reactive) is created by disabling the liveput optimization in Parcae and only enabling the parallelization adaptation mechanism (§8). Parcae-Reactive can be classified as a throughput-optimized system and used to highlight the advantages of our proactive, liveput-based approach. We generate the synthetic trace from the H_AS_P trace by scaling the number of preemption events from 3 to 30 within one hour. The performance gap between Parcae-Reactive and Parcae-Proactive becomes larger as the preemption intensity increases, showing that our proactive approach can be more effective for scenarios with more frequent preemptions.

Case study. As a case study, we compare the liveput-optimized Parcae with throughput-optimized Parcae-Reactive in detail using GPT-2 and partial H_AD_P trace. Figure 15a shows each interval's instance availability, parallel configuration ($D \times P$), and average throughput as time elapses. We observe that for intervals with stable availability, Parcae-Reactive can select configurations with relatively higher throughput. However, greedily selecting execution plans that maximize throughputs suffers when preemptions or allocations happen because it neglects high reconfiguration costs. It can barely make training progress when the available instances frequently change. In contrast, Parcae carefully chooses parallel configurations by considering the future instance availability and adapting efficient migration strategies accordingly to ensure high training efficiency while mitigating expensive reconfiguration. For example, in the first 8 intervals, Parcae selects a pipeline depth of 7 and avoids changing pipeline depth as Parcae-Reactive does (e.g., 8 and 13). Although resulting in some unused instances, the progress made is still larger than running with Parcae-Reactive because of its reconfiguration overheads. Similar observation exists in the last 10 consecutive intervals, where Parcae maintains the same parallel configurations but leverages lightweight inter- and intra-stage migrations to adapt to dynamic preemptions



(a) Parallel configuration ($D \times P$) and average throughput inside each interval (i.e., 1 minute). (b) Accumulated tokens.

Figure 15: The comparison between Parcae-Reactive and Parcae-Proactive approaches for GPT-2 on $H_A D_P$ trace.

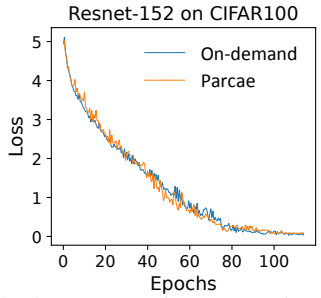


Figure 16: The loss curve of ResNet-152 on CIFAR100.

and allocations. As a result, Parcae achieves 16% more accumulated tokens within 40 minutes (Figure 15b).

10.5 Convergence Preservation

Figure 16 visualizes the convergence curve of training loss between Parcae running on spot instances and the baseline on on-demand instances. We observe that both convergence rates are very close, and Parcae reaches the same training loss of 0.058 as the baseline after training for 110 epochs. This verifies that the Parcae design and implementation align with the model convergence.

11 Related Work

Dynamic preemptible instances. There is a trend of using preemptible instances on modern clouds for cheap service. Tributary [16] studies the latency issues from preemptions and proposes to switching preemptible offerings from clouds with different preemption likelihoods. BurScale [9] employs autoscaling to handle transient queuing in web service traffic. SciSpot [22] presents a reliability model for temporally constrained preemptions to optimize the job scheduling for scientific computing. HotSpot [42] transparently migrates spot VMs in lower price and achieves higher cost-efficiency. Snape [52] improves spot resources' availability by dynamically mixing on-demand VMs with spot eviction predictions. SpotServe [29] realizes fast and reliable serving of LLMs on cheap preemptible instances with dynamic reparallelization and optimal context migration. Prior work has demonstrated the cost benefits of spot instances in cloud computing and motivates the following related research.

Preemptible distributed DNN training. Recently, using preempible instances for machine learning tasks is becoming popular as they are much more cost effective, like what is done in Varuna [8] and Bamboo [47]. SageMaker in AWS [3] automatically pauses the training job when a spot instance is interrupted and resumes from the checkpoint in S3 if the spot instance becomes available again. CM-DARE [26] analyzes distributed training under transient cloud GPU servers and provides a performance modeling methodology. SpotTune [27] leverages spot instances to paralleize hyper-parameter tuning for ML models. SkyPilot [54] migrates training workload to spot resources from other clouds and relaunch jobs using the periodical checkpoint from cloud storage. These approaches make meaningful explorations in this direction but are still suffering from the limited performance due to preemptions.

Oobleck [20] and Gemini [50] are concurrent works for quick failure recovery in distributed DNN training. Oobleck introduces pipeline reinstantiation with precomputed pipeline templates. Gemini uses in-memory checkpoints and orchestras checkpoint traffic schedule. Both are reactive approaches. Besides, Gemini targets dedicated instances and relies on high network bandwidth to reduce checkpointing time, while the bandwidth is low for spot instances.

12 Conclusion

In this work, we present an efficient distributed training system over spot instances, Parcae. The key idea is to proactively adjust the parallelization strategy using a novel metric, liveput, considering both training throughput and instance availability. With holistic system mechanisms and implementation optimizations, Parcae significantly outperforms checkpoint- and redundancy-based solutions in evaluations.

Acknowledgement

We thank the anonymous reviewers and our shepherd, Le Xu, for their comments and helpful feedback. This material is based upon work supported by NSF awards CNS-2147909, CNS-2211882, and CNS-2239351, and awards from Amazon, Cisco, Google, Meta, Oracle, Qualcomm, and Samsung.

References

[1] Amazon ec2 spot instances. https://aws.amazon.com/ec2/spot/.

[2] Use azure spot virtual machines. https://learn.microsoft.com/en-us/azure/virtual-machines/spot-vms.

[3] Amazon sagemaker spot training. https://docs.aws.amazon.com/sagemaker/latest/dg/model-managed-spot-training.html, 2018.

[4] Nvidia nccl. https://developer.nvidia.com/nccl, 2021.

[5] Operating etcd clusters for kubernetes. https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/, 2021.

[6] Pytorch elastic. https://github.com/pytorch/elastic, 2021.

[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2016.

[8] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 472–487, 2022.

[9] Ataollah Fatahi Baarzi, Timothy Zhu, and Bhuvan Urgaonkar. Burscale: Using burstable instances for cost-effective autoscaling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 126–138, 2019.

[10] Léon Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer, 2012.

[11] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[12] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[13] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[15] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 431–445, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Aaron Harlap, Andrew Chung, Alexey Tumanov, Gregory R. Ganger, and Phillip B. Gibbons. Tributary: spot-dancing for elastic services with latency slos. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 1–14. USENIX Association, 2018.

[17] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R Ganger, and Phillip B Gibbons. Proteus: agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 589–604, 2017.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[19] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 103–112, 2019.

[20] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 382–395. ACM, 2023.

[21] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML'19, 2019.

[22] Jcs Kadupitiya, Vikram Jadhao, and Prateek Sharma. Scispot: Scientific computing on temporally constrained cloud preemptible vms. *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[23] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[24] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[25] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment*, 13(12).

[26] Shijian Li, Robert J Walls, and Tian Guo. Characterizing and modeling distributed training with transient cloud gpu servers. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 943–953. IEEE, 2020.

[27] Yan Li, Bo An, Junming Ma, Donggang Cao, Yasha Wang, and Hong Mei. Spottune: Leveraging transient resources for cost-efficient hyper-parameter tuning in the public cloud. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 45–55. IEEE, 2020.

[28] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[29] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. *Proceedings of ASPLOS Conference*, 2024.

[30] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *Proc. VLDB Endow.*, 16(3):470–479, 2023.

[31] Ashish Kumar Mishra, Brajesh Kumar Umrao, and Dharmendra K Yadav. A survey on optimal utilization of preemptible vm instances in cloud computing. *The Journal of Supercomputing*, 74(11):5980–6032, 2018.

[32] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. {CheckFreq}: Frequent,{Fine-Grained}{DNN} checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216, 2021.

[33] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[34] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel DNN training. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 7937–7947. PMLR, 2021.

[35] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty, Apurva Samudra, Prashasti Baid, James Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, and Chunqiang Tang. Ras: Continuously optimized region-wide datacenter resource allocation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 505–520, New York, NY, USA, 2021. Association for Computing Machinery.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.

[37] David A. Patterson, Joseph Gonzalez, Quoc V. Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *CoRR*, abs/2104.10350, 2021.

[38] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[39] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[40] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[41] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. {ZeRO-Offload}: Democratizing {Billion-Scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[42] Supreeth Shastri and David E. Irwin. Hotspot: automated server hopping in cloud spot markets. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*, pages 493–505. ACM, 2017.

[43] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[44] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016.

[45] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[46] Yang Song, Jascha Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[47] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large dnns. *CoRR*, abs/2204.12013, 2022.

[48] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Patrick S. McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 267–284. USENIX Association, 2022.

[49] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[50] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. GEMINI: fast failure recovery in distributed training with in-memory checkpoints. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 364–381. ACM, 2023.

[51] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and

Yangqing Jia. Antman: Dynamic scaling on gpu clusters for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[52] Fangkai Yang, Lu Wang, Zhenyu Xu, Jue Zhang, Liqun Li, Bo Qiao, Camille Couturier, Chetan Bansal, Soumya Ram, Si Qin, et al. Snape: Reliable and low-cost computing with mixture of spot and on-demand vms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 631–643, 2023.

[53] Sheng Yang, Samir Khuller, Sunav Choudhary, Subrata Mitra, and Kanak Mahadik. Scheduling ml training on unreliable spot instances. In *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, pages 1–8, 2021.

[54] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. SkyPilot: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 437–455, Boston, MA, April 2023. USENIX Association.

[55] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 559–578. USENIX Association, 2022.

Table 3: Overview of the five DNNs evaluated.

Model	mini-batch	micro-batch	Dataset
ResNet-152 [18]	2048	32	CIFAR-100 [24]
VGG-19 [45]	2048	32	CIFAR-100 [24]
BERT-Large [14]	1024	8	WikiText-2 [28]
GPT-2 (1.5B) [38]	128	1	WikiText-2 [28]
GPT-3 (6.7B) [12]	64	1	WikiText-2 [28]

Table 4: Migration costs in our experiments on AWS.

Cost Terms	Magnitude (s)	Interfering Factor
Start process	< 1	Instance state
Rendezvous	$0 \sim 10$	
Init CUDA context	$0 \sim 10$	
Load data	$0 \sim 10$	Dataset
Build model	$0 \sim 10$	Model, Configuration
Update comm. groups	$0 \sim 20$	
Model states transfer	$0 \sim 60$	Model, Configuration Preemption Scenario

A Addition Details of Migration Costs

Table 4 lists detailed costs of migrations and their magnitudes. All of them are profiled multiple times and averaged over five DNN models (see Table 3).

B Additional Details of ARIMA

The ARIMA time-series forecasting algorithm is sensitive to trivial perturbations in inputs, which may impede its understanding of essential patterns from previous instance history. We introduce a few optimizations to ensure its predictions are faithful. First, we flatten random spikes that last for only 1-2 intervals in previous instance history, since such trivial noise will likely cause abrupt rise and falls in prediction. ARIMA also likes to simulate the tendency of the entire input curves. When input curves have multiple "hops", we ensure that ARIMA only learns from the most recent variations that are indeed beneficial for prediction. Second, though ARIMA can accurately capture intermediate fluctuations, its prediction can be so steep that it easily hits the upper and lower boundaries of available instances on intervals of sudden increase and decrease. To do so, we set upper and lower boundaries to limit the predicted curves based on observations of all spot instance traces we have. Additionally, our empirical study on traces indicate most intervals have a limitation on the extent of growth. Thus, we would also apply such constraints on predictions. We also apply additional penalty to flatten excessively steep predictions such as their predictions follow the essential patterns of AWS traces. We take care to reset ARIMA mispredictions when the generation deviates seriously from the input. With these rules and modifications, we ensure the ARIMA model can sufficiently describe future scenarios by learning from the past history.

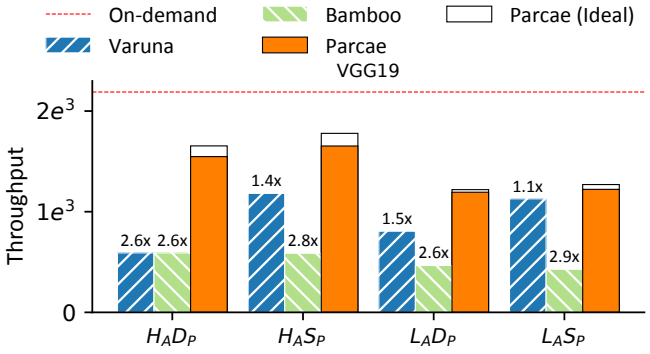


Figure 17: Training throughput comparison of VGG19 among existing frameworks and Parcae on four traces. The dotted on-demand line shows the best throughput with on-demand instances. The numbers over the bars represent the speedup of Parcae over Varuna and Bamboo respectively.

C Additional Experimental Details

C.1 End-to-End Evaluation Setting

We select five popular DNNs for various applications and summarize them in Table 3. For all the models, we used Adam optimizer with half precision (i.e., FP16) for training.

Parallel Configuration. Parcae and Varuna will adjust parallel configurations according to instance availability during training. The parallel configuration of Parcae is decided by migration manager, while it is decided by job morphing for Varuna. We follow the settings of Vauna and first run a one-time profiling to collect primitive parameters of the hardward and the DNN model. Varuna will automatically decide the optimal parallel configuration considering DNN models and number of availability instances. Table 5 summarizes the parallel configurations used for Bamboo in our evaluation. Bamboo maintains a fixed pipeline depth and its redundant computation consumes a huge amount of memory. For different models, we tuned the number of pipeline stages and partitions to find an optimal parallel configuration for Bamboo. We find it requires at least 20 stages for Bamboo to run GPT-3 even with activation checkpointing [13] enabled, and Bamboo performs best for $P = 23$.

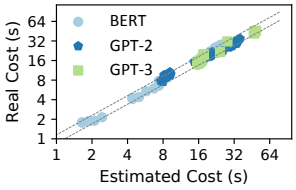
VGG Results Figure 17 shows the end-to-end evaluation results of VGG19. Parcae significantly outperforms Varuna and Bamboo, except for trace L_AS_P, where Varuna achieves comparable performance with Parcae. We move these results in the appendix due to the limited page space.

C.2 Parcae Components Evaluation

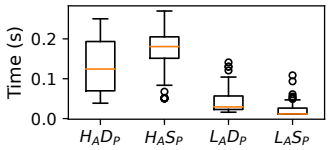
Cost Estimation Accuracy. The cost estimator estimates migration cost for different preemption scenarios and parallel configurations. An accurate estimator is important for accurate liveput optimization. We compare the estimated migration cost predicted by cost estimator with the real migration time measured by actual executions. Figure 18a shows the results

Table 5: The parallel configuration of Bamboo in evaluation.

Model	D	P
ResNet-152 [18]	8	4
VGG-19 [45]	8	4
BERT-Large [14]	4	8
GPT-2 (1.5B) [38]	2	16
GPT-3 (6.7B) [12]	1	23



(a) The accuracy of cost estimator.



(b) The cost of migration advisor.

Figure 18: (a) Comparison between the estimated and actual reconfiguration time for different models. (b) Optimization time of looking ahead 12 intervals for GPT-2.

for different DNN models. The dashed lines indicate a relative difference of -15% and 15% between real and estimated migration cost, respectively. The results demonstrate that our cost estimator is appropriate to evaluate the migration cost for different preemption scenarios and models.

Optimization Cost. ParcaeScheduler periodically runs online liveput optimization to suggest the parallel configuration for the next interval. We evaluate the optimization time it takes to look ahead 12 intervals for one run on one CPU machine. Figure 18b shows the results of GPT-2 on different trace segments. Overall, one optimization takes less than 0.3 seconds, which is negligible compared with interval length. Therefore, the liveput optimization will not delay the training process.

Accelerating Neural Recommendation Training with Embedding Scheduling

Chaoliang Zeng*, Xudong Liao*, Xiaodian Cheng, Han Tian, Xinchen Wan, Hao Wang, Kai Chen
iSING Lab, Hong Kong University of Science and Technology

Abstract

Deep learning recommendation models (DLRM) are extensively adopted to support many online services. Typical DLRM training frameworks adopt the parameter server (PS) in CPU servers to maintain memory-intensive embedding tables, and leverage GPU workers with embedding cache to accelerate compute-intensive neural network computation and enable fast embedding lookups. However, such distributed systems suffer from significant communication overhead caused by the embedding transmissions between workers and PS. Prior work reduces the number of cache embedding transmissions by compromising model accuracy, including oversampling hot embeddings or applying staleness-tolerant updates.

This paper reveals that many of such transmissions can be avoided given the predictability and infrequency natures of in-cache embedding accesses in distributed training. Based on this observation, we explore a new direction to accelerate distributed DLRM training without compromising model accuracy, i.e., embedding scheduling—with the core idea of proactively determining *"where embeddings should be trained"* and *"which embeddings should be synchronized"* to increase the cache hit rate and decrease unnecessary updates, thus achieving a low communication overhead. To realize this idea, we design Herald, a real-time embedding scheduler consisting of two main components: an adaptive location-aware inputs allocator to determine where embeddings should be trained and an optimal communication plan generator to determine which embeddings should be synchronized. Our experiments with real-world workloads show that Herald reduces 48%-89% embedding transmissions, leading up to 2.11× and up to 1.61× better performance with TCP and RDMA, respectively, over 100 Gbps Ethernet for end-to-end DLRM training.

1 Introduction

Deep learning-based recommendation systems have been extensively applied to a wide range of online services [9, 53], consuming significant infrastructure capacity and compute cycles across production datacenters [2]. Training a deep learning recommendation model (DLRM) poses challenges in both memory and computation. A typical DLRM (§2.1) consists of (1) *embedding tables*, which are large lookup tables that store millions to billions of semantic embedding vectors (embeddings for short) and consume a tremendous memory footprint (a few KB per embedding [29] and up to tens of GBs to TBs in total [30, 47, 52–54]), and (2) *multi-layer perceptron (MLP)*, which makes up the dense model and contributes to most of the computation cycles. These hybrid requirements make it challenging to train a DLRM efficiently. Specifically, hardware accelerators notably GPUs, which are widely incorporated into deep learning training, can execute high-performance dense model computation, but fail to provide large enough memory capacity to fully support the embedding tables component.

To address this dilemma, a common practice [2, 21, 29, 38] separates dense model computations and embedding table operations (lookup and update) in DLRM training. It typically adopts the parameter server (PS) architecture [26] to maintain globally shared embeddings in memory-optimized and cost-effective CPU servers while leveraging GPUs to accelerate the dense model computation with data parallelism. GPU workers will further cache a few hot embeddings in their local memory to accelerate the embedding lookup operation. However, a single training sample may involve up to thousands of embeddings in production workloads [2]. A naive cache-enable system still suffers from significant communication overhead for embedding transmissions[1] in bulk synchronous parallel (BSP) [12] training (§2.2), due to stale cached-embedding updates and embedding gradient synchronizations.

Prior efforts [3, 31] mitigate this communication overhead by *reducing the number of embedding transmissions between workers and PS*. They either oversample training data containing only hot embeddings with fast inter-server collective communication [3] or apply a staleness-tolerant embedding update manner [31]. Both approaches deviate from the vanilla BSP training and do not provide any theoretical guarantee of model accuracy. However, a high and stable model accuracy is critical in production. For example, even an order of 0.1% accuracy loss is intolerable in Meta recommendations [2]. Therefore, DLRM is more favorable to BSP training without a bias on training embeddings [34, 38].

In this paper, we explore a new direction, i.e., *embedding scheduling* (§3), to accelerate distributed DLRM training without compromising model accuracy. The core idea is to determine (1) where embeddings should be trained by distributing

* Equal contribution.

[1] In this paper, we refer to both the transmission of embedding value and the transmission of embedding gradient as embedding transmission.

batch training samples to likely cache-hitting[2] workers and (2) which embeddings should be synchronized by identifying embeddings to be trained in upcoming iterations for a low communication overhead. These optimization opportunities come from two crucial characteristics of in-cache embedding accesses during the DLRM training.

- **Predictability:** since the embeddings required by training samples and the current cache snapshot of each worker are visible before the computation, the upcoming cache accesses and their results (hit or not) are predictable under a partition of batch samples.
- **Infrequence:** the physical accesses to most in-cache embeddings are infrequent enough that the popularity of an embedding may be less than the total number of training samples assigned to a worker in the entire training process.

These two characteristics imply that with proactive embedding scheduling during the input generation, there exists a potential to train each infrequently accessed embedding in a fixed worker, respectively, which can effectively increase the cache hit rate and avoid unnecessary synchronizations for consistency (as this embedding is not required by other workers). Furthermore, our theoretical analysis proves that accelerating DLRM training with embedding scheduling can preserve the model consistency and hence the exactly same model accuracy as vanilla synchronous training.

When considering the whole DLRM training data flow, it is important to make the embedding scheduling decision in real time for two main reasons. First, production recommendation systems require online training from the streaming data for high-velocity inference queries [21, 29, 38]. Second, even for offline training, preparing scheduling decisions for the entire training process in advance is difficult, if not impossible, given the huge scheduling space and the unpredictable number of training iterations (and thus the number of scheduling decisions), which depends on the real-time evaluation result of the model. As a result, for offline training, we can only prepare the scheduling result for the early iterations, and we still need real-time scheduling for the rest training iterations.

Nevertheless, designing a real-time embedding scheduler is non-trivial. The *scheduling budget*, i.e., the available scheduling time, to conduct a scheduling decision for an iteration is limited, as the scheduling overhead should not be larger than the training time of an iteration to prevent introducing additional overhead when overlapping training and scheduling. Moreover, the scheduling budget varies among different workloads and training settings, so the embedding scheduler must be adaptive to different training tasks.

To this end, we propose Herald[3] (§4), a real-time embedding scheduler that leverages information including the required embeddings of input samples and the locations of those embeddings, to reduce embedding transmissions and thus achieve efficient DLRM training. Herald addresses the above challenges with a decoupling idea. Specifically, it determines *"where embeddings should be trained"* via an adaptive location-aware inputs allocation (LAIA) algorithm. The adaptive LAIA algorithm leverages the diverse ratios of infrequently-accessed embeddings among embedding tables, and conducts partition decisions based on a subset of selective tables for the batch inputs partition. Therefore, it effectively meets various scheduling budgets with a high cache hit rate. Herald further identifies *"which embeddings should be synchronized"* for this iteration in conjunction with the batch partition result of the next iteration by enabling sample prefetching. Herald figures out a minimal list of embeddings that should be synchronized to minimize gradient transmissions via an optimal communication plan generator.

We have implemented Herald on top of HET [31] (§5) and evaluated it via extensive simulations and a small-scale testbed with 8 Nvidia 3090 GPU workers on typical real-world workloads (§6). Evaluation results show that compared with HET, Herald reduces 48%-89% embedding transmissions. As a result, Herald delivers up to 2.11× and up to 1.61× better performance with TCP and RDMA, respectively, over 100 Gbps Ethernet for end-to-end DLRM training.

This paper makes the following contributions:

- We explore embedding scheduling as a new direction to accelerate DLRM training by leveraging two key characteristics of in-cache embedding accesses, i.e., predictability and infrequency (§3).
- We design Herald, a real-time embedding scheduler to meet various scheduling budgets while preserving a high scheduling quality (§4).
- We verify the benefit of embedding scheduling via testbed experiments on our Herald implementation (§5 and §6).

This work does not raise any ethical issues. Herald is available at https://github.com/HKUST-SING/herald.

2 Background & Motivation

2.1 Deep Learning Recommendation

A deep learning recommendation system models the recommendation decision as a problem to predict the probability of a specific event, e.g., the likelihood of a web viewer watching the recommended content (video or article). To fully exploit high-dimensional (categorical) features, the technique of representation learning is widely applied to project a category ID to a dense feature vector, also called *embedding*. This embedding is used as a representation of the category ID to apply in numerical computation for recommendation processing.

DLRM architecture. Figure 1 illustrates the high-level architecture of a DLRM with the example of apps recommendation. There are two primary components: *embedding tables* and

[2] In this paper, we refer to hitting the latest version of an embedding in the cache as a cache hit for short.

[3] We presented a preliminary idea of Herald in an earlier workshop paper [49].

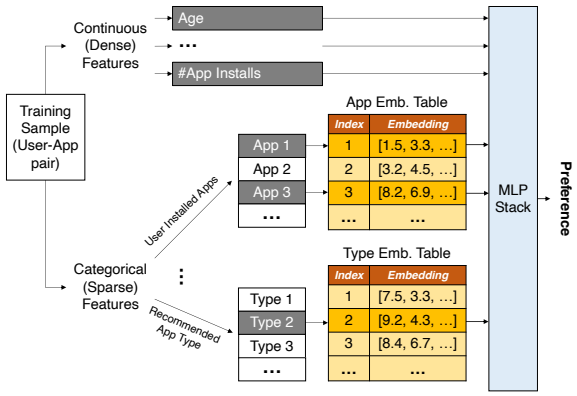


Figure 1: A high-level architecture of a typical DLRM with the example of apps recommendation.

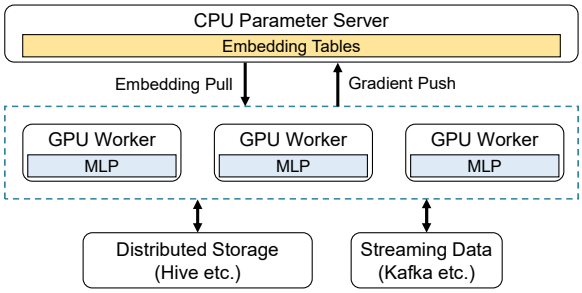


Figure 2: The data flow of DLRM training in production environments. We omit the data flow of MLP synchronization, which is not the focus of this paper.

multi-layer perceptron (MLP). The DLRM leverages embedding tables to project sparse features into dense representations, i.e., embeddings, by looking up every table with the corresponding category IDs as indexes. All these dense features, including embeddings and dense inputs, are processed by the MLP layer for the final prediction result. The value of embeddings, together with the parameters in MLP, will be iteratively updated during the training process.

Overview of distributed DLRM training systems. Figure 2 shows the data flow of the typical distributed DLRM training [2, 21, 29, 38]. DLRM training systems usually support different types of data I/O interfaces, e.g., Hive and Kafka, to provide good compatibility for various scenarios. Given the diverse requirements on computing and memory for MLP and embedding tables, respectively, DLRM training systems tend to separate the training of these two components. They train the MLP layer on compute-optimized GPU workers with data parallelism. To accommodate the scaling up of embedding table sizes (tens of GBs to TBs [30, 47, 52–54]), they maintain embeddings in memory-optimized CPU PS. This separation design introduces significant communication overhead when applying BSP for the non-degraded and reproducible model accuracy [5, 22, 34, 38]. In every training iteration, workers pull embeddings from PS on demand and push embedding gradients to PS at the end of this iteration.

Embedding cache. Given the skewed popularity distribution of embeddings, recent work [3, 31, 52] reduces embedding communications by exploiting an embedding cache in GPU workers. However, a naive cache-enable system provides limited performance improvement for BSP training, as it saves an embedding pull only when the queried embedding hits the cache (with the latest version) and does not save any embedding synchronizations. Therefore, they optimize the communication cost by compromising the BSP training procedure. For example, HET [31] tolerates a bounded staleness of each embedding by tracking the embedding version. Upon retrieving an embedding from the cache, it first compares the local version with the global version in PS and only pulls the embedding from PS if the difference between these two versions exceeds a threshold. A similar behavior happens for gradient synchronizations. HET still communicates with PS in every iteration for version checking, but these costs are much smaller than those of embedding transmissions.

	Model	Dataset
W1	Wide & Deep [7]	Criteo AD [10]
W2	Neural Collaborative Filtering [19]	MovieLens 25M [17]
W3	DeepFM [16]	Avazu [23]
W4	Deep & Cross [43]	Criteo Sponsored Search [40]

Table 1: Workloads in our case studies.

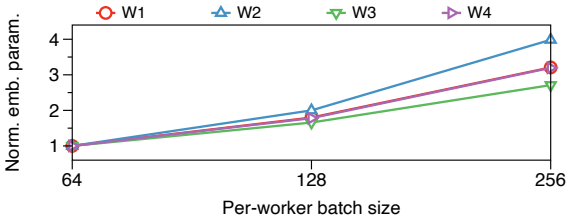


Figure 3: The number of embeddings used in an iteration increases with the per-worker batch size. The results are normalized to that number with a batch size of 64 for each workload.

2.2 Embedding Communication Matters

To quantify the contribution of embedding communication in the end-to-end DLRM training with BSP, we study typical DLRM workloads as listed in Table 1.

In the DLRM training, we find that a larger batch size (the number of training samples in an iteration) may lead to more embedding transmissions. The reason is that different samples may involve different sparse features and thus operate on different embeddings. Therefore, the number of operated distinct embeddings in an iteration grows with a larger group of samples (a larger batch size). Figure 3 demonstrates that the number of distinct embeddings used in an iteration can increase 2.7×-4.0× when the per-worker batch size increases from 64 to 256. However, the number of synchronized parameters in the MLP layer only depends on the MLP architecture, and thus does not change with the batch size.

We evaluate the distributed training efficiency of HET [31], the state-of-the-art cache-enabled embedding model training framework. The evaluation configuration follows the default setting as described in §6.1. We disable the embedding

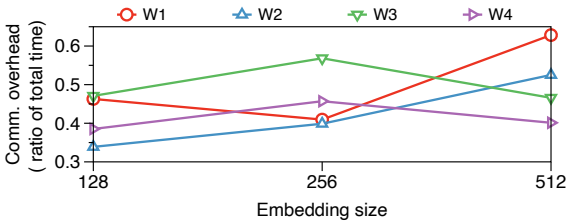


Figure 4: Embedding communication overhead can consume up to 63% of the end-to-end training time. The embedding size is the dimensions of an embedding or the number of columns of embedding tables.

prefetching[4] to precisely quantify the embedding communication overhead and its breakdown. The result is shown in Figure 4, where the embedding communication overhead for many of these workloads is high despite using state-of-the-art communication libraries like NCCL, and consumes up to 63% of end-to-end DLRM training time. It is worth noting that the ratio of the communication time to the total training time may not monotonically increase with the embedding size (the number of dimensions of an embedding or the number of columns of embedding tables), as the embedding size increases both computation time and communication time.

The embedding communication includes worker pull and worker push, which can be caused by either cache miss or cache update. When a cache miss occurs, the worker will pull the required embedding and push an evicted embedding if the cache is full. For cache updates, the worker will push the embedding gradients and pull the required embeddings with the latest version updated by other workers. Based on our evaluation, cache update is the major reason that causes embedding transmissions (71%-95%). In terms of the communication direction, pull and push contribute to a similar overhead (less than 9% coefficient of variance).

3 Embedding Scheduling

To reduce the number of worker pulls/pushes during the training, we propose a new direction to accelerate distributed DLRM training without compromising the model accuracy: *embedding scheduling*, which utilizes an embedding scheduler to determine where embeddings should be trained and which embeddings should be synchronized.

3.1 Rationales

In the forward propagation, a cache hitting on the required embedding can prevent worker pull and potential worker push caused by a cache eviction. Therefore, the embeddings to be trained can be scheduled to appropriate workers, where the training embeddings in each worker are most likely to hit the cache. Meanwhile, accessing an embedding in the forward propagation will incur a corresponding update in the backward propagation. However, synchronizing every embedding update is unnecessary, even in synchronous training. The reason is that an embedding is related to a sparse feature, and this feature may not be trained in the following iterations by other workers. It may happen in two scenarios: (1) this feature does not appear in the later training samples, or (2) this feature is only trained by the same worker. In other words, the updated embeddings only need to be synchronized to PS when they are required by other workers in the following training. Putting all this together, we derive two philosophies to reduce the embedding transmissions:

$\mathcal{P}1$. training in-cache embeddings as much as possible, and

$\mathcal{P}2$. performing on-demand synchronizations.

These philosophies can be followed by proactively partitioning the batch samples among workers and identifying the necessary embeddings for synchronizations.

Figure 5 describes a contrived and illustrative example to elaborate on how embedding scheduling reduces communication overhead. The vanilla training without embedding scheduling (Figure 5b) neither optimizes for cache hit rate nor avoids unnecessary embedding synchronizations due to the ignorance of workers' cache content and the following inputs. As a result, the cache miss or hit during the training is totally random, and every updated embedding should be synchronized. However, such embedding communication can be optimized with embedding scheduling (Figure 5c).

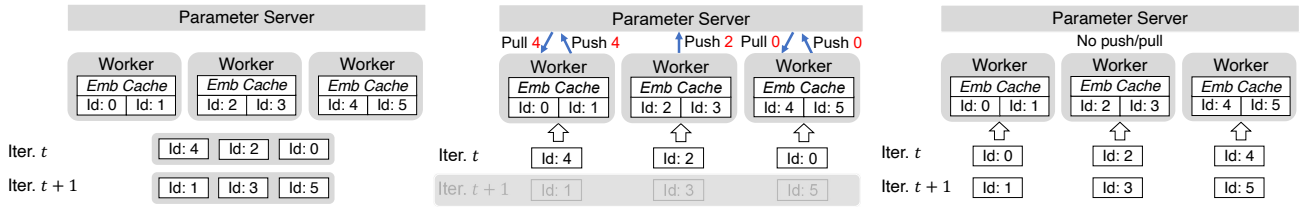
We can formulate the scheduling problem as a Markov Decision Process (MDP) as described in Appendix A. However, given the complexity of finding a globally optimal scheduling decision for the entire training process, we downgrade the scheduling problem to find a local quality decision for every iteration in this work.

3.2 Opportunities

Embedding scheduling is feasible given two characteristics of in-cache embedding accesses.

Predictability. There are two prerequisites for embedding scheduling: knowing current cache snapshots as well as predicting and determining future embedding accesses. Fortunately, both are achievable in DLRM training. As modern training frameworks decouple the training computation and the input preparation, we can foresee and decide proper embedding accesses a-priori when proceeding with input generation for each worker. The input generation is usually performed in a data loader, which partitions a batch of inputs into multiple micro-batches for every worker. Meanwhile, maintaining cache snapshots in the data loader is trivial. Therefore,

[4] HET supports prefetching the embeddings that will be used in the next iteration from the PS at the beginning of an iteration. We will enable this feature in end-to-end evaluation. However, prefetching brings limited improvement in vanilla BSP training, since most embedding pulls are for the updated embeddings as shown in the later paragraph, and these updated embeddings are valid after finishing the synchronization of the current iteration.



(a) Toy example (b) Training w/o embedding scheduling (c) Training w/ embedding scheduling

Figure 5: (a) A contrived example showing performance gain with embedding scheduling, assuming that all caches have empty slots for new embeddings. (b) The vanilla training incurs a total of 2 times worker pulls and 3 times worker pushes at iteration t. (c) The training with embedding scheduling does not incur any embedding transmissions at iteration t, since all trained embeddings hit caches and their updates are not required by the others in the later iteration.

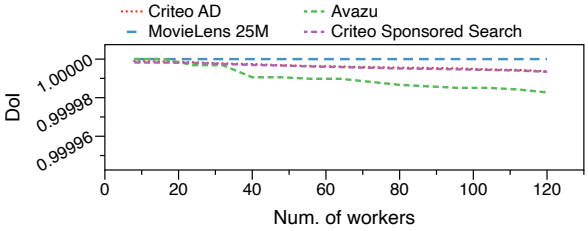


Figure 6: The degree of infrequence is consistently high across different scales of training clusters in studied datasets.

a tailored data loader with an embedding scheduler can allocate input samples to workers based on their in-cache embeddings and identify embedding dependencies among iterations for on-demand synchronizations.

Infrequency. Although a small set of "hot" embeddings could contribute to the majority of the total number of accesses [3,31], the access of individual in-cache embeddings is usually infrequent. We define that an in-cache embedding is infrequently accessed when the popularity of this embedding is less than the total number of samples trained by a worker during the entire training epoch. The more infrequently-accessed embeddings there are the more potential transmissions can be optimized. To evaluate the *degree of infrequence (DoI)*, i.e., the ratio of infrequently-accessed embeddings over the whole in-cache embeddings, we make profiling on real-world datasets as listed in Table 1, and consider each worker can cache 10% of the total embeddings. As shown in Figure 6, our profiled datasets exhibit consistently high DoI ($> 99\%$) even in large training clusters. The high DoI indicates that most in-cache embeddings can each be trained in a consistent worker and their updates can avoid global synchronizations.

3.3 Model Consistency Analysis

Accelerating distributed DLRM training under BSP with embedding scheduling can preserve model consistency and thus the exactly same model accuracy. Compared with vanilla distributed training, embedding scheduling makes two changes: a tailored input partition algorithm (vs. a random partition algorithm) and on-demand synchronizations (vs. full-set synchronizations). In BSP, on-demand synchronizations ensure that all workers use the latest parameters for the incoming training iteration, which is the same behavior as the vanilla distributed training performs.

We prove that the training model will not be affected by the choice of input partition algorithm under BSP. Considering a parameter optimizer followed by the SGD algorithm [14], the gradient calculation for model weights w on a given batch of n training samples is expressed as follows:

$$\nabla w = \frac{1}{n}\Sigma_{i=1}^{n}\frac{\partial L(x_i, w)}{\partial w}, \tag{1}$$

where x_i is the i-th training sample of the batch, and L is the loss function. Based on Equation 1, the gradient of the batch is the sum of the individual gradient for each training sample in the batch. Since the individual gradient depends only on input samples and current model weights, which have been synchronized before this iteration begins under BSP, partitioning the batch into m micro-batches takes no effect on the gradient result:

$$\frac{1}{n}\Sigma_{i=1}^{n}\frac{\partial L(x_i, w)}{\partial w} = \frac{1}{n}\Sigma_{i=1}^{m}\Sigma_{j=1}^{n/m}\frac{\partial L(x_{ij}, w)}{\partial w}, \tag{2}$$

where x_{ij} is the j-th training sample in the i-th micro-batch (for worker i). Therefore, any partition result generated by any partition algorithm preserves the same gradients as in BSP throughout training and hence converges to the same model.

4 Design

Based on the above ideas, we design an embedding scheduler, called Herald, to accelerate distributed DLRM training on top of the state-of-the-art cache-PS architecture [3,31]. Real-time scheduling is a crucial property when 1) training over streaming samples [21,29,38], as pre-processing the whole dataset is difficult, if not impossible, in this case, and 2) the number of training iterations (scheduling decisions) is unpredictable. In this work, we target to support real-time scheduling, where only a few batch inputs instead of the whole dataset are available at a time and the available batch inputs keep updating during the training procedure. We leave discussions on breaking this assumption in §7.

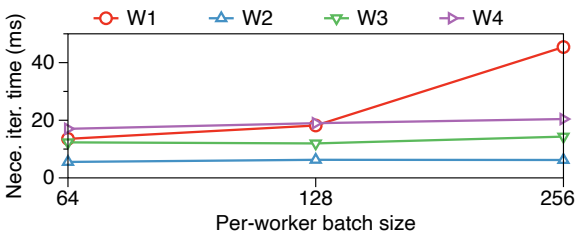


Figure 7: The necessary iteration time varies among different datasets and different per-worker batch sizes, where the embedding size is 256.

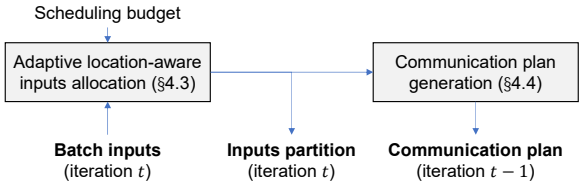


Figure 8: Herald overview.

4.1 Challenges

Integrating real-time embedding scheduling into training frameworks is challenging due to limited and varied *scheduling budgets*, i.e., the available scheduling time.

Limited scheduling budget for real-time scheduling. For a scheduling problem, there exists a tradeoff between scheduling quality and scheduling overhead. High-quality embedding scheduling can effectively reduce communication overhead, but at the cost of consuming a long scheduling time. Such long scheduling time, however, prevents the scheduling from being hidden by the iterative training, and thus introduces extra training overhead. The scheduling budget of real-time scheduling should be smaller than the iteration training time.

Varied scheduling budgets across different workloads and training settings. The scheduling budget is not static and is affected by many factors, including workload and training setting. We regard *necessary iteration time*, i.e., the training time excluding the embedding communication time in an iteration, as a valid scheduling budget, and show these results in Figure 7[5] with the same setting as Figure 4. It reveals an up to 78% coefficient of variance across different workloads within the same setting and an up to 67% coefficient of variance across different settings within the same workload. Varied scheduling budgets indicate that a static scheduling algorithm is not generic enough to support highly variant workloads and settings.

4.2 Overview

Figure 8 overviews Herald, which decouples the schedule targets to support real-time scheduling. Specifically, Herald determines *"where embeddings should be trained"* via an adaptive location-aware inputs allocation (LAIA) algorithm (§4.3), which meets varied scheduling budgets with a high cache hit rate. Herald determines *"which embeddings should be synchronized"* under a given inputs partition via an optimal communication plan generator (§4.4). The above scheduling process is executed on every worker instead of a centralized orchestrater, so that the scheduling can be computed locally to save the distribution time of scheduling results.

[5]Following the industry practice [2], we focus on *weak scaling*, which linearly increases the total batch size with the number of workers, instead of *strong scaling*, which keeps the per-iteration total batch size constant.

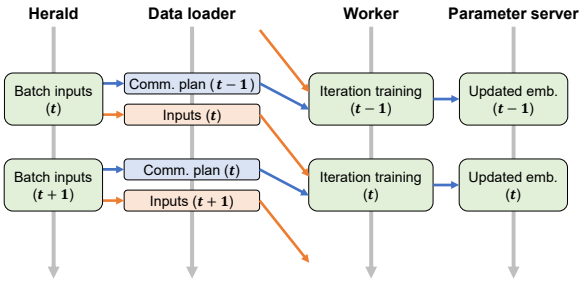


Figure 9: Herald training data flow. Herald leverages input prefetching to generate the communication plan in time.

As discussed before (§3.1 and Appendix A), the scheduling scope in the current design choice is within the single batch to achieve a low scheduling overhead. Given a batch of inputs of iteration t, Herald first generates a partition solution, which allocates each input to a proper worker for iteration t based on current embeddings' location information. However, there is a lag in the generation of the communication plan, which depends on the requirements of the next iteration. To guarantee that each worker can retrieve the latest embeddings under the generated partition solution, Herald figures out an optimal communication plan, which lists a minimal list of embeddings that each worker should synchronize with PS, for the previous iteration, i.e., iteration $t-1$. We discuss extending the scheduling scope to multiple batches in §7.

Figure 9 illustrates the data flow of the training with Herald. From a worker's point of view, in every iteration, the worker receives both training samples and a communication plan from the data loader before proceeding computation, and synchronizes the embeddings listed in the communication plan after backward propagation. Due to the time lag of communication plan generation, Herald needs to partition the next batch inputs before proceeding to that batch, which is possible as prefetching next batch samples is already commonly adopted in today's data loaders [1, 6, 36].

It is worth noting that Herald only controls the embedding update from workers to PS (i.e., worker push), but remains the update manner from PS to workers (i.e., worker pull) the same as existing cache systems operate. Herald relies on the internal cache consistency protocol in existing cache systems (e.g., version check in HET [31]) to pull the updated embeddings from PS and cache them in individual workers. Therefore, Herald preserves the same cache consistency as the underlying cache system. Moreover, as the cache miss results in only a minor fraction of embedding transmissions (§2.2), Herald does not optimize the policy of embedding

Algorithm 1: Static LAIA

```
input  : Batch samples (Inputs) and worker list (Workers)
output : Inputs partition (Alloc)
1  Init Alloc;
2  Init all workers as available;
3  capacity = size(Inputs)/size(Workers);
4  for i in Inputs do
5      for w in Workers do
6          score_(i,w) = |cache(w) ∩ embs(i)|;
7      end
8      Find worker w with the largest score among the
         available workers;
9      Alloc_(i,w) = 1;
10     if Σ_i Alloc_w == capacity then
11         Mark w as unavailable;
12     end
13 end
14 return Alloc;
```

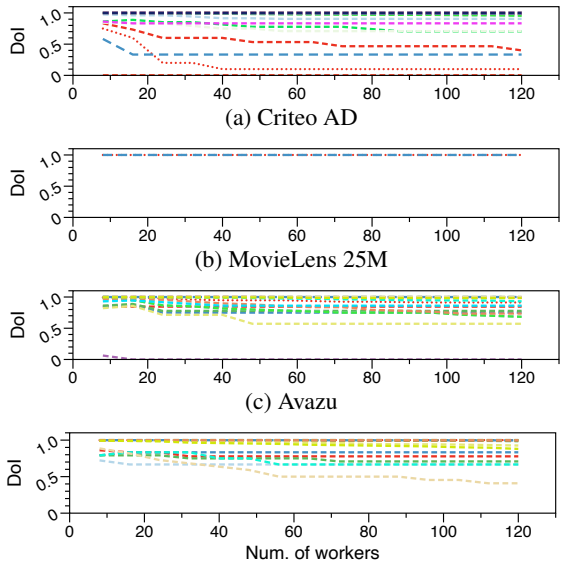
cache evictions caused by the cache miss to avoid additional scheduling overhead.

Scheduling budget measurement. To preserve real-time scheduling, the scheduling budget should not be larger than the iteration time, which consists of both computation time and communication time. However, with embedding scheduling, the communication depends heavily on the scheduling results, which are hard to predict precisely before training[6]. Therefore, we utilize the necessary iteration time (as defined in §4.1), which can be profiled in advance, as a loose scheduling budget. For Herald scheduling, as only the input partition can adapt to the scheduling budget, and the upper bound of the communication plan generation time is predictable given the workload and the training setting, the real scheduling budget used in Herald should be the necessary iteration time minus the communication plan generation time. As our measurement is conservative (by considering a loose metric), it is not likely to overestimate the scheduling budget. Furthermore, our evaluation (§6.2.2) shows that Herald is resilient to the underestimation of the scheduling budget.

4.3 Location-aware Inputs Allocation

We first elaborate a static LAIA algorithm that heuristically partitions batch inputs into proper workers based on the current cache snapshots and a full set of table information. Then we observe that embedding tables are not equally important for scheduling, with the diverse DoIs among embedding tables. Based on this observation, we introduce an adaptive LAIA algorithm, which conducts the partition with selective embedding tables to meet the scheduling budget.

[6]Even during training, the communication time after scheduling can vary among iterations.



(a) Criteo AD
(b) MovieLens 25M
(c) Avazu
(d) Criteo Sponsored Search

Figure 10: The DoI of different embedding tables exhibits a large variance. Each line represents an embedding table.

Static LAIA. As it is impractical to conduct a brute-force search in real time for the optimal partition solution that maximizes the cache hit rate, we propose a heuristic partition algorithm, LAIA, as shown in Algorithm 1. LAIA calculates a score to quantify the relevance between every input sample and worker (Line 6). The score is defined as the number of embeddings that reside in the worker cache (with the latest version) and are required by the input sample simultaneously. After scoring, LAIA allocates each input to the worker with the highest score (Line 8-9) while ensuring evenly distributed inputs among workers (Line 10-12).

Furthermore, when there is a tie in finding the largest score (Line 8), LAIA will randomly select a worker from the candidates. This randomization tends to evenly cache embeddings across all workers. Therefore, in the later scheduling, LAIA can achieve a good balance of the total score (i.e., cache hit rate) among workers. We do not proactively treat the load balance as a scheduling goal in Algorithm 1 for a low scheduling overhead. With consecutive iterations, LAIA moves towards gathering embeddings that tend to be accessed together in many training samples on the same worker.

DoI diversity among tables. We observe that since every embedding table represents a unique physical feature, the distribution of embedding popularities in different tables may exhibit different skewness and hence diverse table DoIs. Figure 10 shows this DoI variance, where we measure the DoI of an individual table as the ratio of the number of infrequently accessed embeddings over the total number of in-cache embeddings of this table. Among the profiled datasets (except for MovieLens 25M which has only two tables[7]), there exist

[7]For those datasets that have a small number of tables, although the DoI diversity is not obvious, they have a low demand for this property, as their scheduling overhead is inherently small.

both tables with consistently high DoI ($> 99\%$) and tables with relatively low DoI (as low as $< 1\%$). Recalling that the potential performance gain achieved by embedding scheduling comes from the infrequent embedding updates (§3.2), this DoI diversity indicates that the embedding tables are not equally important for scheduling.

Being adaptive. Based on the above observation, we can accommodate Algorithm 1 to varied scheduling budgets. In Algorithm 1, scoring (Line 4-7) is one of the most time-consuming functions due to its three-layer loop, including inputs iteration, workers iteration, and tables iteration (implicitly shown in Line 6). Therefore, by only considering scheduling-worthy (i.e., high-DoI) embedding tables, the time overhead of scoring and thus Algorithm 1 can be reduced. Given the scheduling budget, we can determine the maximum number, k, of embedding tables that should be considered in scoring.

Table profiling. To identify the k embedding tables with the highest DoI in runtime, we incorporate a table profiler into Herald to continuously measure the DoI of each table. Specifically, the profiler counts the appearance of every embedding in every batch input and maintains the DoI of each table dynamically. The profiling runs parallel to the allocation function to avoid interference with the scheduling process. The profiler periodically updates the k embedding tables with the highest DoI to LAIA for scheduling. The interval to update the profiling result can be decided by either the change of the selective embedding tables or the profiling time.

4.4 Communication Plan Generation

To figure out the necessarily synchronized embeddings for on-demand synchronizations, we introduce a concept of *embedding dependency*. An embedding dependency appears when an embedding with the latest version is cached on a worker and there are other workers assigned to train this embedding in this iteration. The worker caching the latest version should synchronize this embedding to PS before this iteration begins. Therefore, all embedding dependencies of this iteration become the communication plan of the previous iteration. As discussed in §4.2, Herald can timely generate the communication plan with the inputs prefetching.

Based on the above idea, Herald generates the communication plan for each worker given the inputs partition result of the next iteration, as described in Algorithm 2. The *embs_by_others* are all embeddings required by workers except for worker w itself (Line 2) in the next inputs, and the embedding dependencies (communication plan) of worker w are the intersection of *embs_by_others* and its cached embeddings (Line 3). The generated communication plan is optimal between two iterations, as it contains the minimal list of embeddings required by other workers in the next iteration. Since we focus on the PS architecture, the embedding dependencies of a worker do not need to be distinguished among target workers. We will discuss how to support on-demand synchronization with peer-to-peer communication in Appendix B.

Algorithm 2: Embedding dependency generation

input : Batch samples (*Inputs*), worker list (*Workers*), and inputs allocation (*Alloc*)
output : Embedding dependency (*Dep*)

1 **for** *w in Workers* **do**
2 $\quad embs_by_others = embs(Inputs - Alloc_w)$;
3 $\quad Dep_w = embs_by_others \cap cache(w)$;
4 **end**
5 **return** *Dep*;

5 Implementation

We have implemented a system prototype of Herald with C++ and Python. Besides the design described in §4, Herald maintains cache snapshots of all workers inside the scheduler to provide information on embedding locations. These cache snapshots are used for the scoring function in inputs allocation, and updated at the end of the allocation based on the partition result, which avoids synchronizations among schedulers in different workers. Cache snapshots follow the same caching policy as workers' for consistency.

HET plugin. We have integrated Herald in HET[8]. We replace the data loader implementation in HET with communication plan support. Herald data loader returns training samples and synchronized embedding indexes as sparse inputs. To support prefetching batch inputs while preventing from interfering with the training process, Herald is launched by dedicated threads, and the scheduling results from Herald are transmitted to the data loader via shared memory.

Multi-threading. We utilize spare CPU resources to accelerate the scheduling by enabling multi-threading. We manage a thread pool for multi-threading processing. The parallelism can be easily applied to most Herald functions, including scoring (Line 4-7 in Algorithm 1), cache snapshots update, and communication plan generation (Algorithm 2). However, it is non-trivial to parallelize the function of worker allocation (Line 8-12 in Algorithm 1) due to its dependency between two allocation actions. Preserving the consistency of the allocation result inevitably introduces a lock and limits the parallelism of this function. We implement two versions of the worker allocation function: a single-threading version and a lock-free multi-threading version which may compromise the result consistency. In the multi-threading version, we evenly divide the worker capacity and training samples among threads. Therefore, each thread can independently allocate its samples to the workers without considering the allocation result of other threads. The allocation result of each thread is then merged into the final allocation.

[8] https://github.com/Hsword/Hetu

6 Evaluation

Our evaluation seeks to answer the following three questions:
How does Herald improve cache embedding communication overhead? Simulation experiments (§6.2.1) with representative datasets show that Herald reduces the average number of embedding transmissions by 48%-89%, where LAIA algorithm contributes to the major improvement according to the performance breakdown.
How lightweight and effective is Herald for real-time and adaptive scheduling? Deep dive profiling (§6.2.2) on Herald shows that even adopting the single-threading worker allocation function, multi-threading benefits 92% of scheduling time. Furthermore, adaptive Herald preserves consistently high scheduling quality with less than 1.11× transmission increase compared with static Herald across different scales of training clusters.
How does Herald perform in end-to-end DLRM training? Testbed experiments (§6.3) with real-world workloads show that Herald provides 1.09×-2.11× and 1.02×-1.61× speedup over HET in end-to-end training with TCP and RDMA, respectively, over 100 Gbps Ethernet. Herald preserves the effectiveness in the multi-GPU and multi-node scenario.

6.1 Evaluation Settings

Our evaluations include both cache simulation for micro-benchmarks and testbed for end-to-end training. Unless mentioned otherwise, both the simulation and testbed contain 8 workers and 1 PS. Each worker contains an embedding cache following LRU policy. The embedding cache can house 10% of PS-side embedding tables, following the caching strategy studied in [25, 31, 48]. We adopt the single-threading version of the worker allocation function whenever possible for the best scheduling quality, and adopt the multi-threading version when evaluating the scalability of LAIA algorithm (§6.2.2). The default thread pool size in Herald is 16.
Testbed. Both workers and PS are equipped with a 20-core Intel Xeon Gold 5218R CPU at 2.1 GHz and 64 GB (256 GB in PS) of RAM. Each worker is also equipped with an Nvidia 3090 GPU. These workers and PS are connected at 100 Gbps Ethernet with Mellanox ConnectX-5 NICs. The default underlying transport is TCP. We also incorporate RDMA into HET implementation. All machines run Ubuntu 18.04 (Linux 5.4.0), CUDA 11.3.1[9], cuDNN 8.2.0, and NCCL 2.9.9.
Baseline. We compare Herald with HET [31] and FAE [3]. FAE is designed in a static caching manner. For a fair comparison, we implement FAE on top of HET. As discussed in §1 and §8, both baselines reduce the embedding transmissions by compromising model accuracy and are orthogonal to Herald. Therefore, we adopt vanilla BSP training in HET and FAE. Other settings are configured as same as Herald. Baselines and Herald leverage a hybrid synchronization model [24], where MLP parameters are synchronized in ring-based all-reduce and embedding parameters are synchronized with PS. Baselines and Herald enable the embedding prefetching.

[9]We upgrade HET from CUDA 10 to CUDA 11.

Optimization	Reduced embedding transmissions		
	Pull	Push	Overall
Vanilla (w/o embedding scheduling)	-	-	-
LAIA (§4.3)	54%	17%	35%
Communication plan (§4.4)	0%	13%	7%
Herald	54%	63%	59%

Table 2: Breakdown of contribution by each optimization with the W1 dataset.

Workloads. We use four real-world models with representative datasets for end-to-end experiments, as listed in Table 1. We exclude the first 10 iterations for warm-up and report the performance for the subsequent iterations. We do not include training accuracy results because Herald can preserve a consistent model accuracy, as analyzed in §3.3. The default batch size is 128 in the simulation and 256 in the testbed, and the default embedding size is 512. Since Herald with multi-threading can meet the scheduling budget of the above workloads in the default training setting based on our evaluation (§6.2.2), we adopt static LAIA (Algorithm 1) rather than its adaptive version in end-to-end experiments by default.

6.2 Micro-benchmarks

6.2.1 Cache Performance

We first study the cache behaviors by comparing Herald with vanilla training manner, i.e., a random inputs partition and full-set embedding synchronizations.
Cache behaviors. Figure 11 shows the number of cache embedding transmissions by normalizing to that of vanilla training across different datasets. Overall, applying Herald can reduce 48%-89% embedding transmissions. We further decompose the cache embedding transmissions based on the communication direction. It shows that Herald effectively reduces the number of worker pulls by 43%-88% and worker pushes by 51%-90%.
Improvement breakdown. To figure out the reason behind these improvements, we further break down the contribution made by each optimization with the Criteo AD dataset (W1), as shown in Table 2. We find that LAIA contributes to the major improvements. For worker pull, LAIA distributes the embeddings required by inputs to most likely hit worker caches (reducing the number of worker pulls by 54%). Meanwhile, the worker push performance is jointly optimized by LAIA and communication plan, where the location-aware partition mechanism reduces embedding dependencies while on-demand synchronizations make such dependencies reduction benefit to embedding transmissions reduction. Therefore, the optimization of LAIA and communication plan alone only reduces the number of worker pushes by 17% and 13%, respectively, but the combination of the two, i.e., Herald, can reduce the number of worker pushes by 63%.

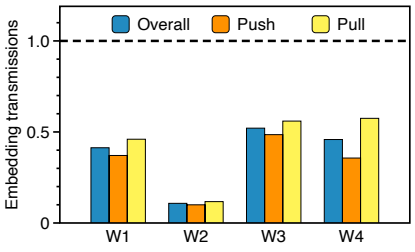


Figure 11: Herald can reduce 48%-89% embedding transmissions.

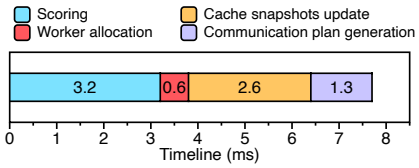


Figure 12: Breakdown of scheduling overhead with the W1 dataset.

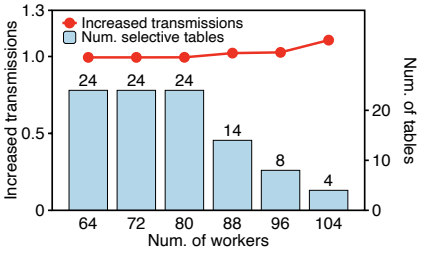


Figure 13: Adaptive LAIA is scalable with less than 1.11× transmissions increase.

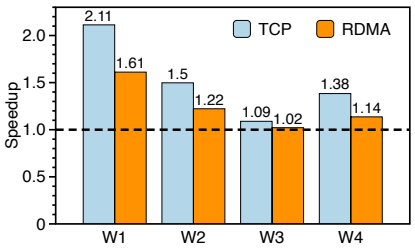


Figure 14: Herald improves end-to-end training by up to 2.11× speedup over TCP and up to 1.61× over RDMA.

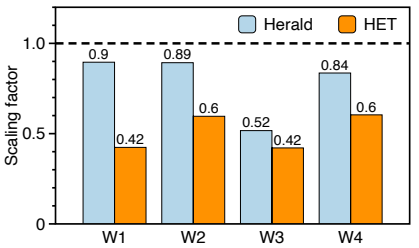


Figure 15: Herald utilizes distributed computing resources efficiently with high scaling factors.

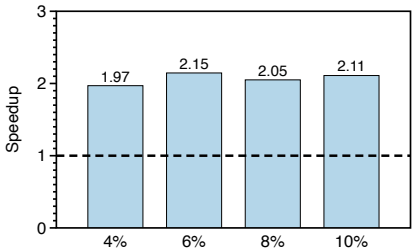


Figure 16: Herald preserves a consistent effectiveness under limited cache sizes.

6.2.2 Scheduler Performance

We profile the Herald scheduler overhead with the same machine as the testbed (§6.1) to evaluate its time consumption and scheduling quality. In this part, we focus on the Criteo AD dataset only.

Overhead breakdown. Through our measurement, the per-batch scheduling overhead is 7.7 ms (16 threads) on average. Figure 12 shows the breakdown of this scheduling overhead, where the major overhead is caused by the scoring function, consuming 3.2 ms (42%). 92% of the overhead, including scoring, cache snapshots update, and communication plan generation, are accelerated by multi-threading. Although the worker allocation function runs in a single thread, it contributes to only 0.6 ms (8%) overhead.

Scalability of adaptive LAIA. We further evaluate the adaptiveness of LAIA across different scales of training clusters. We assume that the scheduling budget[10] does not change when increasing the scale of training clusters. Thereby, we find the number of tables that LAIA can be considered during the scoring, so that the whole scheduling time is not larger than the scheduling budget, i.e., 18.2 ms in this case. In this experiment, we use a multi-threading worker allocation function for adaptive LAIA. We report the results of different cluster scales, where the number of selective tables is smaller than the total number of tables (i.e., 26) in Criteo AD dataset. The increased transmissions are the ratio of the embedding transmissions incurred by adaptive LAIA to that incurred by static LAIA. As shown in Figure 13, adaptive LAIA introduces less than 1.11× transmission increase by scoring with at least 4 tables. This result indicates that Herald is scalable and adaptive to different scales of training clusters. Moreover, it also reveals that Herald is resilient to the underestimation of the scheduling budget by preserving a high scheduling quality with a small set of selective tables.

Herald vs. brute-force search. The performance of Herald compared with the brute-force search is shown in Appendix C due to the space limitation.

[10]The scheduling budget increases with the scaling training clusters due to the larger dense parameters synchronization time. Therefore, we can score with a larger number of embedding tables in adaptive LAIA than we report.

6.3 End-to-end Training

We demonstrate that Herald accelerates end-to-end training, improves scalability, and preserves performance superiority even under limited cache sizes and large-scale clusters.

Training speedup. Figure 14 shows the training speedup of Herald over HET across different workloads. We observe that Herald can achieve 1.09×-2.11× speedup over TCP and 1.02×-1.61× over RDMA. Specifically, we find that the speedup of W1 is the highest among all, which accords with the results in Figure 4 that W1 has the highest embedding communication ratio. The above speedups are attributed to the communication reduction of Herald in the number of embedding transmissions, as explained in §6.2.1.

Scalability. The high training performance makes the distributed training system more scalable. We evaluate the scalability of Herald and HET with the metric of scaling factor defined as in [51]: $\frac{T_N}{NT}$ where T is the throughput of a single worker, N is the number of workers, and T_N is the overall throughput of a cluster with N workers. Note that T in Herald

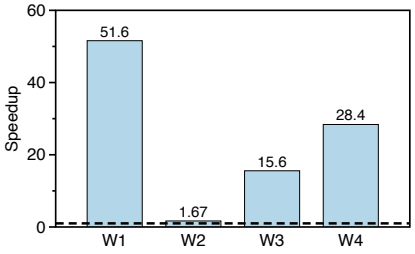


Figure 17: Herald achieves tremendous speedup compared with FAE-like static caching.

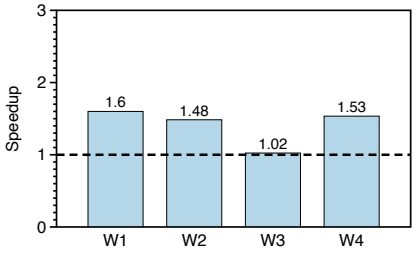


Figure 18: Herald preserves the performance superiority in a large-scale testbed.

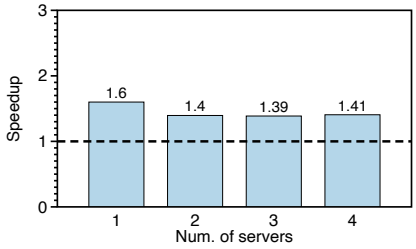


Figure 19: Herald preserves the performance superiority when increasing the number of servers for PS.

equals that in HET, as both frameworks behave the same in the single-worker training. We measure the throughput of a single worker with a single GPU server, where the PS runs on CPUs in the same server to eliminate network communication. Figure 15 shows the scaling factor of Herald and HET in every workload. Herald improves the scaling factor from 0.42-0.60 to 0.52-0.90, which indicates that incorporating Herald into distributed DLRM training can utilize resources more efficiently.

Training performance under limited cache sizes. Given the ever-expanding size of the embedding tables and the relatively slow evolution of accelerator device memory, it is likely that the ratio of in-cache embeddings becomes smaller. We study the impact of different embedding cache sizes on W1. The cache ratio, i.e., the ratio of the number of in-cache embeddings to the total number of embeddings, ranges from 4% to 10%. As shown in Figure 16, Herald preserves the performance superiority over HET and the speedup (1.97×-2.15×) does not vary a lot across different sizes of embedding cache. The reason is that, as discussed in §2.2, embedding transmissions are dominated by the embedding update, and this dominance holds across different embedding cache sizes. Therefore, the distributed DLRM training framework still benefits from reducing embedding pulls and pushes in embedding updates even under a small embedding cache size. This experiment verifies the consistent effectiveness of Herald under limited cache sizes.

Compared with FAE (static caching). FAE [3] is quite a different cache model from Herald and HET. First, FAE is a static cache, i.e., the cached embeddings are predetermined and fixed before training. Second, all workers in FAE cache exactly the same embeddings, and they synchronize all cached embeddings with all-reduce as dense parameters. We build FAE-like static caching on top of HET. Our FAE-HET implementation supports caching at most 1% embeddings in each worker and training workloads with a per-worker batch size of 128. For implementation efficiency, we change embedding interactions from concatenation to sum pooling in all workloads. As shown in Figure 17, Herald outperforms FAE with a 1.67×-51.59× speedup. The huge improvement mainly comes from two reasons. First, since Herald's workers can cache arbitrary embeddings, Herald's overall cache size is larger than FAE's, thus leading to a higher cache hit rate. Second, the embedding scheduling in Herald further improves the efficiency of the cache usage (by reducing the number of worker pulls and worker pushes) compared with FAE.

Multi-GPU and multi-node scenario. Lastly, we evaluate Herald in a multi-GPU and multi-node testbed, consisting of 10 × 8-GPU servers as workers. Each worker contains 80 CPU cores, and the thread pool size is also set to 80. The other settings remain the same as the default. At such a large scale, Herald adopts adaptive scheduling (§4.3) to bound the scheduling time. Figure 18 shows the results. By comparing to HET, Herald can achieve 1.02×-1.60× speedup among four workloads by leveraging 77%-100% of tables for scheduling. We further evaluate the Herald performance when increasing the number of servers for PS on the W1 workload. As shown in Figure 19, the improvement of Herald may have a slight decrease (achieving about 1.4× speedup) when the capability of PS increases. The reason is that increasing the number of servers for PS can relieve or even eliminate the network bottleneck in the PS architecture [41].

7 Discussions

Scheduling without batch prefetching. If the streaming dataset only provides a single batch of inputs at a time, i.e., no batch prefetching support, Herald will fail to generate communication plans for the on-demand synchronizations. In this case, Herald's LAIA algorithm still works to provide a cache-friendly inputs partition. Based on the optimization breakdown (Table 2), scheduling with only LAIA can preserve a considerable overall improvement.

Scheduling among multiple batches. If the scheduling budget is sufficient or even unlimited (offline scheduling in this case), Herald's scheduling scope can be extended to multiple batches to achieve a better scheduling quality.

Inputs partition. Instead of finding a locally high-scoring allocation of a single batch as in Algorithm 1, the partition algorithm for multiple batches should seek an allocation with a globally high score. However, searching for such globally high-scoring allocation is onerous, as allocation scores are dependent on batches. Different allocation results of the front batch may result in totally different cache snapshots, and thus affect the allocation score of the following batch. A potential

heuristic method to reduce the exploration space is to leverage the power of two choices [32], which considers the top-2 highest scoring allocations of a batch, explores the next batch based on these two allocations in parallel, and finally returns the allocation with the highest overall scoring.

Beyond preserving the same batch order when allocating the multi-batch inputs, there exists work to reorder inputs among training iterations for embedding transmissions saving [4]. However, the non-random ordering of training samples may increase the gradient variance during the SGD training and lead to overfitting, as the model may learn some spurious patterns from these handcrafted sample orders [13].

Communication plan generation. In current practice, workers may execute different sizes of communication plans in an iteration. This kind of work imbalance will result in idle workers at the synchronous barrier of the iteration. This issue can be improved by considering multiple batches when generating the communication plan for both embedding pull and push, as discussed in Appendix A, with additional embedding dependency checking. For embedding pull, the worker can prefetch an embedding if it is not trained by other workers until the time this worker accesses it. For embedding push, the worker can synchronize more embeddings beyond the requirement in the next iteration. Based on this idea, we can re-schedule communication plans to balance the communication workload among workers, and thus reduce the overall communication time.

8 Related Work

Distributed recommendation systems. Several specialized systems have been proposed for scalable and efficient training upon DLRMs. Persia [28] advocates mixing the synchronous and asynchronous mechanisms to update MLP and embedding tables, respectively. However, the asynchronous scheme is not scalable and can compromise accuracy with an increasing number of workers [34]. XDL [21] proposes optimizations including hierarchical sample compression, workflow pipeline, and zero copy. It provides systematical optimizations on the DLRM training pipeline, and can benefit from the embedding scheduling to further optimize inter-worker/PS communication. We acknowledge that the embedding scheduling only works for data parallelism. For the system adopting a hybrid parallelism strategy, like Neo [33], embedding scheduling can provide a partial system acceleration.

Communication acceleration. There are many efforts exerted to accelerate communication for deep learning training. A line of work speeds up individual messages with efficient collective communication design. Besides those designed for deep neural networks [8, 35, 39, 41, 42, 45], many collective communication approaches [11, 24, 27, 37] are proposed to optimize the synchronization of sparse parameters. Another line of work exploits communication scheduling [18, 20, 50], which organizes the message transmission order of different layers to overlap communication with computation. All the above communication acceleration methods try to answer *"how to efficiently transmit messages"*. In contrast, Herald accelerates communication by answering *"which messages should be transmitted"*.

Serving large embedding tables. There are two common architectures to resolve the large memory requirement on embedding tables. The first one applies model parallelism directly across multiple GPU workers, where each GPU stores a shard of tables on its high-bandwidth memory (HBM) [27,44]. However, this manner is sub-optimal, as the GPU resource is usually scarce and expensive. The second architecture is the cache-PS architecture [52] as the focus of this paper. This architecture leverages the skewness feature of datasets to accelerate embedding accesses with high popularity, while Herald further identifies the infrequency feature among those cached embeddings. To reduce the communication between cache and PS, existing works apply accuracy-compromising optimizations including oversampling hot inputs (accessing only hot embeddings) [3], reordering training samples among batches [4], and updating embeddings with staleness-tolerance [31]. On the contrary, Herald optimizes the embedding communication with embedding scheduling, which can preserve the model (accuracy) consistency theoretically. Moreover, when the accuracy is not that sensitive, Herald can also benefit from the above philosophies to optimize the scheduling process. Another orthogonal line of optimization focuses on cache prefetching by scheduling the embedding IO and the computation inside a worker [5, 15, 25].

9 Conclusion

This paper presents Herald, a runtime embedding scheduler for efficient cache-enabled recommendation model training. By leveraging the predictability and infrequency of embedding cache accesses, Herald applies an adaptive location-aware inputs allocation mechanism and an on-demand synchronization strategy to reduce the embedding transmissions between workers and PS during training. Extensive experimental results show that Herald can significantly reduce the embedding communication overhead and thus boost the end-to-end recommendation model training.

Acknowledgments

We thank the anonymous NSDI reviewers and our shepherd Mohammad Alizadeh for their constructive comments. We thank Yilun Jin for insightful discussions, and appreciate HET maintainers for their timely responses to our issues. This work is supported in part by Hong Kong RGC TRS T41-603/20R, GRF 16213621, ITF ACCESS, NSFC 62062005, Key-Area Research and Development Program of Guangdong Province (2021B0101400001), and the Turing AI Computing Cloud (TACC) [46]. Kai Chen is the corresponding author.

References

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

[2] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. Understanding training efficiency of deep learning recommendation models at scale. In *HPCA*, 2021.

[3] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J Nair. Accelerating recommendation system training by leveraging popular choices. In *VLDB*, 2021.

[4] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J Nair. Heterogeneous acceleration pipeline for recommendation system training. In *arXiv*, 2022.

[5] Saurabh Agarwal, Ziyi Zhang, and Shivaram Venkataraman. Bagpipe: Accelerating deep recommendation model training. In *arXiv*, 2022.

[6] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *arXiv*, 2015.

[7] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *DLRS*, 2016.

[8] Minsik Cho, Ulrich Finkler, and David Kung. Blueconnect: Novel hierarchical all-reduce on multi-tired network for deep learning. In *SysML*, 2019.

[9] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *RecSys*, 2016.

[10] CriteoLabs. Criteo display ad challenge. https://www.kaggle.com/c/criteodisplay-ad-challenge.

[11] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *SIGCOMM*, 2021.

[12] Saeed Ghadimi, Guanghui Lan, and Hongchao Zhang. Mini-batch stochastic approximation methods for nonconvex stochastic composite optimization. In *Math. Program.*, 2016.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. In *MIT press*, 2016.

[14] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. In *arXiv*, 2017.

[15] Huifeng Guo, Wei Guo, Yong Gao, Ruiming Tang, Xiuqiang He, and Wenzhi Liu. Scalefreectr: Mixcache-based distributed training system for ctr models with huge embedding table. In *SIGIR*, 2021.

[16] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. Deepfm: a factorization-machine based neural network for ctr prediction. In *arXiv*, 2017.

[17] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. In *TIIS*, 2015.

[18] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. In *MLSys*, 2019.

[19] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *WWW*, 2017.

[20] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. In *MLSys*, 2019.

[21] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, et al. Xdl: an industrial deep learning framework for high-dimensional sparse data. In *DLP-KDD*, 2019.

[22] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *OSDI*, 2020.

[23] Kaggle. Avazu mobile ads ctr. https://www.kaggle.com/c/avazu-ctr-prediction.

[24] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In *EuroSys*, 2019.

[25] Youngeun Kwon and Minsoo Rhu. Training personalized recommendation systems from (gpu) scratch: look forward not backwards. In *ISCA*, 2022.

[26] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *NeurIPS*, 2014.

[27] Shengwei Li, Zhiquan Lai, Dongsheng Li, Yiming Zhang, Xiangyu Ye, and Yabo Duan. Embrace: Accelerating sparse communication for distributed training of deep neural networks. In *ICPP*, 2022.

[28] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, et al. Persia: An open, hybrid system scaling deep learning-based recommenders up to 100 trillion parameters. In *SIGKDD*, 2022.

[29] Zhuoran Liu, Leqi Zou, Xuan Zou, Caihua Wang, Biao Zhang, Da Tang, Bolin Zhu, Yijie Zhu, Peng Wu, Ke Wang, et al. Monolith: real time recommendation system with collisionless embedding table. In *arXiv*, 2022.

[30] Michael Lui, Yavuz Yetim, Özgür Özkan, Zhuoran Zhao, Shin-Yeh Tsai, Carole-Jean Wu, and Mark Hempstead. Understanding capacity-driven scale-out neural recommendation inference. In *ISPASS*, 2021.

[31] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. Het: Scaling out huge embedding model training via cache-enabled distributed framework. In *VLDB*, 2021.

[32] Michael Mitzenmacher. The power of two choices in randomized load balancing. In *TPDS*, 2001.

[33] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *ISCA*, 2022.

[34] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, et al. Deep learning training in facebook data centers: Design of scale-up and scale-out systems. In *arXiv*, 2020.

[35] NVIDIA. Collective communications library (nccl). https://developer.nvidia.com/nccl.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

[37] Cèdric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. Sparcml: High-performance sparse communication for machine learning. In *SC*, 2019.

[38] Haidong Rong, Yangzihao Wang, Feihu Zhou, Junjie Zhai, Haiyang Wu, Rui Lan, Fan Li, Han Zhang, Yuekui Yang, Zhenyu Guo, et al. Distributed equivalent substitution training for large-scale recommender systems. In *SIGIR*, 2020.

[39] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. In *arXiv*, 2018.

[40] Marcelo Tallis and Pranjul Yadav. Reacting to variations in product demand: An application for conversion rate (cr) prediction in sponsored search. In *arXiv*, 2018.

[41] Xinchen Wan, Hong Zhang, Hao Wang, Shuihai Hu, Junxue Zhang, and Kai Chen. Rat - resilient allreduce tree for distributed machine learning. In *APNet*, 2020.

[42] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. Towards domain-specific network transport for distributed dnn training. In *NSDI*, 2024.

[43] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. Deep & cross network for ad click predictions. In *ADKDD*, 2017.

[44] Zehuan Wang, Yingcan Wei, Minseok Lee, Matthias Langer, Fan Yu, Jie Liu, Shijie Liu, Daniel G Abel, Xu Guo, Jianbing Dong, et al. Merlin hugectr: Gpu-accelerated recommender system training and inference. In *RecSys*, 2022.

[45] Jiacheng Xia, Gaoxiong Zeng, Junxue Zhang, Weiyan Wang, Wei Bai, Junchen Jiang, and Kai Chen. Rethinking transport layer design for distributed machine learning. In *APNet*, 2019.

[46] Kaiqiang Xu, Xinchen Wan, Hao Wang, Zhenghang Ren, Xudong Liao, Decang Sun, Chaoliang Zeng, and Kai Chen. Tacc: A full-stack cloud computing infrastructure for machine learning tasks. *arXiv preprint arXiv:2110.01556*, 2021.

[47] Xinyang Yi, Yi-Fan Chen, Sukriti Ramesh, Vinu Rajashekhar, Lichan Hong, Noah Fiedel, Nandini Seshadri, Lukasz Heldt, Xiang Wu, and Ed H Chi. Factorized deep retrieval and distributed tensorflow serving. In *MLSys*, 2018.

[48] Chunxing Yin, Bilge Acun, Carole-Jean Wu, and Xing Liu. Tt-rec: Tensor train compression for deep learning recommendation models. In *MLSys*, 2021.

[49] Chaoliang Zeng, Xiaodian Cheng, Han Tian, Hao Wang, and Kai Chen. Herald: An embedding scheduler for distributed embedding model training. In *APNet*, 2022.

[50] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters. In *ATC*, 2017.

[51] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. Is network the bottleneck of distributed training? In *NetAI*, 2020.

[52] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. In *MLSys*, 2020.

[53] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. Deep interest evolution network for click-through rate prediction. In *AAAI*, 2019.

[54] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *SIGKDD*, 2018.

A The Problem Definition of Embedding Scheduling

Prerequisites. Considering a distributed learning system with n workers and a per-worker batch size of m, the overall batch size is $n \times m$. The embedding table length, i.e., the total number of embeddings in the table, is d. The total number of iterations is T. The goal of embedding scheduling is to minimize the communication overhead, specifically the operation time of push and pull operations between workers and the PS.

For modeling simplicity, we delay the operation of the embedding push to the next iteration without violating the BSP training. In this model, an iteration proceeds with the following operations: retrieving the batch inputs, pushing the embeddings, pulling the embeddings, DLRM forward propagation, DLRM backward propagation, and dense parameter synchronization. There exists a synchronous barrier between two operations. Finally, we can model the problem as a Markov Decision Process (MDP) with the following components.

State space $\mathcal{S}$. An iteration can be described by a state including the training batch of the current iteration and the current state of embeddings. The training batch is related to the dataset and the current iteration number. The state of embeddings S is a matrix with d rows and $n+1$ columns, where the value of $s_{i,j}$ is 1 if the j-th worker ($j = n+1$ refers to the PS) has the latest version of the i-th embedding, a value between 0 to 1 if the j-th worker has a non-aggregated gradient (a partially aggregated gradient for PS) of the i-th embedding, and 0 for the rest.

Action space $\mathcal{A}$. The action space is defined by three action matrices: a sample assignment matrix P, an embedding push matrix Q, and an embedding pull matrix R. State actions are enforced based on these three matrices sequentially.

P is a matrix with $n \times m$ rows and n columns, where the value of $p_{i,j}$ is 1 if the i-th sample is assigned to the j-th worker, and 0 otherwise. Each sample can only be assigned to one worker, i.e., $\sum_j p_{i,j} = 1$. The sample assignment matrix P can transfer to the embedding assignment matrix U according to the accessing embeddings of each sample. The embedding assignment matrix U is a matrix with d rows and n columns, where the value of $u_{i,j}$ is 1 if the i-th embedding is assigned to the j-th worker, and 0 otherwise.

Q is a matrix with d rows and n columns, where the value of $q_{i,j}$ is 1 if the j-th worker pushes the i-th embedding to the PS, and 0 otherwise. Q should preserve the availability of the latest version of embeddings at PS if this embedding is not available by its assigning workers:

$$q_{i,j} = 1, \text{ if } 0 < s_{i,j} < 1 \text{ and } \Sigma_{j' \in n} u_{i,j'} >= 1$$
$$\text{or } s_{i,j} = 1 \text{ and } s_{i,n+1} = 0 \text{ and } \Sigma_{j' \in n, j' \neq j} u_{i,j'} >= 1$$

R is a matrix with d rows and n columns, where the value of $r_{i,j}$ is 1 if the j-th worker pulls i-th embedding from the PS, and 0 otherwise. R is subject to two rules that ① the worker

should pull the assigning embedding if it does not maintain the latest version of this embedding, and ② the PS has the latest version of the embedding when there is a pull request:

$$① \; r_{i,j} = 1, \text{ if } u_{i,j} = 1 \text{ and } s_{i,j} \neq 1$$
$$② \; r_{i,j} = 1 \Rightarrow s_{i,n+1} = 1$$

There is an implicit assumption that the PS has the latest version of the i-th embedding when performing rule ①. This assumption is maintained by Q.

State transition. The transition of the training batch is trivial. In terms of the state of embeddings, the transition is as follows: for $j \in [1,n]$ (workers),

$$s'_{i,j} = \begin{cases} \frac{1}{\Sigma_{j' \in n} u_{i,j'}}, & \text{if } u_{i,j} = 1 \\ 0, & \text{if } u_{i,j} = 0 \text{ and } \exists j' \text{ such that } u_{i,j'} = 1 \\ & \text{or } \Sigma u_{i,j} = 0 \text{ and } q_{i,j} = 1 \text{ and } s_{i,j} \neq 1 \\ s_{i,j}, & \text{otherwise} \end{cases}$$

for $j = n+1$ (PS),

$$s'_{i,n+1} = \begin{cases} 0, & \text{if } \Sigma_{j' \in n} u_{i,j'} >= 1 \\ min(1, s_{i,n+1} + \Sigma_{\forall j', q_{i,j'}=1} s_{i,j'}), & \text{otherwise} \end{cases}$$

Reward function $\mathcal{R}$. The reward is the negative of the communication latency. We assume the latency to push or pull an embedding remains the same. Since we preserve a strict order between the operations of push and pull, and the latency of these two operations depends on the worker with the largest workload, we define the reward function as follows:

$$\mathcal{R}(s,a) = -[\max_j \sum_{i=1}^{d} q_{i,j} + \max_j \sum_{i=1}^{d} r_{i,j}]$$

Objective. Based on the above-defined MDP, our objective is to find a policy $\pi(\mathcal{S}) \rightarrow \mathcal{A}$ that maps states to actions, which maximizes the expected cumulative reward $\mathbb{E}\left[\sum_{t=0}^{T} \mathcal{R}(\mathcal{S}_t, \mathcal{A}_t)\right]$.

A conventional solution to solve the above problem is dynamic programming. However, it does not work well in practice given the curse of dimensionality (the number of iterations in this problem), not to mention the fact that sometimes it is difficult to retrieve the entire dataset in advance from the streaming data and hard to predict the number of training iterations. Herald focuses on single batch scheduling, i.e., finding a quality action for the current state to get a high reward. Therefore, both the pull operation and the push operation perform on demand, and existing cache systems (like HET [31]) work with on-demand embedding pulls by design. Moreover, delaying the push operation after retrieving the batch inputs may be different from the practice, where the embedding push operation happens during the synchronization of the previous iteration. According to our analysis, the push action (Q) is mainly affected by the sample assignment. Therefore, in practice, we can decide on the push operation with an early sample assignment for the next iteration.

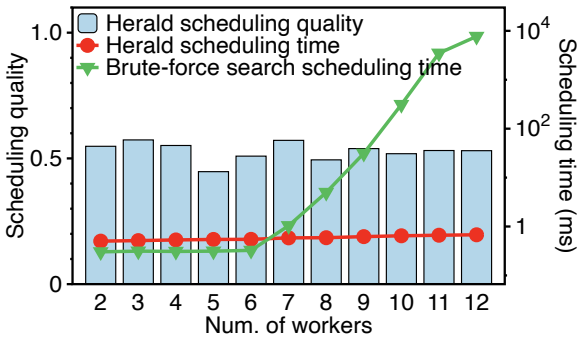


Figure 20: Herald vs. brute-force search (scheduling time is in log scale).

B Point-to-point Embedding Synchronization

In this paper, we follow the same distributed cache model as HET's [31], where each worker only communicates with PS. In this cache model, embedding synchronization requires at least two steps, one worker push and one worker pull. We can reduce the synchronization path to one step by direct P2P synchronization between two workers. Moreover, P2P embedding synchronization can eliminate the potential network bottleneck caused by the PS architecture [41]. Herald can support P2P embedding synchronization by distinguishing synchronization targets during the communication plan generations. These targets finally become a receiving communication plan to indicate which embeddings would be received in an iteration for each worker.

C Herald vs. Brute-force Scheduling

To evaluate the effectiveness of Herald scheduler, we also implement a scheduler that leverages brute-force search as the baseline. The brute-force search scheduler explores all possible allocations and returns an allocation that incurs minimal embedding transmissions. Since the search space expands exponentially with the number of workers and per-worker batch size, we restrict the per-worker batch size to 1. We define the scheduling quality of Herald as the ratio of the number of embedding transmissions incurred by the brute-force search scheduler to that incurred by Herald.

We measure the scheduling quality and the scheduling time across different numbers of workers and report their average values for 90 iterations after 10 warm-ups used to fill the cache. For easy parallelism, the thread pool size is identical to the number of workers for both schedulers. Figure 20 demonstrates that Herald can achieve a scheduling quality from 0.45 to 0.57, indicating that there is still room to exploit embedding scheduling in the future. In terms of scheduling time, Herald consistently preserves a low scheduling overhead (< 0.7 ms). Meanwhile, the scheduling time of the brute-force search scheduler increases rapidly, consuming up to a few seconds when there are more than 10 workers in this tiny search space. This result reveals that the brute-force search scheduler is impractical for large-scale DLRM training.

DISTMM: Accelerating Distributed Multimodal Model Training

Jun Huang*
The Ohio State University

Zhen Zhang
Amazon Web Services

Shuai Zheng
Boson AI, Inc

Feng Qin
The Ohio State University

Yida Wang
Amazon Web Services

Abstract

Multimodal model training takes multiple types of inputs to process with differently structured submodules, and aggregates outcomes from the submodules to learn the relationship among various types of inputs, e.g., correlating text to image for text-to-image generation. The differences of submodule architectures as well as their inputs lead to heterogeneity in terms of computation efficiency. Failing to account for such heterogeneity, existing distributed training systems treat all submodules as a monolithic entity and thus have sub-optimal performance. Moreover, the outcome aggregation phase introduces cross-sample dependencies with contrasting positive and negative sample pairs (i.e., contrastive loss). Such dependencies make the existing pipeline parallelism scheduling algorithms not applicable for multimodal training with contrastive loss.

To address the limitations of existing solutions, we propose DISTMM. For a given multimodal model, DISTMM exploits the heterogeneity among submodules, applying different distributed parallelism strategies for each submodule, e.g., using Tensor Parallelism for a computation-intensive submodule, and Data Parallelism for a submodule with a small number of parameters. DISTMM balances the computation of parallelized submodules to reduce the computing resource idle time of waiting for the slowest submodule. DISTMM further optimizes the locality of submodules by leveraging the heterogeneous bandwidth of interconnections among accelerators. To address the limitation of existing pipeline execution schedules, we propose a new pipeline execution primitive, called batch-sync instruction, and a corresponding schedule, called DISTMM-Pipe. We build a prototype of DISTMM and evaluate it with existing solutions on models with various sizes ranging from 1.1 billion to 26 billion parameters and observe 1.32-3.27× speedup over Megatron-LM.

*Work was done when interned at AWS.

1 Introduction

Deep learning has recently witnessed the rise of foundation unimodal models focused on processing unimodal data such as GPT [4], LLaMA [36] and ViT [9], where models are pre-trained on web-scale text or image data. However, real-world data typically do not exist in isolation so human experience itself is multimodal. For example, a movie is a blend of visuals, dialogue, and music. To bridge the gap between isolated data modalities, multimodal models have emerged as a transformative paradigm, which offers a holistic representation to enhance performance in real-world scenarios, capturing intricate relationships that remain elusive to unimodal models.

A multimodal model has unique characteristics in terms of model structure, which consists of multiple submodules. Each submodule has a specific functionality, e.g., transforming image inputs to feature vectors, or combining multiple feature vectors, etc. As the functionalities are different, the architecture and the scale of the submodules vary accordingly. For example, Contrastive Language–Image Pre-training (CLIP) [29] adopts a deeper and wider structure for processing image inputs, while using a relatively shallow design of a submodule for transforming text inputs. Furthermore, the data input sizes also vary across submodules, e.g., the images in Common Objects in Context (COCO) dataset [20] have an input size of 512×512 pixels, while the input length of corresponding text captions is from 5 to 20 words. The heterogeneity among submodules leads to different demands of computing powers. However, existing distributed training systems are mainly designed for training unimodal models with homogeneous computing power requirements, leading to suboptimal performance in multimodal model training.

Moreover, multimodal models require a large number of data samples to compute the contrastive loss function due to model quality requirements [5]. In the distributed training setting, scaling out multimodal model training with data parallelism and tensor parallelism linearly increases the maximum batch size for contrastive loss. However, scaling out multimodal model training with pipeline parallelism results

in a bounded maximum batch size for contrastive loss per forward and backward. The absence of pipeline parallelism support hinders the scalability of the multimodal model both in training speed and model size.

To address the aforementioned challenges (more details in Sec 3) for efficient multimodal model training, this paper proposes DISTMM, which is, to the best of our knowledge, the first distributed training system specifically designed for multimodal models. DISTMM introduces four components that work collaboratively to optimize multimodal model training. For a given multimodal model training task and a cluster configuration, *the modality-aware partitioner* of DISTMM adaptively applies parallelism strategies for different submodules to minimize the overhead from parallelization. With efficient parallelism for each submodule, *the load balancer* redistributes the training data to ensure balanced computations among the submodules, which minimizes the computing resource idle time of waiting for the slowest submodule. *The placement manager* then assigns devices for each submodule based on the parallelism solution and the interconnect topology of devices, so that DISTMM can utilize heterogeneous bandwidth. Finally, to achieve a targeted sample size required by the training task, *the pipeline executor* generates the execution schedule (i.e., when and what to do for computations, communications, or memory movements) for combining outputs across multiple micro-batches of input data.

In summary, we make the following contributions:

- We propose an idea to partition and parallelize the submodules of a multimodal model based on their modalities and redistribute the training data, resulting in balanced computation among submodules with high computation efficiency.
- We design a submodule placement mechanism to reduce the communication volume by aligning the model's heterogeneity with the heterogeneous bandwidth in the cluster.
- We propose a new pipeline parallelism instruction to increase the supported batch size by fully utilizing the memory capacity, and a corresponding pipeline parallelism schedule to avoid dependency overheads introduced by the new instructions.
- Based on the above ideas, we build DISTMM and evaluate it on eight Amazon EC2 p3.16xlarge instances with 64 GPUs. Our results show that DISTMM achieves 1.32-3.27× speedup over Megatron-LM on three structurally different multimodal models, i.e., CLIP, CoCa, and LiT.

2 Background

2.1 Unimodal and Multimodal Models

Unimodal models. The unimodal models are designed to interpret and learn from one specific type of data modality, e.g., text, images, or audio. The model architecture is thus homogeneous with sequential execution order for consistently processing a single type of data. A typical unimodal model architecture consists of multiple transformer-based layers [4, 7, 36, 37]. Within each layer, the structure is the same, i.e., attention mechanisms and feed-forward networks.

Multimodal models. On the other hand, multimodal models aim to combine and align multiple data modalities, e.g., processing image and text data simultaneously. It relies on specialized architectures or configurations for processing each modality and aligning the processed feature vectors [9,29,38]. Typically, a multimodal model consists of multiple submodules each processing one modality independently. As an example, CLIP [29] employs two specialized modal submodules, namely, text and image. The image submodule is a Vision Transformer (ViT) [9] while the text submodule is a traditional Transformer submodule [37]. On top of multiple modal submodules, the modality interactive submodule takes the outputs from each modal submodule to compute the correlation among them. An exception is the Multimodal Large Language Model (MLLM) [1,21,41], whose LLM submodule integrates both the functionality of modality interactive submodule and text modal submodule.

2.2 Multimodal Model Training

To perform various multimodal tasks, multimodal models require different learning approaches. Fusion models integrate multimodal data to perform tasks including classification, detection, or prediction. This integration is either conducted through contrastive learning or simply achieved by fusing the features through operations such as cross-attention [38]. Co-learning models leverage information from one modality to improve or supplement the learning in another modality. It is achieved by contrastive learning to identify a unified representation containing multimodal information. Generative models generate content based on contextual understanding across multiple data modalities. For generative models such as MLLM, modal submodules are trained separately with contrastive learning first, and then the whole model will be trained with multimodal instruction tuning [21,41].

Contrastive learning. Out of all multimodal learning methods used in various multimodal models, contrastive learning is the most fundamental learning method. Since aligning the representation across features of different modalities is crucial for the downstream task, contrastive learning generates positive and negative samples to contrast them to learn a robust unified representation across different modalities. Compared to cross-modal learning or reconstruction-based representation learning, contrastive learning is more effective by bridging the processing abilities between different modal submodules according to previous study [38]. Most multimodal models adopt contrastive learning [5] as the only training method or combine it with other unimodal learning methods.

Unlike only comparing each sample's output feature vector

with the corresponding label in unimodal learning, contrastive learning systematically compares the feature vector of each data point with all feature vectors across diverse modalities. Specifically, positive pairs are constructed by comparing the feature vectors of a single sample across different modalities, affirming their shared identity. Conversely, negative pairs are formed by comparing feature vectors from distinct samples and modalities, establishing dissimilarity. Multimodal models learn effective representations and understand relationships between modalities by distinguishing positive and negative pairs. To achieve this, the core of contrastive learning is to compute a similarity matrix, which measures pairwise similarities between feature vectors. During multimodal model training, the loss of contrastive learning maximizes the similarity scores of positive pairs and minimizes the similarity scores of negative pairs. For example, CLIP model performs contrastive learning by using dot products of their text and vision submodules' feature vectors to compute a similarity matrix. With similarity matrix, the objective of CLIP training is to minimize a cross-entropy loss over similarity scores.

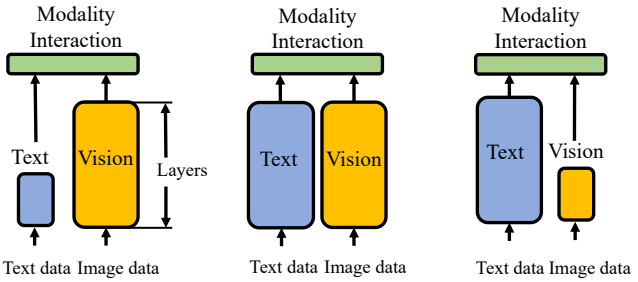
The effectiveness of contrastive learning relies on a large number of positive and negative samples from feature vectors generated by the same model update [2]. For instance, SimCLR [6] used a batch size of 4096 to generate 4096 positive samples and 33,554,432 negative samples. CLIP [29] used a batch size of 32,768 to generate 32,768 positive samples and 2,147,483,648 negative samples. Some low-cost alternatives utilize feature vectors from the previous model updates to approximate comparison in a large batch size. One problem of such approximation is the downgraded model quality since feature vectors are no longer consistent. PIRL [24] used a memory bank to store the previous feature vectors. Meanwhile, MoCo [12] replaced the memory bank with a queue of feature vectors generated by its momentum encoders. According to theory in [17], a larger number of consistent positive and negative samples enhance the model's discriminative power, generalization ability, and understanding of complex cross-modal relationships.

3 Motivation

This section discusses the problems in existing solutions and corresponding objectives in designing a multimodal model training system. The existing solutions did not take the unique model architecture and special learning paradigm of multimodal models into account.

3.1 Multimodal Model Characteristics.

Heterogeneous submodules. As discussed in Section 2.1, multimodal models have multiple modal submodules and one or more modality interactive submodules. Influenced by the design choices and architectural complexity, the sizes of submodules in multimodal models can vary significantly, which introduces heterogeneity.



(a) Vision > Text (b) Vision = Text (c) Text > Vision

Figure 1: Three categories of vision-language models. The height of each rectangle denotes the number of layers and the width denotes hidden dimension of each layer. Text and Vision denote text submodule and vision submodule.

Each modality (e.g., text, image, and audio) has unique characteristics and features that may require specialized processing to effectively capture modality-specific patterns. The focus of the multimodal task may also influence the processing strategy, which requires more understanding of fine-grained details on one modality than on others. The inherent complexity of each modality and the task requirements result in different computational demands among the modal submodules. Figure 1 illustrates three different archetypes of vision-language models. CLIP model belongs to Figure 1(a). The heterogeneity of CLIP model is driven by its vision-focused requirements, leading to a larger vision submodule. The LiT [10] model belongs to the category of Figure 1(c), whose text submodule has more parameters. The Contrastive captioner (CoCa) [38] model falls in the category of Figure 1(b), which uses equal-sized vision and text submodules.

Heterogenous submodules have varied GPU utilization. The parameter size differences result in varied hidden dimensions between heterogenous submodules, which lead to differently scaled operations to execute. The scale of operations affects the choices of kernels for underlying libraries. Large-scale operations are likely to be represented with more efficient kernels than small-scale operations. Additionally, fixed overheads such as kernel launches and memory transfers are amortized by large-scale operations. The training efficiency is limited by the submodules with fewer parameters, as they may not utilize the hardware as efficiently as the larger submodules.

Imbalanced input sizes. Depending on the modality type, the input sizes can vary significantly as well, e.g., the text input length is 77 words while the image input size is 512×512 pixels for CLIP training. Larger input sizes lead to more efficient computation by leveraging the parallelism and vectorization capabilities of the hardware. This results in optimized memory utilization and computational throughput. Within the

multimodal model consists of submodules with imbalanced inputs, the training efficiency is limited by the submodule with smaller input sizes.

Large batch size requirement. According to the observations in [5, 29, 39] and the theory in [17], a larger number of negative and positive samples are beneficial for learning robust and generalizable representations in contrastive learning, which directly results in a trained model with higher quality (i.e., more accurate, more robust, and more reliable). Furthermore, prior study [6] shows that training with a larger number of negative and positive samples not only converges faster but converges to a model that performs better, no matter how long it is trained. Increasing the batch size for the similarity matrix is the only way to generate the sufficient number of negative and positive samples without bringing inconsistency. For example, given a batch size of N, there are N positive pairs and $N^2 - N$ negative pairs in a similarity matrix. As a result, multimodal model training in practice requires a large batch size to ensure expected model quality.

3.2 System Challenges.

Although 3-dimension (3D) parallelism strategies (data, tensor, and pipeline parallelism) are effective for most unimodal models, it is inefficient to apply them to train multimodal models. We first lay out the reasons why applying the 3D parallelism to multimodal model training is inefficient, then conclude with the objectives on how to optimize 3D parallelism for multimodal models.

Memory overhead and imbalanced computation of Data Parallelism (DP). For models that can easily fit into one device, data parallelism is the go-to strategy to parallelize the training. Data parallelism only partitions batch dimensions and evenly distributes the data batch to all devices. Applying data parallelism to a multimodal model training results in all submodules being colocated on each device. Due to the colocation, the heterogeneity and imbalanced input sizes among different submodules lead to uneven computation scales and hardware utilization. Moreover, the colocation of different submodules also reduces the available memory per submodule, which limits the batch size for computation and eventually reduces the computation efficiency. In conclusion, the heterogeneous nature of the multimodal model cannot be fully utilized under the colocation solution of data parallelism. Therefore, it is needed to have a non-colocation solution to support more efficient multimodal model training.

Unnecessary partition overheads of Tensor Parallelism (TP). For models that cannot fit into a single device, tensor parallelism is used to evenly split the model and parallelize the model partition's execution. For multimodal model training, tensor parallelism homogeneously partitions submodules with different sizes according to the entire model's memory consumption. Since the smaller submodule only contributes little to the entire model's memory consumption, tensor parallelism will overly partition the smaller submodule, leading to unnecessary overhead. To address this issue, it is needed to have an adaptive partitioning method that considers the model sizes and structures of different submodules.

Unexpected training semantics with Pipeline Parallelism (PP). As discussed in 3.1, multimodal model's model quality is associated with the training batch size. However, all existing pipeline parallelism schedules, including 1F1B [11], GPipe [13], and PipeDream [25] split a global batch into multiple micro-batches and conduct pipelined forward and backward step on every micro-batch. Applying such pipeline parallelism to multimodal model training results in model quality degradation compared to non-pipeline parallelism approaches. Therefore, a new pipeline parallelism scheme for multimodal model training is needed.

4 DistMM Overview

DistMM is a distributed training system optimized for multimodal model training workloads by tackling the challenges discussed in Section 3.2. The design of DistMM improves the computation efficiency of each submodule in a multimodal model, reduces communication overheads in multi-node distributed training, and ensures the model quality when pipeline parallelism is used for training large models. By design, DistMM treats submodules within a model separately with independent parallelism strategies to maximize efficiency. In addition, it aligns the computation duration of each submodule to minimize the cost of interactions among submodules. As shown in Figure 2, DistMM has four components, which are Modality-aware partitioner, Data load balancer, Heterogeneity-aware placement manager, and Pipeline executor. We overview the components in the following text and leave the detailed design and analysis in Section 5.

Modality-aware partitioner. Modality-aware partitioner splits the whole multimodal model into submodules based on their input modalities. The neural network architecture and configuration of each submodule are typically different from each other, due to the input modality differences, e.g., using ViT [9] for vision inputs and BERT [7] for text inputs. Based on the submodule sizes, Modality-aware partitioner applies independent parallelism strategies to avoid overprovisioning parallelism degrees for small submodules. Modality-aware partitioner only handles model-level partitioning by transforming model description input into submodule partitions, leaving batch dimension partitioning to Data load balancer. In principle, this allows DistMM to maintain high computation efficiency for each submodule.

Data load balancer. Without realizing the structural differences, a naive way is to assign the same amount of computing

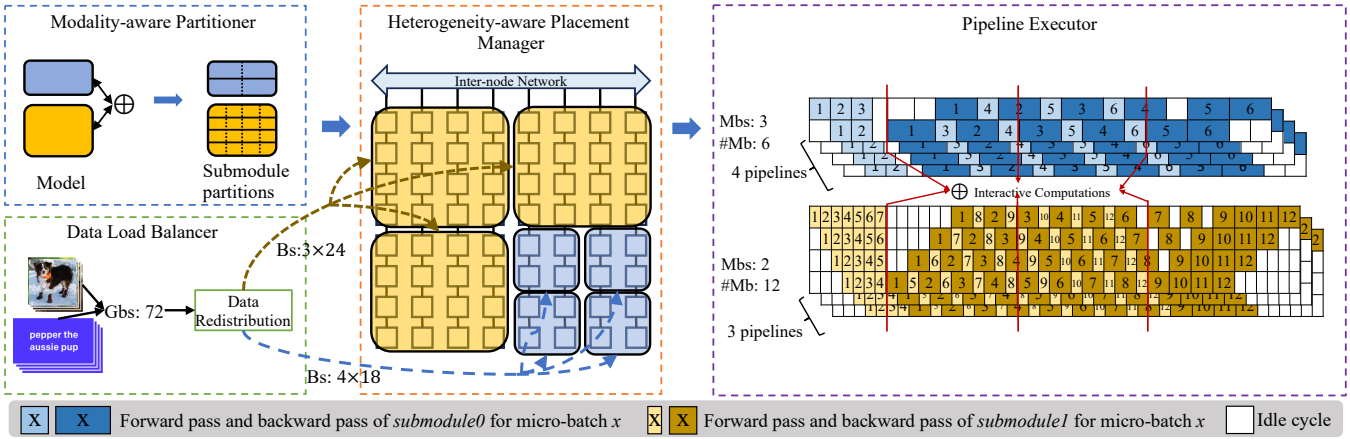


Figure 2: DISTMM overview.

devices and utilize the same batch size across all the submodule partitions of different modalities. This can easily lead to computing resources idling at the interactive computation phase because some submodules need fewer computation cycles than others. To achieve overall high computation efficiency, submodule partitions require a careful design for the resource allocation and adjustment of the batch size for each modality input. Data load balancer takes submodule partitions and cluster configuration as inputs, assigning different numbers of devices to different submodule partitions based on their estimated memory consumption and measured execution time. To minimize device idle time introduced by the heterogeneous resource placement, Data load balancer redistributes batch sizes of different modalities to balance the computation duration.

Heterogeneity-aware placement manager. Computing clusters typically have much higher bandwidths for intra-node communication (e.g., NVLink) than that for inter-node communication (e.g., Ethernet). Heterogeneity-aware placement manager groups submodule partitions within the same modality and places them close to each other to exploit the high-bandwidth links for frequent communication. Furthermore, Heterogeneity-aware placement manager places submodule partitions from different modalities on separate nodes so that the infrequent communication volume (e.g., gradient AllReduce) on low-bandwidth links will be reduced.

Pipeline executor. Pipeline executor generates a customized pipeline parallelism execution schedule for each submodule. The customized schedule maintains the semantics of the training procedure and minimizes hardware idle time. Pipeline executor inserts *mid-point synchronization* to maintain the semantics of the modality interaction among different modal submodules, in which each submodule gathers activations of prior micro-batches (from last mid-point synchronization till now) from submodules with different modalities and itself to complete the modality interaction.

An illustrated example. Figure 2 provides an example of training a vision-language model with DISTMM in a cluster with 64 devices across eight nodes. Assuming the multimodal model has more parameters for encoding image data (yellow) than that of processing text inputs (blue).

Modality-aware partitioner assigns different parallelism strategies for each modal submodule, shown in the top-left part. Different submodules are replicated differently (four versus three) with different numbers of devices by Data load balancer, so that submodules can complete the computation of different batch sizes (18 versus 24) for a given global batch at the same time.

Heterogeneity-aware placement manager deploys vision submodules and text submodules on different devices and groups the devices with the same modality, which reduces both intra-node and inter-node communication volume.

Pipeline executor generates customized pipeline execution plans for each device which integrates a synchronization instruction to preserve the same training semantics with non-pipeline parallelism solution. The synchronization instruction gathers feature vectors to conduct interactive computation. In this case, a gathered global image feature vector consists of 3 pipeline parallelism groups' 4 previous image feature vectors, where each image feature vector's batch size is 2. Similarly, a gathered global text feature vector consists of 4 pipeline parallelism groups' 2 previous text feature vectors, where each text feature vector's batch size is 3. The workloads on submodules for different modalities are balanced by Data load balancer which ensures that they arrive at the synchronized interactive computation at the same time.

5 DistMM Design

5.1 Modality-aware Partitioner

The heterogeneity of the model structure and the difference of the inputs across modalities are the root cause of the low overall computation efficiency in multimodal training (detailed in Section 3.1). Modality-aware partitioner therefore applies adaptive partition strategy to improve the computation efficiency, which accounts for heterogeneity among submodules.

Adaptive Partition Allocation. We design a new partition strategy that fully utilizes the heterogeneous structure with the multimodal model. Modality-aware partitioner takes model definition as an input, which contains the configuration of each modal submodule and modality interactive submodule.

Modality-aware partitioner first parallelizes submodules with independent parallelism strategies to satisfy memory constraints while maintaining low overheads. As discussed in 3.1, modality interactive submodule focuses on aligning high-level information, which does not require a complex computation. It therefore has a highly parallelizable nature. So Modality-aware partitioner equally distributes the computation load of modality interactive submodule to every device.

Modality-aware partitioner then individually applies adaptive partitioning to each modal submodule to maximize efficiency. This component follows the common practice [15,34] of efficiently training an unimodal model, which involves determining the minimum parallelism degree required and selecting parallelism strategies accordingly, with careful consideration of the model size. For modal submodules that can fit into a single device, our approach advocates the use of data parallelism to maximize efficiency. Alternatively, for modal submodules that cannot fit within a single device, employing tensor parallelism is recommended. Notably, for modal submodules that surpass the capacity of a single node, our strategy favors pipeline parallelism, as extant research [26] indicates that extending tensor parallelism across nodes introduces significant overheads. The resulting model partition on each device after Modality-aware partitioner is an adaptively partitioned modal submodule and evenly parallelized modality interactive submodules.

An illustrative example. We construct examples with hypothetical performance numbers to illustrate how Modality-aware partitioner works. Figure 3 (a) shows an example of applying the tensor parallelism in a homogeneous way for the entire model. This homogeneous way of applying tensor parallelism leads to the sequential execution of the first and smaller submodule (blue colored) with 10% GPU utilization, and the second and larger submodule (orange colored) with 30%. As a result, its average GPU utilization for each GPU is $2 \cdot 10\% + 3 \cdot 30\% = 22\%$, where the smaller submodule's 10% utilization is the performance bottleneck. Figure 3 (b) shows the partition result by Modality-aware partitioner, where the

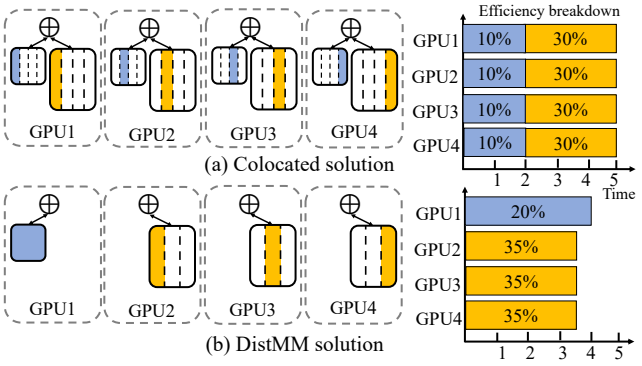


Figure 3: Modality-aware partitioner example. Placements (on left) describe how submodules are partitioned. ⊕ denotes distributed modality interactive submodules. Efficiency breakdown (on right) includes each submodule's duration with its GPU utilization. The x-axis stands for each submodule's duration and y-axis represents the GPU id. Percentages in the duration stand for each submodule's GPU utilization.

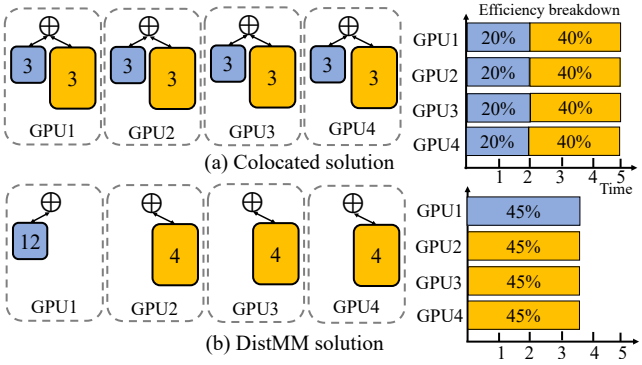


Figure 4: Data load balancer example. Placements (on left) describe the data sample assignment for each submodule. Numbers inside the submodule are the number of assigned data samples per batch.

smaller submodule is not partitioned and the larger submodule is partitioned into three parts via the tensor parallelism. The separation of submodules and adaptive tensor parallelism partitioning allow the smaller submodule to achieve a much better GPU utilization. The GPU utilization of the smaller submodule can be improved from 10% to 20% due to the increase in computational workload. The improved average GPU utilization helps reduce the duration of the entire model. However, it also introduces additional overhead of imbalanced duration among GPUs (i.e., GPU1 versus other GPUs).

5.2 Data Load Balancer

While the adaptive partition strategy reduces partition overheads, it does not address the heterogeneity of inputs. Modality-aware partitioner does not assign the batch size of input data to submodule partitions. Different modal submod-

ule partitions have different computation patterns and inputs which results in different execution durations and memory consumptions under the same batch size. Without considering the differences, some modal submodules may finish first and have to wait for others to gather the outputs.

We view Data load balancer as a resource manager, and the goal is to balance the computing duration among different modal submodules under memory constraints. Data load balancer takes a cluster setup, i.e., the number of nodes and the number of devices per node, and training configuration, i.e., the model partitions generated by Modality-aware partitioner and the global batch size, as inputs. Note that each model partition consists of an adaptively partitioned modal submodule for a specific modality and an evenly partitioned modality interactive submodule. Data load balancer produces a resource assignment plan, which assigns the number of devices and the data batch size for each model partition. This resource assignment problem is equivalent to minimizing the time taken by the slowest model partition, and has the optimal substructure property: a resource assignment plan that maximizes throughput is composed of resource assignments for partial modalities that maximize throughput for smaller clusters. DISTMM uses dynamic programming to find the optimal solution.

An illustrative example. We construct examples with hypothetical performance numbers to illustrate how Data load balancer works. Figure 4 provides an example resource assignment plan from Data load balancer and compares it with a colocated plan. The colocated plan colocates multiple modal submodules and replicates each submodule onto all devices (Figure 4 (a)). In contrast, Data load balancer together with Modality-aware partitioner separates modal submodules with different modalities and replicates them differently, i.e., the smaller submodule has one replica, and the larger submodule has three replicas (Figure 4 (b)). With a smaller number of replicas for each submodule, the number of samples per device is increased, especially for the smaller submodule. Thus, the computation efficiency improves for both submodules.

5.3 Heterogeneity-aware placement manager

The generated resource assignment plan from Data load balancer does not include the model placement to specific devices. Heterogeneity-aware placement manager deploys the resource assignment plan by placing the submodules in a communication-efficient way. It takes bandwidth heterogeneity into consideration to optimize the communication overhead in distributed multimodal model training.

5.3.1 Intra-submodule placements

Heterogeneity-aware placement manager prioritizes the communication patterns within a single modal submodule for placement assignments. The priority is determined by the communication frequency and the data volume transmitted in each communication pattern. Heterogeneity-aware placement manager generates placement assignments where communications with higher priority (TP) are conducted in the network with higher bandwidth (NVLinks within node) and communications with lower priority (PP and DP) are conducted in the lower bandwidth network (Ethernet).

5.3.2 Inter-submodule placements

As a non-colocated solution, Heterogeneity-aware placement manager only places one single-modality partition together with evenly parallelized modality interactive submodules on a single device, which reduces the communication volume involved in modality interaction and gradient synchronization.

Overhead reduction in modality interaction. As described in section 2.2, multimodal model's modality interaction consists of similarity matrix computation. For the multimodal models with vision modality and text modality, the similarity computation on each device consists of two dot products in the distributed setting, the first dot product is between the local image feature vectors and gathered entire text feature vectors and the second dot product is between the local text feature vectors and gathered image feature vectors. With non-colocated placement, since there is only one modal submodule on each device, the similarity computation is reduced to one dot product between the feature vectors of local modality and the gathered feature vectors of opposite modality. So Heterogeneity-aware placement manager halves the communication volume of modality interaction compared with the colocated placement since image and text feature vectors share the same tensor shape.

Table 1: The list of symbols frequently used in the paper.

Symbol	Description
Mbs	Micro-batch size
Rbs	Required micro-batch size for modality interaction
K	Required number of micro-batches ($K = Rbs/Mbs$)
N	Total number of GPUs
P	The number of pipeline parallelism degree
M	Memory capacity per GPU
M_s	Static (Weight, gradient, state) memory consumption
M_g	Gradient memory consumption
M_a	Activation memory consumption

Note: Above symbols are of the whole multimodal model. We add index 1 and 2 to the above symbols to stand for the first and second submodules.

Overhead reduction in gradient synchronization. For communication in gradient all-reduce, Heterogeneity-aware placement manager reduces the communication volume from the entire model's gradients in colocated solution to the gradients of the submodules hosted in this specific device. In the whole

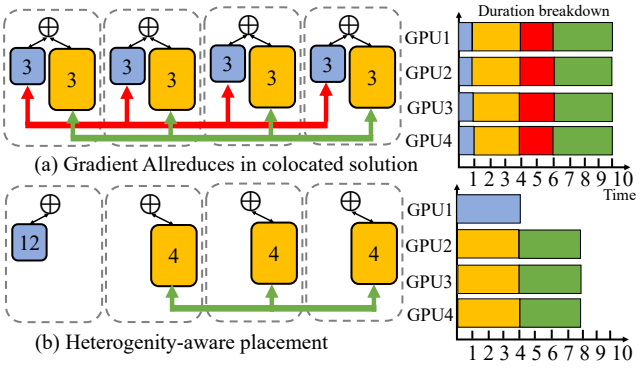


Figure 5: Communication reduction example. Placements (on left) describe the patterns of gradient AllReduce (green and red) under different placements. Duration breakdown (on right) includes the computation and communication durations within each device. The x-axis stands for each submodule's computation or communication duration.

system's perspective, the volume is reduced from $M_{g1}+M_{g2}$ to max (M_{g1}, M_{g2}). As shown in examples with hypothetical duration numbers described by Figure 5, gradient communication volume is reduced from the whole model's gradients (red and green) to only the gradient of the larger submodule (green). The gradient all-reduce volume does not change with the mini-batch size. The computing duration increases when the mini-batch size increases, which further reduces the communication duration ratio in total training time.

5.4 Pipeline Executor

When applying existing pipeline parallelism to multimodal model training, the available memory after static memory consumption (weight, gradient, and state) is divided into P blocks to store P micro-batches' activation to ensure efficient pipelining. Since static memory consumption and activation are divided into P partitions, the maximum micro-batch size is $((M-M_s/P)/(M_a/P))/P = (M-M_s/P)/M_a)$, which is bounded by M/M_a. Such bounded micro-batch size is much smaller than what is needed for training a quality multimodal model in practice. Note that Table 1 shows the meaning of the symbols used in this paper. As a comparison, data parallelism's maximum micro-batch size is $(M \cdot N - M_s \cdot N)/M_a) = N \cdot (M - M_s \cdot)/M_a)$ and the tensor parallelism's maximum micro-batch size is $(M \cdot N - M_s)/M_a)$, which means scaling out the cluster size N can linearly increase the maximum micro-batch size.

To address this issue, we propose a new instruction in pipeline parallelism called batch-sync instruction used by the Pipeline executor. This instruction can preserve the required semantics by ensuring the modality interactive submodule is executed with the needed large batch size. We also propose a new pipeline parallelism schedule called DISTMM-Pipe which adopts batch-sync instruction without introducing any extra idle cycles.

Batch-sync instruction. Given a model quality requirement of batch size Rbs and $K = Rbs/Mbs$, Pipeline executor's batch-sync instruction consists of the following four steps: (1) the memory movement step that concatenates K feature vectors computed by previous K forward instructions into a continuous feature vector, (2) the forward pass of modal interactive submodule is executed with the continuous feature vector, (3) the backward pass of modality interactive submodule is executed, which produces gradients corresponding to the continuous feature vector, and (4) the memory dispatching step that dispatches the continuous gradients to K gradients corresponding to the feature vectors of each micro-batch.

However, integrating batch-sync instruction to pipeline parallelism schedule introduces overhead due to dependency issues. Since the batch-sync instruction depends on the previous forward instructions and vice versa, the backward instructions depend on the batch-sync instruction. The batch-sync instruction works as a barrier between the involved forward and backward instructions. In 1 forward and 1 backward (1F1B) schedule [25], batch-sync instruction can only be inserted between the 1 forward and 1 backward of a single micro-batch due to dependency, which limits the maximum batch size for modality interaction to a single micro-batch. In GPipe schedule [13], batch-sync instructions can be inserted between the required number of forward and backward instructions. As shown in Figure 6, applying the batch-sync instruction without a customized schedule results in dependency issues, where forward and backward instructions are separated by the corresponding batch-sync instruction and cause extra idle cycles.

DISTMM-Pipe schedule. To bypass the dependency barrier so as to mitigate the overhead introduced by the batch-sync instruction, we propose DISTMM-Pipe schedule. DISTMM-Pipe launches $2 \cdot K$ micro-batches with $Mbs/2$ as the micro-batch size for each gradient accumulation cycle. The doubled micro-batches bypass the dependency barrier since each batch-sync instruction only introduces dependency issues between K forward and backward instructions. As shown in Figure 6, after each batch-sync instruction, backward instruction and forward instruction belonging to different K micro-batches are executed in an interleaved way under DISTMM-Pipe schedule, which ensures there are no extra idle cycles caused by dependency issues.

In terms of model quality, Pipeline executor's maximum batch size to the modality interaction can be formulated as $((M-M_s/P)/(M_a/(2 \cdot P))) = (M \cdot P - M_s)/(2 \cdot M_a)$, which grows linearly with the number of pipeline stages P. Compared to existing pipeline parallelism's maximum batch size $(M-M_s/P)/M_a$ which is bounded by M/M_a, Pipeline executor can preserve any model quality requirements by scaling out the cluster size while existing pipeline parallelism cannot.

Schedule alignment. With the pipeline parallelism degree

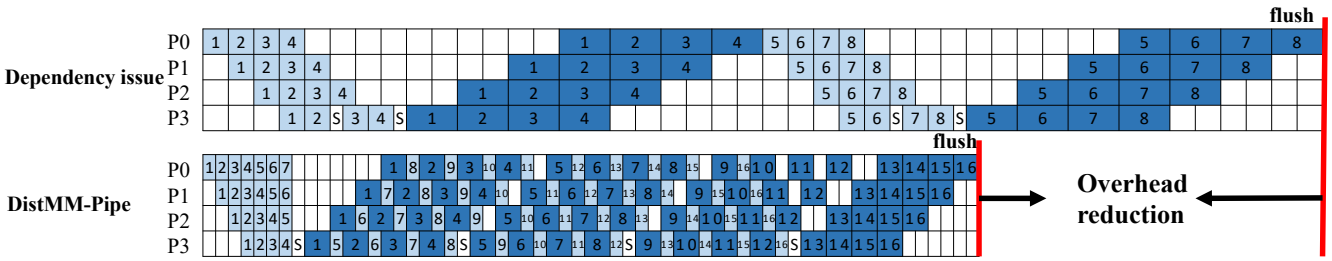


Figure 6: Comparison between pipeline parallelism schedule with dependency issue and DISTMM-Pipe schedule. S stands for batch-sync instruction. The modal quality requirement is $2 \cdot Mbs$.

assigned by Modality-aware partitioner and resource assignment plan by Data load balancer, Pipeline executor generates the DISTMM-Pipe schedule for each modal submodule. The schedules for different modal submodules are aligned through batch-sync instructions, which have a gathering communication for the global similarity matrix. Since the computation durations of each submodule are already balanced by Data load balancer, this alignment will not introduce any overhead caused by imbalanced durations between different schedules.

6 Implementation

We have built DISTMM with ∼2600 lines of code in Python. DISTMM is built on top of PyTorch [28]. It provides simple APIs to transform model scripts written for training on a single device to distributed multi-node multi-device setups. The goal of DISTMM is to minimize the user effort while fully utilizing the hardware resources.

APIs. There are four types of APIs: *partition annotation*, *training loop*, *data loading*, and *initialization*. Users annotate each submodule with the *partition annotation* API to allow DISTMM to partition them accordingly based on the pattern described by annotation. The *training loop* API provides an integrated function that includes both forward and backward computation. Users can use this integrated function to train the model, instead of constructing the training loop by themselves. For logging purposes, we provide hooks to log the intermediate results, e.g., loss values. For loading data samples correctly for load balancing purposes (Section 5.2), users need to wrap their data loading procedure using the *data loading* API. The *data loading* API provides the same user interface as the one from PyTorch framework, allowing a drop-in replacement for the training script with PyTorch. Before the computation starts on the cluster, training scripts are required to call the *initialization* API with configurations of the cluster and the model.

Configuration. DISTMM includes the multimodal model implementations on OpenCLIP [14] with DISTMM's partition annotations. The implemented models can be launched by specifying the model type in the training script. For models that have not been implemented, users need to provide the model description and add partition annotations to the corresponding operations and submodules, which helps Modality-aware partitioner to correctly recognize and partition the whole model. Users need to replace forward and backward calls in the training loop with DISTMM's integrated functions, which return the equivalent result as a single forward and backward computation. For submodules with pipeline parallelism partitioning, the actual computation is a sequence of forward and backward computations under DISTMM-Pipe schedule. Besides DISTMM's forward and backward replacement, users need to call DISTMM's initialization and specify the data source with DISTMM's data loading API in the training loop. Users also need to provide the environment configuration including cluster size, node size, and GPU memory limits for Heterogeneity-aware placement manager. To control the partitioning process, users can provide a partition strategy configuration including the partition strategy candidates and corresponding partition degree range to limit the search or simply specify a particular partitioning solution.

7 Evaluation

In this section, we seek to answer the following questions:

- How well does DISTMM perform on models with different structures?
- How well does DISTMM perform on models ranging from small scale to large scale?
- How effective is each component? Specifically, What is the impact of Modality-aware partitioner on the efficiency improvement of submodules with different structures? What is the impact of Data load balancer on the efficiency improvement with fixed submodule partitions? What is the impact of Heterogeneity-aware placement manager on reducing inter-node communication volume with fixed submodule partitions and fixed resource allocation plans?

Our experiments are conducted on a cluster of 8 Amazon EC2 p3.16xlarge nodes where each node contains dual 24-core Intel Xeon CPUs and 8 NVIDIA Tesla V100 (16GB) GPUs. The GPUs within a node are connected with NVLinks. The nodes are connected to 25 Gbps networks. All nodes run Ubuntu 20.04, CUDA 11.6 and PyTorch 2.0.1.

We measure the iteration time of training three different

models, namely, CLIP [29], CoCa [38], LiT [39]. Each model has unique characteristics in terms of the model structure. Specifically, CLIP model, which is designed for image-text retrieval (ITR), has a larger submodule for image modality input; CoCa model, which is used in natural language for visual reasoning (NLVR), has a relatively balanced (i.e., sizes of the submodules are similar) submodules for image and text inputs; LiT model, which is designed for visual question answering (VQA), has a larger submodule for text inputs. These selected models are representative, as they comprehensively cover the design space for multimodal models with two input data modalities, e.g., vision and language in our evaluation.

We further scale up the model sizes to study the system performance characteristics according to scaling law [16]. In our evaluation, we measure the system performance of model sizes in three categories: 1) *single-device scale*: the model is trainable on a single device; 2) *single-node scale*: the model requires memory more than the capacity of a single device, but the aggregated memory of devices in a single node is sufficient; 3) *multi-node scale*: the model is only trainable on multiple nodes. We summarize the configuration of models in Table 2.

We use Megatron-LM [34] as the baseline. Megatron-LM system provides state-of-the-art performance for training transformer-based unimodal models. As elaborated in Section 5.4, existing pipeline parallelism schedules cannot preserve the semantics of the user-defined training configuration. Therefore, we only use Tensor and Data Parallelisms from Megatron-LM. According to the convention of multimodal training for vision and language tasks, we use sequence length of 77 words and images sized 336×336 pixels [9].

7.1 End-to-End Performance

As shown in Figure 7, for models sized at *single-device scale*, DISTMM is 1.32–1.39× faster than Megatron-LM. These models include CLIP(760M, 350M), CoCa(760M,760M), and LiT(350M, 760M) [1]. At this model scale, Megatron-LM colocates two submodules in every GPU. In comparison, DISTMM uses Modality-aware partitioner to separate submodules, and balances the computation workloads among submodules via Data load balancer. DISTMM also optimizes the placement with Heterogeneity-aware placement manager to lower the communication cost.

For models that are not trainable on a single GPU, but can be trained on eight GPUs on a single node, i.e., single-node scale, DISTMM is 1.48–1.67× faster than Megatron-LM. The evaluated models include CLIP(6.7B, 2.7B), CoCa(6.7B, 6.7B), and LiT(2.7B, 6.7B). Megatron-LM partitions two submodules using the same tensor parallelism degrees. On the other hand, DISTMM applies modality-aware partitioning and assigns the most computation-efficient tensor parallelism degree to each submodule depending on their parameter sizes.

[1](X, Y) denotes the sizes of vision and text submodule, respectively.

When scaling the model size to *multi-node scale*, DISTMM outperforms the baseline by up to 3.27×. The evaluation includes CLIP(13B, 6.7B), CoCa(13B, 13B), and LiT(6.7B, 13B). For these models, we enable pipeline parallelism in DISTMM using Pipeline executor. Megatron-LM cannot maintain the batch size requirement of the training (detailed in Section 5.4), thus only the tensor parallelism is enabled. To support model sizes at the multi-node scale, Megatron-LM uses tensor parallelism across multiple nodes, e.g., CoCa (13B, 13B) is partitioned into 16 tensor parallelism. Its 16 partitions are placed on two nodes. In comparison, DISTMM uses tensor parallelism within each node, and leverages pipeline parallelism crossing nodes. Thus, we have much lower communication overhead [27, 34] as compared to Megatron-LM.

7.2 Effectiveness of Individual Components

To understand the sources of improvement, we designed a sequence of experiments, each dedicated to activating a single component of DISTMM at a time. We tailored the model setup and cluster configuration for each targeted component to mitigate potential side effects and dependencies on other components. Modality-aware partitioner and Data load balancer contribute to the training duration reduction by improving GPU utilization through adaptive partitioning and load balancing. In this paper, the GPU utilization is calculated by:

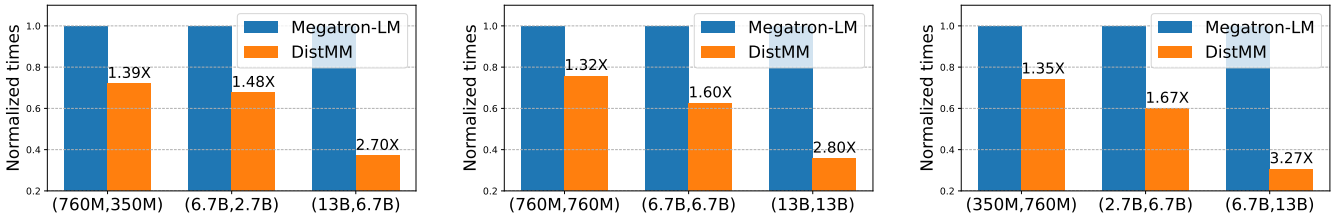
$$\frac{\text{Model FLOPs}}{\text{FLOPS} \cdot T}$$

, where Model FLOPs is the number of floating-point operations required to perform a single forward and backward pass, FLOPS represents the number of floating-point operations per second supported by a GPU, and T denotes duration.

To evaluate the Modality-aware partitioner, we choose the single-node scaled models that need tensor parallelism to demonstrate the GPU utilization differences between DISTMM's adaptive partitioning and Megatron-LM's homogenous partitioning. In the evaluation of Data load balancer, we need to mitigate the side effects of Modality-aware partitioner. To do this, we choose the single-device scaled models that do not need tensor parallelism to demonstrate the GPU utilization differences after Data load balancer's load balancing. Heterogeneity-aware placement manager contributes to the training duration reduction by reducing inter-node communication volume through communication-efficient placement. To evaluate it excluding DISTMM's computation optimization side effects, we choose the single-device scaled models to demonstrate the communication percentage differences between Megatron-LM and Heterogeneity-aware placement manager's placement without load balancing. Since DISTMM-Pipe is the only pipeline parallelism schedule that can be applied to multimodal model training without changing the training semantics as described in Section 5.4, DISTMM-Pipe is effective by ensuring the required modal quality.

Model	Vision submodule			Text submodule			Total # of Parameters
	Layers	Hidden Size	# of Parameters	Layers	Hidden Size	# of Parameters	
CLIP	24	1536	760M	24	1024	350M	1.1B
	32	4096	6.7B	32	2560	2.7B	9.3B
	40	5140	13B	32	4096	6.7B	19.7B
CoCa	24	1536	760M	24	1536	760M	1.52B
	32	4096	6.7B	32	4096	6.7B	13.4B
	40	5140	13B	40	5140	13B	26B
LiT	24	1024	350M	24	1536	760M	1.1B
	32	2560	2.7B	32	4096	6.7B	9.4B
	32	4096	6.7B	40	5140	13B	19.7B

Table 2: Configuration details of the models: CLIP, CoCa, and LiT.



(a) CLIP (b) CoCa (c) LiT

Figure 7: End-to-end iteration times of DISTMM normalized to Megatron-LM on three models with varied parameter configurations. The label of x-axis (X, Y): (size of vision submodule, size of text submodule)

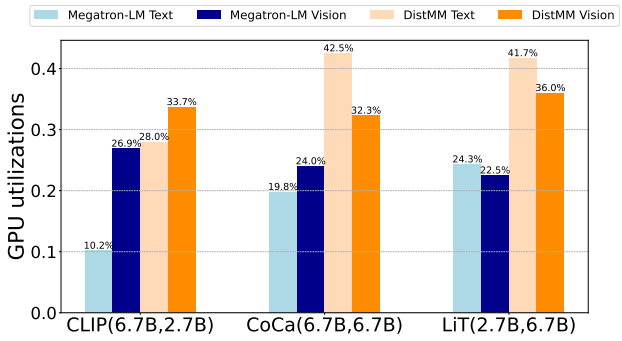


Figure 8: GPU utilization comparison on text and vision submodules between DISTMM and Megatron-LM's data parallelism and tensor parallelism combined solution.

7.2.1 Effectiveness of Modality-aware Partitioner

Experimental setup. We use performance timers to measure the computing duration when running CLIP(6.7B, 2.7B), CoCa(6.7B, 6.7B), and LiT(2.7B, 6.7B) in the 8 Amazon EC2 p3.16xlarge instances setting. Megatron-LM applies tensor parallelism degree 8 to both text and vision submodules and applies data parallelism to replicate the two submodules as a whole. In DISTMM's solution, the Modality-aware partitioner applies adaptive partitioning individually to two submodules.

Results. Figure 8 shows the GPU utilization differences of both text and vision submodules between DISTMM and Megatron-LM. For CLIP(6.7B, 2.7B), the text submodule's GPU utilization is largely increased, since the tensor parallelism is reduced from 8 to 4. For CoCa(6.7B, 6.7B), Megatron-LM's colocated solution limits the GPU utilization of text submodule by sharing the same batch size as vision submodule. In DISTMM's solution, text submodule and vision submodule share the same tensor parallelism partition degree but have larger batch sizes for increasing available memory under non-colocation. For LiT(2.7B, 6.7B), its lower activation memory consumption leads to a larger global batch size, resulting in higher GPU utilization than CLIP(6.7B, 2.7B).

7.2.2 Effectiveness of Data Load Balancer

Experimental setup. We use performance timers to measure the computation duration when running CLIP(760M, 350M), CoCa(760M, 760M), and LiT(350M, 760M) in the 8 Amazon EC2 p3.16xlarge instances setting. Megatron-LM applies data parallelism to the whole model. In DISTMM's solution, the Data load balancer redistributes the data to different submodules, resulting in more balanced workloads.

Results. As shown in Figure 9, the GPU utilization of both submodules have improved. The improvement amount is associated with the size of the submodule. DISTMM's Data load balancer puts more workload on the smaller submodule to balance the duration and efficiency. In CLIP(760M, 350M), the text submodule's batch size is increased from 8 to 150 while the vision submodule's batch size is increased from 8 to 10. The resulting GPU utilization improvement comes from increased batch sizes. For CoCa(760M, 760M), even though the parameter sizes are the same for text and vision submodules, the sequence length differences lead to the workload

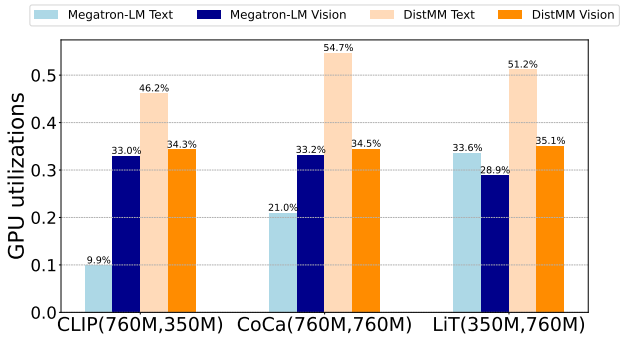


Figure 9: GPU utilization comparison on text and vision submodules between DISTMM and Megatron-LM.

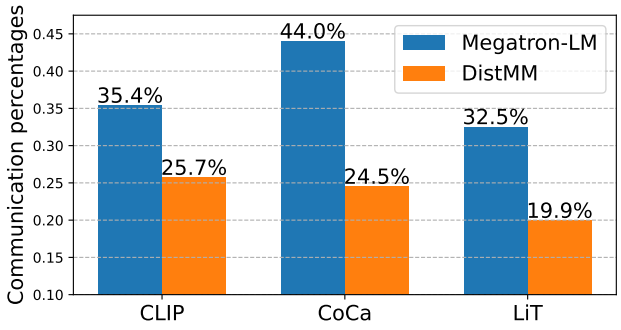


Figure 10: Communication duration's percentage in end-to-end training comparison between DISTMM and Megatron-LM's data parallelism solution.

imbalance. After the data load balancing, the text submodule gains a large GPU utilization improvement, where text submodule replicates 16 times and vision submodule replicates 48 times. For LiT(350M, 760M), since the submodules are balanced before Data load balancer, both submodules gain large GPU utilization improvements for increased batch sizes.

7.2.3 Effectiveness of Heterogeneity-aware Placement Manager

Experimental setup. We use performance timers to measure communication duration when running CLIP(760M, 350M), CoCa(760M, 760M), and LiT(350M, 760M) in the 8 Amazon EC2 p3.16xlarge instances setting. The communication duration involves all-gather communication for distributed modality interaction and all-reduce communication for gradient synchronization. Megatron-LM applies data parallelism to the whole model. In DISTMM's solution, Heterogeneity-aware placement manager places submodules of different modalities onto different nodes without load balancing.

Results. As shown in Figure 10, the communication percentages of each model have been reduced due to the communication volume reduction. DISTMM's solution achieves largest communication reduction in CoCa(760M, 760M), since its text and vision submodules share a similar parameter size.

8 Related Work

Model reordering. IOS [8] explores the parallel opportunities within the modal structure similar to DISTMM through reordering the operators on the same device to improve efficiency, while DISTMM reorders operators on different devices. Rammer [23] orders operators by fusing parallelizable operations without dependencies into a fused kernel to improve resource utilization, which focuses on inference with an extremely small batch size instead of DISTMM's training workload. DeepSpeed-MoE [30] proposes expert parallelism to reorder and parallelize homogeneous MoE experts, while DISTMM focuses on heterogeneous modal submodules.

Model partitioning. Several systems implement one or many strategies out of the 3D parallelism partitioning. Pytorch's DDP [19] and Horovod [32] replicate a model on every device to use data parallelism, while ZeRO [31] splits both weights and model states across all devices to accommodate larger models for data parallelism. FlexFlow [22], Mesh-Tensorflow [33], and Gshard [18] split operations in a way to represent data parallelism and model parallelism. Megatron [34] and DeepSpeed-Megatron [35] support 3D parallelism to parallelize the models in a manually optimized way. Alpa [40] and Pathway [3] combine single program multiple data (SPMD) and multiple programs multiple data (MPMD) abstraction to enable automatic 3D parallelism parallelization. However, unlike DISTMM, these systems do not consider the heterogeneous nature of multimodal models.

9 Conclusion

This paper introduced DISTMM, a distributed multimodal model training system. It consists of four heterogeneity-aware system components designed specifically for multimodal model training. DISTMM utilizes the parallelization opportunities within the multimodal model structure, and successfully solves the performance bottlenecks through adaptively partitioning each submodule and load balancing the partitions. To preserve the model quality, DISTMM also orchestrates a parallel execution by introducing new pipeline parallelism instruction and corresponding schedule. Our experiments show that DISTMM can outperform the state-of-the-art training system for models with varied structures and varied sizes.

10 Acknowledgments

The authors would like to thank our shepherd Dr. Deepak Narayanan and the anonymous reviewers for their valuable feedback and thoughtful suggestions.

References

[1] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. Flamingo: a visual language model for few-shot learning. *Advances in Neural Information Processing Systems*, 35:23716–23736, 2022.

[2] Sanjeev Arora, Hrishikesh Khandeparkar, Mikhail Khodak, Orestis Plevrakis, and Nikunj Saunshi. A theoretical analysis of contrastive unsupervised representation learning. *arXiv preprint arXiv:1902.09229*, 2019.

[3] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. Pathways: Asynchronous distributed dataflow for ml. *Proceedings of Machine Learning and Systems*, 4:430–449, 2022.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[5] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020.

[6] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[8] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. Ios: Inter-operator scheduler for cnn acceleration. *Proceedings of Machine Learning and Systems*, 3:167–180, 2021.

[9] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[10] Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. Palm-e: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378*, 2023.

[11] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[12] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9729–9738, 2020.

[13] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[14] Gabriel Ilharco, Mitchell Wortsman, Ross Wightman, Cade Gordon, Nicholas Carlini, Rohan Taori, Achal Dave, Vaishaal Shankar, Hongseok Namkoong, John Miller, Hannaneh Hajishirzi, Ali Farhadi, and Ludwig Schmidt. Openclip. *https://doi.org/10.5281/zenodo.5143773*, 2021.

[15] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 402–416, 2022.

[16] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[17] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. Supervised contrastive learning. *Advances in neural information processing systems*, 33:18661–18673, 2020.

[18] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[19] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[20] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pages 740–755. Springer, 2014.

[21] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. *arXiv preprint arXiv:2304.08485*, 2023.

[22] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 553–564. IEEE, 2017.

[23] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 881–897, 2020.

[24] Ishan Misra and Laurens van der Maaten. Self-supervised learning of pretext-invariant representations. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6707–6717, 2020.

[25] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[26] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[27] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[29] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021.

[30] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International Conference on Machine Learning*, pages 18332–18346. PMLR, 2022.

[31] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

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

[33] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31, 2018.

[34] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[35] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

[36] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[38] Jiahui Yu, Zirui Wang, Vijay Vasudevan, Legg Yeung, Mojtaba Seyedhosseini, and Yonghui Wu. Coca: Contrastive captioners are image-text foundation models. *arXiv preprint arXiv:2205.01917*, 2022.

[39] Xiaohua Zhai, Xiao Wang, Basil Mustafa, Andreas Steiner, Daniel Keysers, Alexander Kolesnikov, and Lucas Beyer. Lit: Zero-shot transfer with locked-image text tuning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 18123–18133, 2022.

[40] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[41] Deyao Zhu, Jun Chen, Xiaoqian Shen, Xiang Li, and Mohamed Elhoseiny. Minigpt-4: Enhancing vision-language understanding with advanced large language models. *arXiv preprint arXiv:2304.10592*, 2023.

Approximate Caching for Efficiently Serving Text-to-Image Diffusion Models

Shubham Agarwal
Adobe Research

Subrata Mitra[*]
Adobe Research

Sarthak Chakraborty[†]
UIUC

Srikrishna Karanam
Adobe Research

Koyel Mukherjee
Adobe Research

Shiv Kumar Saini
Adobe Research

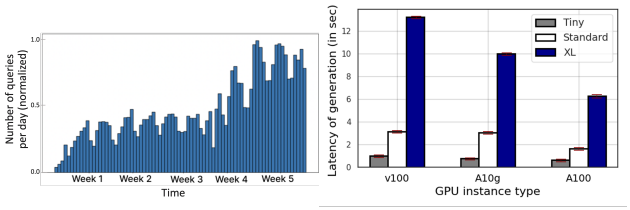
Abstract

Text-to-image generation using diffusion models has seen explosive popularity owing to their ability in producing high quality images adhering to text prompts. However, diffusion-models go through a large number of iterative denoising steps, and are resource-intensive, requiring expensive GPUs and incurring considerable latency. In this paper, we introduce a novel *approximate-caching* technique that can reduce such iterative denoising steps by reusing intermediate noise states created during a prior image generation. Based on this idea, we present an end-to-end text-to-image generation system, NIRVANA, that uses approximate-caching with a novel cache management policy to provide 21% GPU compute savings, 19.8% end-to-end latency reduction, and 19% dollar savings on two real production workloads. We further present an extensive characterization of real production text-to-image prompts from the perspective of caching, popularity and reuse of intermediate states in a large production environment.

1 Introduction

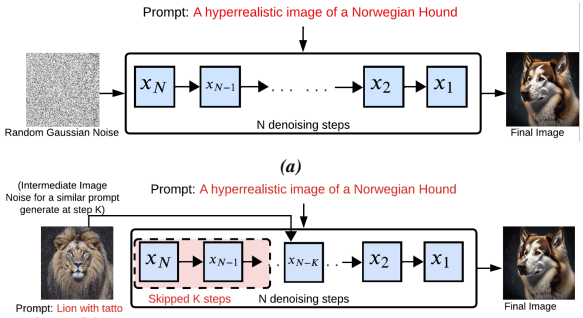
Text-to-image generation has drastically matured over the years [34, 87] and has now become a widely popular feature offered by various companies [8, 14], being integrated into various new creative workflows [10]. The popularity of text-to-image models has become massive. Adobe recently reported [10] that over 2 billion images were created using Firefly [9] text-to-image service. Similar popularity has also been reported for Dall-E-2 from OpenAI [12]. Figure 1a shows the staggering growth over time in the numbers of prompts submitted to a portal running stable-diffusion-based text-to-image model, as captured by the DiffusionDB dataset [79].

State-of-the-art in text-to-image: In text-to-image generation, given a text *prompt* describing certain desirable characteristics of an image, a deep neural network model generates an image capturing the descriptions provided in the prompt. While researchers have been attempting to design viable and consistent text-to-image models for quite some time using VAEs [52], GANs [42] and other techniques [37], the most popular text-to-image production systems of today [9, 17] are based on *Diffusion-Models* (DMs) [36, 49]. The widespread adaptation of DMs can be attributed to their capability to generate superior quality images and to condition the generation more accurately according to the input prompt.



(a) *(b)*

Figure 1: (a) Shows normalized growth in workload over 5 weeks in DiffusionDB. (b) Latency for `tiny`, `standard` and `XL` stable-diffusion models from Hugging Face [51] repository on three different GPU architectures in AWS. The image quality produced by the larger `XL` model is far superior to the smaller two models but comes with a substantially large latency overhead which also implies higher costs.



(a)

(b)

Figure 2: (a) Shows how vanilla `DMs` work. (b) Shows how `DMs` with approximate-caching works, where first K denoising steps are skipped after an intermediate noise belonging to a different prompt present in the cache is retrieved and reused.

Problems with `DMs`: Text-to-image products should offer real-time interaction like search platforms. However, image generation using DMs is a computational resource-intensive task and suffers from relatively long latency. During the inference phase in DMs (that is generating an image from text), essentially a Gaussian noise is iteratively denoised, using the

[*]Corresponding author (subrata.mitra@adobe.com)
[†]Work done at Adobe Research

input prompt as a condition, to produce the output image in a Markov process [39]. The traditional DMs use as much as 1000 such iterative diffusion steps [36] for this. Some subsequent optimizations [61,76] enable such denoising to be done with approximately 50-100 iterative steps [2]. Still, even with 50 iterative steps, image generation with DMs is a resource-intensive and slow task that prohibits interactive experience and results in huge computational costs on expensive GPUs.

In Figure 1b, we show the latency (in seconds) for different GPU architectures (i.e., A100, A10g, and V100 from NVIDIA) that are available on Amazon Web Services (AWS) and for a few DMs of different sizes (i.e., # of parameters): `Stable Diffusion XL`, `Stable Diffusion 2.1`, `Tiny Stable Diffusion` from Hugging Face repository [51] with typical 50 diffusion steps for image generation during inference. It can be observed that while smaller models (e.g., `Tiny` and `Standard`) can provide significantly low inference latency compared to a larger model(e.g., `XL`), it usually comes with significant degradation in the quality of the generated image [67]. Therefore, using a smaller model to reduce latency for better user experience might actually defeat the purpose. Latency can also be reduced by using a more powerful GPU for inference, as can be observed in Figure 1b, but pricing for cloud instances with powerful GPUs such as NVIDIA A100s is significantly high. For instance, in the US East region [1] the V100 is priced at approximately $3.06 per hour, the A10g at $8.144 per hour, and the A100 at $32.77 per hour making an A100 instance 4X costlier than A10g and more than 10X costlier than a V100. Therefore, while inference latency can be reduced with more powerful GPUs, inference cost per image also significantly increases.

In this paper, we introduce an efficient text-to-image generation system called NIRVANA that uses novel *approximate caching* technique to significantly reduce the computational cost and latency by effectively reusing intermediate states created during image generation for prior prompts.

Figure 2 illustrates the key idea behind NIRVANA. For an input prompt shown in *red* in Figure 2a we show how the vanilla or standard DMs work through multiple iterative denoising steps. Here, the process starts from a Gaussian noise at state x_N and then performs N denoising steps with the input prompt as the condition to finally produce a coherent image in state x_0. Figure 2b shows how in *approximate caching*, first K steps are skipped, and directly a suitable intermediate noise from a different prompt is retrieved and used. The value of K depends on the similarity between the new input prompt and the prompt from which the noise was retrieved. Therefore, the amount of compute and latency savings can vary across prompts based on the availability of similar prompts in the cache that stores the intermediate noises/states produced during the image generation from previously encountered prompts. The phrase *approximate caching* emphasizes the fact that in this system, we are not directly reusing the retrieved object from the cache, rather we are retrieving an

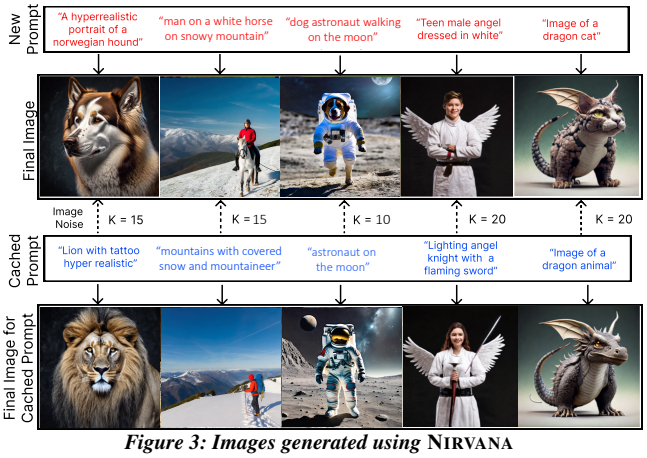


***Figure 3: Images generated using* NIRVANA**

intermediate state and further conditioning those to tailor the generated image according to the new prompt. Thus, NIRVANA is very different from a retrieval-based system such as GPT-CACHE [7], PINECONE [4] that proposes direct retrieval of an image from a cache based on the input prompt.

NIRVANA selects the value of K for a particular input prompt by effectively controlling the *hit-rate* vs. compute savings trade-off. *Hit-rate* here means how likely an incoming prompt can be matched to a *similar enough* prompt in the cache, while K denotes the # steps we skip at the beginning of the DMs when using the retrieved noise from the cache. The *hit-rate*, can be made significantly high if we are planning to skip a very small number of steps at the beginning (i.e., lower value for K). The reason being, if we are skipping less diffusion steps at the beginning, noise from very dissimilar prompts can still be denoised effectively to produce a coherent image. While for high K, the scope for further denoising conditioned on the new prompt becomes limited. However, it provides huge compute savings when done for certain amenable prompts. Careful design of NIRVANA can navigate this complexity by calculating a suitable value for K for each incoming prompt to maintain high quality as well as optimize for maximum compute savings.

Figure 3 shows some real examples using prompts from DiffusionDB [79] to illustrate how NIRVANA can transform a noise from a seemingly different prior prompt to a coherent and high-quality image while providing significant latency and compute reduction at the same time.

Furthermore, in NIRVANA, we design a novel approximate-cache management policy, called *Least Computationally Beneficial and Frequently Used* (`LCBFU`), that manages the storage of noises in such a manner that for a given cache storage size, we optimize the space for the noises that is likely to give the best computational efficiency to NIRVANA.

Overall, NIRVANA can maintain generated image quality very close to vanilla diffusion-models (i.e., DMs without approximate-caching) while providing 21 % savings in GPU computation, 19.8 % reduction in latency, and 19% amortized savings on dollars spent on image generation. Our study in-

volving 60 users with 1000 images shows that 79% users liked NIRVANA generated images which is far better than the best baseline GPT-CACHE [7] with only 31% likes, and much closer to the quality of images generated by the expensive and slow VANILLA model, liked by 86% users.

We summarize our contributions in this paper as follows:

1. We introduce the novel idea of *approximate caching* that provides significant computation saving in the production pipeline of diffusion models for text-to-image generation.
2. We propose an effective cache-management mechanism, called LCBFU, that can optimize the reuse of computation states and computation savings.
3. We present end-to-end design details and rationale for NIRVANA, which is our optimized text-to-image deployment system on the cloud.
4. We characterize real production prompts for text-to-image models from the perspective of reusability and caching.
5. We present extensive evaluation with two real and large production prompts from text-to-image models, along with a human evaluation and several sensitivity studies.

2 Background

2.1 Diffusion Models (DMs)

Diffusion models (DMs) progressively denoise a random Gaussian noise to generate an image conditioned on text. The training procedure contains a forward diffusion process, which obscures an image by adding noise repeatedly in a Markov process until it saturates to Gaussian noise. In the backward diffusion process, the original image is recovered by removing noise repeatedly. Each denoising step is called "sampling" since the model generates a sample by removing noise, and the method used for sampling is called the *sampler*.

In the forward diffusion process, Gaussian noise gets progressively added to an initial image x_0 for T steps to get x_T. With the Markov chain assumption, it is expressed as:

$$q(x_{1:T}|x_0) := \prod_{t=0}^{T} q(x_t|x_{t-1}) \tag{1}$$

$$q(x_t|x_{t-1}) := \mathcal{N}(x_t|\sqrt{1-\beta_t}x_{t-1}, \beta_t I) \tag{2}$$

where $q(x_t|x_{t-1})$ is the posterior probability, and $\beta_1, ..., \beta_T$ is the noise schedule (either learned or fixed) to regulate the noise level at each diffusion step. Similarly, the backward diffusion process can be written as:

$$p_\theta(x_{0:T}) := p(x_T) \prod_{t=0}^{T} p_\theta(x_{t-1}|x_t) \tag{3}$$

$$p_\theta(x_{t-1}|x_t) := \mathcal{N}(x_{t-1}|\bar{\mu}_t, \bar{\beta}_t I) \tag{4}$$

where $p_\theta(.)$ denotes the probability of observing x_{t-1} given x_t. Here, $p(x_T) = \mathcal{N}(x_T|0, I)$. Here, $\bar{\mu}_t$ and $\bar{\beta}_t$ is learned. The objective is to learn $p(\theta)$ that maximizes the likelihood of training data in the backward/reverse diffusion process. Recent optimizations approximate the backward diffusion process by skipping certain intermediate states at predetermined timesteps [54, 90], thus reducing inference steps from $T \approx 1000$ steps to $N \approx 50$ steps. This is achieved by learning a sampler that predicts how much noise will remain after T/N step for every one diffusion step. Notably, each step consumes equal time and compute as it uses the same denoising process on the same diffusion model [87]. Even with these optimizations, image generation still takes 10 seconds on A10g and 6 seconds on A100 GPUs for large models (Figure 1b).

Diffusion model backbone is based on the U-Net architecture [70]. When a text prompt is given, the image generation process is conditioned through cross-attention within the model [69]. Thus, it develops a text-to-image framework capable of generating visually coherent and contextually relevant images based on textual descriptions.

In practice, each x_i can either be the actual image or its latent representation computed by an image encoder [69]. Generally, the latter approach, termed as latent diffusion model (LDM) is preferred since it captures the hidden characteristics of an image. We use the same in our work. To generate the final image from its latent space x_0 at the end of backward diffusion, x_0 is decoded with the inverse of the image encoder.

2.2 Dynamics of Image Generation

The amount of reconditioning needed for the retrieved noise to suit the new prompt depends on K. We observe that various concepts/characteristics of an image, like, *layout*, *color*, *shape or objects*, *style*, etc. are not easy to modify beyond certain K. With initial noise being random Gaussian, similar to recent works [89] (see **Appendix** E for details), we observed for our dataset that LDM models [69] decide the layout of the objects first within the initial ∼20% of diffusion steps. Color map for the overall image then gets decided within ∼40% steps followed by the shape and the size of the objects and then the artistic style of the image. Overall, we observed that after ∼50% of the steps, the concepts get frozen and no further attempts to recondition the intermediate image are reflected in the final generated image. For example, when attempting to recondition the image of a brown horse grazing in a green field (Figure 4), the color can be modified after $K = 10$, but not after $K = 15$. Similarly, actions, objects, and backgrounds could be modified only till certain steps and not beyond that.

3 Understanding User Prompts

We first show some characterization of real user prompts in DiffusionDB dataset [79] having 2 million prompts ranging from 8 Aug, 2022 to 22 Aug, 2022. Each entry has the user-id, prompt, timestamp, and the generated image in the dataset.

(1) Top-Y% most popular prompt clusters over days

We cluster the prompt queries for each day using DBSCAN algorithm [73] on their CLIP text embeddings [68] with multiple *eps* values (*eps* controls the maximum distance between two samples for one to be considered the neighbor of other). We find top-Y% (with Y = 1, 2, 5, and 10) clusters and extract the most popular prompts on Day 1. We then track how many of these top-Y% clusters remain in the top-Y% on the following day (Day 2 to Day 18) and report this fraction as the

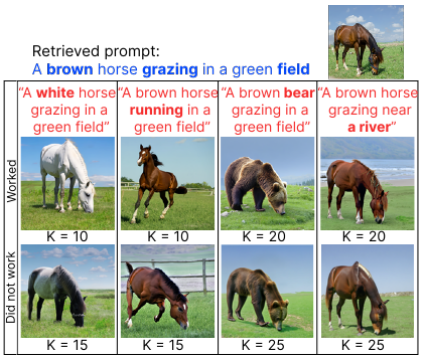


Figure 4: A noise from a brown horse successfully transforms into a white horse at K=10, while K=15 fails. The same noise becomes a bear even at K=20.

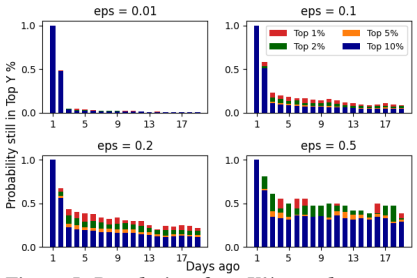


Figure 5: Popularity of top Y% popular prompt clusters over days for different similarity threshold between prompts (eps value)

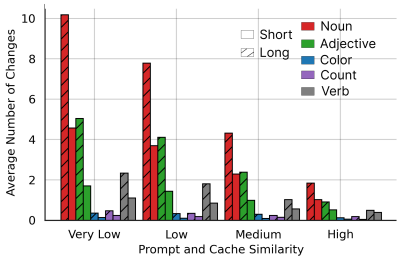


Figure 6: Average number of noun, adjective, color, count, and verb changes across similarity for both short and long prompts

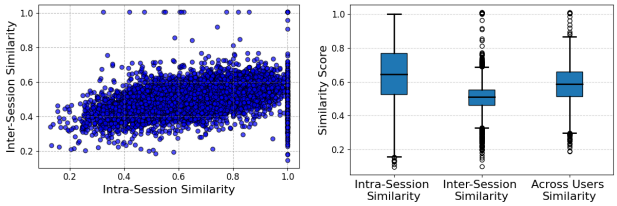
longevity probability of popular prompt clusters in Figure 5. We repeat this analysis with different *eps* values (*eps* = 0.01, 0.1, 0.2, 0.5) in DBSCAN algorithm.

Takeaway. From Figure 5, we see that most of the clusters in the top-Y% prompt clusters do not remain popular in top-Y% even on Day 2 when the clusters are tightly packed (*eps*=0.01). There is a more significant drop in popularity after 2 days. However, as we increase *eps* and make clusters more loosely packed, it effectively gathers more prompts within a cluster and hence the decay rate of popular prompts is less. We empirically verified that with increasing *eps*, higher number of prompts and more dissimilar prompts are clustered together, and hence the longevity of the top-Y% cluster increases. This shows the extent of similarity in prompts over time and the potential for reuse. *Approximate-caching* idea works even with less similar prompts, as it can recondition the retrieved noise. Thus, the high longevity of popular cluster, as seen in plots for *eps*=0.2 and *eps*=0.5, shows potential of *approximate-caching* for generating images in production environments.

(2) Most similar short and long prompt pairs

We divide the prompt queries into short and long prompts, based on the 70th percentile word count (≈15 words). For each set, we then form pairs of most similar prompts using cosine-similarity between their CLIP embeddings and group them into 4 buckets based on their similarity scores: *Very Low* (less than 0.65), *Low* (0.65 to 0.8), *Medium* (0.8 to 0.9), and *High* (more than 0.9). For each bucket, we analyze what attributes (e.g., noun, adjective, verb, color, count) changed between the pair of prompts within the same bucket and show the average number of changes along these attributes, for both long and short classes of prompts in Figure 6.

Takeaway. In Figure 6, we see that as the similarity of the prompts within a pair increases, the average number of changes decreases. We also see that within each bucket, the average number of noun changes is the most, followed by adjective and verb changes. Longer prompts have more changes as compared to the shorter prompts within the same similarity bucket. This indicates that even when several attributes in the text of the prompt change, the CLIP embedding failed to appropriately distinguish the difference between the two prompts. This highlights a limitation in identifying similar prompts when prompts are very long, as two prompts can be misleadingly retrieved as very similar.



(a) *(b)*

Figure 7: Query prompt similarity (a) for inter v/s intra-session per user, (b) range for intra, inter per user and across diff users

(3) Intra vs. inter-session similarity in prompt queries

We group the prompts per user, then divide the prompts from that user into 1-hour sessions. We then compare the similarity of prompts within the same sessions (*intra*) and across different sessions (*inter*) of the *same user* in Figure 7a. In Figure 7b, we further compare these with similarity scores between prompts from sessions of 100 random *other users*.

Takeaway. Figure 7a shows, many users exhibit high intra-session similarity, meaning users tend to use similar prompts within a session with a lot of repetitions (indicated by a cluster of points at 1). Also, there is high inter-session similarity, but it is lower than intra-session, indicating diverse queries across sessions. In Figure 7b, we observe that prompts within a session (*intra-session*, left) exhibit high similarity. Notably, prompts across sessions (*inter-session*, middle) of the same user and even with prompts from different users (*across-users*, right) also demonstrate significant similarity. Interestingly, the similarity between prompts from different users is notably higher, suggesting a promising opportunity for efficient approximate caching utilizing the power of the masses.

4 NIRVANA Overview

4.1 Approximate Caching

For a new query prompt $\mathcal{P}_Q$, NIRVANA uses *approximate-caching* to reduce computation by retrieving an intermediate state that was created after K^{th} iteration of a previous image

generation process and directly reusing and reconditioning that for $\mathcal{P}_Q$ for the remaining $N-K$ steps.

Analytical Modeling: Let $\mathcal{L}$ denote the total end-to-end latency of image generation using approximate caching. Within this, $\mathcal{C}$ represents the cumulative GPU computation time for N diffusion model steps. The set of possible values for K is denoted as $\mathcal{K}$. Each search operation in the vector database (VDB) incurs a latency cost denoted as l_s, and retrieving the intermediate state from the cache introduces a latency denoted as l_r. We use f_c to indicate the overall compute savings.

Therefore, for prompts effectively utilizing *approximate caching* with a cache generated at K, the total latency experienced can be expressed as:

$$l_s + C \cdot \frac{N-K}{N} + l_r \tag{5}$$

In contrast, prompts for which NIRVANA cannot locate a match in the cache will undergo a total latency of $l_s + C$.

This distinction arises from the design, where it attempts to retrieve an intermediate state from the file system (incurring latency overhead l_r) only when a hit is confirmed in the VDB, ensuring the existence of the state in the cache.

Let us denote the *hit-rate@K* for approximate caching as $h(K)$ which is defined as the likelihood that when an intermediate state from K^{th} diffusion step is used, it takes at most $N-K$ diffusion steps to generate a faithful reconditioned image where N is fixed. That is, $(1-h(K))$ fraction of cache exists, which we cannot recondition by running $N-K$ steps.

Now at $K=0$ (running diffusion model from scratch), *all* historical prompts are theoretically usable since an image can be reconditioned in at most $N-0$ steps, leading to $h(0) = 1.0$. As K increases, $h(K)$ decreases, since we can use only a smaller fraction of intermediate states from K^{th} step to recondition an image by running diffusion at most $N-K$ steps. For lower values of K, $h(K)$ is less than 1.0 but can still be relatively high. That is, the diffusion models can effectively recondition the retrieved state if the state is from the initial diffusion steps, resulting in the generation of faithful images.

The decrease in $h(K)$ is influenced by how dissimilar the prompts are. When K surpasses a certain threshold, denoted as K_T, the retrieved state is no longer suitable for further reconditioning, as discussed in § 2.2, and thus, $h(K \geq K_T) = 0$.

Consequently, the effective fraction of savings in GPU computation for a given K can be expressed as:

$$f_C = h(K) \cdot \frac{K}{N} \tag{6}$$

It is evident that substantial savings can be achieved when both K and $h(K)$ are sufficiently high. However, the challenge lies in the fact that as K increases, $h(K)$ tends to decrease while aiming to maintain the quality $\mathbb{Q}$ of the generated images[1]. This trend is described in § 5.2 for DIFFUSIONDB dataset [79] across different discrete K values.

[1] For example, if we assume that $h(K)$ decreases linearly from 1.0 at $K=0$ to 0 at $K=K_T$ following the equation $h(K) = -\frac{K}{K_T} + 1$, then the optimal single value of K_{OPT} that maximizes fractional savings will be: $K_{OPT} = K_T/2$, resulting in effective compute savings of $f_C^{max} = K_T/4N$. Similarly, for a slowly decaying quadratic form expressed as $h(K) = -\left(\frac{K}{K_T}\right)^2 + 1$, $f_C^{max} = \frac{2K_T}{3\sqrt{3}N} > \frac{K_T}{4N}$. Therefore, the slower the decay of $h(K)$ with respect to K the higher the compute savings we can expect.

Now we define $h_{opt}(K)$ as the fraction of cache stored at K^{th} diffusion step that is used to exactly recondition an image for $N-K$ steps. For example, with $N=50, K=5$, we get $h_{opt}(K)$ is the fraction of cache that can be used to recondition an image by running diffusion steps for exact 45 steps. Thus,

$$h_{opt}(K) = h(K) - h(K'), \text{ where } \arg\min_{K'}(K' > K) \tag{7}$$

$$h(\min \mathcal{K}) = \sum_{K \in \mathcal{K}} h_{opt}(K) \tag{8}$$

In essence, $h_{opt}(K)$ quantifies the probability that K represents the maximum potential savings for incoming prompts. $h(\min \mathcal{K})$ represents the overall hit-rate, i.e., fraction of $\mathcal{P}_Q$ having a cache hit.

Our primary objective is to minimize end-to-end latency ($\mathcal{L}$) while maintaining the quality ($\mathbb{Q}$) of generated images. We operate under the constraint that reconditioning of an image with a cache at the selected K values must ensure a specified level of quality $\mathbb{Q}$ compared to when the image is generated from scratch. The goal is to find the optimal K value that satisfies these objectives and the below quality constraint.

Thus, for a given incoming prompt $\mathcal{P}_Q$ and its corresponding cached prompt $\mathcal{P}_c$

Objective (Minimize $\mathcal{L}$): (following Eq. 5)

$$\min_K \mathcal{L} = \sum_{K \in \mathcal{K}} \left(l_s + h_{\text{opt}}(K) \cdot l_r + h_{\text{opt}}(K) \cdot C \cdot \frac{N-K}{N} \right) \tag{9}$$

Quality Constraint:

$$\mathbb{Q}(I_K^c | \mathcal{P}_{c_K}, \mathcal{P}_Q) > \alpha \cdot \mathbb{Q}(I_0 | \mathcal{P}_Q) \tag{10}$$

where I_K^c represents the image generated by using cache c at K and then reconditioning for $N-K$ diffusion steps. $\alpha \in [0,1]$ represents the tolerance threshold over the quality of images generated and is such that I_K^c is not much worse than I_0. In our implementation (§ 5.3.1), we employ the CLIPScore metric [46] to define $\mathbb{Q}$ with $\alpha = 0.9$. We opt for CLIPScore due to its widespread use in evaluating image quality.

In our use case, where $l_r, l_s << \mathcal{C}$, the objective reduces to

$$\min_K \mathcal{L} = \sum_{K \in \mathcal{K}} \left(h_{\text{opt}}(K) \cdot C \cdot \frac{N-K}{N} \right) \tag{11}$$

which maximizes K and $h_{opt}(K)$ to obtain minimal latency.

This framework demonstrates the relationship between latency and quality in the context of approximate caching.

Notably, the wasted overhead of NIRVANA can be captured as $(1-h(K)) \cdot l_s$ where the vector database is queried and results in a cache miss. This means if we operate in a setting where $h(K)$ is low even if K is high, NIRVANA may not provide latency benefits if l_s is comparable to GPU compute

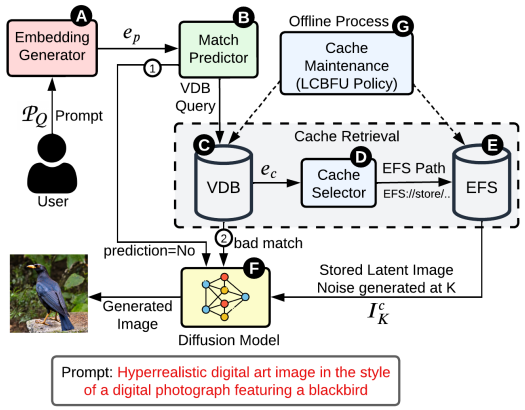


Figure 8:* NIRVANA *Overview

latency $\mathcal{C}$ and such overhead reduction can be an important design aspect as we discuss in § 5.2. It is important to note that, due to the disparate costs of high-end GPUs, even if there is not much latency reduction, it can still be significantly cost-effective to use NIRVANA as it drastically cuts expensive GPU compute costs on the cloud. In our setting, we observed l_s is of the order of 100 *ms* whereas $\mathcal{C}$ is of the order of 10 *s*.

4.2 System Components

Figure 8 shows the main components and the execution paths of NIRVANA. At first, an embedding vector e_p is generated for the current input prompt $\mathcal{P}_Q$ using an `embedding-generator` (A). Then an optional `match-predictor` module (B) predicts whether there would be a *close enough* match for this embedding in the *vector-database* (`VDB`) (C). If the presence of a similar enough cached entry is likely, then NIRVANA starts the process of retrieving an intermediate state from the cache. First, a search query is sent to the `VDB` to find the closest cached embedding of e_p, denoted by e_c. A `VDB` uses efficient approximate nearest neighbor (ANN) [62] search to find such closest embeddings. For each cached historical prompt in the `VDB`, NIRVANA stores several intermediate states during the vanilla diffusion process. Which of these intermediate states corresponding to the match prompt e_c is optimal for the new prompt vector e_p is calculated using a heuristic by the `cache-selector` module (D). Then this particular intermediate state is retrieved from `EFS` storage (E) (i.e. Elastic File System for NIRVANA) using the pointer of the storage location pointed by the search result of the `VDB` query and the particular intermediate state number (i.e. K as explained in § 4.1) calculated by the `cache-selector` module. An intermediate state I_K^c is an $L \times D$ dimensional latent representation captured during the denoising process of e_c, after K_{th} step. Finally, this retrieved intermediate state I_K^c is passed to the `DM` (F) along with e_p for it to be reconditioned for image generation for the rest of the denoising steps. However, there are two situations when NIRVANA directly falls back to the vanilla diffusion model to generate an image from scratch (using Gaussian noise), sacrificing any optimization: (i) when `match-predictor` module predicts that a close entry in the `VDB` is unlikely (arrow (1)), and (ii) when `VDB` query returns a match e_c that is very dissimilar to e_p (arrow (2)) .

To maintain the cache under a fixed size of storage and to prevent the `VDB` from arbitrarily growing and keep on storing stale entries, a `cache-maintainer` module (G) works in the offline mode and implements the novel `LCBFU` protocol to keep both the cache and `VDB` entries fresh.

5 NIRVANA Design Details

We present the detailed design of various components in NIRVANA. The full algorithm can be found in **Appendix** A.

5.1 Embedding Generator

Similar to vanilla `DMs`, NIRVANA first computes a 768-dimensional vector CLIP embedding [68] (e_p) from the text prompt. CLIP effectively positions visually similar prompts closer in the embedding space, which optimizes the likelihood of cache hits in our case.

5.2 Match Predictor

Using e_p, NIRVANA could directly make a search query to `VDB` for the closest prompt in the cache. If cache exists, there will be substantial savings in GPU usage for image generation. However, in case of *cache miss*, the search to `VDB` becomes a latency overhead without getting any reduction in the GPU computation. Now, several factors including: a) if it is in the same LAN vs. far away from the GPUs, b) the particular architecture of the `VDB` and its internal indexing mechanism, and c) the compute resources dedicated to it etc., dictate the magnitude of the wasted latency overhead during each *miss*.

To reduce this overhead, NIRVANA uses a component called `match-predictor`, ((B) in Figure 8), which *predicts* if an embedding close enough to e_p is *likely* to be present in the `VDB`. If the prediction says it is unlikely, then NIRVANA simply bypasses cache retrieval flow altogether, reducing the wasted overhead. Additionally, `match-predictor` also reduces `VDB` load corresponding to search misses, improving scalability.

Internally, `match-predictor` uses a lightweight classifier for predictions that runs on the CPU of the same node where the `DM` runs on the GPU. This reduces the classification latency by orders of magnitude compared to a query to the `VDB`, making latency overhead insignificant.

Recall from § 4.1 that analytically the latency overhead is $(1-h(K)) \times l_s$. Now let c_p denote the *precision* of the `match-predictor` classifier. The effective overhead of NIRVANA with an active `match-predictor` is then:

$$1 - max(h(K), c_p) \cdot l_s, \text{ where } h(K), c_p \in [0,1]$$

This means that either when $h(K) = 1.0$, i.e., prompts are so similar that for every incoming prompt, there is a suitable match available in the cache, or when $c_p = 1.0$, i.e., the classifier is perfect in predictions, the system will not have any wasted latency overhead. Note that, when $c_p < 1.0$, NIRVANA would miss some opportunity of compute savings during false-negative cases as it would directly fall back to vanilla `DM` and

Algorithm 1 *CacheSelector*-Profiling($\mathcal{K}$, I_K^c, α)

for K in $\mathcal{K}$ **do**
  $[I_K] \leftarrow \text{model}(\mathcal{P}_Q, I_K^c, K) \quad \forall \quad \mathcal{P}_Q$
  $\text{min_sim} \leftarrow \min\{\text{sim } s \mid \forall I \in [I_K], \text{quality}(I) > \alpha\}$
  $\text{sim_K_map}[K] \leftarrow \text{min_sim}$
end for
return sim_K_map

generate an image from scratch instead of attempting to retrieve an intermediate state. For this classifier, NIRVANA uses One-Class Support Vector Machine (One-Class SVM) [20] which constructs a decision function for outlier detection. This is trained by utilizing all prompt embeddings stored in the VDB and assigning them a positive label of 1. To achieve high precision and recall, we effectively overfit the model to the existing prompt embedding space. Further, we use Stochastic Gradient Descent (SGD) to enable faster retraining of the classifier when embeddings in VDB change significantly (i.e., > 5%). NIRVANA achieves a $c_p = 0.95$ for production prompts.

Why not a Bloom Filter? Similar to our motivation, Bloom filters [29] are popularly used as a low-cost mechanism to check cache entries in various web services and database systems [19]. Bloom-filters use hashing algorithms to calculate if an item is likely to be present in the cache with zero false negative rate but with some false positive rate. However, Bloom filters are not suitable for NIRVANA as we do not search for an *exact match* for cache entries, we search for the *nearest neighbors* in the VDB by using ANN algorithms.

5.3 Cache Retrieval

Cache Retrieval and `cache-maintainer` are the main components of NIRVANA. In this section, we elaborate on the design of the cache retrieval phase which has three internal components VDB, `cache-selector`, and storage-system. In § 5.4 we discuss the design of `cache-maintainer`.

Vector Database: NIRVANA stores the embedding of the historical prompts in a vector-database (VDB) for fast and efficient similarity search with the embedding of incoming prompts. A VDB uses indexing methods like quantization, graphs, or trees to store and perform high-dimensional similarity search over vectors. For a query embedding vector, it can find m approximate nearest neighbors. In NIRVANA, for each incoming search with e_p, VDB already populated with the embeddings, its payload points to the path where the corresponding intermediate states of e_c at different Ks are stored. NIRVANA uses `cosine-similarity` as a measure to find the nearest neighbor. While NIRVANA can work with any VDB such as `Qdrant` [21], `Milvus` [18], `Weaviate` [23], `Elasticsearch` [13], in this paper we present results with `Qdrant` as benchmarking [3] shows low read latency at scale and moderately low update and delete latency.

Elastic File System as Storage: NIRVANA stores the actual intermediate noise states on AWS Elastic File System (EFS). After comparing with LustreFS [16] on read latency, throughput, and storage cost, we chose EFS as our storage system.

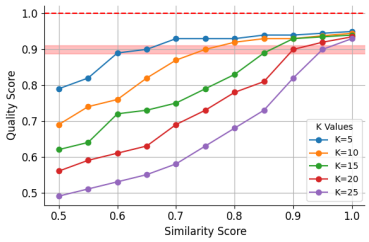


Figure 9: Quality of image generation with cache vs. similarity score across K. Thresholds for cache usage at different K values.

```
def cache_selector(s):
  if s > 0.95: k = 25
  elif s > 0.9: k = 20
  elif s > 0.85: k =
     15
  elif s > 0.75: k =
     10
  elif s > 0.65: k = 5
  else: k = 0
  return k
```

Figure 10: Determining K based on similarity score (s).

The size of the files containing intermediate-state depends on the architecture of the DM. For NIRVANA the default DM uses an intermediate-state of the size of 144 KB and stores for 5 distinct values of $K \in \mathcal{K} = \{5, 10, 15, 20, 25\}$ as increasing beyond $K = 25$ compromises output quality significantly and more granular Ks does not necessarily yield proportional gains but increases algorithmic complexity.

5.3.1 Cache Selector

The `cache-selector` component primarily determines which $K \in \mathcal{K}$ for e_c from EFS should be retrieved to recondition the intermediate state for the rest of the $N - K$ steps with prompt $\mathcal{P}_Q$ for maximum compute saving while maintaining acceptable image quality.

How do we choose the K? We observed empirically that the number of steps that can be skipped is correlated to the similarity between e_p and e_c. Based on this observation, we perform an offline profiling/characterization (Algorithm 1) to find the appropriate K such that the intermediate-state generated at K^{th} diffusion step is optimal based on the similarity score between the two prompt's embeddings (Eq. 10). The algorithm ("*CacheSelector*-Profiling(.)") generates images at each value of K for a set of prompts with their nearest cache prompt. It then finds the minimum similarity score such that all generated images are above a quality threshold α. This is then chosen as the minimum similarity at which the value of K works and is stored to be used later at run-time. As shown in Figure 9, we plot the image quality (higher the better) against various similarity scores between the query and the cached prompts across multiple values of K.

We use the profiled information to determine the optimal value of K for image generation at run-time. Using the similarity score s between e_p and e_c obtained from VDB, we choose a K such that the quality score at the plot $(x = K, \textit{similarity_score} = s)$ is $\geq \alpha$. We choose $\alpha = 0.9$. In simple words, we search the inverse map stored by Algorithm 1. The red band in Figure 9 represents the threshold α. Each line in the figure, corresponding to different K values, intersects this red band at the specific similarity scores above which that particular K value is applicable. Thus, if similarity score is high, we can skip a greater number of diffusion steps and choose higher K to achieve acceptable image quality. But, if similarity score is low, we can skip only a limited number of iterations and hence, a lower K. Figure 10 shows the logic

for computing the optimal K given a similarity score s between two prompts that is identified for DiffusionDB dataset using the previously discussed profiling technique. Based on this, we query the cache storage to obtain I_K^c. Once NIRVANA retrieves the appropriate intermediate noise from the cache , DM denoises it for N-K steps to generate the final image.

Quality vs. performance tradeoff: NIRVANA is designed to extract as much compute savings as possible without degrading the quality of images. However, if more aggressive compute savings is needed for certain use cases that are ready to sacrifice some accuracy, NIRVANA can trivially expose a *knob* that can be used to trade off quality vs. compute savings. The cache-selector heuristic can be biased to select higher values of K. This will provide more compute reduction while letting image quality degrade, as NIRVANA will get fewer steps to recondition the retrieved noise according to the new prompt. At the extreme, for $K = N$, NIRVANA will behave exactly like a pure image retrieval system [4, 7].

5.4 Cache Maintenance: LCBFU Policy

In popular production text-to-image system, there is a large stream of incoming prompts. The cost associated with total storage used increases, and VDB performance also degrades beyond a certain limit of entries if we keep on storing the cache. Therefore, even though NIRVANA can theoretically support infinite cache, the storage cost and increasing search latency would eventually make it unattractive. The cache-maintainer component works in the background to maintain the entries in the cache storage in EFS and in VDB.

To achieve this, NIRVANA uses a novel cache maintenance policy *Least Computationally Beneficial and Frequently Used* (LCBFU) customized for approximate caching in DMs. The fundamental idea is that not all intermediate states are equally beneficial for overall compute and latency savings. Intermediate states with high K values can provide huge compute savings but can be used only when $\mathcal{P}_Q$ has high similarity to one of the cached items. Items stored at low K are usable with a variety of prompts since they can be reconditioned even when the similarity with $\mathcal{P}_Q$ is relatively low, but provides low compute savings. With LCBFU, NIRVANA can limit a total of 1 TB cache in EFS without degrading any performance or quality, which on average corresponds to 1.5 million intermediate-states or noises and 300k unique prompt embeddings stored in the VDB.

Drawbacks of traditional cache policies: Due to the unique compute model of approximate-caching, the traditional cache policies such as LRU (least recently used), LFU (least frequently used), and FIFO (first-in first-out) are not very useful in NIRVANA, since they treat each item in the case homogeneously and only focus on the access patterns or arrival sequence while evicting an item. As discussed in § 2.1, hit-rate alone does not determine the efficiency of NIRVANA as items with large K values, even though less frequently accessed can provide significant overall compute reduction.

LCBFU: We design the LCBFU that takes into account both access frequency of items as well as potential compute benefit in case of a *hit*. It evicts items with least *LCBFU-score* which for each item i is calculated as $f_i \times K_i$. Here f_i is the access frequency of item i and K_i denotes which step of the denoising process this intermediate state belongs to. Notably, since the cache-selector heuristic determines K based on the similarity of e_p and e_c, an aggressive heuristic (high K for low similarity), will force high f_i for high K noises and hence low K noises will be evicted. For example, an item with $K_i = 25$ was accessed 100 times, its LCBFU-score is 2500, while an item with $K_j = 5$ was accessed 200 times, its LCBFU-score is 1000. Thus, the item i has a higher LCBFU-score since it will provide better compute savings while generating an image. LCBFU prioritizes compute savings while managing the trade-offs for optimal user experience at different K values.

The complete mechanism of LCBFU is described as follows:

- **Insertion:** With every cache miss, we directly insert all the intermediate states generated at diffusion denoising step $K \in \mathcal{K}$ to the cache-storage and the corresponding embedding of the prompt to the VDB. Thus $|\mathcal{K}|$ are stored in the cache storage per prompt. The insertions are performed without any eviction until we reach the target storage limit. After that, every insertion is preceded by an eviction.
- **Eviction:** For eviction, LCBFU maintains a running list of *LCBFU-score* in a K-min heap, and evict the top-$|\mathcal{K}|$ items from the heap root just before inserting $|\mathcal{K}|$ intermediate-states for a new prompt. The LCBFU-score evicts image noises which contribute least to compute savings.

With this cache eviction policy, cases can arise for a particular prompt where noises at some Ks are evicted, while noises at other Ks are still in the cache. This creates ***holes*** in the intermediate-states stored.

Handling of holes: Once eviction policy creates a hole, no straightforward way exists to fill that hole, as targeted regeneration of the intermediate state is not possible without running the full diffusion process. However, the heuristic used by cache-selector (§ 5.3.1) is oblivious to the existence of these holes while determining an appropriate K to be used with the retrieved prompt. NIRVANA handles this situation by choosing the intermediate state with the largest value K that is less than or equal to the optimal K (e.g. it can use K=10, if a hole is at K=15). This ensures that NIRVANA continues to generate high-quality images, albeit with little sacrifice in potential compute savings when it encounters the holes. However, we observed that such cases arise only in **4-5%** of the prompts, resulting in imperceptible performance degradation.

NIRVANA does not actively use any strategy to clean such holes. Only when all the intermediate-states for a prompt corresponding to all the $|\mathcal{K}|$ values turn into holes, LCBFU marks that prompt embedding as dirty and removes it from VDB as well as corresponding metadata from the storage-system.

cache-maintainer performs both insertions and deletions on VDB in batches, and at the same time, the classifier in the

match-predictor is also retrained with the fresh entries in the VDB. Recall, as mentioned in § 5.2 and § 5.3, both VDB update and classifier updates are fast and takes only around 7.5 and 0.04 seconds respectively for 10k records.

Discussions: In **Appendix** C, we explore NIRVANA's adaptability to image diversity, workload shifts, the generalizability of caching, LCBFU policy, and its use with ML optimizations.

6 Implementation

NIRVANA is implemented in Python using PyTorch [65] for diffusion model architecture, enabling user-friendly modules and seamless support for SYSTEM-X production integration. Our design also enables easy integration with image generation frameworks like Stable Diffusion. More details on batching and framework used are given in **Appendix** B

System Components: Our system components consist of (i) a classifier that uses *SGDOneClassSVM* [20] from scikit-learn [66] with $\nu = 0.001$, where ν controls the trade-off between training errors and support vectors, (ii) VDB hosted in a Docker container on an AWS m5.4xlarge EC2 instance with HNSW indexing of 256 for prompt embedding search [62], (iii) AWS EFS system that offers web and file system access, object storage, and scalability, (iv) MySQL database to record accesses of cache items, and (v) a text-to-image generator powered by a larger version of stable diffusion-based model with DDIM (Denoising Diffusion Implicit Model) sampler [76]. It generates images in $N = 50$ iterations utilizing approximately 8 GB of memory for a batch size of 1.

7 Evaluation

We first evaluate the overall effectiveness of NIRVANA in terms of quality and in providing significant compute and end-to-end latency savings, throughput, and cost savings for serving on the AWS cloud platform. We perform a user study with 60 participants to compare the image generation quality against baselines. We present LCBFU benefits against common cache-management policy to demonstrate the effectiveness of NIRVANA's internal component design. We also perform and present ablation studies. The key takeaways are:

- NIRVANA generates **high quality** image while **reducing** both GPU usage and end-to-end latency by up to **50%**.
- NIRVANA **reduces** all three - **cost**, **latency** and **compute** requirements of DM by **∼20%** on average.
- With a **27% improvement** in system **throughput**, NIRVANA ensures a stable user experience with minimal response time variations.

7.1 Methodology

Base Diffusion Model: As the base text-to-image model, we use a larger Stable Diffusion based model (which we call **VANILLA**), having approximately 1.5 times the number of parameters as compared to the 2.3B parameter Stable Diffusion XL model [67]. Our model operates within a 96-pixel latent space and performs $N = 50$ denoising steps to generate an image in 8.59 seconds, on average, on an A10g GPU. The final image generated is of size $768 \times 768 \times 3$.

Experimental Setup: We run the DM and embedding-generator of NIRVANA on a single NVIDIA A10g GPU (24 GB GPU memory), while the other components are run on a 32-core AMD EPYC 7R32 2.8 GHz CPU with 128 GB CPU memory. The GPU and the CPU machines were attached to a sub-network which included EFS and VDB. We optimize GPU usage by reusing prompt embeddings with diffusion model embeddings, reducing GPU overhead.

Dataset: We evaluate NIRVANA on two production datasets.

- **DiffusionDB** [79]: A total of 2M images for 1.5M unique prompts, with a total dataset size of $\approx 1.6\ TB$. We filtered the dataset and removed NSFW images
- SYSTEM-X: Prompts from production setup SYSTEM-X spanning over 8 weeks, containing over 7M images for 6.2M unique prompts, with a total size of $\approx 5\ TB$.

Unless otherwise mentioned, we evaluate the results on DIFFUSIONDB dataset, since SYSTEM-X data is proprietary.

Baselines: We compare two versions of NIRVANA: (1) with match-predictor (referred as NIRVANA) and (2) *without* match-predictor (referred as NIRVANA-w/oMP) against the following baselines:

- **GPT-CACHE**: Retrieves image for the closest prompt based on BERT embedding similarity. Otherwise, generates an image from scratch [7]
- **PINECONE**: Retrieves image for the closest prompt based on CLIP text embedding similarity. Otherwise, generates an image from scratch [4]
- **CRS (Clip Retrieval System)**: Clip Retrieval System [5] is another image retrieval method that uses the embedding of the final image generated by the previous prompts when retrieving the closest image for a given input prompt.
- **SMALLMODEL**: A smaller diffusion model [69] with 860M parameters, consuming only 33% of compute/latency compared to VANILLA, generating an image by 50 diffusion steps in 3.05 seconds on average on A10g.

Workload Generator and Cache Preload: The workload generator dispatched prompts in the order in which they arrived, using the dataset's arrival timestamp field to create a stream of prompts. Each prompt query was dispatched to the NIRVANA immediately upon the completion of the preceding query. This query stream began only after the initial 10k prompts were employed to preload the VDB and EFS with their respective caches for kick-starting the system.

Evaluation Metrics: We evaluate NIRVANA on various metrics covering both quality and efficiency aspects.

1. **Quality Metrics:**
   - **FID Score (Fréchet Inception Distance):** Computes the difference between two image datasets and correlates with human visual quality perception [47].
   - **CLIP Score:** Evaluates the alignment between generated images and their textual prompts [46]

- **PickScore:** A metric designed for predicting user preferences for generated images [53]

2. **Efficiency Metrics:** To evaluate the efficacy of NIRVANA in providing system efficiency, we report an average of 5 runs for % savings in GPU usage time, % reduction in end-to-end latency of image generation, % increase in throughput as number of images generated per second in a cluster and also amortized dollar-cost per image generation. We also report *hit-rate* $= h(min\mathcal{K})$ (Eq. 8) for NIRVANA.

7.2 Overall Performance on Quality

In this section, we evaluate how NIRVANA performs significantly better with respect to image generation quality.

Quantitative Generation Performance: Table 1 summarizes NIRVANA's improvements in terms of the quantitative image quality metrics. When compared against the retrieval-based baselines (§7.1), NIRVANA and NIRVANA-w/oMP improve performance significantly as captured by all three metrics [77]. Retrieval-based baselines directly retrieve the image generated from the most similar previous prompt for the query prompt. Hence, these methods fail to capture the differences between the incoming prompt and the retrieved prompt, since similarity metrics are unable to capture these (§3). Thus, all the retrieval-based baselines incur a significant hit in the quality of the image generated, which is of utmost importance in production user-facing text-to-image use cases. Compared to SMALLMODEL, both NIRVANA and NIRVANA-w/oMP perform far superior, which shows bigger models retain quality even with *approximate-caching*.

The metrics for VANILLA show the generated image quality without any kind of approximations. The performance of NIRVANA is very close to VANILLA when compared against CLIP and Pick score for both datasets. This shows that *approximate-caching* does not hurt the overall performance.

We measure FID (lower the better) of baselines against images generated using VANILLA for the same prompts. We also compute FID of VANILLA to indicate the inherent variability of the generated images without any change in the base model. To calculate FID for VANILLA, we generated 4 sets of images with 4 different seeds and calculated the FID between these resulting ${}^4C_2$ sets of images. As can be seen for both datasets, NIRVANA and NIRVANA-w/oMP exhibit much lower FID values than even the internal dissimilarities between different sets of generations by the VANILLA model. This means that images generated by NIRVANA will be indistinguishable from the VANILLA model - which is the design goal.

NIRVANA performs slightly better than NIRVANA-w/oMP due to the presence of `match-predictor` which generates an image from scratch during a predicted cache miss. However, this comes with a very small hit on NIRVANA's efficiency, compared to NIRVANA-w/oMP (see §7.3).

User Study for Accessing Quality: We conduct a user study with 60 participants to demonstrate the qualitative analysis. We evaluate 1000 randomly chosen prompts from DIFFUSIONDB, where each user was presented with 15 random `<prompt, generated-image>` pairs and were asked to vote `Yes`, or `No` based on whether the generated image properly represents the given prompt. Out of these 15 prompt-image pairs, 5 pairs were generated by GPT-CACHE, the best-performing image-retrieval-based baseline, next 5 pairs generated by NIRVANA and, next 5 using VANILLA. The images from these three types were shuffled and presented to each user in a random order and no prompt was repeated within a session. Users were also given the option to disclose reasons for `No`.

Dataset	Models	Quality		
		FID ↓	*CLIP Score* ↑	*PickScore* ↑
DiffusionDB	GPT-CACHE	7.98	25.84	19.04
	PINECONE	10.92	24.83	18.92
	CRS	8.43	24.05	18.84
	SMALLMODEL	11.14	25.64	18.65
	NIRVANA − *w/o*MP	4.94	28.65	20.35
	NIRVANA	**4.68**	**28.81**	**20.41**
	VANILLA	6.12-6.92	30.28	20.86
SYSTEM-X	GPT-CACHE	8.15	26.32	19.11
	PINECONE	10.12	24.43	18.83
	CRS	8.38	23.81	18.78
	SMALLMODEL	11.35	25.91	18.92
	NIRVANA − *w/o*MP	4.48	28.94	20.31
	NIRVANA	**4.15**	**29.12**	**20.38**
	VANILLA	5.42-6.12	30.4	20.71

Table 1: We compare* NIRVANA *against several baselines* GPT-CACHE, PINECONE, CRS *that are pure retrieval-based techniques, a smaller-model, and with vanilla diffusion model. CLIP score and PickScore are based on text-to-image score. FID is image-to-image comparisons, using* VANILLA *images as Ground Truth. Classifier has 0.96 Precision, 0.95 Recall.

In Figure 11, we show the ratio between `Yes` and `No` responses for each of the three models aggregated across all users. It can be seen that GPT-CACHE gets the lowest number of `Yes` votes, while NIRVANA is just marginally beneath the upper bound VANILLA. When analyzed for reasons of `No` (∼12% of the negative responses), we discovered that 80% of these reasons were for the images generated by GPT-CACHE. These insights underscore the challenges of relying solely on retrieval-based solutions for image generation and hence we do not use these baselines in further evaluation.

This reinforces the fact that NIRVANA *is much superior in maintaining image quality, which is of paramount importance for commercial deployment.*

7.3 System Efficiency of NIRVANA

We compare NIRVANA in terms of image generation latency, hit rate, compute savings, and cost of running NIRVANA.

Latency: Latency in Table 2 refers to the average end-to-end latency across two million prompt queries tested. We plot the nth percentile over the median of response latencies for some baselines. An image generation system should exhibit low latency for enhanced user experience. However, pure retrieval techniques like GPT-CACHE provide low latency but produce low-quality images and experience significant fluctuations in their end-to-end latency since they need to

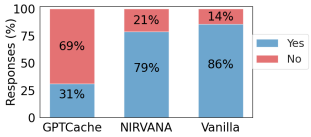


Figure 11: User survey response

	Latency(s)	90th/median	95th/median	99th/median
GPTCache	2.8	29.52	29.64	29.75
NIRVANA	6.9	1.20	1.21	1.21
VANILLA	8.6	1.01	1.02	1.02

Table 2: Average latency and n^{th} percentile over median values of the response latencies for approaches

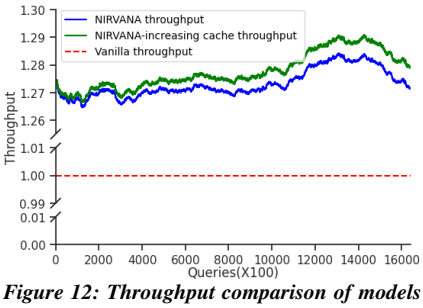


Figure 12: Throughput comparison of models against a stream of queries over time

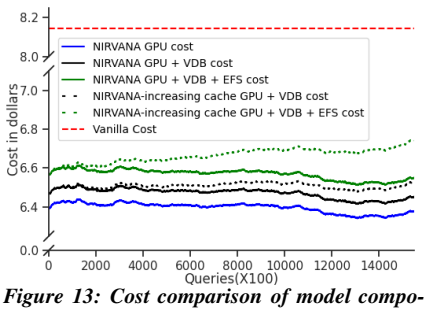


Figure 13: Cost comparison of model components against a stream of queries over time

run diffusion model from scratch for certain prompts that are dissimilar from the cached prompts, thus impacting the user experience [26] as shown in Table 2. NIRVANA reduces such variance in latency compared to the baselines and also the overall latency compared to VANILLA providing a much more stable and faster user experience. The key to minimizing response time variability lies in the smoother transition in compute time across different prompts, attributable to NIRVANA's ability to retrieve caches at various values of K.

This observation underscores that NIRVANA *not only reduces overall latency by 19.8% but also minimizes variance in response times across different prompts, ultimately providing a consistent and stable user experience [30].*

Throughput: We quantify throughput as the number of prompts processed per minute by the system. To evaluate this, we replay the stream of prompts from the DiffusionDB dataset. We assess the relative throughput of NIRVANA in comparison to VANILLA, considering two cache settings. The first setting employs LCBFU with a cache size of 1.5 million items, while the second setting involves an increasing cache configuration where no cache eviction occurs, effectively providing a theoretically infinite cache size. Figure 12 represents our findings, where the x-axis corresponds to the stream of queries, and we plot the relative throughput of the system. Notably, our results indicate that NIRVANA achieves ∼1.28 times higher throughput than VANILLA for both settings.

Cost of Image Generation: To provide a detailed cost breakdown for NIRVANA and VANILLA, we considered specific AWS components and their associated expenses, as per AWS pricing [1]. We use the g5.24xlarge GPU (96 GB GPU) instance from the US East region which costs $8.144 per hour. This cost was used to estimate the GPU-related expenses incurred by VANILLA and NIRVANA. VANILLA solely uses the GPU resources for each image generation. However NIRVANA also relies on additional components besides GPU, namely VDB [6] and EFS. For VDB, the cost is estimated at $0.12 per hour which covers storage and search operations performed over the prompt embeddings. Additionally, NIRVANA incurs costs associated with EFS. In the US East region, utilizing the standard storage type with 20% frequent access over elastic throughput cost amounts to $0.09 per hour. To determine the overall amortized cost of NIRVANA, we calculated it by dividing the GPU cost by the throughput and then adding the costs related to VDB and EFS. It is important to note that this cost analysis was conducted under identical settings for both systems. Despite the inclusion of these additional expenses, Figure 13 highlights that NIRVANA manages to achieve a remarkable 19% reduction in cost compared to VANILLA. This significant cost efficiency underscores the practical advantages of NIRVANA in real-world deployment.

Hit-Rate and Compute Savings: Figure 15 highlights the *hit-rate* and *compute-savings* of NIRVANA across K (x-axis). Averaging over all Ks, NIRVANA achieves a substantial *hit-rate* of 88% and noteworthy *compute savings* of 21% as indicated by the blue and black dotted lines. The cumulative *hit-rate* curve illustrates how the *hit-rate* varies with different values of K and plots $h(K)$. Additionally, the bar plot showcases the potential savings achievable at each K and the actual savings realized at specific K values. For instance, at $K = 25$, there is a potential savings of 50%, while the actual *hit-rate* is 8%, resulting in an actual savings of 4%.

7.4 LCBFU Performance

We experiment the effectiveness of LCBFU against common caching techniques like FIFO, LRU, and LFU. To ensure a fair evaluation, we maintained the same workload generator settings across all experiments with cold cache setting. FIFO removed the earliest added noises from the cache. LRU and LFU evict cache items based on their access frequency and recency. Table 3(a) reports *hit-rate* (Eq. 8) and % compute savings (% of GPU time savings when compared against VANILLA) for various cache eviction policies across different cache sizes. The *hit-rate* of LCBFU is comparable to LRU and LFU while outperforming FIFO. Notably, as highlighted in Table 3(b), the proposed LCBFU offers substantial compute savings compared to all other policies since it is designed to incorporate K with image access frequency for eviction. LCBFU is not designed to have the best hit rate.

7.5 Sensitivity Analysis

We now present some sensitivity analysis regarding NIRVANA's design choices. Additional sensitivity analyses are present in the **Appendix** D.

Embedding type: We conducted a comparison of NIRVANA using various types of embeddings for query and/or cache. Some embeddings are applied to the query as well as cache prompts, while other is applied just to the cached

Cache Size	#noises in cache	*FIFO*	*LRU*	*LFU*	LCBFU
1*GB*	1500	0.58	0.65	0.64	0.65
10*GB*	15000	0.68	0.77	0.78	0.77
100*GB*	150000	0.74	0.85	0.83	0.82
1000*GB*	1500000	0.83	0.95	0.94	0.93

(a) Hit rate

Cache Size	#noises in cache	*FIFO*	*LRU*	*LFU*	LCBFU
1*GB*	1500	0.11	0.12	0.12	0.12
10*GB*	15000	0.13	0.14	0.14	0.15
100*GB*	150000	0.14	0.16	0.16	**0.18**
1000*GB*	1500000	0.17	0.20	0.19	**0.23**

(b) % Compute savings

Table 3: Performance of different eviction techniques. Compute savings with `LCBFU` eviction is significant.

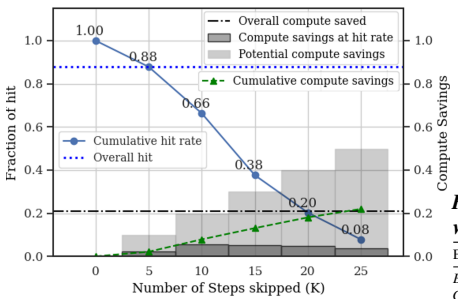


Figure 15: Hit rate and compute saved across K's

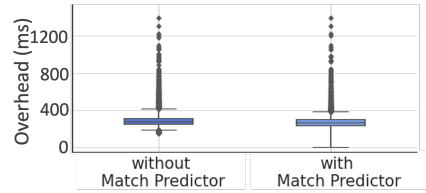


Figure 16: Overhead latency distribution with and without `match-predictor`.

Embedding	Prompt/Cache	FID↓	CLIPScore↑	HitRate↑	ComputeSavings↑
BERT	Query & Cache	4.53	28.94	0.77	0.16
CLIPImage	Only Cache	4.85	28.49	0.84	0.18
CLIP	Query & Cache	4.94	28.65	**0.93**	**0.23**

Table 4: Quality of generation for different embeddings.

prompt and CLIP embedding is maintained for the query prompt. This is indicated by the column 'Prompt/Cache'. The results, summarized in Table 4, indicate that the quality of image generation, assessed through metrics such as FID, CLIP Score, and PickScore remains consistent across different embedding types. However, the hit rate and compute savings achieved with the CLIP Text embeddings significantly outperform the other two embedding types. Hence, we selected CLIP Text embeddings for our system design.

Effectiveness of Match-Predictor: We conducted an evaluation to assess the effectiveness of using the `match-predictor` within NIRVANA, where we measured the average overhead in image generation latency. Figure 16 shows that `match-predictor` contributes significantly to lowering the overhead latency values as a considerable fraction of queries has negligible overhead, approximately equal to zero. This behavior is attributed to the `match-predictor`'s ability to promptly predict whether a particular prompt is present in the cache or not. It reduces the requirement for I/O-related activities, resulting in decreased latencies. Network delays in `VDB`/`EFS` cause right-tail latency. About 4% of these cases result from cache misses, reduced to zero with `match-predictor`. However, right-tail latency persists due to network call delays, even with `match-predictor`.

8 Related Works

ML optimizations. Various techniques like model distillation [63, 72], pruning [38], quantization [60] and others [50, 57, 85] exists for large `DMs` but often degrades the quality as well. DeepSpeed [35] is an optimization library for distributed inference and implements multiple techniques for the same. These can make the base model faster but are orthogonal to NIRVANA as they do not fundamentally change the nature of the iterative denoising process that NIRVANA exploits for compute reduction.

Model-serving in Cloud. Past research [33, 44, 58, 83, 86] explored efficient ML model serving to reduce inference latency. Clipper [33] implements optimizations like layer caching, ensembling methods, and straggler mitigation. Cocktail [44] designs a cost-effective ensemble-based model serving framework along with proactive autoscaling for resource management to provide high throughput and low latency. Other works [31, 32, 86] optimized the prediction pipeline cost during load variations. However, these are complimentary to NIRVANA and can be integrated easily. Tabi [78] uses multiple heterogeneous models for Large Language Models.

Text-to-image models. Diffusion model is one of the main classes of text-to-image models, which was popularized by Dall-E [12], Imagen [71] and Stable Diffusion [69]. Several other enterprises like Midjourney [17], Deci [15], and Adobe [9] have built their own diffusion models. Algorithmic optimizations include designing of faster sampling step [49, 54, 76, 80, 90] and parallel sampling [75, 90] and hence trading off compute for speed. However, our work is orthogonal, and the underlying diffusion model can be chosen from any of the above-mentioned works.

Caching. Caching in DNN inference has been explored in the past [33, 43, 55, 56] including caching intermediate layer outputs to avoid running every layer again on different input [33, 56]. Kumar *et al.* [56] coins the term *approximate caching* for above, but our semantics are orthogonal since we cache the intermediate image noise, and not the model layer outputs. None of the caching policies implemented in these works (LRU, static cache) work in our case. Other caching techniques [28, 84] are non-trivial to extend for our purpose. Retrieval-based works [4, 5, 7] uses caching to retrieve images for the most similar prompt, but suffer in quality (see §7).

Approximations in System Design: Various forms of approximations are employed in system designs to reduce redundant computation, enhancing efficiency in domains such as big data [24, 25, 27, 40, 41, 48, 64, 74], mobile computing [45, 81], and video processing [82, 88]. However, none of these approaches apply to `DMs`.

9 Conclusion

In this paper, we introduced the design and implementation of NIRVANA that uses a novel technique called *approximate-caching* to significantly reduce compute cost and latency during text-to-image generation using `DMs` by caching and reusing intermediate-states created while processing prior text prompts. We also presented a new cache management technique to optimize performance under a fixed storage.

References

[1] Aws product and service pricing | amazon web services. https://aws.amazon.com/pricing/?aws-products-pricing.sort-by=item.additionalFields.productNameLowercase&aws-products-pricing.sort-order=asc&awsf.Free%20Tier%20Type=*all&awsf.tech-category=*all. (Accessed on 09/20/2023).

[2] Complete guide to samplers in stable diffusion - félix sanz. https://www.felixsanz.dev/articles/complete-guide-to-samplers-in-stable-diffusion. (Accessed on 09/21/2023).

[3] Hugging face model repository. https://huggingface.co/models.

[4] Making stable diffusion faster with intelligent caching | pinecone. https://www.pinecone.io/learn/faster-stable-diffusion/. (Accessed on 09/21/2023).

[5] rom1504/clip-retrieval: Easily compute clip embeddings and build a clip retrieval system with them. https://github.com/rom1504/clip-retrieval. (Accessed on 09/21/2023).

[6] Vector search database | qdrant cloud. https://cloud.qdrant.io/calculator. (Accessed on 09/20/2023).

[7] zilliztech/gptcache: Semantic cache for llms. fully integrated with langchain and llama_index. https://github.com/zilliztech/GPTCache. (Accessed on 09/21/2023).

[8] Adobe express with ai-powered firefly integration now commercially available. https://news.adobe.com/news/news-details/2023/Adobe-Express-With-AI-Powered-Firefly-Integration-Now-Commercially-Available/default.aspx, 2023.

[9] Adobe firefly. https://www.adobe.com/sensei/generative-ai/firefly.html. 2023.

[10] Adobe unleashes new era of creativity for all with the commercial release of generative ai. https://news.adobe.com/news/news-details/2023/Adobe-Unleashes-New-Era-of-Creativity-for-All-With-the-Commercial-Release-of-Generative-AI/default.aspx, 2023.

[11] Aitemplate. https://github.com/facebookincubator/AITemplate, 2023.

[12] Dall-e 2. https://openai.com/dall-e-2. 2023.

[13] Elasticsearch. https://www.elastic.co/, 2023.

[14] Intel labs introduces ai diffusion model, generates 360-degree images from text prompts. https://www.businesswire.com/news/home/20230621842353/en/Intel-Labs-Introduces-AI-Diffusion-Model-Generates-360-Degree-Images-from-Text-Prompts, 2023.

[15] Introducing decidiffusion 1.0: : 3x the speed of stable diffusion with the same quality. https://deci.ai/blog/decidiffusion-1-0-3x-faster-than-stable-diffusion-same-quality/. 2023.

[16] Lustrefs. https://www.lustre.org/, 2023.

[17] Midjourney. https://www.midjourney.com/home/. 2023.

[18] Milvus - vector database. https://milvus.io/, 2023.

[19] Myrocks and bloom filters. https://mariadb.com/kb/en/myrocks-and-bloom-filters/, 2023.

[20] Oneclass svm. https://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html, 2023.

[21] Qdrant - vector database. https://qdrant.tech/, 2023.

[22] Stable diffusion batch prediction with ray data. https://docs.ray.io/en/latest/data/examples/stablediffusion_batch_prediction.html, 2023.

[23] Weaviate - vector database. https://weaviate.io/, 2023.

[24] Shubham Agarwal, Gromit Yeuk-Yin Chan, Shaddy Garg, Tong Yu, and Subrata Mitra. Fast natural language based data exploration with samples. In *Companion of the 2023 International Conference on Management of Data*, pages 155–158, 2023.

[25] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. {GRASS}: Trimming stragglers in approximation analytics. In *11th USENIX symposium on networked systems design and implementation (NSDI 14)*, pages 289–302, 2014.

[26] Xiao Bai, Ioannis Arapakis, B Barla Cambazoglu, and Ana Freire. Understanding and leveraging the impact of response latency on user behaviour in web search. *ACM Transactions on Information Systems (TOIS)*, 36(2):1–42, 2017.

[27] Martin Beck, Pramod Bhatotia, Ruichuan Chen, Christof Fetzer, Thorsten Strufe, et al. {PrivApprox}:{Privacy-Preserving} stream analytics. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 659–672, 2017.

[28] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. {LHD}: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, 2018.

[29] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[30] Peter M Broadwell. *Response time as a performability metric for online services*. Computer Science Division, University of California, 2004.

[31] Lequn Chen, Weixin Deng, Anirudh Canumalla, Yu Xin, Matthai Philipose, and Arvind Krishnamurthy. Symphony: Optimized model serving using centralized orchestration. *arXiv preprint arXiv:2308.07470*, 2023.

[32] Daniel Crankshaw, Gur-Eyal Sela, Corey Zumar, Xiangxi Mo, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. Inferline: Ml prediction pipeline provisioning and management for tight latency objectives, 2020.

[33] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A {Low-Latency} online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[34] Florinel-Alin Croitoru, Vlad Hondru, Radu Tudor Ionescu, and Mubarak Shah. Diffusion models in vision: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.

[35] deepspeed.ai. Deepspeed. https://www.deepspeed.ai/, 2023.

[36] Prafulla Dhariwal and Alexander Nichol. Diffusion models beat gans on image synthesis. *Advances in neural information processing systems*, 34:8780–8794, 2021.

[37] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[38] Gongfan Fang, Xinyin Ma, and Xinchao Wang. Structural pruning for diffusion models. *arXiv preprint arXiv:2305.10924*, 2023.

[39] Paul A Gagniuc. *Markov chains: from theory to implementation and experimentation*. John Wiley & Sons, 2017.

[40] Shaddy Garg, Subrata Mitra, Tong Yu, Yash Gadhia, and Arjun Kashettiwar. Reinforced approximate exploratory data analysis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 7660–7669, 2023.

[41] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 383–397, 2015.

[42] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, 2014.

[43] Edouard Grave, Armand Joulin, and Nicolas Usunier. Improving neural language models with a continuous cache. In *International Conference on Learning Representations*, 2017.

[44] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. Cocktail: A multidimensional optimization for model serving in cloud. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1041–1057, 2022.

[45] Peizhen Guo, Bo Hu, Rui Li, and Wenjun Hu. Foggycache: Cross-device approximate computation reuse. In *Proceedings of the 24th annual international conference on mobile computing and networking*, pages 19–34, 2018.

[46] Jack Hessel, Ari Holtzman, Maxwell Forbes, Ronan Le Bras, and Yejin Choi. Clipscore: A reference-free evaluation metric for image captioning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7514–7528, 2021.

[47] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30, 2017.

[48] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. Deepdb: Learn from data, not from queries! *arXiv preprint arXiv:1909.00607*, 2019.

[49] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

[50] Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2021.

[51] Jina.ai. Benchmark vector search databases with one million data. https://jina.ai/news/benchmark-vector-search-databases-with-one-million-data/, 2022.

[52] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[53] Yuval Kirstain, Adam Polyak, Uriel Singer, Shahbuland Matiana, Joe Penna, and Omer Levy. Pick-a-pic: An open dataset of user preferences for text-to-image generation. *arXiv preprint arXiv:2305.01569*, 2023.

[54] Zhifeng Kong and Wei Ping. On fast sampling of diffusion probabilistic models. In *ICML Workshop on Invertible Neural Networks, Normalizing Flows, and Explicit Likelihood Models*, 2021.

[55] Roland Kuhn. Speech recognition and the frequency of recently used words: A modified Markov model for natural language. In *Coling Budapest 1988 Volume 1: International Conference on Computational Linguistics*, 1988.

[56] Adarsh Kumar, Arjun Balasubramanian, Shivaram Venkataraman, and Aditya Akella. Accelerating deep learning inference via freezing. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[57] Fan Lai, Yinwei Dai, Harsha V Madhyastha, and Mosharaf Chowdhury. {ModelKeeper}: Accelerating {DNN} training via automated training warmup. In *NSDI*, 2023.

[58] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. {PRETZEL}: Opening the black box of machine learning prediction serving systems. In *OSDI*, 2018.

[59] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[60] Xiuyu Li, Yijiang Liu, Long Lian, Huanrui Yang, Zhen Dong, Daniel Kang, Shanghang Zhang, and Kurt Keutzer. Q-diffusion: Quantizing diffusion models. *arXiv*, 2023.

[61] Luping Liu, Yi Ren, Zhijie Lin, and Zhou Zhao. Pseudo numerical methods for diffusion models on manifolds. *arXiv preprint arXiv:2202.09778*, 2022.

[62] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[63] Chenlin Meng, Robin Rombach, Ruiqi Gao, Diederik Kingma, Stefano Ermon, Jonathan Ho, and Tim Salimans. On distillation of guided diffusion models. In *CVPR*, 2023.

[64] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. Verdictdb: Universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1461–1476, 2018.

[65] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[66] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[67] Dustin Podell, Zion English, Kyle Lacey, Andreas Blattmann, Tim Dockhorn, Jonas Müller, Joe Penna, and Robin Rombach. Sdxl: improving latent diffusion models for high-resolution image synthesis. *arXiv preprint arXiv:2307.01952*, 2023.

[68] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021.

[69] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022.

[70] O Ronneberger, P Fischer, and T Brox. Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015 Conference Proceedings*, 2022.

[71] Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily L Denton, Kamyar Ghasemipour, Raphael Gontijo Lopes, Burcu Karagol Ayan, Tim Salimans, et al. Photorealistic text-to-image diffusion models with deep language understanding. *Advances in Neural Information Processing Systems*, 35:36479–36494, 2022.

[72] Tim Salimans and Jonathan Ho. Progressive distillation for fast sampling of diffusion models. In *International Conference on Learning Representations*, 2021.

[73] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. Dbscan revisited, revisited: why and how you should (still) use dbscan. *ACM Transactions on Database Systems (TODS)*, 42(3):1–21, 2017.

[74] Nikhil Sheoran, Subrata Mitra, Vibhor Porwal, Siddharth Ghetia, Jatin Varshney, Tung Mai, Anup Rao, and Vikas Maddukuri. Conditional generative model based predicate-aware query approximation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 8259–8266, 2022.

[75] Andy Shih, Suneel Belkhale, Stefano Ermon, Dorsa Sadigh, and Nima Anari. Parallel sampling of diffusion models, 2023.

[76] Jiaming Song, Chenlin Meng, and Stefano Ermon. Denoising diffusion implicit models. In *International Conference on Learning Representations*, 2020.

[77] Jianyi Wang, Kelvin CK Chan, and Chen Change Loy. Exploring clip for assessing the look and feel of images. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 2555–2563, 2023.

[78] Yiding Wang, Kai Chen, Haisheng Tan, and Kun Guo. Tabi: An efficient multi-level inference system for large language models. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 233–248, 2023.

[79] Zijie J Wang, Evan Montoya, David Munechika, Haoyang Yang, Benjamin Hoover, and Duen Horng Chau. Diffusiondb: A large-scale prompt gallery dataset for text-to-image generative models. *arXiv preprint arXiv:2210.14896*, 2022.

[80] Zike Wu, Pan Zhou, Kenji Kawaguchi, and Hanwang Zhang. Fast diffusion model, 2023.

[81] Mengwei Xu, Xiwen Zhang, Yunxin Liu, Gang Huang, Xuanzhe Liu, and Felix Xiaozhu Lin. Approximate query service on autonomous iot cameras. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 191–205, 2020.

[82] Ran Xu, Jinkyu Koo, Rakesh Kumar, Peter Bai, Subrata Mitra, Sasa Misailovic, and Saurabh Bagchi. {VideoChef}: Efficient approximation for streaming video processing pipelines. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 43–56, 2018.

[83] Neeraja J Yadwadkar, Francisco Romero, Qian Li, and Christos Kozyrakis. A case for managed and model-less inference serving. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 184–191, 2019.

[84] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518, 2021.

[85] Miao Yin, Yang Sui, Siyu Liao, and Bo Yuan. Towards efficient tensor decomposition-based dnn model compression with optimization framework. In *CVPR*, 2021.

[86] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. {MArk}: Exploiting cloud services for {Cost-Effective},{SLO-Aware} machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, 2019.

[87] Chenshuang Zhang, Chaoning Zhang, Mengchun Zhang, and In So Kweon. Text-to-image diffusion model in generative ai: A survey. *arXiv preprint arXiv:2303.07909*, 2023.

[88] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. Live video analytics at scale with approximation and {Delay-Tolerance}. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 377–392, 2017.

[89] Yuxin Zhang, Weiming Dong, Fan Tang, Nisha Huang, Haibin Huang, Chongyang Ma, Tong-Yee Lee, Oliver Deussen, and Changsheng Xu. Prospect: Expanded conditioning for the personalization of attribute-aware image generation. *arXiv preprint arXiv:2305.16225*, 2023.

[90] Hongkai Zheng, Weili Nie, Arash Vahdat, Kamyar Azizzadenesheli, and Anima Anandkumar. Fast sampling of diffusion models via operator learning. In *International Conference on Machine Learning*, pages 42390–42402. PMLR, 2023.

A NIRVANA Algorithm

NIRVANA first uses the `match-predictor` to predict if there is a close match for the prompt query $\mathcal{P}_Q$. If the prediction is yes, it makes a `VDB` call to get the nearest prompt and finds the K using heuristics (see 5.2). Next, it retrieves the intermediate image noise at K^{th} step from `EFS` and passes it to the diffusion model to generate the final image in $N-K$ steps. If there is no match predicted or found, NIRVANA takes fallback to generate an image from scratch. Note, using `match-predictor` reduces the case where `VDB` call is made for *cache miss* cases.

Algorithm 2 GenerateImageCache($\mathcal{P}_Q$)

$P_e \leftarrow$ Embed(P)
if MatchPredictor(P_e) is $True$ **then**
  // Generate using cache
  (neigh, score) $\leftarrow$ search_VDB(P_e)
  $K \leftarrow$ heuristics_K(score)
  if $K \neq 0$ **then**
    $path_K \leftarrow$ neigh['payload']['noise'][K]
    $c_noise \leftarrow$ retrieve_EFS($path_K$)
    $I \leftarrow$ model(P, c_noise, K)
  else
    // No suitable cache found, generate from scratch
    $I \leftarrow$ model(P, null, 0)
  end if
else
  // Generate from scratch
  $I \leftarrow$ model(P, null, 0)
end if
return I

B Implementation Details

Batch Processing: While deploying, our diffusion model takes up around 80% of GPU memory with batch size 1, so concurrent batching isn't supported.

AITemplate (AIT): We leverage AITemplate (AIT) [11], a Python framework to accelerate inference serving of the PyTorch-based diffusion model by converting it into CUDA (NVIDIA GPU) / HIP (AMD GPU) C++ code. It provides high-performance support during the inference process.

C Discussions

Image diversity. Since NIRVANA reuses previous intermediate states from the cache, over time the diversity of images generated by the system can be reduced if the majority of the prompts encountered by the system are very similar. This can be addressed in the following ways: first, by actively changing the `seed` of the denoising process after retrieval. We observed that for $K < 35$ this can increase diversity. However, if prompts become too similar, then NIRVANA would attempt to save compute more aggressively using a higher K where `seed` change becomes ineffective. In that case, NIRVANA can

be designed to let an ε fraction of prompts follow the vanilla diffusion process to maintain the diversity of images.

Change in prompt characteristics. If the characteristics of the prompts suddenly change due to some external factors, NIRVANA will find less similar items in the cache and will automatically move towards lower K, or even for more dissimilar prompts the `match-predictor` will kick in and redirect the prompts to vanilla diffusion process. While this will reduce compute savings, *by design* NIRVANA will not let the image quality degrade in case of a sudden change in prompt characteristics.

Generalizability of Approximate Caching: The concept of *approximate caching* easily extends to iterative generation approaches. It can also be applied to other architectures, such as Transformers (in LLMs) in sequential decoding [59], or at the layer level [56] for models like U-Net.

Cache management policy: The proposed `LCBFU` policy prioritizes compute savings, particularly with higher K values. However, there's a nuanced trade-off, notably for frequently accessed prompts at lower K (e.g., $K = 5$) v/s less frequent access at higher K (e.g., $K = 25$). Eviction at lower K may seem to impact the user experience by removing frequently used cache, but a lower $K = 5$ facilitates alternative matches with a lower similarity score requirement. Preserving $K = 25$ is critical for its similarity to specific prompts and substantial compute savings; evicting it might result in the absence of a highly similar prompt. Hence, (`LCBFU`) aims for maximum compute optimization with minimal impact on user experience.

ML optimizations. Several techniques are used to reduce the compute footprint and latency of models such as the use of lower precision [60], distillations [63, 72], pruning [38], batched-inference optimizations [22]. NIRVANA is complementary to such techniques as it can be used on top of those optimized models as well to reduce redundant computation using approximate caching. However, if a new family of generative models emerges that does not require such a large number of iterative steps, then NIRVANA's applicability will become limited. But since as of today, `DMs` with 50 or more denoising steps produce the best and production quality images, NIRVANA provides an attractive proposition for compute reduction.

D Additional Results from Sensitivity Analysis

D.1 Match-Predictor Settings

The *SGDOneClassSVM* `match-predictor` can produce binary predictions (0 or 1) by employing various thresholds. These thresholds influence the Precision (P) and Recall (R) values obtained from the `match-predictor`. To determine the optimal settings, we conducted an ablation study, measuring the overhead latency under different P and R configurations. The resulting plot in Figure 17 led us to select the settings with a $P = 96$ and $R = 95$. The choice is made to prioritize high precision, aiming to minimize false positives. Simultaneously, we aim to maintain a high recall to avoid missing opportunities for using *approximate-caching*.

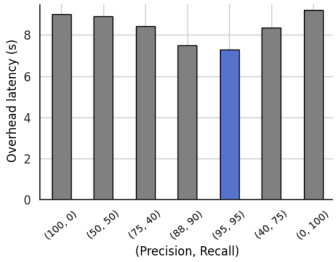


Figure 17: Hit rate and compute saved across K's

D.2 Decomposition of End-to-End Latency in NIRVANA

In Figure 18 we show the end-to-end latency of VANILLA diffusion model and also how different components of NIRVANA contribute towards its end-to-end latency. We can see noise retrieval from EFS and VDB search is the main contributor to the overhead.

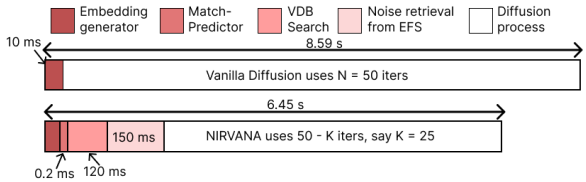


Figure 18: Time taken by different components of VANILLA and NIRVANA for generating an image using 50 steps on A10g GPU instance.

D.3 Image Quality across Long vs. Short Prompts

As we discussed in §3, the prompt queries can be either long or short. We perform ablation to see how the system works with them. The ablation results presented in Table 5 indicate that NIRVANA performs more effectively with slightly shorter prompts compared to very lengthy ones. This disparity in performance can be intuitively attributed to the retrieval technique employed. Longer prompts tend to challenge the ability of embeddings (in our case, CLIP) to capture the context adequately. If the prompt retrieved from the cache significantly deviates from the query despite high similarity based on *CLIPText* embeddings (as discussed in §3), it may result in the generation of incoherent images

Prompt	*FID* ↓	*CLIPScore* ↑
Short	7.96	28.48
Long	11.48	28.96

Table 5: Generation of short v/s long prompts. Less than 15 words are considered short.

D.4 Quality with different Caching Policy

We conducted an evaluation to compare image quality metrics, including *FID*, *CLIPScore*, and *PickScore*, while using different cache eviction techniques. The results, presented in Table 6, demonstrate that image generation quality remains

consistent across all caching techniques. Therefore, the choice of eviction mechanism should prioritize improved compute savings and hit rates.

Policy	*FID* ↓	*CLIPScore* ↑	*PickScore* ↑
FIFO	5.12	28.25	20.31
LRU	4.82	28.54	20.38
LFU	4.98	28.61	20.42
LCBFU	4.94	28.65	20.41

Table 6: Quality of generation for different eviction techniques with 1500 GB cache.

E Concept Development in Image Generation

This section gives a motivating example of how various concepts/characteristics develop during image generation [89]. In Figure 19, we present an illustrative example of image

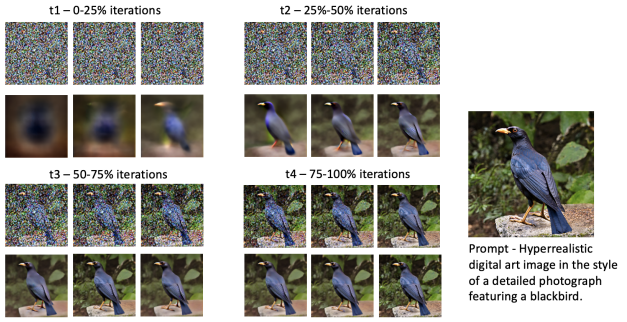


Figure 19: Generation across K

generation using a prompt that encompasses various aspects, including color, layout, content, size, style, and more. We divide the generation steps into four distinct time buckets, labeled t1, t2, t3, and t4, with each bucket having a unique role in shaping different facets of the image. The formation of color initiates in t1 and becomes relatively stable by t2. Style, here referring to generating a digital art image in the style of a detailed photograph, commences in t1 and experiences significant development in t2. The image's content, featuring a bird and leaves, begins to manifest partially towards the end of t2, with substantial development occurring in t3. Layout, which dictates the positioning of elements like the blackbird, starts its formation in t1 and progresses towards the end of this phase. By the time we reach t4, most aspects of the image are firmly established, with this final bucket primarily responsible for fine-tuning and enhancing details and clarity.

THC: Accelerating Distributed Deep Learning Using Tensor Homomorphic Compression

Minghao Li[††], Ran Ben Basat[†], Shay Vargaftik[§], ChonLam Lao[††],
Kevin Xu[††], Michael Mitzenmacher[††], Minlan Yu[††]

Harvard University [††], University College London [†], VMware Research [§]

Abstract

Deep neural networks (DNNs) are the de facto standard for essential use cases, such as image classification, computer vision, and natural language processing. As DNNs and datasets get larger, they require distributed training on increasingly larger clusters. A main bottleneck is the resulting communication overhead where workers exchange model updates (i.e., gradients) on a per-round basis. To address this bottleneck and accelerate training, a widely-deployed approach is compression. However, previous deployments often apply bi-directional compression schemes by simply using a uni-directional gradient compression scheme in each direction. This results in significant computational overheads at the parameter server and increased compression error, leading to longer training and lower accuracy.

We introduce Tensor Homomorphic Compression (THC), a novel bi-directional compression framework that enables the direct aggregation of compressed values and thus eliminating the aforementioned computational overheads. Moreover, THC is compatible with in-network aggregation (INA), which allows for further acceleration. Our evaluation shows that training representative vision and language models with THC reaches target accuracy by 1.40× to 1.47× faster using INA and 1.28× to 1.33× faster using a software PS compared with state-of-the-art systems.

1 Introduction

In the past decade, the scale of machine learning training and data volume has increased dramatically due to the growing demand for various ML applications [11,13,23,54,62,70,72,76]. Alibaba's general-purpose ML platforms also reported a rapid increase in ML training data, from hundreds of gigabytes to tens or even hundreds of terabytes, at an internet scale, within a few years [75]. This trend is expected to continue in the future [59] with the rapid advancements of giant models [11, 13, 44, 52, 60]. To support these large-scale models, we need large-scale distributed training [19, 30, 51, 71].

However, distributed training incurs high communication overhead. Recent research [73] has shown that the synchronization cost of GPT2 [50] and BERT-base [14] in a 8-worker setting can be as high as 42% and 49% of the total time during training, even with state-of-the-art frameworks. As the number of workers increases, the communication overhead rises substantially [59]. Meanwhile, computing devices are pushing more data into the network with specialized ML accelerators [18, 42, 47, 77] and more advanced GPU/TPU hardware, which further increases the communication overhead [65, 80].

To reduce the communication overhead, many compression schemes have been conceived [6, 10, 15, 64, 73, 74]. One common problem of these solutions is that they apply bi-directional compression. For example, in the Parameter Server (PS) architecture, the PS nodes first decompress all gradients, aggregate them, and then compress the aggregated gradients again. Such compression and decompression operations result in significant computational overheads and affect the training convergence time and attainable accuracy (see Section §2.1).

To address these problems, we introduce Tensor Homomorphic Compression (THC), a novel bi-directional compression framework enabling the direct aggregation of compressed tensors (e.g., gradients) without first decompressing them, eliminating much of the aforementioned computational overhead. Additionally, direct aggregation can also run on programmable switches for further acceleration.

This paper focuses on developing a THC framework to reduce the communication overhead of data parallelism for the parameter server architecture. Importantly, PS is effective in GPU/CPU hybrid clusters [28, 39, 46, 73], which are common in clouds [27]. Furthermore, when we colocate a PS with each worker, it essentially functions as an AllReduce [28].

From an algorithmic standpoint, the technical challenge is designing an algorithm that enables workers to accurately compress their gradients in a way that allows aggregating their results without decompressing each worker's message. We propose *Homomorphic Compression* – a property that enables this and develops efficient schemes that satisfy it while optimizing the accuracy.

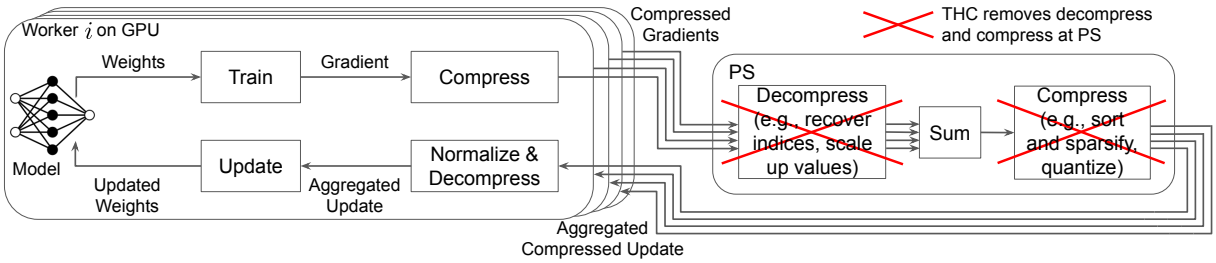
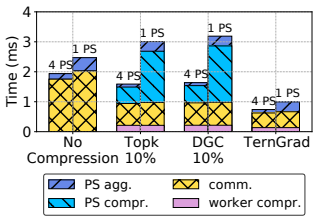
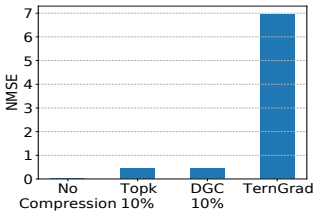


Figure 1: End-to-end workflow when using typical compression algorithms.



(a) Communication round time of one 4MB partition.



(b) NMSE of various compression schemes with four workers.

Figure 2: Communication time and error.

A key idea behind THC's ability to satisfy this property is an initial communication stage with minimal information exchange between the workers at the beginning of each round that allows them to *coordinate* and ensure that their compressed gradients are directly aggregable. To ensure high accuracy, THC does not directly encode the gradients but rather pre- and post-processes them with the GPU-friendly Randomized Hadamard Transform (RHT) to transform the gradients to a different representation that is amenable to accurate quantization. Since RHT preserves the tensor sizes (i.e., norms), by merely exchanging these norms during a preliminary light communication stage (a single float per client), the clients can align their quantization values such that they can be averaged without decompression. Furthermore, this communication step overlaps with applying the RHT transform and thus does not increase the compression time. THC also employs an advanced non-uniform quantization technique we developed and an error-feedback mechanism to further improve the bandwidth to accuracy tradeoff.

We built THC on top of the BytePS [28] PyTorch extension. Our PS can run on either software or programmable switches. We perform extensive evaluation over seven representative DNN models on a local testbed and AWS EC2. Testbed results show that THC achieves the target accuracy 1.42× to 1.47× faster with aggregation on a programmable switch and 1.28× to 1.33× with a software PS, compared to the state-of-the-art distributed training framework (Horovod RDMA). THC with the programmable switch also improves the training throughput by up to 54% over Horovod RDMA. [1]

[1] THC is available at https://github.com/SophiaLi06/BytePS_THC.

2 Background and Motivation

In this section, we give the background on compression and in-network aggregation and motivate the need for direct aggregation on compressed data. Gradient compression, a well-studied approach to lower the network communication of distributed training, reduces the volume of gradients transmitted in synchronization steps at the expense of convergence speed and accuracy. There are two key techniques: sparsification and quantization. Sparsification may filter out coordinates of small magnitude in the gradient tensor. TopK [64] is a straightforward sparsification algorithm that only sends the top k percent (by magnitude) of coordinates and their indices. Sparsification becomes lossy when many coordinates are nonzero. Quantization reduces the size of each element by reducing the precision of gradients. For example, TernGrad [74] reduces the bit length of the gradient coordinates to two bits by converting each float into a value $x \in \{-1, 0, 1\}$. Different compression techniques offer various accuracy, bandwidth, and time complexity tradeoffs.

2.1 Compression Cost and Tradeoffs

Figure 1 shows the bi-directional compression process in existing PS systems [6, 73]. Workers send compressed gradients to the PS. The PS decompresses and aggregates gradients. Then, to reduce the traffic back to workers, the PS compresses the aggregated results again before transmission. In such a design, while compression reduces the communication cost, decompressing and compressing data on PS nodes introduce high compute overhead and additional compression errors.

To quantize the computational overhead and estimation error of compression schemes, we run a microbenchmark transmitting a single 4MB partition (the recommended partition size that balances pipelining efficiency and system overheads as specified in BytePS repository [12]) on our local testbed. Training frameworks usually batch gradients and chunk them into same size partitions before communication [35, 46]. Since communication time grows linearly with the number of partitions, we simply measure the times for a single partition in our microbechmark. We measure the worker-side *compression and decompression time ("worker compr.")*, the PS side *compression and decompression time ("PS compr.")*, the PS *aggregation time ("PS agg.")* and the worker-PS *communication time ("comm.")* as shown in Figure 2a.

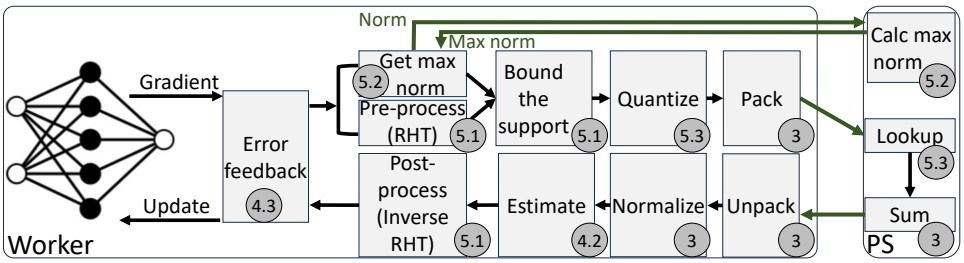


Figure 3: An illustration of THC, with the section numbers of each component.

With one PS and four workers, two sparsification schemes TopK 10% and DGC 10% [38] that communicate the top 10% coordinates by magnitude slow down the end-to-end time by 19.3% and 27.1% of the no-compression round time. This is due to the high PS computational overhead of compression and decompression that contributes up to 56.9% of the round time even when the communication time is reduced. Note that the computational overhead of TopK and DGC grow as the PS aggregates more coordinates, making it a poor choice when the worker-to-parameter server ratio is imbalanced.

When we use colocated PS (i.e., have four PS in total), TopK 10% reduces the communication time by 60.4% of that of no compression but takes an extra 0.54 ms (34.0% of the round time) to run compression/decompression on the PS. The end-to-end round time reduction is therefore diluted to 20.6% of that of no compression.

One can use other compression schemes (e.g., TernGrad) that take less time for decompression/compression at the PS side. However, these schemes have larger quantization errors. Figure 2b shows the NMSE (Normalized Mean Squared Error, $NMSE(x,\hat{x}) = \frac{||x-\hat{x}||_2^2}{||x||_2^2}$), which quantifies the difference between the actual vector x and the vector restored after compression $\hat{x}$. TernGrad results in an NMSE that is by an order of magnitude larger than that of TopK 10% (i.e., 6.95 vs. 0.46 with four workers). This large gap in NMSE means that TernGrad requires more iterations to reach the target accuracy or might fail to reach the target accuracy at all. In fact, provable convergence rates for distributed SGD have a linear dependence on NMSE (e.g., [29]), rendering quantization schemes with large NMSE less appealing for distributed training.

To address these limitations, THC allows direct aggregation of compressed gradients, which eliminates decompression and compression operations at PSes while ensuring high accuracy. A detailed comparison between THC and other compression schemes under different bit budgets is in Section §8.4.

2.2 In-network Aggregation

In-network aggregation [45] is another option to reduce communication overhead. Recent research [33, 57] has demonstrated that programmable switches can aggregate gradients from multiple workers, reducing the switch-to-PS traffic and resulting in a substantial training performance improvement. However, using switches does not reduce the traffic volume generated by the workers as gradient compression does.

Most existing compression solutions are incompatible with in-network aggregation solutions because switches can not easily decompress and compress the data due to their programmability and resource limitations [33, 49]. Since THC supports direct aggregation over compressed data, it only requires the PS to do summation, which programmable switches can readily perform with their ALUs. This also offers new opportunities for incorporating compression with in-network aggregation to further improve training performance.

3 THC Overview

We propose the *Tensor Homomorphic Compression (THC)* framework, which allows the PS to merely aggregate the incoming compressed gradients and transmit the (still compressed!) aggregated values back to the workers. THC hence enables us to avoid the computational overhead of compression and decompression at PS while still having accurate estimation of the gradients' average. We first introduce the homomorphic compression property to model such system constraints. Consider n workers and let ∇_i be the i-th worker's gradient. We define the Uniform Homomorphic Compression (UHC) property as follows:

Definition 1 (Uniform Homomorphic Compression)

$$\widehat{\nabla}_{avg} = \frac{1}{n} \cdot \sum_i \texttt{Decompress}(\texttt{Compress}(\nabla_i))$$
$$= \texttt{Decompress}\left(\frac{1}{n} \cdot \sum_i \texttt{Compress}(\nabla_i)\right).$$

That is, with the Uniform Homomorphic Compression property, the average of the decompressed gradients is mathematically equivalent to decompressing the average compressed gradient. Leveraging this property, the PS simply sums the compressed gradients and sends the result back (still in compressed form). Finally, each worker averages the result and applies the decompression to derive the update $\widehat{\nabla}_{avg}$.

The key challenge for THC is to design compression algorithms that retain the UHC property while ensuring high accuracy. Non-homomorphic compression techniques require the PS to decompress the gradients before aggregation because they rely only on worker-local information. When workers use different quantization ranges (e.g., [4, 53]), the PS must decompress each gradient separately, sum them up, and compress again, increasing processing delay at the PS side and

the errors caused by compression. We know of one previous compression scheme, SignSGD [10], that is homomorphic as it simply counts, for each coordinate, the number of workers which had a positive value for it. However, this scheme is biased, and thus its error does not decrease with the number of workers, making it yield large errors in practice. In THC, we retain an accuracy that is similar to that of the uncompressed baseline and achieve the UHC property.

We present the THC framework, illustrated in Figure 3. The THC workflow starts by having workers compress their gradients, which commonly consist of 32-bit floats. Each coordinate is quantized and then encoded into a *table index* (formally introduced in Section §4.2) that requires a small number of bits. The table indices are then packed and sent to the PS. The PS looks the table indices up in a lookup table to restore the corresponding *table values* and sums up the looked-up table values coordinate-wise. After summing values from all workers, PS packs the result and broadcasts it. Finally, each worker decompresses and normalizes the result to obtain the average gradient's estimate.

The table here serves two purposes: first, it allows an efficient expansion of the indices to wider values that allow summation without overflows. Second, we can use the table to minimize the quantization error, as we later show, by picking the table values in correspondence to the underlying data distribution. Since the table is small (of size 2^b, where b is a small constant) and hardcoded (does not depend on the gradients or number of workers), and lookups do not require arithmetic operations, we consider it as part of the direct aggregation.

Figure 4 visualizes the THC implementation we adopt in our system prototype. Namely, each 32-bit coordinate is encoded into a 4-bit *table index*, which then gets converted into a 8-bit *table value* on the PS. The broadcast summation result also uses 8 bits per coordinate. Therefore, our system prototype provides a $\times 8$ bandwidth reduction from workers to the PS and a $\times 4$ bandwidth reduction in the other direction.

The THC algorithm is described in detail in Section §4; in Section §5, we introduce further optimizations for THC and the end-to-end training procedure; then, Section §6 shows how THC can be seamlessly used in conjunction with in-network aggregation to further accelerate the training.

4 Tensor Homomorphic Compression

In this section, we explain how to achieve the UHC property effectively. We start by giving background on stochastic quantization, a core building block of our THC approach. We then show (Section §4.2) that stochastic quantization with uniform intervals, a technique that has been used previously in compression, has the UHC property. However, its performance in terms of the accuracy per bit is relatively poor, because it is unoptimized. This poor compression performance can lead to worse training time and/or model accuracy than an uncompressed baseline. We, therefore, introduce further optimizations that improve the compression performance. Our

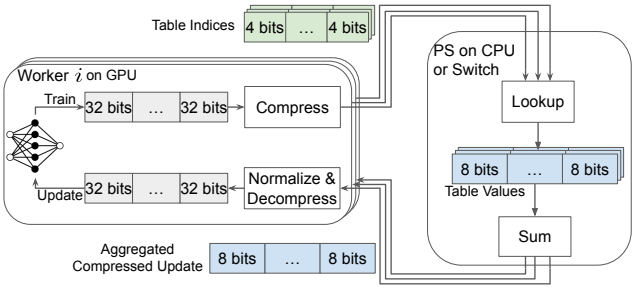


Figure 4: THC's workflow illustration.

main conceptual advance is to use non-uniform quantization intervals while maintaining the UHC property; that is, we show how to optimize the choice of quantization values in Section §4.3. Finally, Section §5 provides important technical optimizations for speed and accuracy.

4.1 Background on Stochastic Quantization

A main building block that is used for gradient compression is *quantization*, a technique that allows representing gradient entries (e.g., 32-bit floats) using a small (e.g., 4) number of bits while bounding the error. At a high level, our Tensor Homomorphic Compression (THC) framework leverages Stochastic Quantization (SQ) that rounds a given real-value a to one of two quantization values q_0, q_1 such that $q_0 \leq a \leq q_1$. SQ quantizes a to q_0 with probability $(q_1 - a)/(q_1 - q_0)$ and to q_1 otherwise. An appealing property of SQ is that the expected value of the quantization is exactly a, i.e., it is unbiased. This is especially useful in distributed scenarios where using SQ results in a decrease in the error of estimating the average as the number of workers increases [68]. Our focus in what follows is minimizing the error introduced by quantization while maintaining the UHC property.

4.2 The Uniform THC Algorithm

As we now show, it turns out that using a variation of SQ with uniform intervals *where all workers use the same set of intervals* already yields a solution that is both unbiased and homomorphic. However, the accuracy per-bit of this solution leaves much to be desired, which is where we focus our efforts in the following sections.

The most popular form of SQ is *Uniform* SQ (USQ), in which the quantization values are uniformly spaced. For example, given the range $[m, M]$ and using 2^b quantization values, their locations are $q_0 = m, q_1 = m + (M-m)/(2^b - 1), \ldots, q_{2^b} = M$. USQ quantizes a value $a \in [m, M]$ to one of its nearest quantization values. We first show that USQ, when all workers globally use the same range $[m, M]$ and b, satisfies the UHC property. Note that for implementation this requires all workers to first obtain the global minimum and maximum to perform quantization, which is different than the standard use of USQ (where each worker quantizes based on their own local minimum and maximum value). For convenience, we henceforth denote $\langle i \rangle = \{0, \ldots, i-1\}$ for any $i \in \mathbb{N}$.

Algorithm 1 Uniform THC

Input: bit budget b

Preliminary stage

Worker i:
1: Compute $m_i = \min\{\nabla_i\}$ and $M_i = \max\{\nabla_i\}$
2: Send (m_i, M_i) to the PS
PS:
3: Compute $m = \min_i\{m_i\}$ and $M = \max_i\{M_i\}$
4: Send (m, M) to workers

Main stage

Worker i:
5: Compute $X_i = USQ(\nabla_i, m, M, b)$
6: Send X_i to PS
PS:
7: Compute $X = \sum_{i\in\langle n\rangle} X_i$
8: Send X to workers
Worker i:
9: Estimate $\widehat{\nabla}_{avg} = m + \frac{1}{n}\cdot X \cdot \frac{M-m}{2^b-1}$

Definition 2 (Homomorphic USQ) *Let $\nabla_0, \nabla_1, \ldots, \nabla_{n-1} \in \mathbb{R}^d$ be the input gradients, and let $m_i = \min\{\nabla_i\}$ and $M_i = \max\{\nabla_i\}$ be the i'th gradient's minimum and maximum. Let $m = \min_i\{m_i\}$ and $M = \max_i\{M_i\}$ and consider a set of uniformly spaced quantization values $\left\{m + k\cdot\frac{M-m}{2^b-1} \mid k \in \langle 2^b\rangle\right\}$. The workers perform stochastic quantization using these quantization values on all input gradients.*

This approach is homomorphic (Definition 1) since (C and D stand for `Compress` and `Decompress`):

$$\frac{1}{n}\cdot\sum_{i\in\langle n\rangle} D(C(\nabla_i)) = \frac{1}{n}\cdot\sum_{i\in\langle n\rangle}\left(m + C(\nabla_i)\cdot\frac{M-m}{2^b-1}\right)$$
$$= m + \left(\frac{1}{n}\cdot\sum_{i\in\langle n\rangle} C(\nabla_i)\right)\cdot\frac{M-m}{2^b-1} = D\left(\frac{1}{n}\cdot\sum_{i\in\langle n\rangle} C(\nabla_i)\right).$$

With this primitive at hand, we introduce a simplified (uniform) variant of the THC framework, which we generalize to a non-uniform setting to obtain better performance.

The pseudo-code of Uniform THC is given by Algorithm 1. It begins with a preliminary communication round where each worker computes and sends the smallest and largest gradient entry to the PS (lines 1-2). In turn, the PS computes the extreme global values and distributes them back to the workers (lines 3-4). This communication round is light and requires each worker to transmit and receive only two floats. Next, each worker quantizes its gradient using the global extremes (i.e., perform Homomorphic USQ) and sends the result to the PS (lines 5-6). Then, the PS sums all the quantized vectors and sends their sum to the worker (lines 7-8). Finally, each worker divides the sum by the number of workers to obtain the estimate of the average (line 9).

As detailed in Section 5, we can reduce the quantization error by pre-processing the input gradient prior to its quantization and post-processing the average's estimate at the end.

4.3 The Non-uniform THC Algorithm

In many cases, it is possible to choose the quantization values in a non-uniform manner to optimize the accuracy-bandwidth tradeoff. However, it is unclear how to leverage existing non-uniform quantization methods (e.g., [8, 9, 53, 66, 67]) to improve our homomorphic compression. Namely, in non-uniform methods, the sender transmits a *table index* z that is then converted to the *table value* $T[z]$ by the receiver. Here, T is a *lookup table* that converts indices to values that may not be uniformly spaced, i.e., $T[z+1] - T[z]$ may not equal $T[z'+1] - T[z']$ for different indices z, z'.

For example, consider quantizing values in the range $[-1, 1]$ using four quantization values. Using USQ, the lookup table is $T_0[0] = -1, T_0[1] = -1/3, T_0[2] = 1/3, T_0[3] = 1$. In this case, any sum of table indices corresponds to a single sum of quantization values. For example, suppose two senders send indices z, z' and consider two cases: (1) $z = 0, z' = 3$ and (2) $z = 1, z' = 2$. In both cases, $T_0[z] + T_0[z'] = 0$. That is, $z + z' = 3$ implies that $T_0[z] + T_0[z'] = 0$. Intuitively, the receiver can sum the indices and deduce the sum of sent values instead of performing two table lookups.

In contrast, consider non-uniform quantization that uses the table $T_1[0] = -1, T_1[1] = -1/2, T_1[2] = 1/2, T_1[3] = 1$. Consider the two cases in which $z + z' = 4$: (1) $z = 1, z' = 3$ and (2) $z = z' = 2$. In the first case, $T_1[z] + T_1[z'] = 1/2$ while in the second $T_1[z] + T_1[z'] = 1$. That is, the sum of table indices does not determine the sum of table values.

Our insight is that we can overcome this by using a subset of the uniformly spaced quantization values. To that end, for a bit-budget of b bits per coordinate, we define a hyperparameter $g \geq 2^b - 1$ called *granularity*. We then use a table $T : \langle 2^b\rangle \to \langle g+1\rangle$ that maps table indices to values that are integers in the larger range. Intuitively, one can think about this as running USQ with $g+1$ quantization values, but where all senders are only allowed the same 2^b indices. A table value $T[z]$ then corresponds to the quantization value $m + T[z]\cdot\frac{M-m}{g+1}$, same as in USQ. For instance, in the above example of quantizing values in the range $[-1, 1]$, the PS can use a table $T_2[0] = 0, T_2[1] = 1, T_2[2] = 3, T_2[3] = 4$ to map each table index into (the larger) range $\langle 5\rangle$. The benefit is that the table values are now directly aggregable; for example, consider three senders with the two cases (1) $z = z' = z'' = 1$ and thus $T_2[z] = T_2[z'] = T_2[z''] = 1$, i.e., all quantization values are $-1 + \frac{1-(-1)}{4} = -\frac{1}{2}$; (2) $z = z' = 0, z'' = 2$ and thus $T_2[z] = T_2[z'] = 0, T_2[z''] = 3$, i.e., the first two quantization values are -1 and the third is $-1 + 3\times\frac{1-(-1)}{4} = \frac{1}{2}$. Notice that the sum of table values is the same in both cases, and so is the sum of quantization values; in contrast, the sum of table indices differ.

Intuitively, g introduces a tradeoff where larger g results in more fine-grained quantization values and lower error but also requires more bits to represent the summation and higher bandwidth requirement from the PS or switch to the workers.

The pseudocode for the non-uniform variant of THC appears in Algorithm 2 (the Preliminary stage is the same as in Algorithm 1). Notice that the lookup table $T_{b,g}$ depends on bit-budget b and the granularity g. When clear from context, we omit the subscripts and write T.

Each worker i then calculates the set of quantization values $Q \subset [m,M]$ given the (global) m,M and the granularity g (Line 1). Next, it stochastically quantizes each coordinate to a value in Q, giving the result $X_i \in Q^d$ (Line 2). The worker then transforms each quantized coordinate into its uniform-HC value in $\langle g+1 \rangle$ using the linear transformation (Line 3). The next stage involves computing which b-bit table indices (in $\langle 2^b \rangle$) would map to the uniformly partitioned values by applying the inverse mapping (Line 4). Finally, it sends the result Z_i to the PS (Line 5).

The PS then gets the set of vectors $\{Z_i\}_{i\in\langle n\rangle}$. It looks up each vector (coordinate-wise) and sums them up, corresponding to the sum of the Y_i's (Line 6). Then, the PS sends the result, Y, to the workers, still in compressed form (Line 7). Observe that the PS only performs table lookups and summation without decompressing the vectors and without additional processing and re-compression that increases the error.

The final part takes place in parallel, where each worker i first computes the average of $\sum_{i\in\langle n\rangle} Y_i$ by dividing Y by n (Line 8). It then applies the inverse transformation to Line 3 to obtain an estimate $\widehat{\nabla}_{avg}$ of the average gradient (Line 9).

To show that our algorithm is homomorphic, we generalize Definition 1 to account for the lookup table (C,D and T stand for `Compress`, `Decompress` and `table lookup`)

Definition 3 (Non-uniform homomorphic compression)

$$\widehat{\nabla}_{avg} = \frac{1}{n}\cdot\sum_{i\in\langle n\rangle} D(T(C(\nabla_i))) = D\left(\frac{1}{n}\cdot\sum_{i\in\langle n\rangle} T(C(\nabla_i))\right).$$

That is, a non-uniform HC (NUHC) scheme generalizes UHC by allowing the PS to apply a lookup table T to the compressed gradients before aggregating them.

Notice that if $g = 2^b-1$ and T is the identity mapping, NUHC is identical to UHC (in that case, the lookup table is redundant). As shown above, for T that is the identity mapping, the compression is uniform homomorphic. We now show that if $0 = T(0) < T(1) < \ldots < T(2^b-1) = g$ then Algorithm 2 is homomorphic. [2] This is because

$$\frac{1}{n}\cdot\sum_{i\in\langle n\rangle} D(T(C(\nabla_i))) = \frac{1}{n}\cdot\sum_{i\in\langle n\rangle} D(T(Z_i)) =$$

$$\frac{1}{n}\cdot\sum_{i\in\langle n\rangle} D(Y_i) = \frac{1}{n}\cdot\left(\sum_{i\in\langle n\rangle} m + Y_i\cdot\frac{M-m}{g}\right) =$$

$$m + \left(\frac{1}{n}\cdot\sum_{i\in\langle n\rangle} Y_i\right)\cdot\frac{M-m}{g} = D\left(\frac{1}{n}\cdot\sum_{i\in\langle n\rangle} T(C(\nabla_i))\right),$$

where we used (i) $C(\nabla_i) = Z_i$, (ii) $T(C(\nabla_i)) = T_{b,g}[Z_i] = Y_i$, and (iii) $D(Y_i) = m + Y_i\cdot\frac{M-m}{g}$.

[2] In fact, it is sufficient that T is injective and satisfies $0, g \in Im(T)$; the above definition is without loss of generality.

Algorithm 2 Non-uniform THC

Input: bit budget b, granularity g, and their lookup-table $T_{b,g}$

Main stage ▷ Preliminary stage same as in Algorithm 1

Worker i**:**

1: Compute $Q = CalcQuantizationValues(m,M,T_{b,g})$
2: Compute $X_i = SQ(\nabla_i, Q)$
3: Compute $Y_i = (X_i - m)\cdot\frac{g}{M-m}$ ▷ $Y_i \in \langle g+1\rangle^d$
4: Compute $Z_i = T_{b,g}^{-1}[Y_i]$ ▷ $Z_i \in \langle 2^b\rangle^d$
5: Send Z_i to PS

PS:

6: Compute $Y = \sum_{i\in\langle n\rangle} T_{b,g}[Z_i]$ ▷ $Y \in \langle g\cdot n+1\rangle^d$
7: Send Y to workers

Worker i**:**

8: Compute $Y_{avg} = \frac{1}{n}\cdot Y$ ▷ $Y_{avg} \in [0,g]^d$
9: Estimate $\widehat{\nabla}_{avg} = m + Y_{avg}\cdot\frac{M-m}{g}$

The above result shows that Algorithm 2 is homomorphic for many choices of lookup tables. As we later demonstrate, very small lookup tables (e.g., for $b=4$ we can usually use $g \in \{16,\ldots,51\}$) are sufficient to obtain accurate quantization.

5 Optimizing THC

We next describe optimizations that improve THC's bandwidth-accuracy tradeoff (§5.1-§5.2) and speed (§5.3).

5.1 Pre- and Post-processing Using the Randomized Hadamard Transform

For pre-processing, we utilize the Randomized Hadamard Transform (RHT) that improves quantization accuracy by reducing the expected range and transforming coordinates to approach a normal distribution. The RHT of a vector $x \in \mathbb{R}^d$ is defined as $\frac{1}{\sqrt{d}}\cdot H\cdot D\cdot x$, where H is the Hadamard matrix [25] and D is a diagonal matrix with i.i.d. Radamacher variables (taking ± 1 with equal probabilities) on its diagonal. An important property RHT is that the special recursive structure of H allows a fast GPU-friendly $O(d\log d)$ time implementation, significantly faster than general matrix multiplication. The post-processing is then the inverse transform, i.e., $RHT^{-1}(x) = \frac{1}{\sqrt{d}}\cdot D\cdot H\cdot x$, which has identical complexity to RHT.

RHT has two key benefits: First, RHT reduces the range of coordinate values, improving accuracy. More concretely, a key quantity that determines the error of a quantization scheme is its range (the difference between the largest and smallest quantization values), $M-m$ (see Appendix A.2 for more details). Intuitively, when the range is large, one is forced to quantize to values that are further away from the encoded quantity, thus increasing the error. As proven by [3], and using M', m' to denote the maximal and minimal value after the RHT transform, $\mathbb{E}[M'-m'] = O\left((M-m)\cdot\sqrt{\log d\ /\ d}\right)$. This decrease in the expected range significantly improves the quantization accuracy.

To further decrease the range, and thus the quantization error, we leverage known results about the distribution of the transformed coordinates to derive a threshold t_p, for an appropriate $p \in (0,1)$, such that approximately a p fraction of the transformed coordinates are expected to fall outside $[-t_p, t_p]$. Namely, it is known that each coordinate in the RHT of a vector $x \in \mathbb{R}^d$ follows a distribution that approaches (for a large enough d) the normal distribution $\mathcal{N}(0, 1/\|x\|^2)$ [68]. In particular, this means that the probability of a coordinate landing outside the range diminishes exponentially in t_p, giving an opportunity to significantly reduce the range at the expense of a small bias and allowing us to pick an appropriate p, as we describe below in § 5.3. Our algorithm then optimizes the quantization values to minimize the error of the coordinates in $[-t_p, t_p]$, and truncates the rest by rounding those larger than t_p to t_p and those smaller than $-t_p$ to $-t_p$.

To address this small bias, we compensate for it using a technique called error-feedback (EF) [29], which includes sending the vector $x = \nabla + e$ that adds the previous error e to the current gradient ∇, and later updating e to account for the quantization error. It is known that when the bias is not too large, EF guarantees the convergence of the training [36]. The effect of rotation and error feedback is evaluated in detail in Appendix D.3, along with comparison to uniform THC.

The second key benefit of RHT is that, since the distribution of the coordinates after applying RHT is known, we can compute the optimal lookup table T *offline*, as we explain next.

5.2 Constructing the Optimal Lookup Table

As explained, after applying RHT and truncating, our goal is to design a lookup table that minimizes the quantization error of the resulting vector. This vector has two types of coordinates: (1) *the truncated coordinates* have no error (beyond the truncation) since, by design, there are always quantization values at $\{-t_p, t_p\}$; (2) the *non-truncated coordinates* have a distribution that approaches the truncated normal distribution. Intuitively, this allows us to design the lookup table T to minimize the quantization error of a truncated normal random variable and thereby minimize the NMSE.

Formally, the optimal lookup table needs to minimize the error in quantizing a truncated normal random variable $(A \sim \mathcal{N}(0,1) \mid A \in [-t_p, t_p])$, where $t_p = \Phi^{-1}(1 - p/2)$ and Φ is the CDF of the normal distribution. As we later show, we can use this pre-computed table for a normal random variable *with any variance* by scaling the quantization values.

Formally, denoting by $P(a,z)$ the probability of sending the index $z \in \langle 2^b \rangle$ given a value $a \in [-t_p, t_p]$, the optimization problem aims to find the table T and probabilities P. Let us further denote by ϕ the pdf of the normal distribution. Then, the following optimization problem gives the optimal lookup table as $T_{b,g,p}[z] = T[z]$:

$$\underset{P,T}{\text{minimize}} \int_{-t_p}^{t_p} \sum_{z \in \langle 2^b \rangle} P(a,z) \cdot (a - T[z])^2 \cdot \phi(a) \cdot da$$

subject to

$$(\textit{Unbiasedness}) \quad \sum_{z \in \langle 2^b \rangle} P(a,z) \cdot T[z] = a \qquad \forall a \in [-t_p, t_p]$$

$$(\textit{Probability}) \quad \sum_{z \in \langle 2^b \rangle} P(a,z) = 1 \qquad \forall a \in [-t_p, t_p]$$

$$P(a,z) \geq 0 \qquad \forall a \in [-t_p, t_p], z \in \langle 2^b \rangle$$

$$(\textit{Granularity}) \quad T[z] \in \left\{ \tfrac{2t_p}{g} \cdot i - t_p \mid i \in \langle g+1 \rangle \right\} \qquad \forall z \in \langle 2^b \rangle$$

As we elaborate in Appendix B, we optimally solve the above problem. To that end, we wrote a specialized ILP solver that leverages various properties of the above optimization problem to reduce the search space and speed up the computation of T. Recall that for any b, g, p, we compute the optimal $T_{b,g,p}$ table only once offline and thus the solver's runtime does not affect THC's performance.

5.3 Accelerating the Preliminary Stage

Heretofore, we discussed a preliminary stage in which the workers exchange information that depends on the transformed vectors to determine the quantization range. A natural implementation would, therefore, include transforming the vectors using RHT and then exchanging the information required for setting M and m.

Instead, we leverage special properties of RHT, namely that (i) it preserves the *norm* of the transformed vector and (ii) that there is a tight connection between the *maximal norm* and the values M, m we are seeking. Accordingly, each client i first computes the norm of its vectors x_i and then parallelizes the following operations: performing the RHT and obtaining the maximum norm among the workers' gradients using the PS. Then it can set $M = (t_p/\sqrt{d}) \cdot \ell, m = -M$ where $\ell = \max \|x_i\|^2$ and proceed to the quantization.

5.4 Putting It All Together

We are now ready to describe our complete THC training process, whose pseudocode is given Algorithm 3, color coding the different steps. In the first learning step, each worker computes its local gradient and adds the error-feedback. Next, the preliminary stage, in which the clients exchange their norms, is parallelized with the RHT part of the pre-processing; later, the transformed vector is normalized and clamped. Next, during main stage, the workers and PS follow our Non-uniform THC algorithm (see Algorithm 2) to obtain an estimate of the average of the pre-processed vectors. Then, each worker post-processes the estimate using the inverse transform. Finally, in the second learning step, the workers update the error-feedback and model.

Algorithm 3 Training with THC

Input: Support parameter p and its threshold t_p. Bit budget b, granularity g, and lookup-table $T_{b,g,p}$.
for $r = 0, 1, \ldots$ **do**
  for worker $i \in \langle n \rangle$ in parallel **do**
    Compute local gradient, ∇_i
    $x_i = \nabla_i + e_i^r$ ▷ Error-feedback for round r
    in parallel
      (i) **Send** $\|x_i\|$ to PS
        Receive $\ell = \max_i \|x_i\|$ from PS
      (ii) $R_i = \texttt{RHT}(x_i)$ ▷ $R = HD^r x_i$
    end in parallel
    $M = (t_p/\sqrt{d}) \cdot \ell \;\; ; \;\; m = -M$
    $x_i' = \texttt{clamp}(R_i, \min = m, \max = M)$ ▷ Truncation
    $Q = CalcQuantizationValues(m, M, T_{b,g,p})$
    $X_i = SQ(x_i', Q)$ ▷ Stochastic Quantization
    $Y_i = (X_i - m) \cdot \frac{g}{M-m}$ ▷ $Y_i \in \langle g+1 \rangle^d$
    $Z_i = T_{b,g,p}^{-1}[Y_i]$ ▷ $Z_i \in \langle 2^b \rangle^d$
    Send Z_i to PS
    Receive $Y = \sum_{i \in \langle n \rangle} Y_i = \sum_{i \in \langle n \rangle} T_{b,g,p}[Z_i]$ from PS
    Compute $Y_{avg} = \frac{1}{n} \cdot Y$ ▷ $Y_{avg} \in [0, g]^d$
    Estimate $\widehat{x}'_{avg} = m + Y_{avg} \cdot \frac{M-m}{g}$
    Compute $\widehat{\nabla}_{avg} = \texttt{RHT}^{-1}(\widehat{x}'_{avg})$ ▷ Global update
    $e_i^{r+1} = x_i - \texttt{RHT}^{-1}(X_i)$ ▷ Quantization error
    Update model using $\widehat{\nabla}_{avg}$
  end for
end for

6 THC with Other System Opportunities

In this section, we discuss how THC leverages the opportunities of in-network aggregation and explore potential optimizations to address issues such as packet loss and stragglers.

Aggregation at Programmable Switches In our THC framework, we can offload PS completely to programmable switches for further hardware acceleration. Our THC design simplifies the PS by removing the compression and decompression operations. Since THC already compresses floating-point gradients to integer table indices, it fits programmable switches well. We do not need additional conversions from floating points to integers at workers as used in previous work [33, 57, 79].

Packet Loss and Stragglers THC is a compression algorithm that tolerates data loss, allowing it to ignore the tail outliers that can negatively impact performance during training introduced by packet loss [69] or straggler problems [22, 78].

Packet losses and stragglers may occur between workers and the PS. For each gradient, if a worker doesn't receive the corresponding aggregation result packet within a specified time threshold, the worker could fill in the missing data with zeros and continue with the received aggregation results. This practice may lead to some workers updating their models with different information. To mitigate the impact, we can implement a synchronization scheme, where workers coordinate their model parameters after every epoch by choosing to copy the parameters of another worker when encountering severe packet loss. Our simulation results shows that there is no significant impact on THC model accuracy or convergence within the reasonable data center packet loss rate range (less than 1% [21, 37]) (see Section §8.4). This result aligns with the observations in [69].

To handle packet losses from workers to the PS, we can perform partial aggregation: the PS broadcasts partial aggregation results once it hears from the majority (e.g., 90%) of workers. We evaluate the impact of stragglers and partial aggregation in Section §8.4 and show that THC with partial aggregation over 90% of workers reach the baseline accuracy.

7 Implementation

THC Worker. We develop THC prototype of workers atop BytePS's PyTorch extension [28]. During each iteration, the BytePS worker receives the calculated gradient from the front-end PyTorch and passes it to our *compression module*. Our compression module runs the THC algorithm on GPUs and employs a GPU-friendly implementation of RHT. It also keeps the error-feedback records to compensate for the biased quantization as mentioned in Section 5.

The communication module is a C++ module developed based on Data Plane Development Kit (DPDK), which provides kernel bypassing so that applications can directly receive data from the NIC using busy polling. The communication module assembles packets based on compression results and communicates with the PS.

Another approach is to adopt the RDMA protocol, such as RoCEv2 [5]. However, adopting the RDMA protocol requires additional header parsing functions on the switch side. SwitchML [57] has demonstrated that the RoCEv2 protocol can be used with in-network aggregation by carefully parsing the header. For THC, we consider this as future work.

THC Parameter Servers. We implement two versions of parameter servers: the software version written in C++ and the programmable switch version implemented on the Intel Tofino switch [26]. In the programmable switch version, the PS performs table lookup using the "Table" control block. After the table lookup, the switch sends packets carrying table values through recirculation ports. The "Register" extern then takes care of value aggregation. Please see Appendix C.2 for the resource usage of the programmable switch PS.

8 Evaluation

We evaluate our THC prototype by training popular computer vision and language models on a local four-worker testbed and AWS EC2. Experiments show that employing THC gives shorter time-to-accuracy (1.40× to 1.47× speedup) and higher throughput (25% to 54% increase) on our local

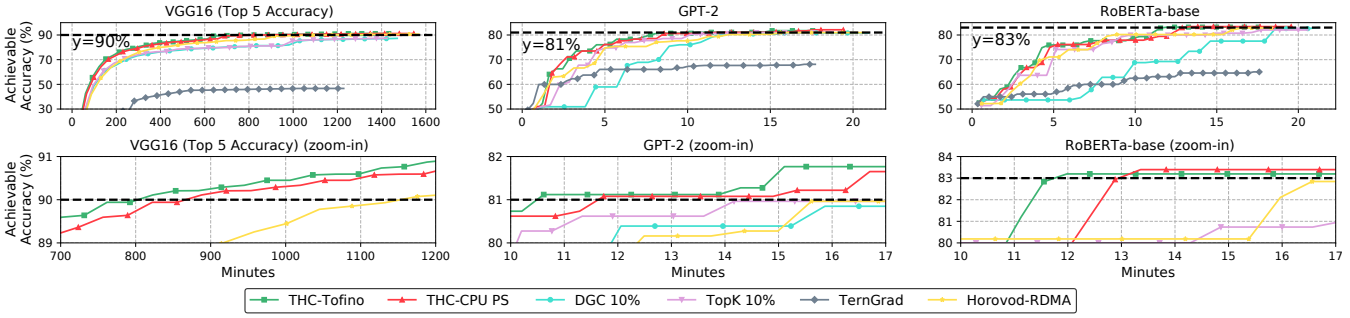


Figure 5: Time to accuracy (TTA) comparison over one image processing task (i.e., VGG16) and two NLP tasks (i.e., GPT-2 and RoBERTa-base). The second row zooms in on the competitive results.

testbed with a programmable switch, compared to state-of-the-art training frameworks. THC also outperforms DGC-10%, TopK-10%, and TernGrad in time-to-accuracy as it has little impact on accuracy and eliminates the overhead at the PS.

Testbed Setup. Our local testbed has four GPU machines as workers, each with one NVIDIA A100 GPU and one NVIDIA MCX516A-CCAT ConnectX-5 100G dual-port NIC connecting to a Tofino2 switch. For large-scale experiments, we use eight AWS EC2 p3.16xlarge instances, each with eight NVIDIA V100 GPUs and 25 Gbps network bandwidth.

Systems for Comparison. We run three versions of THC: (1) with a programmable switch as the PS (labeled as *THC-Tofino*), (2) with the software PS process running on a single stand-alone CPU machine (connected to the Tofino2 with a ConnectX-5 100G dual-port NIC) (*THC-CPU PS*), and (3) with colocated PSes for each worker (*THC-colocated*), which build on BytePS's PS architecture and uses BytePS's RDMA module for a fair comparison. We compare THC with two state-of-the-art systems: (1) Horovod which uses RDMA for communication (*Horovod-RDMA*) and (2) BytePS with colocated PSes for each worker (*BytePS*). We also compare THC against three popular compression algorithms: DGC [38] and TopK [64] are sparsification algorithms that only communicate the top k% of coordinates by magnitude (here we set $k = 10$ and refer to them as *DGC 10%* and *TopK 10%*); and TernGrad [74], a quantization algorithm that converts each coordinate into a value $x \in \{-1, 0, 1\}$ (we refer to it as *TernGrad*). TernGrad represents a stream of quantization algorithms [4, 10] with small differences in design. *BytePS*, *DGC 10%*, *TopK 10%*, and *TernGrad* all use BytePS's colocated PSes and RDMA module. We also tried Espresso [73] but faced convergence issues.[3]

On AWS EC2, we deploy THC with software PS built on top of BytePS [28] servers. We compare THC against BytePS with colocated PS (*BytePS*) and Horovod [58]. All systems use the TCP protocol to communicate. Unless noted otherwise, we use the following THC configuration: granularity 30, p-fraction 1/32, and 16 quantization levels. This configuration avoids overflow for up to eight workers, saturates the worker to PS bandwidth of our system prototype, and consistently achieves high accuracy across various models.

Workloads. We evaluate THC with network-intensive [33, 57] computer vision models (VGG16 and VGG19 [62]) and language models (RoBERTa-base, RoBERTa-large [40], Bart-large [34], BERT-base [14], and OpenAI GPT-2 [50]). We train the vision models with the ImageNet1K dataset [55] and train the language models with the GLUE (General Language Understanding Evaluation benchmark) SST2 (The Stanford Sentiment Treebank) task [63]. Unless noted otherwise, we set the per-GPU batch size as 32. We include results for computation-intensive models that don't benefit much from accelerated inter-machine communication in Appendix D.

Metrics. We first measure time-to-accuracy (TTA) as the training time needed to reach a target validation accuracy. We set the target accuracy based on the convergence of our no-compression baseline. We then present the training throughput (images per second or tokens per second referred to as *samples per second*) for all training tasks. We also show the breakdown of computation and communication time to highlight the factors that contribute to THC's improvements.

8.1 End-to-End Training Performance

Time-to-accuracy. Figure 5 shows that for GPT-2, THC-Tofino reaches the 81% target accuracy 1.47× faster than the Horovod-RDMA baseline, and THC-Software PS reaches the target accuracy 1.33× faster. For RoBERTa-base, THC-Tofino achieves the 83% target accuracy 1.43× faster than the Horovod-RDMA baseline; and THC-Tofino also achieves a 1.40× speedup to reach the 90% target accuracy for VGG16. Note that even though our system prototype uses DPDK, which has similiar performance with RDMA, we still notably outperform Horovod-RDMA. THC-software PS reduces network communication with minimal impact on model convergence, thanks to the optimizations explained in Section 5. Using the programmable switch (THC-Tofino) further accelerates the training by reducing the volume of transmitted data through in-network aggregation.

[3] We followed the instructions in Espresso's git repository and installed all desired versions of software. Unfortunately, we couldn't get models to converge (training loss became "nan" within three iterations and the accuracy stalled around 0.1%). We contacted the authors but they do not have time to fix it. See https://github.com/zhuangwang93/Espresso/issues/3.

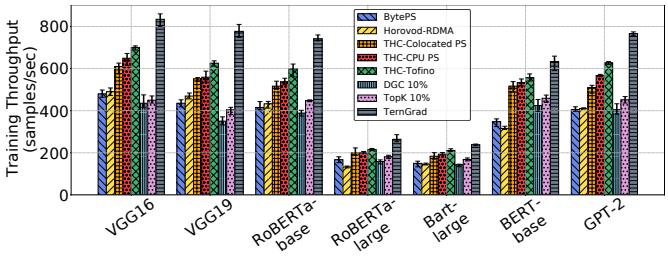


Figure 6: Training throughput with 100Gbps links over different network-intensive architectures.

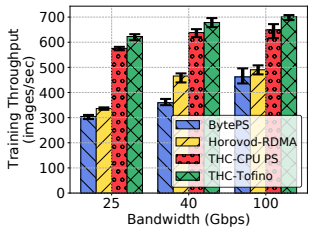


Figure 7: Training throughput for different bandwidths.

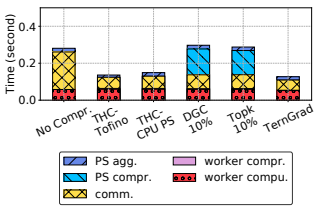


Figure 8: Average training round time breakdown.

Although training with DGC 10% and TopK 10% also approach the target accuracy, they suffer from the high PS compression overhead that leads to longer training epoch time and TTA. TernGrad stalls at low accuracy for all three models despite its high training throughput (Figure 6). This is because TernGrad loses information during its compression and thus cannot improve end-to-end TTA with more training epochs.

Training throughput. To examine the synchronization round time reduction we achieve with THC, we measure the throughput in Figure 6. THC-Tofino provides higher throughput than all alternatives (except TernGrad). For example, THC-Tofino achieves 54% improvement over Horovod-RDMA for GPT-2. THC-colocated has 11% to 37% higher throughput than TopK because THC eliminates the PS-side compression operations. TernGrad provides the highest throughput because it uses fewer bits per coordinate and has shorter PS time and compression overhead. However, TernGrad does not improve TTA as Figure 5 shows due to its low accuracy (Section 8.2).

Effectiveness with different bandwidth. We train the VGG16 architecture under different network bandwidth settings (25, 40, and 100Gbps) in Figure 7. THC-Tofino achieves 1.85×, 1.45×, and 1.43× training throughput over Horovod-RDMA at 25Gbps, 40Gbps, and 100Gbps respectively. When the bandwidth decreases from 40Gbps to 25Gbps, the throughput of Horovod-RDMA drops significantly because it faces more network bottlenecks. Meanwhile, the performance of THC-Tofino and THC-CPU PS downgrades gracefully as the bandwidth decreases, leading to higher training speedups at low bandwidths. We expect THC to offer more benefits under slower networks (e.g., 10Gbps, 1Gbps) which we might encounter in federated learning settings.

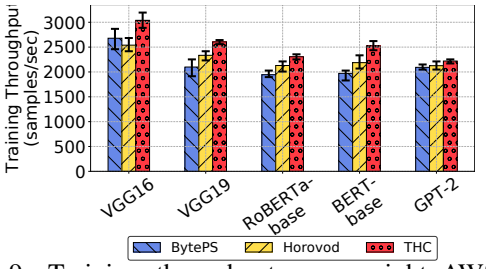


Figure 9: Training throughput across eight AWS EC2 p3.16xlarge instances.

8.2 Breakdown of Network and Compute Time

In Figure 8, we break down the time per iteration into the time spent at the PS, workers, and the communication, when training VGG16 at 100Gbps with THC-Tofino, THC-CPU PS, TopK 10%, and TernGrad. At the PS, we measure the compression time and aggregation time. At the workers, we measure the training time and compression time.

THC-CPU PS reduces the gradient communication time (comm.) to 32.5% of that of the no-compression baseline. As a tradeoff, we introduce compress and decompress operations on the GPU on the worker side. However, these operations only increase the overall worker time by 9.5%. The result demonstrates that in distributed training settings where workers periodically synchronize a large amount of data, saving bandwidth by slightly increasing the worker GPU computation time is worthwhile. THC-Tofino achieves more savings by further reducing communication time through in-network aggregation and offloading the PS to the switches.

For DGC 10% and TopK 10%, they have to run expensive sorting operations on the PS (DGC 10% additionally requires local gradient accumulation), introducing a significant overhead at the PS side. Therefore, although TopK10% gives similar communication time as THC-CPU PS, the overall round time is 46.5% higher than that of THC-CPU PS.

TernGrad uses only two bits per coordinate and requires simple summation at the PS. Thus, it has a short communication and PS time. However, TernGrad produces high NMSE and correspondingly can produce poor TTA results.

8.3 THC Performance on AWS EC2

We measured throughput on AWS EC2 instances equipped with workers containing multiple GPUs at a larger scale (Figure 9). Since AWS instances have 8 GPUs per worker, we have a higher intra-machine communication overhead compared to our local testbed setting. This means that inter-machine communication overhead, which THC optimizes for, takes a smaller portion of training time. Despite this, THC consistently outperformed all the baseline models, resulting in throughput improvements of 1.05× to 1.16×.[4]

[4]Note that Bart-large and RoBERTa-large are not displayed in Figure 9 due to the V100 GPU's memory limitations on the EC2 environment. We used a smaller batch size and reported it in Appendix D separately.

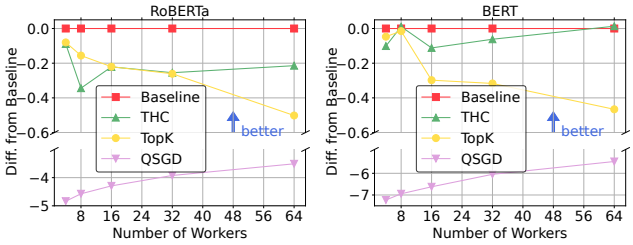


Figure 10: Scalability of THC. Displaying the difference in training accuracy from baseline after two epochs (i.e. if the final training accuracy for 4 worker RoBERTa baseline is 95%, then the accuracy for 4 worker RoBERTa QSGD would be 90%).

8.4 THC Simulation Results

We run simulations to understand THC under different system configurations. As the number of workers increase, the performance of THC increases due to the UHC property. We show that the error of THC scales well from 4 to 64 workers, in contrast to biased compression algorithms like TopK whose error can increase by 9.9× over the same margin. Furthermore, under the synchronization and partial aggregation schemes, THC shows less than a 0.5% drop in training accuracy in the presence of packet loss and straggling workers.

Simulation Environment. We simulate THC on an academic cluster with 4 A100/V100 GPUs per node. Multiple worker training is modeled by storing multiple passes of the backpropagation before performing an update step. This allows us to compress and decompress the aggregated gradient with THC's algorithm (and others) before updating the model, reproducing the communication steps in actual systems.

For the scalability experiments, we train BERT and RoBERTa [40] on SST2 [63] with batch size 8. The configuration uses granularity 36, p-fraction 1/32, and bit budget 4. We choose language tasks for the scalability results because they are more sensitive to small compression errors in the gradient. The other simulations train ResNet50 [23] models on the CIFAR100 [32] dataset with a batch size 128, workers 10, granularity 20, p-fraction 1/512, and bit budget 4.

Scalability. We fine-tune a model for two epochs for each compression algorithm with 4, 8, 16, 32, and 64 workers and then compute the percentage difference from the uncompressed baseline accuracy. We track the difference in accuracy rather than the absolute accuracy because machine learning effects can alter the baseline accuracy as the batch size (number of workers) changes. We compare THC with bit budget 4 against baseline (no compression), TopK [64] and Quantized Stochastic Gradient Descent (QSGD) [4]. QSGD is chosen because we want to compare compression algorithms that have the same compression ratio: QSGD is analogous to an unbiased version of TernGrad/SignSGD with a tunable compression ratio. We choose the k-value and number of intervals of TopK and QSGD respectively to match the overall compression ratio of THC with bit budget 4.

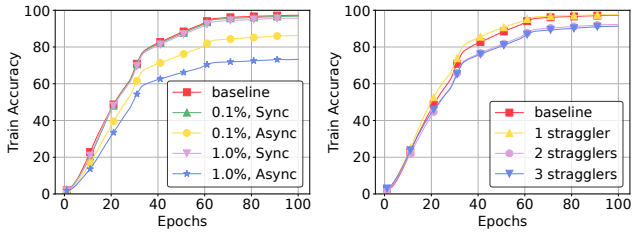


Figure 11: Resiliency to Gradient Losses. Displaying attainable accuracy with packet drops and stragglers.

Figure 10 shows that for BERT, the accuracy of THC actually improves as the number of workers increases, with the error decreasing from −0.1% to 0 (no difference from baseline) from 4 to 64 workers. Although there seems to be some outlier at 8 workers that is likely due to randomness in the training process, these trends match our predictions in Section 4 and can be attributed to the increased accuracy of the estimate of the average gradient. In comparison, the error of TopK quickly inflates by 9.9× from −0.047% to −0.46% in the same region because bias in the compression dominates and causes larger compression errors. The data for RoBERTa show a similar trend, with THC becoming the most accurate at 32 workers and beyond.

Such results are promising for THC because actual system performance depends on both throughput and compression accuracy. As we showed in §8.1, THC has a higher throughput in training than most compression algorithms. At 16 workers and beyond, THC also shows the highest accuracy, implying that THC will have better time-to-accuracy results for a system with many workers. The advantage over other compression algorithms is more apparent in larger scale systems because biased algorithms such as TopK lose accuracy at scale.

However, we cannot increase the number of workers to an arbitrary large size without incurring costs to accuracy. From each worker, the largest value per coordinate aggregated at the switch is equal to the granularity, so the maximum aggregated result is $m = g \times (\#\ of\ workers)$ and the number of bits needed to send this value downstream is $\lceil \log_2 m \rceil$. If we keep the number of bits sent downstream constant, we must decrease the granularity for a larger number of workers to prevent overflow, which increases the error. This can be seen from our results in Figure 15 in Appendix D. One advantage however is that as the granularity decreases, we can also decrease the bit budget for THC, sending fewer bits per coordinate upstream. On the other hand, if we keep the granularity constant, then we must send more bits per coordinate downstream, decreasing the throughput. In these experiments, we kept the granularity constant and instead adjusted the compression ratio of the downstream data (including for TopK and QSGD for fair comparison). It is likely that the optimal strategy is to employ a combination of the options depending on the specifics of the system.

Packet Loss. We simulate packet losses in both directions between workers and the PS. Figure 11 shows that even with

a packet loss rate of 1%, which greatly decreases the final training accuracy with no synchronization, our proposed synchronization scheme reduces the training accuracy drop from 24% to 1.5%. With 0.1% packet loss, synchronization reduces the accuracy discrepancy from 11% to 0.5%, which is nearly indistinguishable from baseline. The corresponding test accuracies are shown in Figure 16 of Appendix D and demonstrate similar results.

Stragglers. To mitigate stragglers, we model a scheme where the PS only waits for the top n% of the workers for aggregation. Figure 11 shows the effect of randomly choosing a 1/2/3 stragglers during each round and dropping their gradients. With 10 workers, this corresponds to waiting for 90%/80%/70% of the workers. Waiting for the top 90% reaches the baseline accuracy, whereas 80% and 70% show only a 5-6% decrease in final training accuracy.

9 Related Work and Discussions

Systems Support for Gradient Compression. Gradient compression systems have been developed to train large models that are increasingly bottlenecked by communication, given the slower growth of bandwidth compared to GPU capability. Previous systems (e.g., HiPress [6] and Espresso [73]) focus on compression awareness and finding compression strategies and work division (e.g., compression on GPU or CPU). These systems maximize the overlap of efficient communication and compression to hide existing overhead. THC, on the other hand, mitigates compression overhead by reducing the number of compress/decompress operations. THC is complementary with these works.

In-network Aggregation for ML. SwitchML [57] and ATP [33] have demonstrated the benefits of aggregating gradients within networks, but they do not support compression as the data is restricted to the format that can be directly aggregated. OmniReduce [17] and ASK [24] propose using a key-value data structure for in-network aggregation. However, these approaches require compressing the gradient data into the key-value format and make assumptions about the data itself. In THC, we support aggregation directly on compressed data, making it orthogonal to previous works. This leads to efficient in-network aggregation with compression.

Supporting Other AllReduces. An important future research direction is incorporating homomorphic compression in other types of all-reduce like ring-based or tree-based. Currently, compression schemes fail to improve the performance of these types [2]. For example, in the widely-deployed ring all-reduce that requires $O(n^2)$ aggregation operations, existing schemes would need an excessive number of decompression and re-compression operations, leading to poor accuracy and slowdown compared to an uncompressed baseline. THC makes the first step towards making compression algorithms ring-based or tree-based all-reduce friendly. For example, we may run the reduce operation directly on gradients compressed with Uniform THC using the same number of bits required for the PS aggregation (e.g., 8). However, this method is not compatible with our various optimizations, such as sending just b (e.g., 4) bits or using the lookup table, and is thus sub-optimal.

Colocated with Other Training Paradigms. While THC primarily focuses on data parallelism, it can seamlessly integrate with state-of-the-art training paradigms that use hybrid approaches combining tensor, pipeline, and data parallelism [43, 61]. Since all these optimizations are clearly separated in different dimensions, THC can be applied to the line of data parallelism without additional adaption. Moreover, Megatron-LM [43] reports that data parallelism remains the dominating factor in training throughput; thus, gradient exchanges will continue to make a significant contribution to computation and communication costs. We believe that increasing the degree of data parallelism is still a better choice, emphasizing THC's crucial role in training optimization.

Compatibility with Security. An extensively studied application of homomorphism is Homomorphic Encryption (HE) [1] in the security field. Although we consider HE as orthogonal to our work, it might be feasible to combine THC with other security practices. For example, applying differential privacy [16] techniques first and then compressing the tensors with THC can be practicable.

10 Conclusion

THC is a novel framework that formally defines homomorphic compression. As homomorphic compression supports direct aggregation of compressed data, it also allows an elegant combination of gradient compression and in-network aggregation. To demonstrate THC's generalizability, we build a distributed DNN training system prototype that employs both the THC algorithm and in-network aggregation to accelerate gradient synchronization. Testbed experiments with four GPU workers, one programmable switch, and 100Gbps network show that our system prototype achieves up to 1.47× TTA improvement when we enable both gradient compression and in-network aggregation.

11 Acknowledgment

We thank the NSDI reviewers and our shepherd, Qun Huang, for their invaluable feedback. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Ran Ben Basat was supported by the Meta Network for AI faculty award. Michael Mitzenmacher was supported in part by NSF grants CCF-2101140, CNS-2107078, and DMS-2023528. We thank Vyas Sekar for proposing the term 'Homomorphic Compression'.

References

[1] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Comput. Surv.*, 51(4), jul 2018.

[2] Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos. On the utility of gradient compression in distributed training systems. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 652–672, 2022.

[3] Nir Ailon and Bernard Chazelle. Approximate nearest neighbors and the fast johnson-lindenstrauss transform. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 557–563, 2006.

[4] Dan Alistarh, Demjan Grubic, Jerry Z. Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1707–1718, Red Hook, NY, USA, 2017. Curran Associates Inc.

[5] InfiniBand Trade Association. InfiniBand Trade Association. RoCE v2 Specification. . https://cw.infinibandta.org/document/dl/7781, 2014.

[6] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. Gradient compression supercharged high-performance data parallel dnn training. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 359–375, New York, NY, USA, 2021. Association for Computing Machinery.

[7] Ran Ben Basat, Michael Mitzenmacher, and Shay Vargaftik. How to send a real number using a single bit (and some shared randomness). In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 25:1–25:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[8] Ran Ben Basat, Shay Vargaftik, Amit Portnoy, Gil Einziger, Yaniv Ben-Itzhak, and Michael Mitzenmacher. QUIC-FL: Quick Unbiased Compression for Federated Learning. *arXiv preprint arXiv:2205.13341*, 2022.

[9] Ran Ben-Basat, Yaniv Ben-Itzhak, Michael Mitzenmacher, and Shay Vargaftik. Optimal and Near-Optimal Adaptive Vector Quantization. *arXiv preprint arXiv:2402.03158*, 2024.

[10] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. signsgd: Compressed optimisation for non-convex problems. In *International Conference on Machine Learning*, pages 560–569. PMLR, 2018.

[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[12] ByteDance. BytePS Environment Variables. https://github.com/bytedance/byteps/blob/master/docs/env.md, 2021.

[13] Aaron Daniel Cohen, Adam Roberts, Alejandra Molina, Alena Butryna, Alicia Jin, Apoorv Kulshreshtha, Ben Hutchinson, Ben Zevenbergen, Blaise Hilary Aguera-Arcas, Chung ching Chang, Claire Cui, Cosmo Du, Daniel De Freitas Adiwardana, Dehao Chen, Dmitry (Dima) Lepikhin, Ed H. Chi, Erin Hoffman-John, Heng-Tze Cheng, Hongrae Lee, Igor Krivokon, James Qin, Jamie Hall, Joe Fenton, Johnny Soraker, Kathy Meier-Hellstern, Kristen Olson, Lora Mois Aroyo, Maarten Paul Bosma, Marc Joseph Pickett, Marcelo Amorim Menegali, Marian Croak, Mark Díaz, Matthew Lamm, Maxim Krikun, Meredith Ringel Morris, Noam Shazeer, Quoc V. Le, Rachel Bernstein, Ravi Rajakumar, Ray Kurzweil, Romal Thoppilan, Steven Zheng, Taylor Bos, Toju Duke, Tulsee Doshi, Vincent Y. Zhao, Vinodkumar Prabhakaran, Will Rusch, YaGuang Li, Yanping Huang, Yanqi Zhou, Yuanzhong Xu, and Zhifeng Chen. Lamda: Language models for dialog applications. In *arXiv*. 2022.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[15] Ron Dorfman, Shay Vargaftik, Yaniv Ben-Itzhak, and Kfir Yehuda Levy. Docofl: Downlink compression for cross-device federated learning. 2023.

[16] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407, 2014.

[17] Jiawei Fei, Chen-Yu Ho, Atal N. Sahu, Marco Canini, and Amedeo Sapio. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ACM SIGCOMM '21, page 676–691,

New York, NY, USA, 2021. Association for Computing Machinery.

[18] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2018.

[19] Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydin Buluc. Integrated model, batch, and domain parallelism in training neural networks. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 77–86, 2018.

[20] Kaja Gruntkowska, Alexander Tyurin, and Peter Richtárik. EF21-P and Friends: Improved Theoretical Communication Complexity for Distributed Optimization with Bidirectional Compression. *arXiv preprint arXiv:2209.15218*, 2022.

[21] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. *SIGCOMM Comput. Commun. Rev.*, 45(4):139–152, aug 2015.

[22] Wang Hao, Qin Yuxuan, Lao ChonLam, Le Yanfang, Wu Wenfei, and Chen Kai. Preemptive switch memory usage to accelerate training jobs with shared in-network aggregation. In *2023 IEEE 30th International Conference on Network Protocols (ICNP)*, pages 1–11, 2022.

[23] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[24] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. A generic service to provide in-network aggregation for key-value streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 33–47, New York, NY, USA, 2023. Association for Computing Machinery.

[25] A Hedayat and Walter Dennis Wallis. Hadamard matrices and their applications. *The Annals of Statistics*, pages 1184–1238, 1978.

[26] Intel. Barefoot Tofino. https://www.barefootnetworks.com/technology/#tofino.

[27] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, July 2019. USENIX Association.

[28] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479. USENIX Association, November 2020.

[29] Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian Stich, and Martin Jaggi. Error feedback fixes signsgd and other gradient compression schemes. In *International Conference on Machine Learning*, pages 3252–3261. PMLR, 2019.

[30] Mehrdad Khani, Manya Ghobadi, Mohammad Alizadeh, Ziyi Zhu, Madeleine Glick, Keren Bergman, Amin Vahdat, Benjamin Klenk, and Eiman Ebrahimi. Sip-ml: High-bandwidth optical network interconnects for machine learning training. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ACM SIGCOMM '21, page 657–675, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Jakub Konečnỳ and Peter Richtárik. Randomized distributed mean estimation: Accuracy vs. communication. *Frontiers in Applied Mathematics and Statistics*, 4:62, 2018.

[32] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[33] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761. USENIX Association, April 2021.

[34] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

[35] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith,

Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[36] Xiaoyun Li, Belhal Karimi, and Ping Li. On distributed adaptive optimization with gradient compression. In *International Conference on Learning Representations*, 2021.

[37] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 481–495, New York, NY, USA, 2016. Association for Computing Machinery.

[38] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018.

[39] Juncai Liu, Jessie Hui Wang, and Yimin Jiang. Janus: A unified distributed training framework for sparse mixture-of-experts models. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 486–498, New York, NY, USA, 2023. Association for Computing Machinery.

[40] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[41] Yurii Lyubarskii and Roman Vershynin. Uncertainty Principles and Vector Quantization. *IEEE Transactions on Information Theory*, 56(7):3491–3501, 2010.

[42] Bradley McDanel, Sai Qian Zhang, H. T. Kung, and Xin Dong. Full-stack optimization for accelerating cnns using powers-of-two weights with fpga validation. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, page 449–460, New York, NY, USA, 2019. Association for Computing Machinery.

[43] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[44] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[45] NVIDIA. NVIDIA Scalable Hierarchical Aggregation and Reduction Protocol (SHARP). . https://docs.nvidia.com/networking/display/SHARPv200, 2020.

[46] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 16–29, New York, NY, USA, 2019. Association for Computing Machinery.

[47] Lucian Petrica, Tobias Alonso, Mairin Kroes, Nicholas Fraser, Sorin Cotofana, and Michaela Blott. Memory-efficient dataflow inference for deep cnns on fpga. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 48–55, 2020.

[48] Constantin Philippenko and Aymeric Dieuleveut. Bidirectional compression in heterogeneous settings for distributed or federated learning with partial participation: tight convergence guarantees. *arXiv preprint arXiv:2006.14591*, 2020.

[49] Dan R. K. Ports and Jacob Nelson. When should the network be the computer? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 209–215, New York, NY, USA, 2019. Association for Computing Machinery.

[50] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[51] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. Cassini: Network-aware job scheduling in machine learning clusters. *arXiv preprint arXiv:2308.00852*, 2023.

[52] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. DeepSpeed-MoE: Advancing mixture-of-experts inference and training to power next-generation AI scale. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 18332–18346. PMLR, 17–23 Jul 2022.

[53] Ali Ramezani-Kebrya, Fartash Faghri, Ilya Markov, Vitalii Aksenov, Dan Alistarh, and Daniel M Roy. Nuqsgd: Provably communication-efficient data-parallel sgd via nonuniform quantization. *J. Mach. Learn. Res.*, 22:114–1, 2021.

[54] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[55] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[56] Mher Safaryan, Egor Shulgin, and Peter Richtárik. Uncertainty Principle for Communication Compression in Distributed and Federated Learning and the Search for an Optimal Compressor. *arXiv preprint arXiv:2002.08958*, 2020.

[57] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

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

[59] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. Compute trends across three eras of machine learning. *arXiv preprint arXiv:2202.05924*, 2022.

[60] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[61] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[62] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[63] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[64] Sebastian U. Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified sgd with memory. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 4452–4463, Red Hook, NY, USA, 2018. Curran Associates Inc.

[65] Peng Sun, Wansen Feng, Ruobing Han, Shengen Yan, and Yonggang Wen. Optimizing network performance for distributed dnn training on gpu clusters: Imagenet/alexnet training in 1.5 minutes, 2019.

[66] Ananda Theertha Suresh, X Yu Felix, Sanjiv Kumar, and H Brendan McMahan. Distributed Mean Estimation With Limited Communication. In *International Conference on Machine Learning*, pages 3329–3337. PMLR, 2017.

[67] Shay Vargaftik, Ran Ben Basat, Amit Portnoy, Gal Mendelson, Yaniv Ben Itzhak, and Michael Mitzenmacher. Eden: Communication-efficient and robust distributed mean estimation for federated learning. In *International Conference on Machine Learning*, pages 21984–22014. PMLR, 2022.

[68] Shay Vargaftik, Ran Ben-Basat, Amit Portnoy, Gal Mendelson, Yaniv Ben-Itzhak, and Michael Mitzenmacher. Drive: One-bit distributed mean estimation. *Advances in Neural Information Processing Systems*, 34:362–377, 2021.

[69] Hao Wang, Jingrong Chen, Xinchen Wan, Han Tian, Jiacheng Xia, Gaoxiong Zeng, Weiyan Wang, Kai Chen, Wei Bai, and Junchen Jiang. Domain-specific communication optimization for distributed dnn training. *arXiv preprint arXiv:2008.08445*, 2020.

[70] Wei Wang, Meihui Zhang, Gang Chen, H. V. Jagadish, Beng Chin Ooi, and Kian-Lee Tan. Database meets deep learning: Challenges and opportunities. *SIGMOD Rec.*, 45(2):17–22, September 2016.

[71] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 739–767, Boston, MA, April 2023. USENIX Association.

[72] Zheng Wang and Michael O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.

[73] Zhuang Wang, Haibin Lin, Yibo Zhu, and T. S. Eugene Ng. Hi-speed dnn training with espresso: Unleashing the full potential of gradient compression with near-optimal usage strategies. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 867–882, New York, NY, USA, 2023. Association for Computing Machinery.

[74] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1508–1518, Red Hook, NY, USA, 2017. Curran Associates Inc.

[75] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. Mlaas in the wild: Workload analysis and scheduling in large-scale heterogeneous gpu clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960. USENIX Association, 2022.

[76] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[77] Xilinx. Vitis AI. https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html, 2023.

[78] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. Using trio: Juniper networks' programmable chipset - for emerging in-network applications. In *Proceedings of the ACM SIGCOMM 2022 Conference*, ACM SIGCOMM '22, page 633–648, New York, NY, USA, 2022. Association for Computing Machinery.

[79] Yifan Yuan, Omar Alama, Jiawei Fei, Jacob Nelson, Dan R. K. Ports, Amedeo Sapio, Marco Canini, and Nam Sung Kim. Unlocking the power of inline Floating-Point operations on programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 683–700, Renton, WA, April 2022. USENIX Association.

[80] Xiang Zhou, Ryohei Urata, and Hong Liu. Beyond 1 tb/s intra-data center interconnect technology: Im-dd or coherent? *Journal of Lightwave Technology*, 38(2):475–484, 2020.

A Uniform THC Preliminaries

Given a bandwidth budget of b bits per coordinate (e.g., per gradient entry), we define a compression scheme using a pair of compression and decompression operators.

Definition 4 (compression operator) *A compression operator $C : \mathbb{R}^d \to \{0,1\}^{b\cdot d}$ takes a real-valued d-dimensional vector and outputs a $(b\cdot d)$-bits compressed representation.*

Definition 5 (decompression operator) *A decompression operator $D : \{0,1\}^{b\cdot d} \to \mathbb{R}^d$ takes a $(b\cdot d)$-bits compressed representation and outputs a real-valued d-dimensional estimate of the input vector.*

More generally, a compression scheme may require sending some additional information. We, therefore, allow $b\cdot d + O(1)$ bits in practice.

For a vector $x \in \mathbb{R}^d$, we denote its estimate by $\widehat{x} = D(C(x))$. The goal of a compression scheme is then, given a bandwidth budget b, to minimize some error metric, e.g., the expected squared error, $\mathbb{E}[\|x-\widehat{x}\|^2]$. Before describing THC, for ease of presentation, we present a simplified (uniform) version of the THC framework and later generalize it.

A.1 Uniform Homomorphic Compression

In distributed deep learning, at each training round, the mean of the workers' gradients forms the update of the model's parameters for the next round. Without compression, we could add all the workers' gradients and divide the results by the number of workers.

To reduce the bandwidth with a minimal impact on accuracy, the Distributed Mean Estimation (DME) problem has been extensively studied [4, 31, 41, 56, 66–68]. Namely, in DME, workers compress their gradients before sending them for aggregation. Most DME works only consider compression in this direction, while messages from the parameter server to workers remain uncompressed. To achieve bidirectional compression, several works further suggest that the server, after decompressing and aggregating the gradients, will recompress the result before sending it back (e.g., [20, 48]), introducing additional delay and error. Avoiding these problems motivates the following definition; for convenience, we henceforth denote $\langle i \rangle = \{0,\ldots,i-1\}$ for any $i \in \mathbb{N}$.

Definition 6 (Uniform Homomorphic Compression) *We say that a compression scheme (C,D) is* uniform homomor-

phic *if for any* $n,d \in \mathbb{N}$ *and* $x_0,x_1,\ldots,x_{n-1} \in \mathbb{R}^d$*, it satisfies*

$$\frac{1}{n}\cdot\sum_{i\in\langle n\rangle} D(C(x_i)) = D\left(\frac{1}{n}\cdot\sum_{i\in\langle n\rangle} C(x_i)\right).$$

That is, a Uniform Homomorphic Compression (UHC) scheme allows averaging the compressed representations and applying a single decompression invocation.

That is, by using UHC, the parameter server can sum up the compressed gradients and send $\sum_i C(x_i)$ back (in compressed form) without increasing the delay and error.

A.2 Uniform Stochastic Quantization

Two desired properties of gradient quantization schemes in a distributed setting are *unbiasedness* (i.e., $\mathbb{E}[\widehat{x}] = x$) and *independence* (i.e., each worker makes the random choice of their quantization values independently). These features are especially useful in distributed deep learning as the errors of the different workers then cancel out on average rather than add up, leading to a better estimation of the mean.

One of the most fundamental compression techniques that offer both properties is *uniform stochastic quantization* (USQ). Intuitively, given a vector, $x \in \mathbb{R}^d$, and denoting its minimum by m and maximum by M, using USQ to quantize each coordinate $x[j]$ to a single bit means rounding it to m with probability $(x[j]-m)/(M-m)$ and to M with probability $(M-x[j])/(M-m)$. That is, the sender encodes the coordinate $x[j]$ using one bit $C(x)[j] \in \{0,1\}$ and the receiver estimates it as $\widehat{x} = D(C(x)) = m + (M-m)\cdot C(x)$. Note that m and M need to be sent to the receiver as well. However, considering that d is large, this overhead is negligible.

This idea generalizes to any number of bits per coordinate b by partitioning the *range* $[m,M]$ into 2^b-1 *uniform* (i.e., equal-length) intervals where each entry is rounded to one of its nearest endpoints c_0,c_1 with probabilities $(x[j]-c_0)/(c_1-c_0)$ and $(c_1-x[j])/(c_1-c_0)$ to make the estimate unbiased. That is, when the message for coordinate $x[j]$, $C(x)[j] \in \langle 2^b\rangle$, is sent to the receiver that estimates the coordinate as $m + C(x)[j]\cdot(M-m)/(2^b-1)$.

Despite its popularity and simplicity, in our context, USQ has two main drawbacks. First, USQ is not homomorphic; this is because each worker has *its own* minimum and maximum values, and accordingly, the b-bit messages describing the same coordinate by different workers are not amenable to aggregation without decompression. Second, USQ's error highly depends on the input vector's distribution, e.g., the difference between the minimum and maximum. For example, for $b=1$, if the input vector is $(1,-1,0,0,\ldots,0)$, all the zero-valued coordinates will be rounded with an error of 1, and the vector's estimate will greatly differ from the input.

Several recent works propose pre-processing each gradient and post-processing the average's estimate. The insight is to change the vector's distribution prior to quantization to avoid bad cases. For example, [66] proposes to preprocess by applying the randomized Hadamard transform to ensure that the range is small with high probability and postprocess using the inverse transform. As another example, Kashin's representation [41, 56] allows projecting the vector into a higher-dimensional space with similar magnitude coefficients.

B Optimally Solving the Lookup Table Optimization Problem

As described in Section 5.2, the solution of the following optimization problem yields the optimal lookup table $T_{b,g,p} = T$. Observe that the problem depends on the parameters b,g,p, where b is the number of bits workers send for each quantized coordinate, g is the granularity (the range of values that the table can take), and p is the expected fraction of transformed and scaled coordinates that are not taken into account when determining the truncation range $[-t_p,t_p]$.

$$\underset{P,T}{\text{minimize}} \int_{-t_p}^{t_p} \sum_{z\in\langle 2^b\rangle} P(a,z)\cdot(a-T(z))^2\cdot\phi(a)\cdot da$$

subject to

$$\begin{aligned}
&(\textit{Unbiasedness}) && \sum_{z\in\langle 2^b\rangle} P(a,z)\cdot T(z) = a && \forall a\in[-t_p,t_p]\\
&(\textit{Probability}) && \sum_{z\in\langle 2^b\rangle} P(a,z) = 1 && \forall a\in[-t_p,t_p]\\
& && P(a,z)\geq 0 && \forall a\in[-t_p,t_p],\, z\in\langle 2^b\rangle\\
&(\textit{Granularity}) && T(z)\in\left\{\tfrac{2t_p}{g}\cdot i - t_p \mid i\in\langle g+1\rangle\right\} && \forall z\in\langle 2^b\rangle
\end{aligned}$$

Recall that, without loss of generality, we may assume that $0 = T(0) < T(1) < \ldots < T(2^b-1) = g$, which significantly narrows down the search range from $(g+1)^{2^b}$ (which is the number of options to choose a table value for each table index). Namely, this observation means that the number of possible options for T is $SaB(g-2^b-1,2^b-1)$ (*SaB* stands for the *stars-and-bars*), where $SaB(n,k) = \binom{n+k-1}{k-1}$ is the number of options for throwing n identical balls into k distinct bins. This is because we can think of 2^b-1 bins representing the values $\{T(i+1)-T(i) \mid i\in\langle 2^b-1\rangle\}$; this way, 'throwing a ball' into the i'th bin corresponds to increasing the difference by 1, and we have $g-2^b-1$ balls as all bins must be non-empty to enforce the strict monotonicity. The number of options is therefore $SaB(g-2^b-1,2^b-1) = \binom{g-3}{2^b-2} \ll (g+1)^{2^b}$. For example, if $b=4, g=51$, we reduce the number of options from $52^{15} \approx 5.5\cdot 10^{25}$ to $\binom{48}{14} \approx 4.8\cdot 10^{11}$. We note that these b,g values are the largest ones that we have found to be of interest as they yield a solution whose accuracy is on par with an uncompressed baseline.

Algorithm 4 Stars-and-Bars (n,k) Enumeration

Initialize $B[0]=n, \quad B[1]=B[2]=\ldots,B[k-1]=0$
`Yield`(B) ▷ This is the 0'th option
for $j=1,2,\ldots,SaB(n,k)-1$ **do**
  $a=\min\{i \mid B[i]>0\}$
  $B[a+1]=B[a+1]+1$
  $S=B[a]-1$
  $B[a]=0$
  $B[0]=S$
  `Yield`(B) ▷ B is the j'th option
end for

To further reduce the number of options, if g is odd, we leverage the symmetry of the normal distribution (i.e., that $\phi(a)=\phi(-a)$ for any $a\in\mathbb{R}$). In particular, together with the fact that the number of table indices is even, this implies that a table index $T(z)$ exists in the optimal table if and only if there exists z' such that $T(z')=g-T(z)$. In particular, this can be manifested as the following additional constraint:

$$(\textit{symmetry}) \qquad T(z)=T(z+2^{b-1})-\tfrac{g+1}{2} \quad \forall z\in\langle 2^{b-1}\rangle$$

Notice that this further reduces the number of options to $SaB\left(\frac{g+1}{2}-2^{b-1}-1,2^{b-1}-1\right)$. Using the $b=4,g=51$ example, we reduced the number of options to just 100947.

This allows us to efficiently solve the problem optimally by computing the target integral for every possible value of T. For each value, we use the fact that stochastic quantization (picking one of the two closest quantization values with probabilities that make it unbiased) is optimal given the quantization values [7]. In particular, this means that we can easily derive the optimal P values and thus compute the integral. For example, if $b=2,g=4$ and $T(0)=0,T(1)=1,T(2)=3,T(3)=4$, we can compute the integral as:

$$\int_{-t_p}^{-t_p/2}\left(\frac{a-(-t_p)}{t_p/2}\cdot(a-(-t_p/2))^2+\frac{-t_p/2-a}{t_p/2}\cdot(a-(-t_p))^2\right)\cdot\phi(a)\cdot da$$
$$+\int_{-t_p/2}^{t_p/2}\left(\frac{a-(-t_p/2)}{t_p}\cdot(a-t_p/2)^2+\frac{t_p/2-a}{t_p}\cdot(a-(-t_p/2))^2\right)\cdot\phi(a)\cdot da$$
$$+\int_{t_p/2}^{t_p}\left(\frac{a-t_p/2}{t_p/2}\cdot(a-t_p)^2+\frac{t_p-a}{t_p/2}\cdot(a-t_p/2)^2\right)\cdot\phi(a)\cdot da$$

Enumerating over the options: The last ingredient in our solver is a simple method to iterate over all the stars-and-bars options. Namely, consider wanting to throw n balls into k bins and let $B[i]$ denote the number of balls in the i'th bin. Then, the enumeration process is given by Algorithm 4.

The resulting solver is quite efficient: we ran it once for each of over 4000 different (b,g,p) combinations and computed all the optimal tables within mere minutes.

C Additional Parameter Server Details

C.1 Pseudocode of Parameter Server (PS)

Pseudocode 1 THC PS processing logic

Input: Gradient packet *pkt* from worker. Workers generate *pkt*.round_num and insert the *pkt*.num_worker along with gradient data for each packet before sending.

1: **if** *pkt*.roundnum < expected_roundnum[*pkt*.agtr_idx] **then**
2: Notify straggler
3: **else**
4: **if** *pkt*.roundnum = expected_roundnum[*pkt*.agtr_idx] **then**
5: recv_count[*pkt*.agtr_idx] += 1
6: **else**
7: recv_count[*pkt*.agtr_idx] = 1
8: expected_roundnum[*pkt*.agtr_idx] = *pkt*.roundnum
9: **end if**
10: Table indices lookup
11: Aggregate table values
12: **if** recv_count[*pkt*.agtr_idx] = *pkt*.num_worker **then**
13: Multicast back aggregation result
14: **else**
15: Drop *pkt*
16: **end if**
17: **end if**

The PS progressing logic is demonstrated as shown in Pseudocode 1. When workers' compressed gradient packets arrive, the PS will first check whether the `pkt.roundnum` is less than the `expected_roundnum` it stores. If so, then this packet is carrying obsolete data, and the PS will discard this packet and notify the sender that it is likely straggling (Line 1-2). Otherwise, the PS regards it as a normal case and updates the corresponding `recv_count` counter. (Line 5-7) After the PS finishes the table lookup and aggregation process for each packet, it will check if the aggregation is complete by comparing its `recv_count` counter and the `pkt.num_worker` (Line 12). If the aggregation is complete, it will multicast the aggregated gradient packet back, or drop the packet (Line 13-16).

C.2 Switch Resources Usage

The programmable switch version of THC Parameter Server has 32 aggregation blocks. Each aggregation block has a copy of the lookup table and can aggregate 32 bits (i.e., four 8-bit table values) in one pass. Overall, the programmable switch PS consumes 39.9 Mb of SRAM and 35 ALUs. THC workers send packets of 1024 table indices, so each packet needs $\frac{1024}{(32\times4)}=8$ passes to have all 1024 elements aggregated. Therefore, we recirculate a packet twice through each of the four pipelines (eight passes in total) and consume up to two recirculation ports per pipeline. When we have N workers, the first $N-1$ packets are dropped after aggregation while the N-th packet is recirculated through all four pipelines once again to collect the aggregation results.

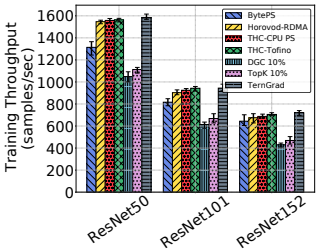


Figure 12: Throughput of training ResNet models on the local testbed.

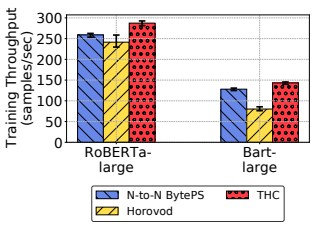


Figure 13: Throughput of training RoBERTa-large and Bart-large on AWS EC2.

D Evaluation Figures

D.1 Computational-intensive Model Training with THC

Figure 12 demonstrates the throughput results of training ResNet [23] models on our local testbed. Due to their computational-intensive nature, ResNet models don't experience much network bottleneck and hence don't benefit from accelerating inter-machine communication. Even with the most aggressive TernGrad compression, we are only able to improve the training throughput by up to 4.5% of that of Horovod-RDMA. So, computational-intensive models are poor candidates for gradient compression and should be trained with full-precision gradients unless the network bandwidth is low.

D.2 AWS EC2 Large Language Models Training Results

Figure 13 show the throughput results of training RoBERTa-large and Bart-large on AWS EC2. We achieve a 1.11× throughput improvement for RoBERTa-large and a 1.12× throughput improvement for Bart-large.

D.3 Optimizations of THC

To see how THC performs with the different optimizations mentioned in Section 4, we train both uniform and non-uniform THC with different optimizations. We use 4 workers on SST2 [63] with RoBERTa. To measure the performance, we enable all optimizations on THC, and then we run uniform THC (UTHC) with and without rotation and error feedback. We keep all optimizations for THC because the algorithm assumes rotation and error feedback to be enabled a prior, so it would not make sense to disable them for the test.

From Figure 14, we see that THC performs the best overall as expected, nearly reaching baseline accuracy. UTHC with and without error feedback seems to perform similarly, although error feedback seems to increase the performance slightly. The largest difference is disabling rotation, which drops the final accuracy by about 5%. This is expected, since removing rotation introduces a large bias into the algorithm, resulting in a large error.

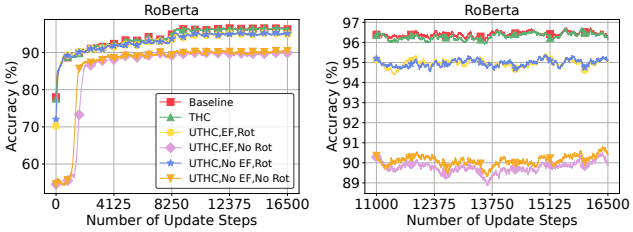


Figure 14: Accuracy of THC with Optimizations. The figure on the right is zoomed in on the last epoch, and accuracies are calculated on a sliding window of 336 batches.

D.4 NMSE for Different Granularities

We repeatedly compute the NMSE of the compression under different granularities. A gradient is first drawn from a lognormal distribution (which well approximate gradients in neural networks) and then copied multiple times to match the number of simulated workers. We run THC on the copies of the gradient and compute the NMSE. Repeating 100 times, we report the average NMSE as we increase the granularity. Figure 15 varies the granularity of THC, while maintaining 10 workers and used a p-fraction of 1/1024. Three curves are plotted with bit budget 2/3/4.

In Figure 15, we first note that the largest discrepancy in NMSE is between different bit budgets where the error decreases by almost an order of magnitude between bit budgets of 2 to 3 to 4. The bit budget corresponds directly to the compression ratio of the algorithm, so it is expected that the accuracy should increase as we use more bits for compression. Furthermore, the NMSE of THC also decreases as the granularity increases since larger granularity values allows for more fine-grained choices of quantization values, though this effect is more difficult to see.

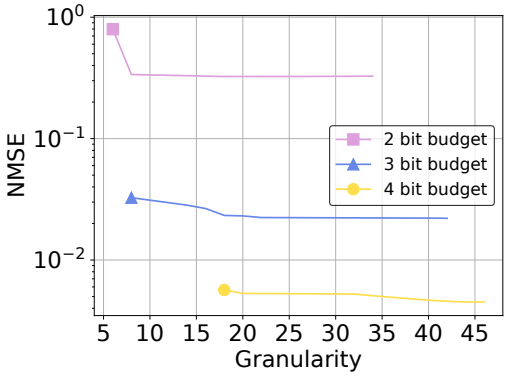


Figure 15: NMSE under different granularities and number of workers (plotted on log scale)

D.5 Test Accuracies for Resiliency Results

Similar to the training accuracy results, the test accuracy shows that (1) synchronization is beneficial when the system is lossy and (2) waiting for the top 90% of workers does not affect the final accuracy. Figure 16 shows that under 1%/0.1% loss, the discrepancy from baseline drops from 6%/3.2% to 1.5%/0.4%. For 80%/70% stragglers, the error from baseline is roughly 0.5%.

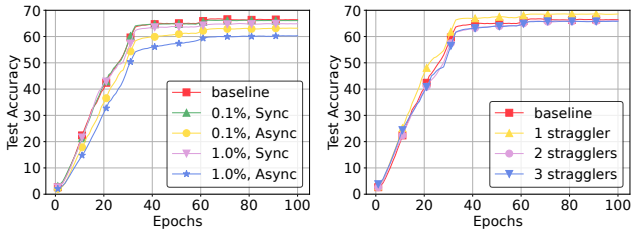


Figure 16: Test Accuracy with Gradient Losses

Accelerating Skewed Workloads With Performance Multipliers in the TurboDB Distributed Database

Jennifer Lam*, Jeffrey Helt*, Wyatt Lloyd*, Haonan Lu†
*Princeton University, †University at Buffalo

Abstract

Distributed databases suffer from performance degradation under skewed workloads. Such workloads cause high contention, which is exacerbated by cross-node network latencies. In contrast, single-machine databases better handle skewed workloads because their centralized nature enables performance optimizations that execute contended requests more efficiently. Based on this insight, we propose a novel hybrid architecture that employs a single-machine database inside a distributed database and present TurboDB, the first distributed database that leverages this hybrid architecture to achieve up to an order of magnitude better performance than representative solutions under skewed workloads.

TurboDB introduces two designs to tackle the core challenges unique to its hybrid architecture. First, Hybrid Concurrency Control is a specialized technique that coordinates the single-machine and distributed databases to collectively ensure process-ordered serializability. Second, Phalanx Replication provides fault tolerance for the single-machine database without significantly sacrificing its performance benefits. We implement TurboDB using CockroachDB and Cicada as the distributed and single-machine databases, respectively. Our evaluation shows that TurboDB significantly improves the performance of CockroachDB under skewed workloads.

1 Introduction

Distributed databases support large applications by sharding data across many machines to provide capacity far greater than what can fit on a single machine. However, these databases often experience severe performance degradation under *skewed workloads* where most requests contend on a small subset of data. This contention results in excessive aborts and retries that are expensive in the distributed setting. Unfortunately, many real-world workloads are highly skewed [3, 9, 10, 62].

In contrast, single-machine databases store all data on one machine. Although these databases cannot support large-scale applications, they handle skewed workloads more efficiently. For instance, Cicada [37], a single-machine database, can achieve much higher throughput and lower latency for TPC-C New-Order than CockroachDB, a representative distributed database, running on 48 servers [54]. This drastic difference stems from two *performance multipliers*, such as local concurrency control and one-stop execution, which single-machine databases can employ due to their centralized nature, but distributed databases cannot.

Local concurrency control techniques handle conflicts more efficiently by leveraging global knowledge of, and centralized control over, transactions, as all transactions access the same machine. However, when data is spread over multiple servers, these techniques, such as memory fences in Silo [58] and shared lock tables in MVTL [1], are infeasible.

One-stop execution, which handles transactions entirely within a single machine, enables shorter transaction lifetimes. For instance, a lock's acquisition and release, as part of transaction execution on a single machine, takes only nanoseconds to microseconds. However, distributed lock management, which requires multiple round trips between servers, takes orders of magnitude longer. Short transaction lifetimes lower the likelihood of conflicts and thus aborts. Aborting and retrying distributed transactions is more costly than aborting local transactions, due to network delays.

Empowered by local concurrency control and one-stop execution, single-machine databases offer a natural solution to the challenge of skewed workloads. Therefore, this paper proposes a novel hybrid architecture that employs a single-machine database within a distributed database to improve performance under skewed workloads. The single-machine database "turbocharges" overall performance under skew while the distributed database scales capacity.

Specifically, one server of the distributed database is designated as the *turbo*, which runs a single-machine database. The turbo co-locates many popular, contended data items, creating a focal point on which the single-machine database can concentrate its performance multipliers. The remaining servers run a distributed database to handle less contended requests, which access less popular data.

TurboDB is the first distributed database to employ this hybrid architecture. The architecture enables it to achieve significantly better performance than representative distributed databases under skewed workloads. However, it requires TurboDB to overcome two new correctness challenges.

First, a transaction may access data on both the single-

machine and distributed databases. Thus, some of its requests are executed with local concurrency control employed by the turbo while the rest of its requests are executed with distributed concurrency control. TurboDB must ensure the transaction as a whole is isolated from other transactions, and all transactions provide the correct consistency guarantees. Second, TurboDB must make the turbo fault tolerant. Although there are replication techniques for single-machine databases, none automatically work for TurboDB because they require full control of transactions, which the turbo does not have, i.e., the turbo is part of the distributed database, and its execution partially relies on the rest of the system.

TurboDB addresses these challenges, while preserving the performance benefits of the turbo, through two novel designs: Hybrid Concurrency Control and Phalanx Replication.

Hybrid Concurrency Control (HCC) leverages timestamp ordering to stitch together local and distributed concurrency control protocols. HCC ensures all requests of the same transaction commit at the same timestamp, and that all transactions are serialized in their timestamp order, thus guaranteeing process-ordered serializability [16, 40]. To maximize the performance advantages of local concurrency control, HCC applies *finale commit*, a serial-commit protocol that avoids unnecessary blocking and aborts on the turbo.

Phalanx Replication tackles the unique challenge of replicating a single-machine database in a distributed setting. A later-received transaction may be assigned a smaller timestamp due to clock skew and network asynchrony. Thus, existing replication techniques for single-machine databases that rely on assigned timestamps being monotonically increasing do not automatically work for TurboDB's setting. To address this challenge, Phalanx uses Frontline, a mechanism that determines the correct replication order even when timestamps may be out-of-order. Phalanx also employs a set of techniques to reduce replication costs, including per-core replication and decoupled log replay. The non-turbo servers are replicated with standard techniques for distributed databases, e.g., Raft [43].

We implement TurboDB using Cicada [37] and CockroachDB [54], which are representative single-machine and distributed databases, respectively. We evaluate TurboDB using YCSB+T and TPC-C, with a variety of read-write ratios and levels of skew. TurboDB achieves up to an order of magnitude higher throughput and 50% lower latency for highly-skewed YCSB+T, and up to 1.6× higher throughput and lower latency for highly contended TPC-C than CockroachDB.

In summary, this paper makes the following contributions:

- A novel system architecture that incorporates a single-machine database within a distributed database to "turbocharge" the performance under skewed workloads.
- TurboDB, the first design that leverages this new architecture using HCC and Phalanx to ensure the correctness of the combination while retaining the performance benefits of the turbo.
- An implementation and evaluation that shows TurboDB outperforms a representative distributed database by up to an order of magnitude under skewed workloads.

2 Background and Motivation

This section provides background on distributed databases and then discusses the challenge of skewed workloads.

2.1 Distributed Databases

Front-end *client* machines translate user requests into transactions whose requests are executed on the *servers* that store the data. Databases run concurrency control protocols to ensure that transactions appear to take effect in an order that satisfies specific consistency guarantees. TurboDB provides *process-ordered serializability* [16, 40], which guarantees there exists a total order amongst committed transactions, and the total order respects the order in which clients issue transactions. Process-ordered serializability is stronger than snapshot isolation and plain serializability [44].

Fault tolerance. Distributed databases tolerate server failures by replicating each server onto multiple replicas through consensus protocols such as Raft [43].

2.2 The Challenge of Skewed Workloads

Many real-world workloads are highly skewed [2, 4, 8–10, 28, 62]. For instance, Facebook's TAO reports the most popular data items are queried several orders of magnitude more often than other objects [3, 9], and Twitter's Twemcache reports an even higher skew [62]. Skewed workloads are difficult in general and particularly adversarial to distributed databases.

First, skewed workloads introduce more *conflicts*—i.e., concurrent transactions access overlapping (popular) data items with at least one write—in a distributed setting, because distributed concurrency control must coordinate multiple servers, thus prolonging transaction execution due to network transmission times. The longer the execution, the more likely it is that transactions conflict. Conflicts often result in aborts, which are especially expensive as retrying distributed transactions takes a long time, due to network delays.

Second, the overall performance of the database is limited by its performance on the few popular data items as they are accessed by most requests. As a result, the excessive aborts and prolonged execution, due to distributed concurrency control that often incurs multiple rounds of inter-machine communication, on the popular data have a disproportionately large negative effect on the overall performance.

A natural solution. Single-machine databases can better handle skewed workloads because their local concurrency control executes conflicting transactions more efficiently by

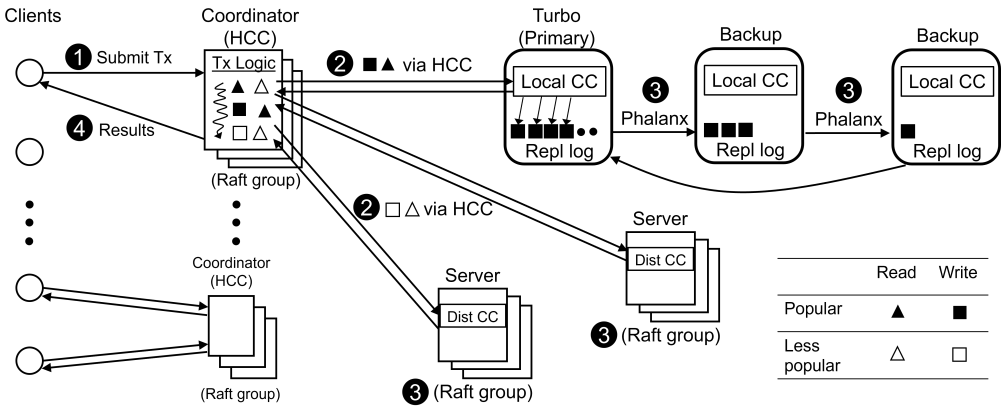


Figure 1: TurboDB designs Hybrid Concurrency Control (HCC) to integrate the turbo and servers. The requests on popular keys are mostly sent to the turbo, which runs local concurrency control and a specialized replication protocol. Less popular requests are processed by servers running distributed concurrency control and a standard replication protocol.

employing techniques that are infeasible in a distributed setting, such as memory fences [58], shared lock tables [1], and single-threaded timestamping [23]. Because no cross-server coordination is needed, transactions can be processed in one stop within the machine, thus greatly shortening the execution time and, in turn, reducing conflicts. These performance multipliers, i.e., local concurrency control and one-stop execution, enable fast processing of popular data items, lifting the performance bottleneck of the overall database.

3 Design Overview

TurboDB is built on a hybrid architecture that incorporates a single-machine database as the turbo. The turbo leverages its performance multipliers to efficiently execute transactions that contend on popular data, enhancing overall performance under skewed workloads.

3.1 A Hybrid Architecture

As shown in Figure 1, TurboDB is built on a standard distributed database and dedicates one of its storage servers to running a single-machine database (the *turbo*). The turbo and remaining *servers* run local and distributed concurrency control protocols, respectively. The turbo can send/receive requests to/from all the servers through RPCs. The servers, as part of the distributed database, are made fault tolerant using standard techniques. The turbo is made fault tolerant through a special technique: Phalanx Replication (§5).

Data placement. To leverage the turbo's performance multipliers for skewed workloads, TurboDB co-locates many popular keys on the turbo. The more popular keys the turbo stores, the greater performance improvement it brings to the system. Given the capacity and load on the turbo, some popular keys may remain on the servers. The information on data locations, i.e., the mapping of a key to the server or turbo that stores it, is stored on each server and can be kept up-to-date via standard techniques, e.g., Zookeeper [66].

3.2 Transaction Life Cycle

As shown in Figure 1, the overall flow of TurboDB executing a transaction with its hybrid architecture is as follows:

❶ A client receives a user request and translates it into a transaction. The client sends the transaction to one of the servers, and this server will serve as the coordinator for processing this transaction.

❷ The coordinator executes the transaction following Hybrid Concurrency Control (§4) by sending its requests to the servers and/or turbo which store the data this transaction accesses.

❸ Committed transactions replicate their state on servers through the standard technique the distributed database uses, e.g., Raft, and replicate their state on the turbo through Phalanx (§5).

❹ The coordinator replies to the client with the results of the transaction after it is committed and replicated, and the client then replies to the user.

Limitations. First, TurboDB assumes that data popularity does not change significantly or abruptly over time, i.e., data popularity changes on relatively slow timescales compared to how fast data can be migrated. To deal with changes in data popularity, TurboDB relies on existing techniques to migrate data between the turbo and servers. Second, co-locating popular keys on the turbo would make it more difficult to react to load spikes, i.e., sudden increases in request rate, and less resilient to failures or slowdowns. TurboDB partially mitigates this issue by not oversubscribing the turbo, i.e., it reserves enough CPUs and memory for moderate load increases.

As the first step in exploring a hybrid database architecture, TurboDB focuses on its core design challenges. We leave investigating the above limitations to future work.

Core challenges. The core challenges in TurboDB's use of a hybrid architecture are ensuring correctness and fault tolerance while preserving the high performance of the turbo.

4 Hybrid Concurrency Control

This section explains Hybrid Concurrency Control (HCC), which orchestrates the local and distributed concurrency control protocols co-existing in TurboDB's architecture. HCC ensures the system as a whole is process-ordered serializable, without significantly sacrificing the turbo's performance.

4.1 HCC Insight

HCC ensures consistency across the whole system by coordinating the local and distributed concurrency control protocols. A naive design would use traditional two-phase locking (with two-phase commit) across the turbo and servers to handle transactions that access both databases. While this ensures that transactions are serialized, it negates the performance benefits of using the turbo. Locking popular keys, even for one RTT across phases, prolongs the transaction's lifetime and reduces concurrency, sacrificing one-stop execution (§2.2).

To maintain the performance multipliers of the turbo, we apply a specialized two-phase protocol (consisting of the execute and commit phases) that does not acquire distributed locks on the turbo in the execute phase and employs *finale commit*, which is a serial two-step mechanism, in the commit phase to reduce unnecessary aborts on the turbo and enable one-stop execution.

To provide process-ordered serializability, HCC leverages timestamp ordering [7] to ensure that both local and distributed concurrency control protocols commit all requests of a transaction at the same timestamp, which represents the transaction's serialization point. HCC thus assumes both local and distributed protocols are timestamp-based, which is true of many existing protocols [14, 37, 41, 54, 58].

4.2 The Execute Phase

The client starts a transaction by generating a unique timestamp, a combination of the client's ID and the current time. Timestamps generated by the same client are strictly increasing. The client sends the transaction and timestamp to a server, which acts as the transaction's coordinator. The timestamp is used to inform the local and distributed protocols to commit all requests of this transaction at this timestamp.

Algorithm 4.1 shows the coordinator logic. In the execute phase, the coordinator buffers writes locally, and issues read requests (lines 3–12). The values returned by these reads may be used to complete any missing key dependencies, e.g., the value returned by a read request determines what to read/write in another request (line 12). The coordinator sends each request and the timestamp to either a server or the turbo, based on the up-to-date key-location mapping it stores (lines 4–8).

Algorithm 4.1: Transaction coordinator logic

```
1  Function HybridConcurrencyControl(tx, t) :
2      results ← {} // transaction results
       // Begin the execute phase
3      for req in tx.read_set do
4          if req.key on turbo then
5              res, is_aborted ← LocalCC(req, t, "read_only")
               // remove turbo reads
6              tx.read_set ← tx.read_set − req
7          else
8              res, is_aborted ← DistributedCC(req, t)
9          if is_aborted is true then
10             exit(tx.abort)
11         results ← results ∪ res
12         UpdateKeyDependencies(tx, res)
       // Begin the commit phase
13     hot_set ← {}
14     for req in tx.write_set do
           // read_set now only has server reads
15         if req.key on turbo then
16             hot_set ← hot_set ∪ req
17             continue
           // Send writes required by DistCC
18         res, is_aborted ← DistributedCC(req, t)
19         if is_aborted is true then
20             exit(tx.abort)
21         results ← results ∪ res
22     for req in hot_set do
23         res, is_aborted ← LocalCC(req, t, "finale_commit")
24         if is_aborted is true then
25             SendAbortMsgToServers(tx)
26             exit(tx.abort)
27         results ← results ∪ res
28     SendCommitMsgToServers(tx)
29     return results
```

Each server executes reads following the distributed concurrency control protocol. The turbo, in contrast, encapsulates these reads as a standalone read-only transaction so that active locks are not left behind after the execute phase, ensuring that these reads do not block other transactions on the turbo. Specifically, the turbo executes the "read-only transaction" with its local concurrency control, by returning the values at the specified timestamp. Each key accessed by these reads stores a piece of metadata, which records that "transaction *tx* read this key at timestamp *t*." This metadata will be used in the commit phase if the same transaction updates the same keys. In such cases, this metadata signals to the turbo that the reads (which were executed as a standalone read-only transaction) in the first phase and the writes in the second phase are from the same transaction, and the turbo will handle the commit accordingly. We discuss the details next.

4.3 The Commit Phase

After all reads are executed and key dependencies are resolved, the transaction enters the commit phase (lines 13–29). A naive design would send the remaining requests, i.e., buffered writes, to the turbo and servers in parallel, committing them with corresponding concurrency control protocols. However, this would cause unnecessary aborts on the turbo, e.g., when the turbo commits its part of the transaction, but the distributed concurrency control aborts the other part, the transaction as a whole must be aborted, wasting the work on the turbo. Moreover, trying to commit in parallel requires extra coordination between the turbo and servers, which would result in prolonged execution on the turbo and forfeit its one-stop execution. Therefore, HCC enforces a serial commit order: it first attempts to commit on the servers before attempting to commit on the turbo.

Finale commit. The coordinator divides the buffered writes into cool and hot sets, which update the servers and the turbo, respectively (lines 13–17). The coordinator sends the cool set and the timestamp to the servers and attempts to commit these writes at the specified timestamp through distributed concurrency control, e.g., two-phase commit (2PC) could be involved. Other messages as part of the distributed concurrency control, e.g., prepare messages for reads, may be sent together, depending on the specific protocol (lines 18–21).

If at least one server decides to abort this transaction, e.g., at the end of the prepare phase of 2PC, the coordinator aborts this transaction without (unnecessarily) attempting to commit the hot set (lines 19 and 20). If the servers agree to commit the transaction, the coordinator sends the hot set and the timestamp to the turbo and attempts to commit these writes at the timestamp through local concurrency control (lines 22–27). If the turbo cannot commit a write because some requests have read the key at this timestamp, the turbo checks if the most recent reads are from the same transaction as this write, e.g., the "read-only transaction" in the execute phase, and allows the write to commit in this case.

If the turbo commits the hot set, the coordinator then sends a commit message to each involved server to finally commit this transaction (line 28); otherwise, the transaction is aborted, and an abort message is sent to each server (line 25). If the transaction commits, the coordinator can respond to the client without waiting for the acknowledgments of the commit messages (line 29), reducing latency by one RTT, a technique used in many systems, e.g., CockroachDB. If the transaction is aborted, it will be retried by the coordinator (lines 24–26).

Finale commit enforces a serial commit order: trying to commit the cool set on servers → servers are ready to commit → trying to commit the hot set on the turbo → the turbo commits → the servers commit. This serial order preserves the turbo's performance multipliers by ensuring that a transaction does not compete for resources on the turbo if it cannot be committed on the servers and that each transaction updates the turbo at most once at the commit time, thus preserving one-stop execution.

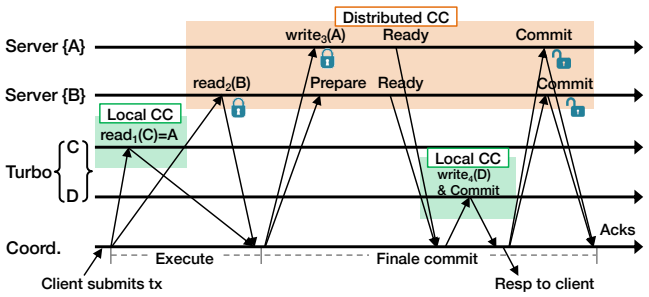


Figure 2: *tx* has 2 reads and 2 writes where the return value of $read_1$ determines which key $write_3$ updates. HCC integrates distributed concurrency control that may hold locks across phases and local concurrency control that does not.

Figure 2 shows an example execution. Transaction *tx* has four requests that include a key dependency: $read_1$ determines which key $write_3$ updates. Keys A and B are on two servers while C and D are stored on the turbo. In the execute phase, $read_1$ is executed by local concurrency control as a standalone read-only transaction without leaving behind locks while $read_2$ holds an active lock following distributed concurrency control (e.g., 2PL). $write_4$ is sent to the turbo in the commit phase only if $read_2$ and $write_3$ are ready to commit on the servers, which will finally commit $read_2$ and $write_3$ if the turbo commits $write_4$.

4.4 Correctness of HCC

This section explains why HCC enforces process-ordered serializability and ensures that transactions eventually terminate.

HCC is safe. HCC guarantees process-ordered serializability by satisfying the following requirements: (1) there exists a total order among all transactions, and (2) the total order respects the process order (§2.1).

First, HCC guarantees (1) by committing all requests of each transaction at the same timestamp, assigned by the client at the transaction's start. Specifically, the pre-assigned timestamp is used by the reads in the execute phase to retrieve values, by the servers to commit the cool set, and by the turbo to commit the hot set. Therefore, the timestamp is the serialization point of each transaction, and the timestamp order is the transactions' commit order. Because timestamps uniquely identify each transaction (§4.2), and timestamp order is a total order, transactions' commit order is total.

Second, HCC guarantees (2). Because each client generates timestamps in a strictly increasing manner (§4.2), later-issued transactions must have larger timestamps than any previously issued transactions by the same client, thus must appear later in the total order, respecting the process order.

HCC is live. Distributed concurrency control guarantees that the requests executed on the servers do not deadlock. Local concurrency control guarantees that the requests executed on the turbo do not deadlock. HCC's finale commit enforces a serial commit order between the servers and turbo, i.e., all transactions follow the same access order: servers $\rightarrow$ the turbo, thus transactions that access both servers and the turbo do not deadlock. Therefore, HCC guarantees that transactions eventually terminate.

5 Phalanx Replication

Another core challenge of TurboDB's hybrid architecture is correctly replicating the turbo for fault tolerance without trading off its performance multipliers.

5.1 Phalanx Insight

A naive solution would be deploying a standard consensus protocol designed for distributed systems, i.e. Raft. However, recent work on single-machine databases [26, 32, 46, 51] has shown that doing so significantly degrades the database's performance. They thus propose special techniques to replicate such databases. Specifically, they assign transactions *strictly increasing* timestamps and ensure that transactions are both committed and replicated in this timestamp order. Leveraging this timestamp order, these techniques remove most of the replication work from the critical path of transaction execution, preserving good performance. While Phalanx leverages the insight of these techniques, it cannot, however, directly apply them to TurboDB's hybrid setting.

The challenge is that these techniques require transactions be timestamped in *strictly increasing* order, which is straightforward for single-machine databases, which have full control over transactions. In contrast, as part of a hybrid database, the turbo passively accepts transactions whose timestamps have been predetermined by the clients. Due to network asynchrony, the turbo may have to execute transactions whose predetermined timestamps are smaller than any it has previously seen. Phalanx must tackle this unique timestamp challenge.

In the following subsections, we first explain the techniques Phalanx uses to preserve the performance of the turbo, then detail the timestamp challenges and Phalanx's timestamp management that tackles the challenges.

5.2 Protocol Basics

Phalanx arranges its replicas in a chain. The head of the chain is the primary; the rest are backups; and the last backup is the tail. The primary is the only replica that communicates with servers and runs local concurrency control, e.g., the "turbo" in Section 4 is this primary replica. Phalanx sequentially propagates the turbo's log, i.e., a sequence of committed updates grouped by transactions, down the chain to each backup. Once the tail receives the log, it sends the primary an acknowledgment (*ack*). Each backup applies the log's updates in an order specified by Phalanx (§5.3).

As a variant of primary-backup [59], Phalanx tolerates f failures with $f+1$ replicas while a coordination service, e.g., ZooKeeper [66], may be used to detect failures and handle membership changes in the replica group.

Decoupled replication. Phalanx preserves the turbo's performance multipliers by decoupling replication from transaction execution, shielding execution from replication delays as much as possible. Specifically, when the primary commits a transaction, it makes its effect immediately visible to future transactions (not yet to users), buffers its response into a response queue, and appends its committed updates to a replication log. The updates in the replication log are asynchronously propagated to the backups, while the primary continues to execute future transactions, i.e., replication does not block transaction execution. The responses of committed transactions are released by the response queue (i.e., they are sent to their coordinators) in order when these transactions' updates have been applied by all backups, i.e., when the primary receives an ack that indicates the completion of replication from the tail.

Allowing the primary to execute future transactions without being blocked by the replication of committed transactions best utilizes the turbo's performance multipliers. It is safe to make committed transactions visible to future transactions before they are replicated, i.e., there is no risk of cascading aborts, because they will certainly commit (the finale-commit guarantees that the turbo being ready to commit the transaction's hot set implies that its cool set must have first been ready to commit on the servers; neither side will abort). Moreover, their results will be visible to the users only after they are successfully replicated.

Per-core replication. Phalanx partitions the replication log across CPU cores. Each core propagates its sub-logs in parallel, reducing inter-core synchronization. Algorithm 5.1 shows the pseudocode. For simplicity, the pseudocode illustrates the primary and backups propagating a single log entry, but our implementation batches entries for performance. When the replication of a transaction starts (line 34), the primary finds an available core, e.g., core 3, and appends a new entry to core 3's sub-log (lines 36-37). This new entry contains this transaction's updates and commit timestamp. Periodically, core 3 propagates new entries to the next replica in the chain (line 41). When the next backup receives these new entries (Algorithm 5.2), it appends them to the sub-log for which its own core 3 is responsible and then propagates these entries to the next replica (lines 46-50). Due to a one-to-one correspondence between cores across all replicas (Figure 3), sub-logs managed by the same core become (eventually) identical across all replicas. The primary's replication log is stored on all backups and partitioned across cores in the same way.

Algorithm 5.1: Phalanx primary handling requests

```
30 Function HANDLEREQUEST(coord, req_msg) :
       // Receive a request msg from a coordinator
31     req ← req_msg.req; t ← req_msg.t; flag_str ← req_msg.flag
32     res, is_aborted ← LOCALCC(req, t, flag_str)
33     if not is_aborted then
34         response_queue.APPEND(coord, res, t)
           // Find a core i to start replication
35         if req is update then
36             log_entry_i ← {req, t}
37             sublog_i.APPEND(log_entry_i)
38             unacked_entries_i ← unacked_entries_i + log_entry_i
               // Update core i's t_s
39             core_i.t_s ← min{core_i.t_s, t}
               // Update the global Frontline if needed
40             t_f ← min{t_f, core_i.t_s}
               // Propagate entry to the next replica
41             tail_ack_i ← next_backup.PROPAGATE(log_entry_i, t_f)
               // Once receiving the ack
42             unacked_entries_i ← unacked_entries_i − log_entry_i
43             core_i.t_s ← min{unacked_entries_i.t}
44             t_f ← min{∀t_s ∈ cores.t_s}
```

Algorithm 5.2: Backup receives propagated entry

```
45 Function PROPAGATE(new_entry, t_f):
       // Append new log entry, then propagate
46     sublog_i.APPEND(new_entry)
47     if this is tail then
           // Tail acks to the primary
48         SENDACK(to:primary, new_entry, t_f)
49     else
           // Propagate down the chain
50         next_backup.PROPAGATE(new_entry, t_f)
       // Apply sublog entries up to t_f (i.e., t < t_f)
51     for entry in sublog_i do
52         if entry.t < t_f then
               // Apply entry to the database
53             this.APPLY(entry)
               // Remove replayed entry from sublog
54             sublog_i ← sublog_i − entry
```

Algorithm 5.3: Primary releases buffered responses

```
55 Function RELEASEBUFFRESPS():
       // Periodically invoked by the primary
56     for resp in response_queue do
57         coord ← resp.coord; res ← resp.res; t ← resp.t
58         if resp.t < t_f then
               // Releases and removes response
59             SENDRESPONSE(to:coord, res, t)
60             response_queue ← response_queue − resp
```

Decoupled log replay. When a backup receives new log entries, it appends these entries to its own sub-log of the corresponding core (line 46), which eventually replays the sub-log, i.e., the core applies the updates in its sub-log to the backup's database (lines 51-54). The backup propagates the new entries to the next backup without waiting for its local replay to complete (line 50). When these new entries finally reach the tail, the tail replays them and sends an *ack* to the primary, indicating that all backups have received these new log entries. Algorithm 5.2 shows the pseudocode.

When the primary receives the *ack* (line 41), the transactions associated with these new entries are considered safe, i.e., their state is stored on all backups and will be replayed by each backup. Yet, Phalanx must ensure a correct log-replay order before the transactions' responses can be returned to their coordinators. That is, Phalanx must guarantee that the transactions take effect on each backup in the same order as the primary, enabling backups to seamlessly and correctly take over the primary's role if the primary fails. Phalanx leverages timestamps to enforce such a log-replay order, and the next section explains a unique challenge Phalanx must tackle and how Phalanx overcomes it.

5.3 Timestamp Challenges & Frontline

Existing solutions execute and replicate transactions in a monotonically increasing timestamp order to ensure that once a transaction finishes, all the transactions before it (e.g., ones whose values this transaction may have read) must have been replicated and returned to their users, since they have smaller timestamps than the current transaction. Yet, the turbo has no control over timestamp generation and thus cannot expect timestamps to be monotonically increasing. Transactions are timestamped by the client machines and may arrive at the turbo in any order. Thus, naively applying existing solutions to TurboDB may lead to two issues.

First, out-of-order timestamps may incur unnecessary stalls, because the system cannot replicate a transaction until it is certain that no future transaction with a smaller timestamp will arrive. That is, later-arriving transactions with smaller timestamps block the replication of earlier transactions that have larger timestamps. For example, in Figure 3, w_3 arrives later than w_1 or w_2 but has a smaller timestamp, then it blocks w_1 and w_2 from replicating. Even worse, the system can never be certain when such transactions (e.g., w_3) may arrive or if they even exist.

Second, we cannot naively disobey the timestamp order, lest incorrect behavior arise during failover, as shown in Figure 3. Let's say w_1 and w_2 belong to tx_1 and tx_2 and have timestamps 8 and 10, respectively. If tx_2 reads the value written by w_1 (from tx_1), then we must ensure that by the time tx_2's w_2 is replicated w_1 must have replayed on all replicas and will not be lost in f failures of $f+1$ replicas. Otherwise, since tx_2 and tx_1 may be handled by different cores, backups may potentially replay w_2 without replaying w_1. If the primary then fails, no backup (or the new primary) will have applied w_1 to the database. As a result, future transactions will only observe w_2, but not w_1, which violates serializability.

This shows that Phalanx must replicate w_1 and w_2 in their timestamp order to preserve the dependencies between them, e.g., w_2 depends on w_1.

Design insight. Phalanx selectively obeys the timestamp order. It replays and returns transactions in timestamp order *only when necessary*: when there exist dependencies among transactions. We say that transaction tx_2 depends on transaction tx_1 (denoted $tx_1 \Rightarrow tx_2$) if their requests access the same key(s), and at least one is an update. Since HCC (§4) guarantees that a dependent (tx_2) always has a larger timestamp than the transaction it depends on (tx_1), Phalanx can preserve dependencies by replaying them in their strictly increasing timestamp order. If a transaction does not depend on another, i.e., $tx_1 \not\Rightarrow tx_2$ and $tx_2 \not\Rightarrow tx_1$, Phalanx can replay them in any order, avoiding unnecessary stalls.

Frontline. Phalanx enables selective timestamp ordering by designing *Frontline*, a timestamp tracking technique. Each core on the primary keeps track of t_s, which is the timestamp of the most recent, safe transaction (log entry) in this core's sub-log. A transaction is safe if it has been appended to the logs of all backups. A core updates its t_s when it receives the *ack* of its propagated entries from the tail. The frontline t_f is the minimum t_s across all cores. t_f represents a threshold timestamp at which all transactions in the database with timestamps less than or equal to t_f are safe.

When a core of the primary begins propagating log entries, it piggybacks the current frontline t_f with them. Upon receiving both the entries and t_f (line 45), the backup core appends the new entries to its sub-log and continues propagating both the entries and t_f to the next backup (line 50). This backup then replays all entries whose timestamps are less than t_f, i.e., these entries (updates) are applied to the database (lines 51–54). Log replay is uninterruptible, that is, cores will not handle new entries until the current replay is complete. Replayed updates are then removed from the sub-log. The tail sends an *ack* to the primary when it has appended these new entries to its log (lines 47–48).

When the primary's core receives the *ack*, it updates its core-local safe time t_s and cross-core frontline t_f accordingly (lines 43–44). The primary periodically loops through the buffered response queue to release the responses of transactions whose timestamps are less than or equal to t_f (algorithm 5.3). These responses are released to their corresponding coordinators in their timestamp order.

When the transactions received by the primary happen to have monotonically increasing timestamps, t_s and t_f are advanced monotonically, similar to existing techniques. When the primary receives a new, out-of-order transaction whose timestamp is smaller than that of earlier transactions *and* the current t_f, the primary immediately lowers t_f and the corresponding core's t_s below the transaction's timestamp (lines 39–40). By immediately lowering t_f, Phalanx prevents the primary from prematurely releasing the responses of future transactions that may depend on this out-of-order transaction. That is, any future dependent transaction will only be released after the out-of-order transaction is finished. Algorithm 5.3 shows pseudocode for releasing buffered responses.

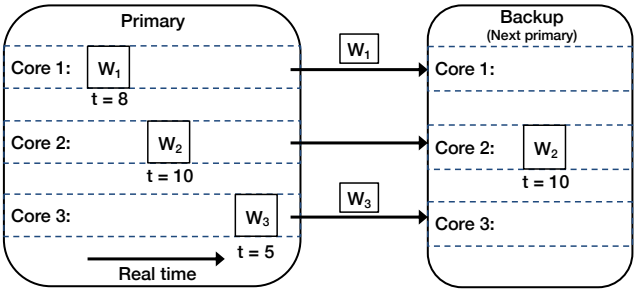


Figure 3: Writes operations w_1, w_2, w_3 from transactions tx_1, tx_2, and tx_3 (not shown), respectively, being replicated in each core's sub-log.

Buffering read responses. Although only updates are replicated to the backups, the primary must also buffer the responses of reads, including the reads in read-write transactions and read-only transactions. The primary handles these responses the same way that it handles update responses: it ensures that the updates observed by these reads have been stored on all replicas and cannot be lost in f failures.

Correctness. Phalanx guarantees that a transaction's response is released (i.e., this transaction is finished) only if its updates have been inserted in the logs of all backups and will eventually be replayed. For all transactions that have dependencies among them: HCC guarantees that they are timestamped in strictly increasing order, which reflects their dependencies. The frontline's forward movement returns them in this order. Otherwise, Phalanx does not block the replay of the current transaction tx_1 for the possible arrival of a later transaction tx_2 that has a smaller timestamp, thereby avoiding unnecessary stalls. This is safe because tx_1 must not depend on tx_2 as tx_1 was executed before tx_2, and tx_2 must not depend on tx_1 because tx_2's timestamp is smaller than tx_1's, thus it is safe to replay them in either order.

Failover. When the primary fails, the next live backup becomes the new primary. The new primary finishes replaying all sub-logs up to the frontline it knows, and then discards the remaining entries in its logs before servicing new requests. This is safe because the responses of these discarded transactions could not have been released to their coordinators. When the coordinators query the status of these discarded transactions, e.g., they have not received any responses for some time, the new primary replies with abort messages that make the coordinators abort these transactions on the servers.

6 Evaluation

We evaluate our system to answer the following questions:

- How well does TurboDB perform, compared to a representative distributed database, under skewed workloads?
- How well does TurboDB scale performance, specifically throughput, compared to the baseline?
- How well does TurboDB perform under different workloads with a variety of read-to-write ratios and levels of skew?

Implementation. We build TurboDB on CockroachDB [54] and Cicada [37], which are written in Go and C++, respectively. We change ∼4 K lines of Go in CockroachDB's codebase. We also employ Cicada's library to implement, replicate (i.e., implement Phalanx), and network (using gRPC [25]) a single-machine database in ∼10 K lines of C++. Of those, direct changes to Cicada's library are 5 lines long.

Baseline. We compare TurboDB with CockroachDB, which is a production distributed database that has been widely adopted by industry [34]. Its distributed concurrency control technique is a combination of timestamp-ordering and locking-based mechanisms, and it tolerates server failures with Raft. Our experiments have fault tolerance enabled for both TurboDB (Phalanx and Raft) and CockroachDB (Raft).

6.1 Experimental Setup

Workloads. We evaluate TurboDB under YCSB+T [17] and TPC-C [56]. YCSB+T contains one-shot key-value transactions, i.e., all requests are sent in one round in parallel as data locations are known a priori. Our experiments use the default parameters: 8 B key, 512 B value, 10 keys per transaction, and 95% reads. There are a total of 160 M keys. We vary the levels of skew by controlling the Zipfian constant (Zipf): uniform workloads have a Zipf of 0.01, medium-skewed workloads have a Zipf of 0.99 (∼8% of requests access the most popular key), and high-skewed workloads have a Zipf of 1.2 (∼25% of requests access the most popular key). We also include experiments that vary the read-to-write ratio.

TPC-C contains complex, multi-shot transactions, i.e., requests must be sent in multiple steps as data read in prior steps determines the read-/write-sets in later steps. TPC-C has five types of transactions: New Order, Payment, Delivery, Order Status, and Stock Level. We only implemented New Order which has the most complex transaction logic. We have 10 districts per warehouse and vary the level of skew by controlling the number of warehouses that are evenly distributed across machines. Our experiments show 64, 16, and 8 warehouses. The fewer warehouses, the more skewed the workload is.

Data placement. For the YCSB+T experiments, we promote hot data items to the turbo. We identify hot data items using a simple queries-per-second (QPS) count. It promotes as many data items to the turbo as can fit in its memory, taking care not to oversubscribe the turbo's CPU and memory capacity: 40 M keys of 160 M. We place the remaining, cool data items on CockroachDB, allowing its default data sharding schemes to balance the load. For the TPC-C experiments, we manually promote the two hottest tables—Warehouse and District—to the turbo. The remaining tables are partitioned by warehouse across the CockroachDB nodes.

Testbed. We run all experiments on CloudLab [22] in one data center. Each machine has 2.0 GHz CPUs with 8 physical (16 virtual) cores, 64 GB RAM, and a 10 Gbps network interface. For YCSB+T experiments, CockroachDB has 8 servers. Raft is run among these 8 servers instead of on a set of separate machines, as suggested by the CockroachDB technical team. TurboDB has 8 servers that handle the workloads. One of the 8 servers is the turbo, and the rest runs CockroachDB. TurboDB employs another 2 standalone machines as the backups, which do not directly handle the workload. Thus, TurboDB has a total of 10 server machines. The YCSB+T scalability experiments use up to 16 servers (18 servers for TurboDB). Similarly, the TPC-C experiments have 9 servers for CockroachDB and 11 servers for TurboDB.

An additional set of machines generate closed-loop client requests. Each experiment lasts 180 seconds. The first 120 warm up the system, e.g., the system shards the data. Performance metrics are collected during the remaining 60.

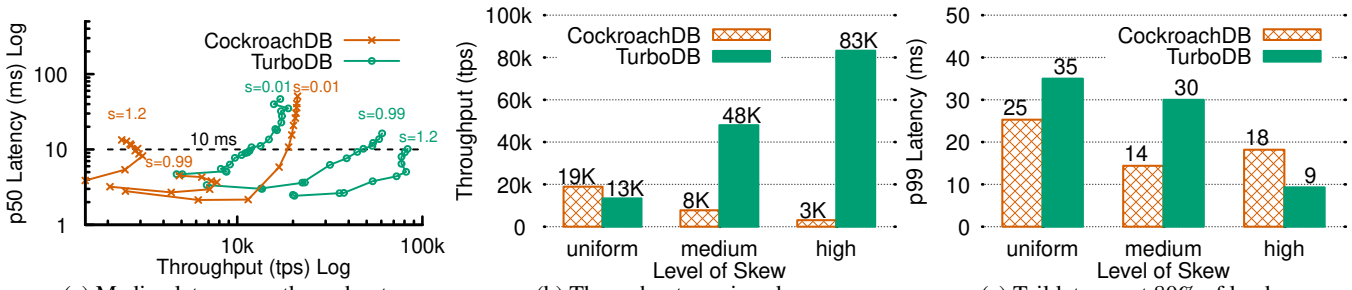
6.2 Result Overview

TurboDB outperforms CockroachDB by an order of magnitude higher throughput and 2× lower latency for YCSB+T, and by 1.6× higher throughput and lower latency for TPC-C, under skewed workloads. TurboDB scales out as well as CockroachDB under uniform workloads and shows much better scalability under medium and highly skewed workloads. TurboDB shows comparable performance to CockroachDB under uniform workloads and more significant performance improvements while skew increases. TurboDB consistently outperforms CockroachDB with different read-to-write ratios.

6.3 Latency & Throughput

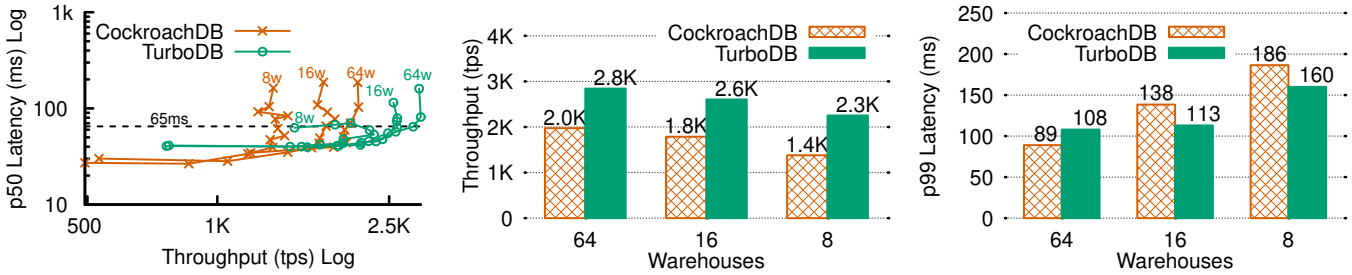
This section compares the performance of TurboDB and CockroachDB in terms of latency and throughput under YCSB+T and TPC-C workloads.

YCSB+T. Figure 4a plots the median latency vs. throughput graph for uniform, medium, and high skew as we increase load on the system. The dashed horizontal line shows a median latency of 10 ms, a reasonable operating point. TurboDB consistently outperforms CockroachDB under medium and high skew, due to its turbo being able to efficiently execute contended transactions. For instance, TurboDB has more than 4× lower abort rate than CockroadhDB with a Zipf of 1.2. Although TurboDB is not designed for uniform workloads,



(a) Median latency vs. throughput (b) Throughput varying skew (c) Tail latency at 80% of load

Figure 4: Latency (medium and tail) and throughput of TurboDB and CockroachDB under YCSB+T workloads with varying levels of skew (s indicates the Zipfian constant.



(a) Median latency vs. throughput (b) Throughput varying skew (c) Tail Latency at 80% of load

Figure 5: Latency (medium and tail) and throughput of TurboDB and CockroachDB under TPC-C New Order workloads with varying levels of skew, controlled by the number of warehouses (w).

its performance is comparable to (slightly worse than) CockroachDB for these workloads, due to the design choices TurboDB makes specifically for skewed workloads, e.g., finale commit and chain-shaped replication may increase latency.

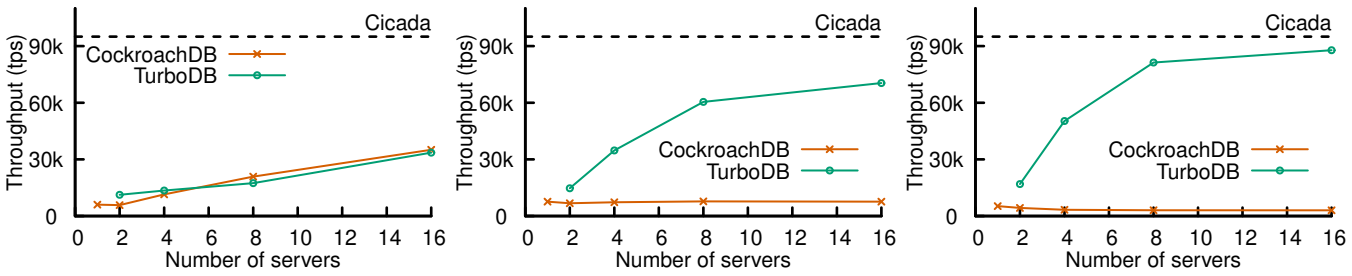
Figure 4b shows the throughput while varying the level of skew at the operating point, i.e., a median latency of 10 ms. TurboDB exhibits more significant throughput advantages when skew increases, e.g., up to 17× improvements. This is because a higher level of skew causes more transactions to abort on CockroachDB, while TurboDB can reduce aborts with its local concurrency control and execute these transactions more quickly with its one-stop execution. Moreover, TurboDB's throughput increases when skew increases because more requests can benefit from the turbo, while the performance of traditional databases often keeps getting worse when the workload becomes more skewed.

Figure 4c shows the tail (p99) latency when both systems operate at 80% of their maximum load. TurboDB exhibits slightly higher latency on uniform and medium skewed workloads. This is because HCC's serial finale commit and Phalanx's chain propagation increase latency, and this affects the tail of the distribution more significantly. However, this latency impact is offset by the latency improvements under high skew workloads where TurboDB has 2× lower tail latency.

TPC-C New Order. Figure 5 shows the performance under TPC-C New Order workloads, which exhibits a similar takeaway of performance improvements enabled by TurboDB while the improvements are not as significant as that for YCSB+T. This is because TPC-C transactions are multi-shot and have much more complex logic than YCSB+T, and because it is non-trivial to partition TPC-C and find the right popular data items to store on the turbo. Our experiments make the turbo store popular District tables, and we expect even greater performance improvements with careful partitioning. That said, TurboDB achieves consistently better performance under medium and high skew, as shown in Figure 5a. TurboDB achieves consistently higher throughput (up to 1.65× higher) with low (64 warehouses), medium (16 warehouses), and high skew (8 warehouses), as shown in Figure 5b. TurboDB has lower tail latency for medium and high skew (up to 1.5× lower), and slightly higher tail latency when skew is low. The low skew setting (64 warehouses) still exhibits some contention and is far from being uniform, and TurboDB's performance improvements may diminish when the number of warehouses is sufficiently large.

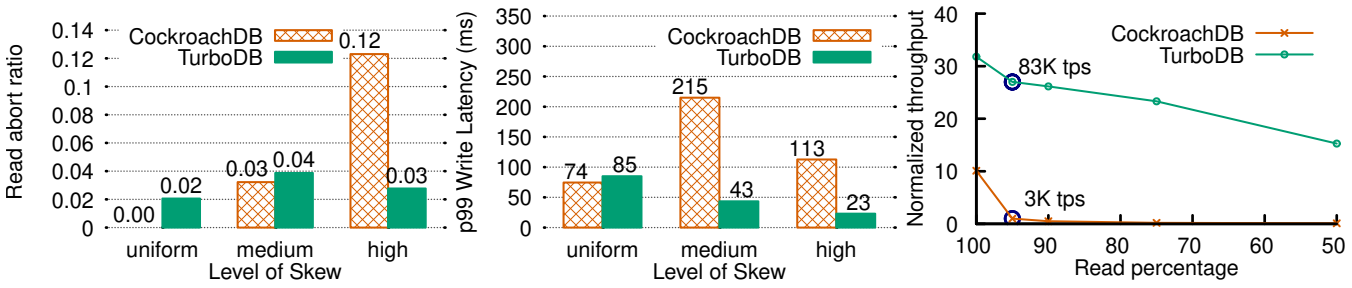
6.4 Scalability

Figure 6 shows peak throughput of TurboDB and CockroachDB under YCSB+T when we increase the number of machines (and the amount of data stored) in the system. Figure 6a shows that TurboDB can scale as linearly as CockroachDB under uniform workloads for which TurboDB is not designed specifically. This shows that the overhead of TurboDB's design under uniform workloads does not much



(a) Uniform ($s = 0.01$) (b) Medium skew ($s = 0.99$) (c) High skew ($s = 1.2$)

Figure 6: Scalability under YCSB+T read-heavy (95% reads) workloads while increasing the number of servers and the amount of data. The dashed line denotes the peak throughput of networked, replicated Cicada (the turbo).



(a) YCSB+T Abort ratio (b) p99 latency of write-only txns (c) Varying Read Fraction (s=1.2)

Figure 7: Additional experiments that report abort ratios, write tail latency, and throughput with different read fractions.

affect the system's overall performance, because the workload is "easy," i.e., uniform, and additional overhead is affordable.

TurboDB scales throughput significantly better than CockroachDB under medium and highly skewed workloads, as shown in Figure 6b and Figure 6c. CockroachDB does not scale under skewed workloads because it is bottlenecked on processing contended transactions that access a small set of popular keys. Adding more machines does not address the bottleneck. In contrast, TurboDB scales throughput linearly until the turbo reaches its capacity, i.e., when networked, replicated Cicada is running at maximum throughput.

TurboDB's throughput approaches that of networked, replicated Cicada, denoted by the dotted horizontal line, with a sufficient number of servers. TurboDB can sustain its throughput and scales no worse than traditional distributed databases after Cicada becomes the bottleneck. Moreover, TurboDB scales up faster at higher skew because more transactions can leverage the performance multipliers of the turbo.

6.5 Additional Experiments

We show more experiments with YCSB+T workloads.

Abort ratio. An important source of improvements enabled by the turbo is the reduction of aborts. (Another source is the fast execution of contended transactions, enabled by one-stop execution.) Figure 7a plots the abort ratio which is the number of the transactions that were ever aborted to the number of committed transactions, under uniform, medium, and high skew. While TurboDB has abort ratios similar to CockroachDB under uniform and medium skewed workloads, TurboDB has a 4× lower abort ratio under high skew, which is enabled by the performance multipliers of the turbo.

Write latency. To fully understand TurboDB's performance improvements, we examine the latency of write-only transactions from Figure 4b's workload. We focus on write requests because they lead to conflicts and because they reflect the costs of replication as only writes are replicated through Phalanx. Figure 7b shows the tail (p99) latency of write-only transactions at different levels of skew. TurboDB has significantly lower write latency at medium and high skew because the turbo can execute these writes with fewer aborts and because one-stop execution avoids cross-phase locking, while CockroachDB requires distributed locks (i.e., write intents) that significantly prolongs the execution time.

Varying read fractions. Figure 7c shows the throughput of both systems when we vary the read-to-write ratio of the highly skewed YCSB+T workloads (Zipf of 1.2). We normalize throughput against the maximum throughput under the default setting of 95% reads. Both TurboDB and CockroachDB have lower throughput with more writes, because write operations are more costly compared to reads and because writes increase the likelihood of conflicts, and TurboDB has consistently higher throughput than CockroachDB.

7 Related Work

TurboDB builds on earlier work on single-machine databases, distributed databases, partitioning techniques, and replication techniques. Section 5 discussed replication techniques. We now review each of the other categories in turn.

Single-machine databases. There exists a large body of research on designing and building fast single-machine databases [23, 24, 29, 31, 33, 37, 47, 58]. Because their data resides on one machine, these databases capitalize on readily available global views of the system to employ sophisticated optimizations (see discussion in §2). In contrast, in distributed databases, global views are non-trivial to construct.

TurboDB's design makes it possible for these optimizations to be used in distributed systems. What's more, TurboDB improves its performance while retaining the distributed system's ability to scale system capacity to support large-scale applications—ones that would have been too large to fit in a single-machine database.

We highlight the constraint that TurboDB requires the turbo (single-machine database) to employ a timestamp-based concurrency control. That said, many existing databases would make good candidates [29, 33, 37, 58].

Distributed databases and systems. There also exists a large body of work on distributed databases and distributed systems [12, 49, 52, 54, 55, 65]. Some achieve good performance by constraining operations of a transaction to access the same logical partition [11, 13]. These systems rely on careful data partitioning, which may be challenging for today's complex applications to achieve. In contrast, TurboDB requires no partitioning constraints.

Some other systems trade off strong consistency for better performance [11, 38, 39, 42, 50]. Unfortunately, weakly consistent transactions complicate application development, yet, are still subject to performance degradation under skewed workloads. In contrast, TurboDB provides strong consistency (process-ordered serializability).

Some other systems support restricted transaction models, e.g., read-only and/or write-only transactions [21, 38, 39]. Unfortunately, this complicates application development. In contrast, TurboDB supports general transactions.

Sequencers and RDMA. Some systems leverage a centralized component, e.g., a sequencer or a shared log, to serialize all transactions [5, 6, 30, 35, 36, 55, 60]. These techniques' insights bears similarity to TurboDB's, i.e., leveraging a powerful centralized entity to tackle the most challenging problems in the system's design. However, unlike TurboDB, they require that all transactions pass through the sequencer. Instead, TurboDB only forwards a fraction of (popular) keys through the turbo, enabling scalability with less internal complexity and fewer resources than recent scalable sequencer-based systems [18, 27].

Some systems leverage specialized hardware and network abstractions [20], e.g., RDMA and DPDK [19]. These networking optimizations are orthogonal to TurboDB's performance improvements, as TurboDB can also adopt and benefit from them.

Partitioning techniques. One line of work handles contention by partitioning the keyspace by workload access patterns, i.e., keys that are likely accessed together by transactions are co-located on the same machine [15, 45, 48, 53, 54, 57, 61, 63]. These works aim to reduce the number of nodes that each transaction must contact—ideally, only one—such that they are as "non-distributed" as possible. For instance, Chiller [64] and Quro [61] co-locate keys that are both popular and often accessed together on the same machine.

While these partitioning techniques benefit workloads where transactions minimally access keys on different machines, their benefits diminish when no such obvious groups of keys exist. In contrast, TurboDB's performance benefits are agnostic to whether the keyspace is partitionable. Even when transactions access both the turbo and the servers, TurboDB's HCC and Phalanx ensure such transactions benefit from the turbo's performance multipliers.

8 Conclusion

Distributed databases are challenged by skewed workloads, which are common in real-world applications. These workloads cause high contention, which are exacerbated by network latencies. TurboDB presents a novel hybrid architecture that integrates a single-machine database in a distributed database to "turbocharge" its overall performance under skewed workloads. TurboDB leverages the single-machine database's performance multipliers to efficiently execute contended transactions. It introduces new designs, HCC and Phalanx, that tackle the challenges of concurrency control and replication under its hybrid architecture. Consequently, TurboDB achieves up to an order of magnitude better performance than a representative distributed database.

Acknowledgments. We thank our shepherd, Dan Ports, and our anonymous reviewers for their invaluable feedback. We also thank Christopher Hodsdon, Yue Tan, Samuel Ginzburg, Mike Wong, and Gongqi Huang for their helpful suggestions. This material is based upon work supported by the National Science Foundation under Grant No. CNS 2241719, CNS 1824130, CNS 2327609, and CNS 2321724. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Availability. Code and experimental scripts are available at https://github.com/princeton-sns/TurboDB.

References

[1] Marcos K Aguilera, Tudor David, Rachid Guerraoui, and Junxiong Wang. Locking timestamps versus locking objects. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2018.

[2] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to adopting stronger consistency at scale. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[3] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: A database benchmark based on the facebook social graph. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1185–1196, 2013.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS and PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[5] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D Davis. CORFU: A shared log design for flash clusters. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2012.

[6] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–340, 2013.

[7] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[8] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE International Conference on Computer Communications (INFOCOMM)*, volume 1, pages 126–134. IEEE, 1999.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference (ATC)*, pages 49–60, 2013.

[10] Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, Nathan Bronson, Natacha Crooks, and Ion Stoica. TAOBench: An end-to-end benchmark for social network workloads. *Proceedings of the VLDB Endowment (PVLDB)*, 15(9):1965–1977, 2022.

[11] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment (PVLDB)*, 1(2):1277–1288, 2008.

[12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 261–264, 2012.

[13] CosmosDB. CosmosDB. https://azure.microsoft.com/en-us/services/cosmos-db/, 2022.

[14] James Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX Annual Technical Conference (ATC)*, Jun 2012.

[15] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: A workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1–2):48–57, Sep 2010.

[16] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with ordering guarantees. In *IEEE International Conference on Data Engineering (ICDE)*, 2004.

[17] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. YCSB+T: Benchmarking web-scale transactional databases. In *IEEE International Conference on Data Engineering (ICDE) Workshops*, pages 223–230, 2014.

[18] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 325–338, 2020.

[19] DPDK. DPDK. http://dpdk.org/, 2020.

[20] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[21] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *ACM Symposium on Cloud Computing (SoCC)*, 2013.

[22] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *USENIX Annual Technical Conference (ATC)*, pages 1–14, 2019.

[23] Jose M Faleiro and Daniel J Abadi. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment (PVLDB)*, 8(11):944–955, 2015.

[24] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment (PVLDB)*, 10(5), 2017.

[25] gRPC. gRPC, 2020. https://github.com/grpc.

[26] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: Replication at the speed of multi-core. In *European Conference on Computer Systems (EuroSys)*, pages 1–14, 2014.

[27] Christopher Hodsdon, Theano Stavrinos, Ethan Katz-Bassett, and Wyatt Lloyd. Mason: Scalable, contiguous sequencing for building consistent services. *Journal of Systems Research (JSys)*, 3(1), 2023.

[28] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of facebook photo caching. In *ACM Symposium on Operating Systems Principles (SOSP)*, November 2013.

[29] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Opportunities for optimism in contended main-memory multicore transactions. *Proceedings of the VLDB Endowment (PVLDB)*, 13(5):629–642, Jan 2020.

[30] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance RDMA systems. In *USENIX Annual Technical Conference (ATC)*, pages 437–450, 2016.

[31] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment (PVLDB)*, 1(2):1496–1499, Aug 2008.

[32] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: Execute-verify replication for multi-core servers. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 237–250, 2012.

[33] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1675–1687, 2016.

[34] Cockroach Labs. CockroachDB, 2020. https://www.cockroachlabs.com/.

[35] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 104–120, 2017.

[36] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 467–483, 2016.

[37] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *ACM International Conference on Management of Data (SIGMOD)*, pages 21–35, 2017.

[38] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 401–416, 2011.

[39] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 313–328, 2013.

[40] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The SNOW theorem and latency-optimal read-only transactions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 135–150, 2016.

[41] Haonan Lu, Shuai Mu, Siddhartha Sen, and Wyatt Lloyd. NCC: Natural concurrency control for strictly serializable datastores by avoiding the timestamp-inversion pitfall. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 305–323, 2023.

[42] MySQL. MySQL 5.6 Reference Manual. https://dev.mysql.com/doc/refman/5.6/en/, 2020.

[43] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (ATC)*, pages 305–319, 2014.

[44] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[45] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *ACM International Conference on Management of Data (SIGMOD)*, pages 61–72, 2012.

[46] Dai Qin, Angela Demke Brown, and Ashvin Goel. Scalable replay-based replication for fast databases. *Proceedings of the VLDB Endowment (PVLDB)*, 10(13):2025–2036, 2017.

[47] Dai Qin, Angela Demke Brown, and Ashvin Goel. Caracal: Contention management with deterministic concurrency control. In *ACM Symposium on Operating Systems Principles (SOSP)*, page 180–194, 2021.

[48] Abdul Quamar, K Ashwin Kumar, and Amol Deshpande. Sword: scalable workload-aware data placement for transactional workloads. In *International Conference on Extending Database Technology (EDBT)*, pages 430–441, 2013.

[49] Kun Ren, Dennis Li, and Daniel J Abadi. SLOG: serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment (PVLDB)*, 12(11):1747–1761, 2019.

[50] riak. riak, 2022. https://riak.com/index.html.

[51] Weihai Shen, Ansh Khanna, Sebastian Angel, Siddhartha Sen, and Shuai Mu. Rolis: A software approach to efficiently replicating multi-core transactions. In *European Conference on Computer Systems (EuroSys)*, pages 69–84, 2022.

[52] Swaminathan Sivasubramanian. Amazon DynamoDB: A seamlessly scalable non-relational database service. In *ACM International Conference on Management of Data (SIGMOD)*, pages 729–730, 2012.

[53] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment (PVLDB)*, 8(3):245–256, 2014.

[54] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruner, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The resilient geo-distributed sql database. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1493–1509, 2020.

[55] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1–12, 2012.

[56] TPC. TPC-C: An on-line transaction processing benchmark. http://www.tpc.org/tpcc/, 2020.

[57] Khai Q Tran, Jeffrey F Naughton, Bruhathi Sundarmurthy, and Dimitris Tsirogiannis. JECB: A join-extension, code-based approach to OLTP data partitioning. In *ACM International Conference on Management of Data (SIGMOD)*, pages 39–50, 2014.

[58] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–32, 2013.

[59] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–104, 2004.

[60] Michael Wei, Amy Tai, Christopher J Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, et al. vCorfu: A cloud-scale object store on a shared log. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 35–49, 2017.

[61] Cong Yan and Alvin Cheung. Leveraging lock contention to improve OLTP application performance. *Proceedings of the VLDB Endowment (PVLDB)*, 9(5):444–455, 2016.

[62] Juncheng Yang, Yao Yue, and KV Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *USENIX Symposium on Operating Systems*

Design and Implementation (OSDI), pages 191–208, 2020.

[63] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. Locality-aware partitioning in parallel database systems. In *ACM International Conference on Management of Data (SIGMOD)*, pages 17–30, 2015.

[64] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *ACM International Conference on Management of Data (SIGMOD)*, pages 511–526, 2020.

[65] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):12:1–12:37, 2018.

[66] ZooKeeper. Apache ZooKeeper. https://zookeeper.apache.org/, 2022.

SIEVE is Simpler than LRU: an Efficient Turn-Key Eviction Algorithm for Web Caches

Yazhuo Zhang*
Emory University

Juncheng Yang†
Carnegie Mellon University

Yao Yue
Pelikan Foundation

Ymir Vigfusson
Emory University & Keystrike

K. V. Rashmi
Carnegie Mellon University

Abstract

Caching is an indispensable technique for low-cost and fast data serving. The eviction algorithm, at the heart of a cache, has been primarily designed to maximize efficiency—reducing the cache miss ratio. Many eviction algorithms have been designed in the past decades. However, they all trade off throughput, simplicity, or both for higher efficiency. Such a compromise often hinders adoption in production systems.

This work presents SIEVE, *an algorithm that is simpler than LRU and provides better than state-of-the-art efficiency and scalability* for web cache workloads. We implemented SIEVE in five production cache libraries, requiring fewer than 20 lines of code changes on average. Our evaluation on 1559 cache traces from 7 sources shows that SIEVE achieves up to 63.2% lower miss ratio than ARC. Moreover, SIEVE has a lower miss ratio than 9 state-of-the-art algorithms on more than 45% of the 1559 traces, while the next best algorithm only has a lower miss ratio on 15%. SIEVE's simplicity comes with superior scalability as cache hits require no locking. Our prototype achieves twice the throughput of an optimized 16-thread LRU implementation. SIEVE is more than an eviction algorithm; it can be used as a cache primitive to build advanced eviction algorithms just like FIFO and LRU.

1 Introduction

Web caches, such as Content Delivery Networks (CDNs) and key-values caches, are widely deployed in today's digital landscape to reduce user request latency [14, 21, 22, 33, 69, 73, 76, 100], network bandwidth [54, 55, 79, 95], and repeated computation [28, 89, 97, 98]. As a critical component of modern infrastructure, these caches often have a large footprint. For example, Netflix used 18,000 servers for caching over 14 PB of application data in 2021 [68]; while Twitter reportedly had 100s of clusters using 100s of TB of DRAM and 100,000s of CPU cores for in-memory caching in 2020 [96].

At the heart of a cache is the eviction algorithm, which plays a crucial role in managing limited cache space. Such algorithms are efficient when they can retain more valuable objects in the cache to achieve a lower miss ratio—the fraction of requested objects that must be fetched from the backend. The quest for high efficiency has spurred a long repertoire of clever algorithms, but most, if not all, trade off simplicity in exchange for efficiency gains. For example, ARC [67], SLRU [55], 2Q [60], and MQ [106] manage multiple least-recently-used (LRU) queues to achieve better efficiency. LHD [16], CACHEUS [75], LRB [79], and GL-Cache [93] use machine learning techniques that further increase system and lookup complexity. Furthermore, many of these algorithms require explicit or implicit parameter tuning to achieve good efficiency on a target workload.

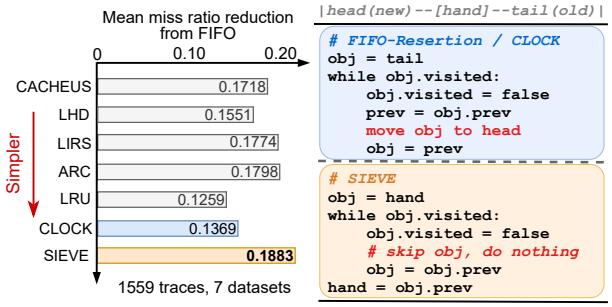


Figure 1: SIEVE is simple and efficient. The code snippet shows how FIFO-Reinsertion and SIEVE find eviction candidates. Minor code changes convert FIFO-Reinsertion to SIEVE, unleashing lower miss ratios than state-of-the-art algorithms.

The conventional wisdom among systems operators is that *simple is beautiful*: simplicity is a key appealing feature for an algorithm to be deployed in production since it commonly correlates with effectiveness, maintainability, scalability, and low overhead. To illustrate, note that most caching systems or libraries in use today, such as ATS [2], Varnish [11], Nginx [7], Redis [9], and groupcache [25], use only FIFO and LRU policies.

We have stumbled upon an easy improvement (Fig. 1) to a decades-old algorithm (FIFO-Reinsertion) that materially improves its efficiency across a wide range of web cache workloads. Instead of moving the to-be-evicted object that

*Equal contribution.
†Corresponding author: Juncheng Yang, sieve@cachemon.com.

has been accessed to the head of queue, SIEVE keeps it in its original position. It should be noted that both SIEVE and FIFO-Reinsertion insert new objects at the head of the queue. The new algorithm is called SIEVE [1]: a simple and efficient turn-key cache eviction policy. We implemented SIEVE in five production cache libraries, which required fewer than 20 lines of change on average, underscoring the ease of real-world deployment.

Despite a simple design, SIEVE can quickly remove unpopular objects from the cache, achieving comparatively high efficiency compared to the state-of-the-art algorithms. By experimentally evaluating SIEVE on 1559 traces from five public and two proprietary datasets, we show that SIEVE achieves similar or higher efficiency than 9 state-of-the-art algorithms across traces. Compared to ARC [67], SIEVE reduces miss ratio by up to 63.2% with a mean of 1.5% [2]. As a comparison, ARC reduces LRU's miss ratio by up to 33.7% with a mean of 6.7%. Moreover, compared to the best of all algorithms, SIEVE has lower miss ratio on over 45% of the 1559 traces. In comparison, the runner-up algorithm, TwoQ, only outperforms other algorithms on 15% of the traces.

SIEVE's design eliminates the need for locking during cache hits, resulting in a boost in multi-threaded throughput. Our prototype implementation in Cachelib [37] demonstrates that SIEVE achieves twice the throughput of an optimized LRU implementation when operating with 16 threads.

Through empirical evidence and analysis, we illustrate that SIEVE's efficiency stems from sifting out unpopular objects over time. SIEVE transcends a single standalone algorithm — it can also be embedded within other cache policies to design more advanced algorithms. We demonstrate the idea by replacing the LRU components in ARC, TwoQ, and LeCaR with SIEVE. The SIEVE-supported algorithms significantly outperform the original LRU-based algorithms. For example, ARC-SIEVE reduces ARC's miss ratio by up to 62.5% with a mean reduction of 3.7% across the 1559 traces.

Our work makes the following contributions.

- We present the design for SIEVE: an easy, fast, and surprisingly efficient cache eviction algorithm for web caches.
- We demonstrate SIEVE's simplicity by implementing it in five production cache libraries by changing less than 20 lines of code on average.
- Using 1559 traces from 7 datasets, we show that SIEVE outperforms all state-of-the-art eviction algorithms on more than 45% of the traces.
- We illustrate SIEVE's scalability using our Cachelib-based implementation, which achieves 17% and 125% higher throughput than optimized LRU at 1 and 16 threads.
- We show how SIEVE, as a turn-key cache primitive, opens new opportunities for designing advanced eviction algorithms, e.g., replacing the LRU in ARC, TwoQ, and LeCaR with SIEVE.

[1] SIEVE sifts out unpopular objects from cache over time (§5).

[2] Due to a large number of traces, the mean miss ratio looks small.

2 Background and Related Work

2.1 Web caches

Web caches are essential components of modern Internet infrastructure, playing a crucial role in reducing data access latency and network bandwidth. Key-value caches, e.g., Memcached [5], Pelikan [8] and Cachelib [37], are widely used in modern web services such as Twitter [97] and Meta [20] to reduce service latency. CDN caches are deployed close to users to reduce data access latency and high WAN bandwidth cost [14, 91, 95, 101].

Cache metrics. Caches are measured along two primary axes: efficiency and throughput performance. Cache efficiency measures how well the cache can store and serve the required data. A cache miss occurs when the requested data is not found in the cache, requiring access to the backend storage to retrieve the data. Common cache efficiency metrics include (1) object miss ratio: the fraction of requests that are cache misses; (2) byte miss ratio: the fraction of bytes that are cache misses. A lower miss ratio indicates higher cache efficiency, as more requests are served directly from the cache, reducing backend load, access latency, and bandwidth costs.

Throughput performance, on the other hand, is as important as efficiency because the goal of a cache is to serve data quickly and help scale the application. Beyond throughput, scalability is also increasingly important [72, 98] as modern CPUs often surpass 100 cores. Scalability measures throughput growth with the number of threads accessing the cache. A more scalable cache can better harness the many cores in a modern CPU.

Access patterns. Web cache workloads typically follow Power-law (generalized Zipfian) distributions [20, 26, 27, 34, 49, 52, 55, 81, 82, 97], where a small subset of objects account for a large proportion of requests. In detail, the ith popular object has a relative frequency of $1/i^{\alpha}$, where α is a parameter that decides the skewness of the workload. Previous works find different α values from 0.6 to 0.8 [26], 0.56 [49], 0.71–0.76 [51], 0.55–0.9 [20], and 0.6–1.5 [97]. The reasons for the large range of α include (1) the different types of workloads, such as web proxy and in-memory key-value cache workloads; (2) the layer of the cache, noting that many proxy/CDN caches are secondary or tertiary cache layers [55]; and (3) the popularity of the service, such as the most popular objects receiving greater volume of requests in more popular (widely-used) web applications. Moreover, web caches often serve constantly growing datasets — new content and objects are created every second.

In contrast, the backend of enterprise storage caches or single-node caches, such as the page cache, often has a fixed size, not regularly observing new objects. Further, many storage cache workloads often have scan and loop patterns [75], in which a range of block addresses are sequentially requested in a short time. Such patterns are rare in web cache workloads according to our observation on 1559 traces from 7 datasets.

2.2 Cache eviction policies

The cache eviction algorithm, which decides which objects to store in the limited cache space, governs the performance and efficiency of a cache. The field of cache eviction algorithms has a rich literature [12, 17–19, 23, 29, 32, 35, 36, 39, 41, 44–46, 53, 58, 62, 63, 71, 74, 78, 83, 86, 88, 90, 102].

Increasing complexity. Most works on cache eviction algorithms focused on improving efficiency, such as LRU-k [70], TwoQ [60], SLRU [61], GDSF [29], EELRU [77], LRFU [39], LIRS [59], ARC [67], MQ [105], CAR [15], CLOCK-pro [58], TinyLFU [42, 43], LHD [16], LeCaR [84], LRB [79], CACHEUS [75], GLCache [93], and HALP [80]. Over the years, new cache eviction algorithms have gradually convoluted. Algorithms from the 1990s use two or more static LRU queues or use different recency metrics; algorithms from the 2000s employ size-adaptive LRU queues or use more complicated recency/frequency metrics, and algorithms from the 2010s and 2020s start to use machine learning to select eviction candidates. Each decade brought greater complexity to cache eviction algorithms. Nevertheless, as we show in §4, while the new algorithms excel on a few specific traces, they do not show a significant improvement (and some are even worse) compared to the traditional ones on a large number of workloads. The combination of limited improvement and high complexity explains why these algorithms have not been used in production systems.

The trouble with complexity. Multiple problems come with increasing complexity. First, complex cache eviction algorithms are difficult to debug due to their intricate logic. For example, we find two open-source cache simulators used in previous works have two different bugs in the LIRS [59] implementation. Second, complexity may affect efficiency in surprising ways. For example, previous work reports that both LIRS and ARC exhibit Belady's anomaly [50, 85]: miss ratio increases with the cache size for some workloads. It's worth noting that FIFO, although simple, also suffers from this anomaly. Third, complexity often negatively correlates with throughput performance. A more intricate algorithm performs more computation with potentially longer critical sections, reducing both throughput and scalability. Furthermore, many of these algorithms need to store more per-object metadata, which reduces the effective cache size that can be used for caching data. For example, the per-object metadata required by CACHEUS is 3.3× larger than that of LRU. Fourth, complex algorithms often have parameters that can be difficult to tune. For example, all the machine-learning-based algorithms include many parameters about learning. Although some algorithms do not have explicit parameters, e.g., LIRS, previous work shows that the implicit ghost queue size can impact the efficiency [85].

Trade-offs in using simple eviction algorithms. Besides works focusing on improving cache efficiency, several other works have improved cache throughput and scalability. For example, MemC3 [47] uses Cuckoo hashing and CLOCK eviction to improve Memcached's throughput and scalability; MICA [64] uses log-structured storage, data partitioning, and a lossy hash table to improve key-value cache throughput and scalability. Segcache [98] uses segment-structured storage with a FIFO-based eviction algorithm and leverages macro management to improve scalability. Frozenhot [72] improves cache scalability by freezing hot objects in the cache to avoid locking. However, it's crucial to note that while these approaches excel in throughput and scalability, they often compromise on cache efficiency due to the use of simpler, weaker eviction algorithms such as CLOCK[3] and FIFO.

2.3 Lazy promotion and quick demotion

Promotion and demotion are two cache internal operations used to maintain the logical ordering between objects[4]. Recent work [94] shows that "lazy promotion" and "quick demotion" are two important properties of efficient cache eviction algorithms.

Lazy promotion refers to the strategy of promoting cached objects only at eviction time. It aims to retain popular objects with minimal effort. An example of lazy promotion is adding reinsertion to FIFO. In contrast, FIFO has no promotion, and LRU performs eager promotion – moving objects to the head of the queue on every cache hit. Lazy promotion can improve (1) throughput due to less computation and (2) efficiency due to more information about an object at eviction.

Quick demotion removes most objects quickly after they are inserted. Many previous works have discussed this idea in the context of evicting pages from a scan [16, 60, 67, 70, 75, 77]. Recent work also shows that not only storage workloads but web cache workloads also benefit from quick demotion [94] because object popularity follows a power-law distribution, and many objects are unpopular.

To the best of our knowledge, our proposed cache eviction algorithm, which we call SIEVE, is the simplest one that effectively achieves both lazy promotion and quick demotion.

3 Design and Implementation

3.1 SIEVE Design

In this section, we introduce SIEVE, a cache eviction algorithm that achieves both simplicity and efficiency.

Data structure. SIEVE requires only one FIFO queue and one pointer called "hand". The queue maintains the insertion order between objects. Each object in the queue uses one bit to track the visited/non-visited status. The hand points to the next eviction candidate in the cache and moves from the tail to the head. Note that, unlike existing algorithms, e.g., LRU, FIFO, and CLOCK, in which the eviction candidate is always the tail object, the eviction candidate in SIEVE is an object

[3] CLOCK was recently shown to be more efficient than LRU [94].

[4] Note that the terms "promotion" and "demotion" are also commonly used in the context of cache hierarchy. In this case, promotion refers to the process of moving data to a faster device, while demotion involves moving the data to a slower device [65, 87].

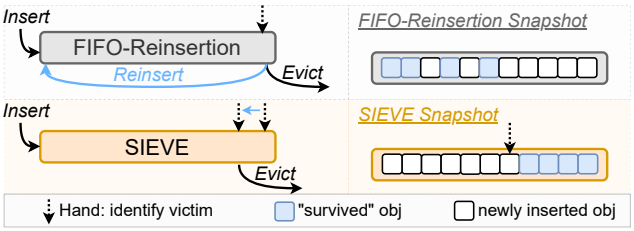


Figure 2: An illustration of SIEVE. Note that FIFO-Reinsertion and CLOCK are different implementations of the same algorithm. We use FIFO-Reinsertion in the illustration but will use CLOCK in the rest of the text because it is more commonly used and is shorter.

somewhere in the queue.

SIEVE operations. A cache hit in SIEVE changes the visited bit of the accessed object to 1. For a popular object whose visited bit is already 1, SIEVE does not need to perform any operation. During a cache miss, SIEVE examines the object pointed by the hand. If it has been visited, the visited bit is reset, and the hand moves to the next position (the retained object stays in the original position of the queue). It continues this process until it encounters an object with the visited bit being 0, and it evicts the object. After the eviction, the hand points to the next position (the previous object in the queue). While an evicted object is in the middle of the queue most of the time, a new object is always inserted into the head of the queue. In other words, the new objects and the retained objects are not mixed together.

At first glance, SIEVE is similar to CLOCK/Second Chance/FIFO-Reinsertion [5]. Each algorithm maintains a single queue in which each object is associated with a visited bit to track its access status. Visited objects are retained (also called "survived") during an eviction. Notably, new objects are inserted at the head of the queue in both SIEVE and FIFO-Reinsertion. However, the hand in SIEVE moves from the tail to the head over time, whereas the hand in FIFO-Reinsertion stays at the tail. The key difference is where a retained object is kept. SIEVE keeps it in the old position, while FIFO-Reinsertion inserts it at the head, together with newly inserted objects, as depicted in Fig. 2.

We detail the algorithm in Alg. 1. Line 1 checks whether there is a hit, and if so, then line 2 sets the visited bit to one. In the case of a cache miss (Line 3), Lines 5-12 identify the object to be evicted.

Lazy promotion and quick demotion. Despite a simple design, SIEVE effectively incorporates both lazy promotion and quick demotion. As described in §2.3, an object is only promoted at the eviction time in lazy promotion. SIEVE operates in a similar manner. However, rather than promoting the object to the head of the queue, SIEVE keeps the object at its original location. The "survived" objects are generally more popular than the evicted ones, thus, they are likely to be accessed again in the future. By gathering the "survived" objects, the hand in SIEVE can quickly move from the tail to the area near the head, where most objects are newly inserted. These newly inserted objects are quickly examined by the hand of SIEVE after they are admitted into the cache, thus achieving quick demotion. This eviction mechanism makes SIEVE achieve both lazy promotion and quick demotion without adding too much overhead.

The key ingredient of SIEVE is the moving hand, which functions like an adaptive filter that removes unpopular objects from the cache. This mechanism enables SIEVE to strike a balance between finding new popular objects and keeping old popular objects. We discuss more in §5.

[5]Note that Second Chance, CLOCK, and FIFO-Reinsertion are different implementations of the same eviction algorithm.

Algorithm 1 SIEVE

```
Input: The request x, doubly-linked queue T, cache size C, hand p
1:  if x is in T then                                   ▷ Cache Hit
2:      x.visited ← 1
3:  else                                                ▷ Cache Miss
4:      if |T| = C then                                 ▷ Cache Full
5:          o ← p
6:          if o is NULL then
7:              o ← tail of T
8:          while o.visited = 1 do
9:              o.visited ← 0
10:             o ← o.prev
11:             if o is NULL then
12:                 o ← tail of T
13:         p ← o.prev
14:         Discard o in T                              ▷ Eviction
15:     Insert x in the head of T.
16:     x.visited ← 0                                   ▷ Insertion
```

3.2 Implementation

Simulation. We implemented SIEVE in libCacheSim [92]. LibCacheSim is a high-performance cache simulator designed for running cache simulations and analyzing cache traces. It supports many state-of-the-art eviction algorithms, including ARC [67], LIRS [59], CACHEUS [75], LeCaR [84], TwoQ [60], LHD [16], Hyperbolic [24], FIFO-Reinsertion/CLOCK [35], B-LRU (Bloom Filter LRU), LRU, LFU, and FIFO. For all state-of-the-art algorithms, we used the configurations from the original papers.

Prototype. Because of SIEVE's simplicity, it can be implemented on top of a FIFO, LRU, or CLOCK cache in just a few lines by adding, initializing, and tracking the "hand" pointer. The object pointed to by the hand is either evicted or retained, depending on whether it has been accessed.

We implemented SIEVE caching in five different open-source cache libraries: Cachelib [20], groupcache [25], mnemonist [6], lru-dict [3], and lru-rs [4]. These represent the most popular cache libraries of five different programming languages: C++, Golang, JavaScript, Python, and Rust. All five of these production cache libraries implement LRU as the eviction algorithm of choice. Aside from mnemonist, which uses arrays, they all use doubly-linked-list-based implementations of LRU. Adapting these LRU implementations to use SIEVE was a low effort, as mentioned earlier.

Table 1: Datasets used in this work. CDN 1 and 2 are proprietary, and all others are publicly available.

trace collections	approx time	# traces	cache type	# request (million)	# object (million)
CDN 1	2021	1273	object	37,460	2,652
CDN 2	2018	219	object	3,728	298
Tencent Photo [103]	2018	2	object	5,650	1,038
Wiki CDN [1]	2019	3	object	2,863	56
Twitter KV [97]	2020	54	KV	195,441	10,650
Meta KV [10]	2022	5	KV	1,644	82
Meta CDN [10]	2023	3	object	231	76

4 Evaluation

In this section, we evaluate SIEVE to answer the following questions.

- Does SIEVE have higher efficiency than state-of-the-art cache eviction algorithms?
- Can SIEVE improve a cache's throughput and scalability?
- Is SIEVE simpler than other algorithms?

4.1 Experimental setup

Workloads. Our experiments use open-source traces from Twitter [97], Meta [10], Wikimedia [1], TencentPhoto [103, 104], and two proprietary CDN datasets. We list the dataset information in Table 1. It consists of 1559 traces that together contain 247,017 million requests to 14,852 million objects. Notably, our research is centered around web traces. We replayed the traces in the simulator and the prototypes as a closed system with instant on-demand fill.

Metrics. Miss ratio serves as a key performance indicator when evaluating the efficiency of a cache system. However, when analyzing different traces (even within the same dataset), the miss ratios can vary significantly, making direct comparisons and visualizations infeasible, as shown in Fig. 3. Therefore, we calculate the miss ratio reduction relative to a baseline method (FIFO in this work): $\frac{mr_{FIFO}-mr_{algo}}{mr_{FIFO}}$ where mr stands for miss ratio. If an algorithm's miss ratio is higher than FIFO, we use $\frac{mr_{FIFO}-mr_{algo}}{mr_{algo}}$. This metric has a range between -1 and 1.

We measure throughput in millions of operations per second (Mops) to quantify a cache's performance. To evaluate scalability, we vary the number of trace replay threads from 1 to 16 and measure the throughput.

Testbed. Our evaluations were conducted on Cloudlab [40] and focused on two key aspects: simulation-based efficiency and prototype-based throughput and simplicity.

We used libCacheSim [92], a high-performance cache simulator, to evaluate the efficiency of different cache algorithms. These simulations ran on various node types at either the Clemson or Utah sites, subject to availability.

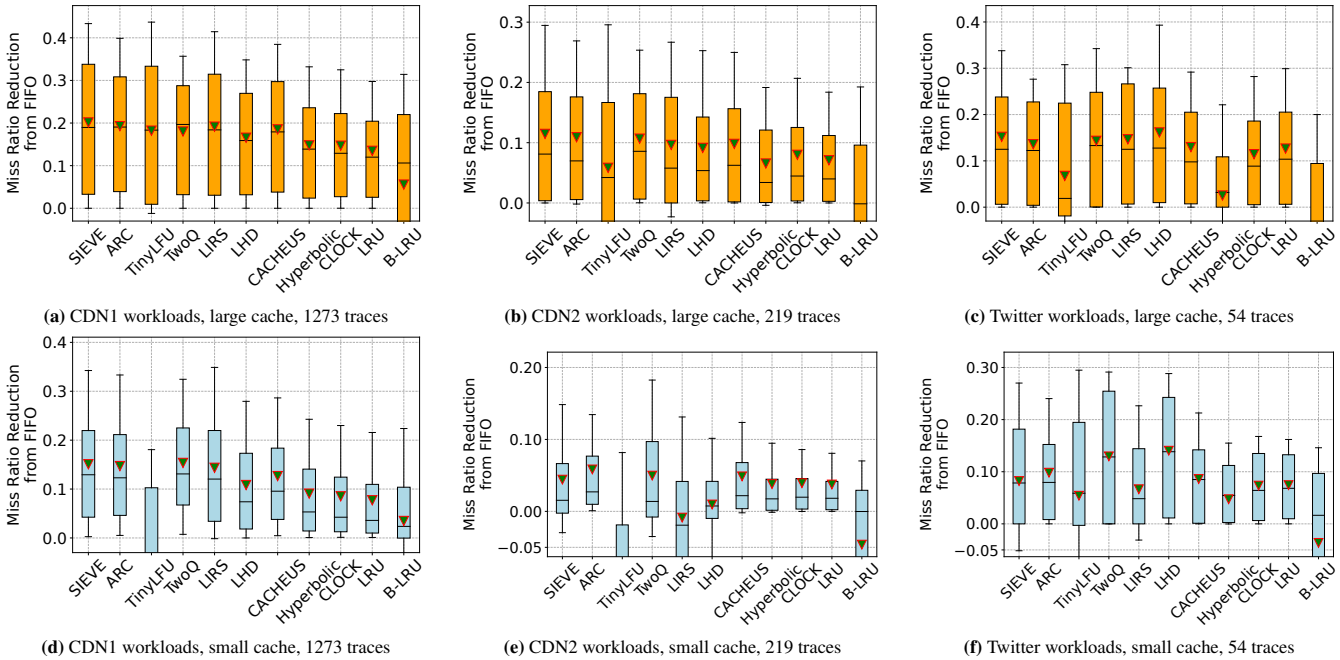
We evaluate the throughput and simplicity using prototypes, as described in §3.2. The prototype evaluations were conducted on the c6420 node from the Clemson site. This node type has a dual-socket Intel Gold 6142 running at 2.6 GHz and is equipped with 384 GB DDR4 DRAM. We turned off turbo boost and pinned threads to CPU cores in one NUMA node in our evaluations. We validated the efficiency results from the simulator and prototype using 60 randomly selected traces and found the same conclusion.

4.2 Efficiency results

In this section, we compare the efficiency of different eviction algorithms. Because many caches today use slab-based space management, in which evictions happen on objects of similar sizes, we do not consider object size in this section. The cache sizes are determined as a percentage of the number of objects in a trace. We evaluate eight cache sizes using 1559 traces from the 7 datasets and present two representative cache sizes at 0.1% and 10% of the trace footprint (the number of unique objects in the trace).

Three large datasets CDN1, CDN2 and Twitter. Fig. 3 shows the miss ratio reduction (from FIFO) of different algorithms across traces. The whiskers on the boxplots are defined using p10 and p90, allowing us to disregard extreme data and concentrate on the typical cases. At the large cache size, SIEVE demonstrates the most significant reductions across nearly all percentiles. For example, SIEVE reduces FIFO's miss ratio by more than 42% on 10% of the traces (top whisker) with a mean of 21% on the CDN1 dataset using the large cache size (Fig. 3a). As a comparison, all other algorithms have smaller reductions on this dataset. For example, CLOCK/FIFO-Reinsertion, which is conceptually similar to SIEVE, can only reduce FIFO's miss ratio by 15% on average. Compared to advanced algorithms, e.g., ARC, SIEVE reduces ARC miss ratio by up to 63.2% with a mean of 1.5%. We remark that a 1.5% mean miss ratio reduction on the huge number of traces is significant. For example, ARC only reduces LRU's miss ratio by 6.3% on average (not shown). A similar observation can be made on the CDN2 dataset. Although LHD is the best algorithm on the Twitter dataset, SIEVE scores second and outperforms most other state-of-the-art algorithms.

When the cache is very small, TwoQ and LHD sometimes outperform SIEVE. This is because TwoQ and LHD can quickly remove newly-inserted low-value objects similar to SIEVE. The primary reason for SIEVE's relatively poor performance is that new objects cannot demonstrate their popularity before being evicted when the cache size is very small. A similar problem also happens with ARC and LIRS. ARC's adaptive algorithm sometimes shrinks the recency queue to very small and yields a high miss ratio. LIRS, which uses a 1% queue for new objects, suffers the most when the cache size is small, as we see its miss ratio on some traces higher than FIFO. In contrast, TwoQ does not suffer from the small cache sizes because it reserves a fixed 25% of the cache space for new objects, preventing overly aggressive demotion. However, we remark that the production miss ratios reported in previous works [13, 55, 97, 98] are close to the miss ratios we observe at the large cache size.



(a) CDN1 workloads, large cache, 1273 traces

(b) CDN2 workloads, large cache, 219 traces

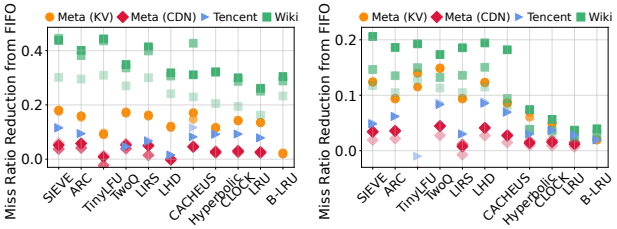
(c) Twitter workloads, large cache, 54 traces

(d) CDN1 workloads, small cache, 1273 traces

(e) CDN2 workloads, small cache, 219 traces

(f) Twitter workloads, small cache, 54 traces

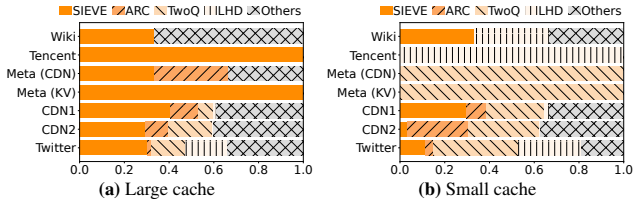
Figure 3: The box shows the miss ratio reduction from FIFO over all traces in the dataset. The box shows P25 and P75, the whiskers show P10 and P90, and the triangle shows the mean. The large cache uses 10% of the trace footprint, and the small cache uses 0.1% of the trace footprint. SIEVE achieves similar or better miss ratio reduction compared to state-of-the-art algorithms.



(a) Large cache

(b) Small cache

Figure 4: Miss ratio reduction on Meta (KV + CDN), Wiki CDN, and Tencent Photo CDN datasets. The different opacity of the same color indicates multiple traces from the dataset. Some negative results are not shown.



(a) Large cache

(b) Small cache

Figure 5: Best-performing algorithms on each dataset. Table 1 shows the number of traces per dataset.

The secret behind SIEVE's efficiency is the ability to quickly remove newly-inserted unpopular objects (quick demotion), the ability to sift out old unpopular objects, and the balance between new and old objects. We discuss more in §5.

Four small datasets: Meta KV, Meta CDN, Wiki, and TencentPhoto. Because each dataset contains fewer than ten traces, we use scatter plots to compare the algorithms. Fig. 4 demonstrates that SIEVE outperforms all other algorithms on all four datasets at the large cache size. When the cache size is small, the observation is similar to that made in Fig. 3. SIEVE is the best algorithm on the Wiki dataset. TwoQ and LHD are the best on Meta and TencentPhoto datasets. Although not the best, SIEVE remains highly competitive.

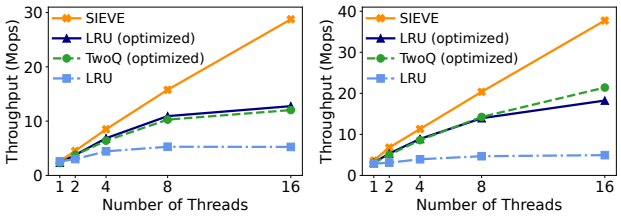
Best-performing algorithm per dataset. We have demonstrated that SIEVE provides larger miss ratio reductions across traces than state-of-the-art algorithms. For a more quantitative comparison, Fig. 5 shows the fraction of traces each algorithm performs the best.

With a large cache size, SIEVE outperforms all other algorithms on the Tencent Photo, Wiki, and Meta KV datasets. On the CDN1 and CDN2 datasets, SIEVE is the best algorithm on 48% and 38% of the 1273 and 219 traces. On the Twitter dataset, although SIEVE is the best on only 30% of the traces, it is important to note that no other algorithms are the best on more than 18% of the traces. When using the small cache size, SIEVE, TwoQ is the best algorithm winning on the two Meta datasets. On the other datasets, SIEVE and LHD have similar shares being the best-performing algorithms. The reason for the observation is similar to that previously explained.

4.3 Throughput performance

Besides efficiency, throughput is the other important metric for caching systems. Although we have implemented SIEVE in five different libraries, we focus on Cachelib's results. Because all other libraries implement strict LRU and do not consider object sizes, evaluations yield the same miss ratio as our simulation. Moreover, strict LRU is not scalable, as we show next.

Fig. 6 shows how throughput grows with the number of



(a) Meta KV trace **(b)** Twitter trace

Figure 6: Throughput scaling with CPU cores on two KV-cache workloads.

Table 2: Lines of code requires modification to add SIEVE to a production cache library.

Cache library	Language	Lines
groupcache [25]	Golang	21
mnemonist [6]	Javascript	12
lru-rs [4]	Rust	16
lru-dict [3]	Python + C	21

trace replay threads using two production traces from Meta and Twitter. To better emulate real-world deployments in which the working set size (dataset size) grows with the hardware specs (#cores and DRAM sizes), we scale the cache size and working set size together with the number of threads. To scale the working set size, each thread plays the same trace with the object id transformed into a new space. For example, the benchmark sends 4× more requests to 4× larger cache size at 4 threads compared to the single-thread experiment. We set the cache size to be $4 \times n_{thread}$ GB for both traces, which gives miss ratios of 7% (Meta) and 2% (Twitter). We remark that the miss ratio is close to previous reports [13, 98].

The LRU and TwoQ in Cachelib use extensive optimizations to improve the scalability. For example, objects that were promoted to the head of the queue in the last 60 seconds are not promoted again, which reduces lock contention without compromising the miss ratio. Cachelib further adds a lock combining technique to elide expensive coherence and synchronization operations to boost throughput [38]. As a result of the optimizations, both LRU and TwoQ show impressive scalability results compared to the unoptimized LRU: the throughput is 6× higher at 16 threads than using a single thread on the Twitter trace. As a comparison, unoptimized LRU's throughput plateaus at 4 threads.

Compared to these LRU-based algorithms, SIEVE does not require "promotion" at each cache hit. Therefore, it is faster and more scalable. At a single thread, SIEVE is 16% (17%) faster than the optimized LRU (TwoQ) and on both traces. At 16 threads, SIEVE shows more than 2× higher throughput than the optimized LRU and TwoQ on the Meta trace.

4.4 Simplicity

Prototype implementations. SIEVE not only achieves better efficiency, higher throughput, and better scalability, but it is also very simple. We chose the most popular cache libraries/systems from five different languages: C++, Go, JavaScript, Python, and Rust, and replaced the LRU with SIEVE.

Table 3: Lines of code (excluding comments and empty lines) and per-object metadata size required to implement each algorithm in our simulator. We assume that frequency counter and timestamps use 4 bytes and pointers use 8 bytes.

Algorithm	cache hit	eviction	insertion	metadata size
FIFO	1	4	3	16B
LRU	5	4	3	16B
ARC	64	108	20	17B
LIRS	96	120	64	17B
LHD	192	81	64	13B
LeCaR	72	76	20	40B
CACHEUS	168	140	150	54B
TwoQ	28	16	8	17B
Hyberbolic	4	20	4	16B
CLOCK	4	9	3	17B
SIEVE	4	9	3	17B

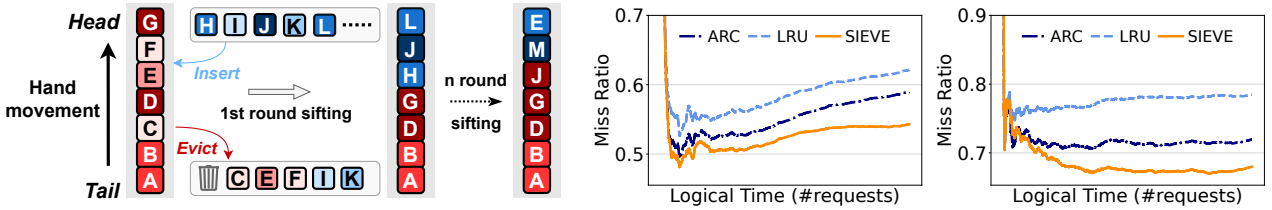
Although different libraries/systems have different implementations of LRU, e.g., most use doubly-linked-list, and some use arrays, we find that implementing SIEVE is very easy. Table 2 shows the number of lines (not including the tests) needed to replace LRU — all implementations require no more than 21 lines of code changes [6].

Advanced algorithms in simulator. Most of the complex algorithms we evaluated in §4.2 are not implemented in production systems. Therefore, we compare the lines of code needed to implement cache hit, insert, and evict in our simulator. Although we implemented our own linked list and hash table data structures in C for our simulator, we do not include the code lines related to list and hash table operations, i.e., appending to the list head or inserting to the hash table requires one line.

Table 3 shows that FIFO requires the fewest number of lines to implement. On top of FIFO, implementing LRU adds a few lines to promote an object upon cache hits. CLOCK and SIEVE require close to 10 lines to implement the eviction function because both need to find the first object that has not been visited. However, we remark that SIEVE is simpler than LRU and CLOCK because SIEVE does not require moving objects to the head of the queue in either hit or miss (evict). Besides these, all other algorithms require one to two orders more lines of code to implement the three functions.

Per-object metadata. In addition to the implementation complexity, we also quantified the per-object metadata needed to implement each algorithm. FIFO does not require any metadata when implemented using a ring buffer. However, such an implementation does not support overwrite or delete. So common FIFO implementation also uses a doubly-linked list with 16 bytes of per-object metadata similar to LRU. CLOCK and SIEVE are similar, both requiring 1-bit to track object access status. When implemented using a doubly linked list,

[6] While most LRU implementations are straightforward to adapt for SIEVE, CacheLib is an exception. Cachelib is highly optimized for LRU-based algorithms. Many optimizations are not needed for SIEVE, making it impractical to quantify code modifications for integration with SIEVE. Therefore, it is not included in Table 2.



(a) Density of colors indicates inherent object popularity (blue: newly inserted objects; red: old objects in each round), and the letters represent object IDs. The first queue captures the state at the start of the first round, and the second queue captures the state at the end of the first round.

(b) Trace 1, full trace (one week)

(c) Trace 2, first two days of a week-long trace

Figure 7: Left: illustration of the sifting process. Right: Miss ratio over time for two traces. The gaps between SIEVE's miss ratio and others enlarge over time.

they use 17 bytes per-object metadata. Compared to SIEVE, advanced algorithms often require more per-object metadata. Many key-value cache workloads have objects as small as 10s of bytes [66, 97], and large metadata wastes the precious cache space.

ZERO parameter. *Besides being easy to implement and having less metadata,* SIEVE *also has no parameters*. Except for FIFO, LRU, CLOCK, and Hyperbolic, all other algorithms have explicit or implicit parameters, e.g., the sizes of queues in LIRS, the learning rate in LeCaR and CACHEUS, and the decay rate and age granularity in LHD. Note that although ARC has no explicit parameters, its adaptive algorithm uses implicit parameters in deciding when and how much space to move between the queues. As a comparison, SIEVE has no parameter and requires no tuning.

5 Distilling SIEVE's Effectiveness

Our empirical evaluation shows that SIEVE is simultaneously simple, fast, scalable, and efficient. In a well-trodden field like cache eviction, SIEVE's competitive performance was a genuine surprise to us as well. We next report our analysis that seeks to understand the secrets behind its efficiency.

5.1 Visualizing the sifting process

The workhorse of SIEVE is the "hand" that functions as a sieve: it sifts through the cache to filter out unpopular objects and retain the popular ones. We illustrate this process in Fig. 7a, where each column (queue) represents a snapshot of the cached objects over time from left to right. As the hand moves from the tail (the oldest object) to the head (the newest object), objects that have not been visited are evicted – the same sweeping mechanism that underlies CLOCK [30, 35]. For example, after the first round of sifting, objects at least as popular as A remain in the cache while others are evicted. The newly admitted objects are placed at the head of the queue — much like the CLOCK policy, but a departure from CLOCK, which does in-place replacements to emulate LRU. During the subsequent rounds of sifting, if objects that survived previous rounds remain popular, they will stay in the cache. In such a case, since most old objects are not evicted, the eviction hand quickly moves past the old popular objects to the queue positions close to the head. This allows newly inserted objects to be quickly assessed and evicted, putting greater eviction pressure on unpopular items (such as "one-hit wonders") than LRU and its variations [67]. As previous work has shown [16, 55, 94], quick demotion is crucial for achieving high cache efficiency.

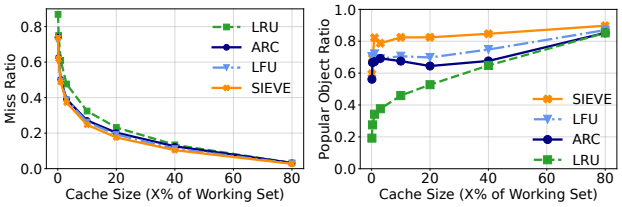
Fig. 7b and Fig. 7c show the cumulative miss ratio over time of different algorithms on two representative production traces. After the cache is warmed up, the miss ratio gaps between SIEVE and other algorithms widen over time, supporting the interpretation that SIEVE indeed sifts out unpopular objects and retains popular ones. A similar observation can be seen in Fig. 10a.

5.2 Analyzing the sifting process

We now analyze the popularity retention mechanism in SIEVE. To clarify the exposition, suppose the SIEVE cache can fit C equally sized objects. Since SIEVE always inserts new objects at the head, and objects that are retained remain in their original positions within the queue, the algorithm implicitly partitions the cache between new and old objects. This partition is dynamic, allowing SIEVE to strike a balance between exploration (finding new popular objects) and exploitation (enjoying hits on old popular objects).

SIEVE performs sifting by moving the hand from the tail to the head, evicting unpopular objects along the way, which we call one round of sifting. We use r to denote the number of rounds. We first enumerate the queue positions p from the tail ($p = 0$) to the head ($p = C - 1$). We then further denote that an object at position p in round r is *examined* (during eviction) or *inserted* at time T_p^r. Note that T effectively defines a logical timer for the examined objects: whenever an object is examined, T increases by 1, regardless of whether the examined object is evicted or retained. In addition, T changes *once* each round for an old object (retained from previous rounds).

For an old object x at position p, we define the "inter-examination time" $I_e(p^r) = T_p^r - T_{p'}^{r-1}$ where p' was the position of x in round $r-1$. Clearly, $p' \geq p$. For a new object inserted in the current round, the inter-examination time is defined as the time between its examination and insertion. We further define an old object x's "inter-arrival time" $I_a(x^r)$ as the time, measured again in the number of objects examined, between the first request to the x in round r and the last re-



(a) Miss ratio over size **(b)** Popular object ratio over size

Figure 8: Miss ratio and popular object ratio on a Zipfian dataset ($\alpha = 1.0$).

quest to x in round $r-1$. For a new object, the inter-arrival time is the time between its insertion and the second request. If an old object is not requested in the last round or a new object does not have a second request, its inter-arrival time is infinite.

In round r, consider two consecutive retained objects x_1 and x_2 at position p_1 and $p_2 = p_1 + 1$. The inter-examination times are $I_e(p_1^r) = T_{p_1}^r - T_{p_1'}^{r-1}$ and $I_e(p_2^r) = T_{p_2}^r - T_{p_2'}^{r-1}$, respectively. The transition yields two invariants:

$$\begin{aligned} T_{p_2}^r - T_{p_1}^r &= 1 \\ T_{p_2'}^{r-1} - T_{p_1'}^{r-1} &\geq 1 \end{aligned}$$

The first equation follows from x_1 and x_2 being consecutively retained objects; the second inequality expresses that other evictions may have taken place between x_1 and x_2 in the previous round. Together, these imply that $I_e(p_1^r) \geq I_e(p_2^r)$. The result generalizes further: for any two retained old objects in the queue, the object closer to the head has a smaller inter-examination time.

Moreover, if an object is retained, its inter-arrival time must be no greater than its inter-examination time. Therefore, for any retained object x at position p_x, its inter-arrival time $I_a(x^r)$ must be smaller than the tail object's inter-examination time:

$$I_a(x^r) \leq I_e(p_x^r) \leq I_e(p_0^r) \tag{1}$$

Using the commonly assumed independent reference model [31, 48, 56, 57] with a Poisson arrival, we can expect any retained object to be more popular than some dynamic threshold set by the tail object's inter-examination time $I_e(p_0^r)$. Since evicting an object keeps the hand pointer at its original position (relative to the tail), the more objects are evicted during a round, the longer the inter-examination time. As a result, SIEVE effectively adapts the popularity threshold so that more objects are retained in the next round.

Following our sifting process metaphor, the mesh size in SIEVE is determined by the tail object's inter-examination time $I_e(p_0^r)$, which is dynamically adjusted based on object popularity change. If too few objects are retained in one round (mesh size too small), then we will have an increased tail inter-examination time $I_e(p_0^r)$ (a larger mesh size) in the next round.

5.3 Deeper study with synthetic workloads

Production trace workloads are often too complex and dynamic to analyze. One consistent finding from past workload characterization work, however, is that object popularity in web cache workloads invariably follows a heavy-tailed power-law (generalized Zipfian) distribution [27, 97]. Therefore, we opted for synthetic power-law workloads for our study. It allows us to easily modify workload features to better understand their impact on performance. Using these synthetic workloads, we further scrutinize SIEVE's effectiveness.

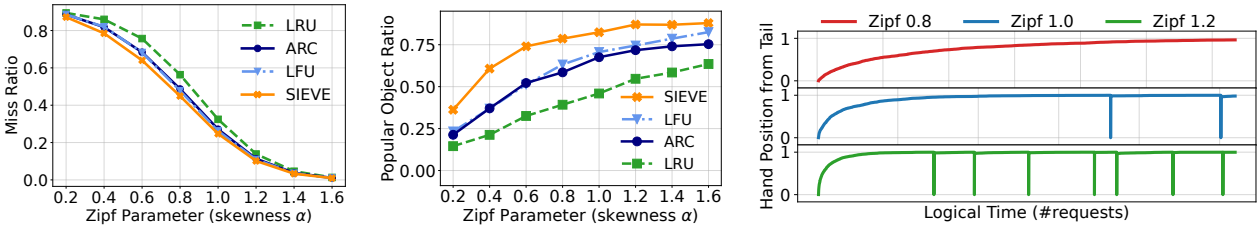
Miss ratio over size. Fig. 8a displays the miss ratio of LRU, LFU, ARC, and SIEVE at different cache sizes. Notably, LFU, ARC, and SIEVE all exhibit lower miss ratios than LRU, demonstrating their efficiency. Despite being considered optimal for synthetic power-law workloads, LFU performs similarly to ARC and is visibly worse than SIEVE. This is because objects with medium popularity, such as objects with ranks around the cache size C, are only requested once before their eviction. LFU cannot distinguish the true popularity of these objects and misses out on an opportunity for better performance. As a comparison, both ARC and SIEVE can quickly remove new and potentially unpopular objects, which allows cached objects to enjoy more time in the cache to demonstrate their popularity. Between the two algorithms, SIEVE further extends the tenure of these objects in the cache because when the hand sweeps through the newly inserted objects, the objects closer to the head must have strictly shorter inter-arrival times (expected to be more popular) to survive.

Popular object ratio over size. To capture how different algorithms manage popular objects, we define a metric called "popular object ratio". Under the assumption of a static and known popularity distribution, the optimal caching policy retains the most popular content within the cache at all times. Given a cache size C and a workload following a power-law distribution, the popular objects are the C most frequent objects in the workload, denoted by H. The popular ratio of objects in the cache at time t is calculated by $I_t = \frac{|H \cap A_t|}{C}$ where A_t denotes the cache contents at time t.

Fig. 8b shows the popular object ratio at different cache sizes. LRU evicts objects based on recency, which only weakly correlates with popularity. In this scenario, LRU stores the least number of popular objects. LFU stores slightly more "popular objects" than ARC. SIEVE, however, successfully filters out unpopular objects from the cache.

Varying the popularity skew. Fig. 8 shows a distribution with Zipfian skewness $\alpha = 1$. We further studied how different concentration of popularity affects SIEVE's effectiveness. Due to space restrictions, we focus on results with large cache sizes for the remainder of this subsection. Results using the small cache size are either similar or do not reveal interesting patterns.

Fig. 9a and Fig. 9b demonstrate the impact of varying skew on miss and popular object ratios. As skew increases, making popular objects more prominent, it becomes easier to identify



(a) Miss ratio **(b)** Popular object ratio **(c)** Hand movement in SIEVE

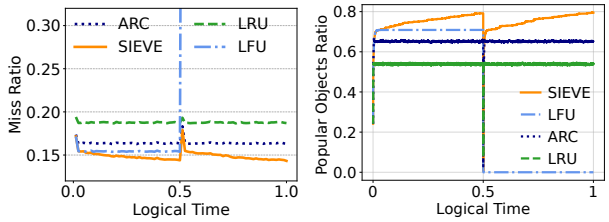
Figure 9: Left two: miss ratio and popular object ratio on Zipfian workloads with different α. Right: hand position in the cache over time in Zipfian workloads.

and cache the popular objects, increasing the popular object ratio and reducing the miss ratio for all tested algorithms. Among ARC, LFU, and SIEVE, we observe that SIEVE always shows a higher popular ratio with a lower miss ratio across skewness, indicating the efficiency of SIEVE is not limited to very skewed workloads.

Fig. 9c illustrates the hand position in the SIEVE cache over time, advancing towards the head with each retained object and pausing during evictions. Therefore, the more objects are retained, the faster the movement. We observe that the hand moves more slowly in the first round than in the later rounds because that is when many unpopular objects are evicted. In subsequent rounds, the hand lingers at positions close to the head for most of the time because SIEVE keeps a new object at position p only if it is more popular (shorter inter-arrival time) than the object at position $p-1$. In other words, SIEVE performs quick demotion [87].

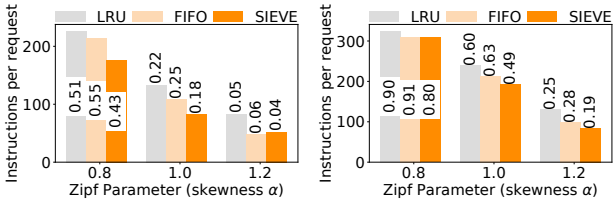
In more skewed workloads, the hand moves quickly due to early arrival and higher request volumes for popular objects, allowing SIEVE to cache most popular objects by the end of the first round. Consequently, the hand rapidly transitions from tail to head with fewer evictions and spends less time near the head, as new objects are more likely to be retained, hastening its progress. Nevertheless, the time of each round varies depending on the frequency of encountering potentially popular objects, highlighting SIEVE's adaptability to workload shifts. When new popular objects appear, the hand accelerates, replacing existing cached objects with the newcomers by giving less time to set their visited bit.

SIEVE is adaptive. To visualize SIEVE's adaptivity via the sifting process, we created a new workload by joining two Zipfian ($\alpha = 1.0$) workloads that request different populations of objects. Fig. 10 shows the interval miss ratio (per 100,000 requests) over time on this conjoined workload. The changeover happens at the 50% midway time mark. We observe that the interval miss ratio of LFU skyrockets to nearly 100% (beyond figure bounds) since new objects cannot replace the old objects. Relative to LRU and ARC, SIEVE's miss ratio spike is larger because it takes time for the hand to move back to the tail before it can evict old objects. However, SIEVE's spike is invisible when the cache size is small (not shown). With respect to the interval miss ratio spike, we observe the popular object ratio of all algorithms (the curves overlap) dropping to



(a) Interval miss ratio **(b)** Popular object ratio over time

Figure 10: Interval miss ratio and popular object ratio over time on a workload constructed by connecting two different Zipfian workloads ($\alpha = 1$).



(a) Large cache **(b)** Small cache

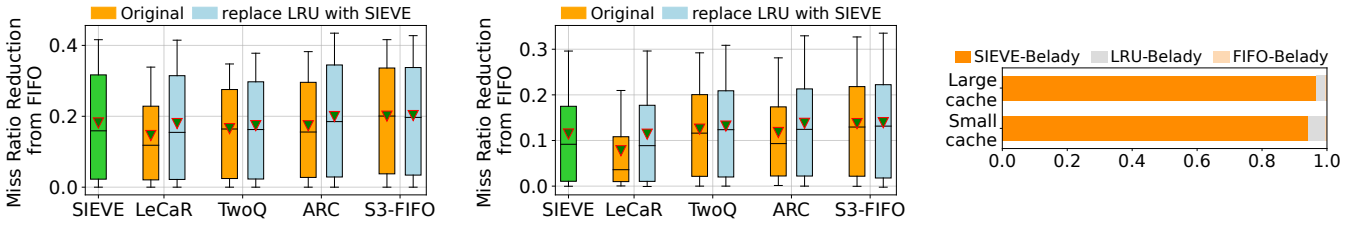
Figure 11: Average number of instructions per request when running LRU, FIFO, and SIEVE caches. The top number denotes the miss ratio.

0 when the workload changes at the midway point. Whereas LFU never recovers from the drop, the popular object ratios in all other algorithms quickly recover to large proportions. Finally, the figures corroborate our interpretation of the sifting process: SIEVE's miss ratio drops over time, while the fraction of popular objects increases over time.

6 SIEVE as a Turn-key Cache Primitive

6.1 Cache primitives

Beyond being a cache eviction algorithm, SIEVE can serve as a cache primitive for designing more advanced eviction policies. To study the range of such policies, we categorize existing cache eviction algorithm designs into four main approaches. **(1)** We can design simple and easy-to-understand eviction algorithms, such as FIFO queues, LRU queues, LFU queues, and Random eviction. We call these simple algorithms *cache primitives*. SIEVE falls under this category. **(2)** We can improve the cache primitives. For example, FIFO-Reinsertion is designed by adding reinsertion to FIFO; LRU-K [70] is designed by changing the recency metric in LRU. **(3)** We can compose multiple cache primitives with objects moved between them. For example, ARC, SLRU, and MQ use multiple LRU queues. **(4)** We can run multiple cache primitives and



(a) Large cache

(b) Small cache

(c) Best-performing algorithms across traces.

Figure 12: Impact of replacing LRU with SIEVE in advanced algorithms (**a**,**b**). The potential of FIFO, LRU, and SIEVE when endowed with foresight (**c**).

craft a decision-maker to select eviction candidates suggested by the primitives. For example, LeCaR [84] uses reinforcement learning to choose between the eviction candidates from LRU and LFU; HALP [80] uses machine learning (MLP) to choose one object from the eight objects at the LRU tail.

Having an efficient cache primitive not only provides an effective and simple eviction algorithm but also enables other approaches to design more efficient algorithms. The ideal cache primitive is simultaneously (1) simple, (2) efficient, and (3) fast — in terms of high throughput. For example, FIFO and LRU meet these requirements and are frequently used to construct more advanced algorithms. However, they are less efficient than complex algorithms.

While we have shown that SIEVE is simple, efficient, and fast in §4, to further understand SIEVE as a cache primitive, we compare the number of instructions needed to run FIFO, LRU, and SIEVE caches. We remark that the number of instructions may not necessarily correlate with latency or throughput but rather a rough metric of CPU resource usage. We used `perf stat` to measure the number of instructions for serving power-law workloads (100 million requests, 1 million objects) in our simulator. We then deduct the simulator overhead by measuring a no-op cache, which performs nothing on cache hits and misses.

Fig. 11 shows that SIEVE generally executes fewer instructions per request than FIFO and LRU, a difference accentuated in skewed workloads and larger cache sizes. Compared to LRU, SIEVE requires fewer instructions since SIEVE needs only to check and possibly update a Boolean field on cache hits, which is much simpler than moving an object to the head of the queue. Besides LRU, SIEVE also requires fewer instructions than FIFO because of the difference in miss ratios. Because SIEVE has a lower miss ratio than FIFO, fewer objects need to be inserted due to cache misses, leading to fewer instructions per request on average. The only exception is when SIEVE and FIFO have similar miss ratios, in which case, FIFO executes fewer instructions than SIEVE. Overall, SIEVE requires up to 40% and 24% fewer instructions than LRU and FIFO, respectively.

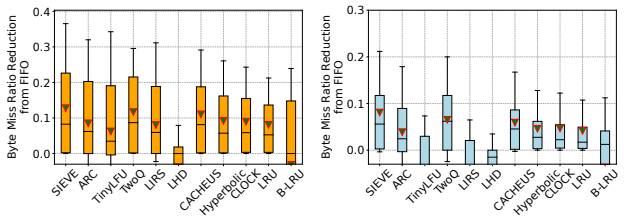
6.2 Turn-key cache eviction with SIEVE

As a cache primitive, SIEVE can facilitate the design of more advanced eviction algorithms. To understand the benefits of using a better cache primitive, we replaced the LRU in LeCaR, TwoQ, and ARC with SIEVE. Note that for ARC, we only replace the LRU for frequent objects.

We evaluate these algorithms on all traces and show the miss ratio reduction(from FIFO) in Fig. 12a and Fig. 12b. Compared to SIEVE, LeCaR has much lower efficiency; however, when replacing the LRU in LeCaR with SIEVE, it significantly reduces LeCaR's miss ratio by 4.5% on average. TwoQ and ARC achieve efficiency close to SIEVE; however, when replacing the LRU with SIEVE, the efficiency of both algorithms gets boosted. For example, ARC-SIEVE achieves the best efficiency among all compared algorithms at both small and large cache sizes. It reduces ARC's miss ratio by 3.7% on average and up to 62.5% on the large cache size (recall that ARC reduces LRU's miss ratio by 6.3% on average). ARC-SIEVE also reduces SIEVE's miss ratio by an average of 2.4% and up to 40.6%.

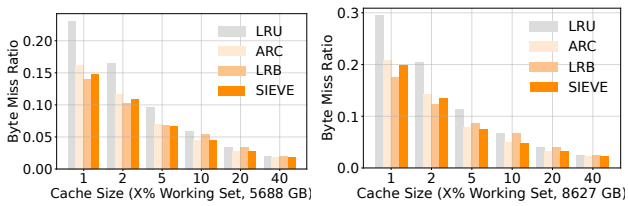
To understand the potential in suggesting eviction candidates, we evaluated the efficiency of FIFO, LRU, and SIEVE, granting them access to future request data. Each eviction candidate is either evicted or reinserted, depending on whether the object will be requested soon. We assume that an object will be requested soon if the logical time (number of requests) till the object's next access is no more than $\frac{C}{mr}$, where C is the cache size and mr is the miss ratio. This mimics the case that we have a perfect decision-maker choosing between the eviction candidates suggested by multiple simple eviction algorithms. Fig. 12c shows that when supplied with this additional information, SIEVE achieves the lowest miss ratio on 97% and 94% of the 1559 traces at the large and small cache size, respectively.

These results highlight the potential of SIEVE as a powerful cache primitive for designing advanced cache eviction algorithms. Leveraging lazy promotion and quick demotion, SIEVE not only performs well on its own but also bolsters the performance of more complex algorithms.



(a) Large cache **(b)** Small cache

Figure 13: Byte miss ratio across all CDN traces.



(a) Wiki2018 trace **(b)** Wiki2019 trace

Figure 14: Byte miss ratios at different cache sizes on two Wiki CDN traces used in LRB evaluation.

7 Discussion

7.1 Byte miss ratio

To gauge SIEVE's efficiency in reducing network bandwidth usage in CDNs, we analyzed its byte miss ratio by considering object sizes. We chose the cache size at 10% and 0.1% of the trace footprint in bytes. Fig. 13a and Fig. 13b show that SIEVE presents larger byte miss ratio reductions at *ALL* percentiles than state-of-the-art algorithms at both cache sizes, showcasing its high efficiency in CDN caches.

We further compared SIEVE with LRB [79], the state-of-the-art machine-learning-based cache eviction algorithm optimized for byte miss ratio. Due to LRB's long run time, we only evaluated LRB on the two open-source Wiki traces provided by the authors. Fig. 14a and Fig. 14b show that LRB performs better at small cache sizes (1% and 2%), while SIEVE excels at larger cache sizes. We conjecture that at a small cache size, the ideal objects to cache are popular objects with many requests, which LRB can more easily identify because they have more features (most of LRB's features are about the time between accesses to an object). When the cache size is large, most objects in the cache have few requests. Without enough features, a learned model can provide little benefits [94, 99]. In summary, compared to complex machine-learning-based algorithms, SIEVE still has competitive efficiency.

7.2 SIEVE is not scan-resistant

Besides web cache workloads, we evaluated SIEVE on some block cache workloads. However, we find that SIEVE sometimes shows a miss ratio higher than LRU. The primary reason for this discrepancy is that SIEVE is not scan-resistant. In block cache workloads, which frequently feature scans, popular objects often intermingle with objects from scans. Consequently, both types of objects are rapidly evicted after insertion. Since SIEVE does not use a ghost cache — a shadow cache that keeps track of recently evicted items to make smarter future eviction decisions — it cannot recognize the popular objects when they are requested again. This problem is less severe on the large cache size, but when the cache size is small, we observe that having a ghost is critical to be scan-resistant. We conjecture that not being scan-resistant is probably the reason why SIEVE remained undiscovered over the decades of caching research, which has been mostly focused on page and block accesses.

7.3 TTL-friendliness

Time-to-live (TTL) is a common feature in web caching [97, 98]. It specifies the duration during which an object can be used. After the TTL has elapsed, the object expires and can no longer be served to the user, even if it may still be cached. Most existing eviction algorithms today do not consider object expiration and require a separate procedure, e.g., scanning the cache, to remove expired objects. Similar to FIFO, SIEVE maintains objects in insertion order, which allows objects in TTL-partitioned caches, e.g., Segcache [98], to be sorted by expiration time. This provides a convenient method for discovering and removing expired objects.

8 Conclusion

We design SIEVE, a simple, efficient, fast, and scalable cache eviction algorithm for web caches that leverages "lazy promotion" and "quick demotion". The high efficiency in SIEVE comes from gradually sifting out the unpopular objects. SIEVE *is the first and the simplest cache primitive that supports both lazy promotion and quick demotion*. This serves as the foundation for SIEVE's high efficiency and high performance. Evaluated on 1559 traces from 7 datasets, we show that SIEVE outperforms complex state-of-the-art algorithms on over 45% of the traces. We implemented SIEVE in five open-source production libraries using less than 20 lines on average.

Availability

The code and data used in this work are open-sourced at `https://github.com/cacheMon/NSDI24-SIEVE`. This repository includes both the simulator and prototypes.

Additionally, we have engineered cache libraries based on SIEVE for various programming languages. More information is available at `https://sievecache.com`.

Acknowledgments

We thank the anonymous reviewers and our shepherd Kay Ousterhout for constructive suggestions. We are grateful to the individuals and organizations that have generously open-sourced and shared production traces. We thank Cloudlab [40] for the infrastructure support for running experiments. We also appreciate the members of SimBioSys and PDL Consortium for their interest, insights, feedback, and support.

References

[1] Analytics/data lake/traffic/caching. https://wikitech.wikimedia.org/wiki/Analytics/Data_Lake/Traffic/Caching. Accessed: 2023-04-27.

[2] Apache traffic server. https://trafficserver.apache.org/. Accessed: 2023-04-27.

[3] lru-dict. https://github.com/amitdev/lru-dict. Accessed: 2023-04-27.

[4] lru-rs. https://github.com/jeromefroe/lru-rs. Accessed: 2023-04-27.

[5] Memcached - a distributed memory object caching system. http://memcached.org/. Accessed: 2023-04-27.

[6] mnemonist. https://github.com/Yomguithereal/mnemonist. Accessed: 2023-04-27.

[7] Nginx. https://nginx.org/. Accessed: 2023-04-27.

[8] pelikan. https://github.com/pelikan-io/pelikan. Accessed: 2023-04-27.

[9] Redis. http://redis.io/. Accessed: 2023-04-27.

[10] Running cachebench with the trace workload. https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_FB_HW_eval. Accessed: 2023-04-27.

[11] Varnish cache. https://varnish-cache.org/. Accessed: 2023-04-27.

[12] Ismail Ari, Ahmed Amer, Robert B Gramacy, Ethan L Miller, Scott A Brandt, and Darrell DE Long. ACME: Adaptive Caching Using Multiple Experts. In *WDAS*, volume 2, pages 143–158, 2002.

[13] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[14] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S. Berger. Caching with Delayed Hits. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, pages 495–513, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with Adaptive Replacement. In *3rd USENIX Conference on File and Storage Technologies*, FAST'04, 2004.

[16] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX symposium on networked systems design and implementation*, NSDI'18, pages 389–403, 2018.

[17] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture*, HPCA'15, pages 64–75, Burlingame, CA, USA, February 2015. IEEE.

[18] Nathan Beckmann and Daniel Sanchez. Maximizing Cache Performance Under Uncertainty. In *2017 IEEE International Symposium on High Performance Computer Architecture*, HPCA'17, pages 109–120, Austin, TX, February 2017. IEEE.

[19] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[20] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX symposium on operating systems design and implementation*, OSDI'20, pages 753–768. USENIX Association, November 2020.

[21] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. RobinHood: Tail latency aware caching – dynamic reallocation from Cache-Rich to Cache-Poor. In *13th USENIX symposium on operating systems design and implementation*, OSDI'18, pages 195–212, Carlsbad, CA, October 2018. USENIX Association.

[22] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. AdaptSize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX symposium on networked systems design and implementation*, NSDI'17, pages 483–498, 2017.

[23] Adit Bhardwaj and Vaishnav Janardhan. Pecc: Prediction-error correcting cache. In *Workshop on ML for Systems at NeurIPS*, volume 32, 2018.

[24] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX annual technical conference*, ATC'17, pages 499–511, Santa Clara, CA, July 2017. USENIX Association.

[25] bradfitz. group cache. https://github.com/golang/groupcache. Accessed: 2023-04-27.

[26] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: evidence and implications. In *Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 126–134 vol.1, New York, NY, USA, 1999. IEEE.

[27] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. On the implications of Zipf's law for web caching. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1998.

[28] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. mPart: miss-ratio curve guided partitioning in key-value stores. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*, ISMM'18, pages 84–95, Philadelphia PA USA, June 2018. ACM.

[29] Pei Cao and Sandy Irani. Cost-Aware WWW Proxy Caching Algorithms. In *USENIX Symposium on Internet Technologies and Systems*, USITS'97, Monterey, CA, December 1997. USENIX Association.

[30] Richard W. Carr and John L. Hennessy. WSCLOCK: a simple and effective algorithm for virtual memory management. In *Proceedings of the eighth ACM symposium on Operating systems principles*, SOSP '81, pages 87–95, New York, NY, USA, December 1981. Association for Computing Machinery.

[31] H. Che, Z. Wang, and Y. Tung. Analysis and design of hierarchical Web caching systems. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, volume 3, pages 1416–1424, Anchorage, AK, USA, 2001. IEEE.

[32] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX symposium on networked systems design and implementation*, NSDI'16, pages 379–392, 2016.

[33] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX annual technical conference*, ATC'17, pages 321–334, Santa Clara, CA, July 2017. USENIX Association.

[34] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[35] Fernando J Corbato. A paging experiment with the multics system. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[36] Peter J Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[37] Meta developers. Cachelib. https://cachelib.org. Accessed: 2023-04-27.

[38] Meta developers. Distributed mutex. https://github.com/facebook/folly/blob/2c00d14adb9b632936f3abfbf741373871cd64a6/folly/synchronization/DistributedMutex.h. Accessed: 2023-04-27.

[39] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, S.H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, December 2001.

[40] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[41] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive Software Cache Management. In *Proceedings of the 19th International Middleware Conference*, pages 94–106, Rennes France, November 2018. ACM.

[42] Gil Einziger, Ohad Eytan, Roy Friedman, and Benjamin Manes. Lightweight robust size aware cache management. *ACM Transactions on Storage*, 18(3), August 2022.

[43] Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Transactions on Storage*, 13(4):1–31, December 2017.

[44] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX symposium on networked systems design and implementation*, NSDI'19, pages 65–78, Boston, MA, February 2019. USENIX Association.

[45] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using non-volatile memory for storing deep learning models. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of machine learning and systems*, volume 1 of *mlsys'20*, pages 40–52, 2019.

[46] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. It's time to revisit LRU vs. FIFO. In *12th USENIX workshop on hot topics in storage and file systems*, hotStorage'20. USENIX Association, July 2020.

[47] Bin Fan, David G Andersen, and Michael Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *10th USENIX symposium on networked systems design and implementation*, NSDI'13, pages 371–384, 2013.

[48] Philippe Flajolet, Daniele Gardy, and Loÿs Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics*, 39(3):207–229, 1992.

[49] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. Youtube traffic characterization: a view from the edge. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 15–28, 2007.

[50] Xiaoming Gu and Chen Ding. On the theory and potential of lru-mru collaborative cache management. *SIGPLAN Not.*, 46(11):43–54, jun 2011.

[51] Lei Guo, Enhua Tan, Songqing Chen, Zhen Xiao, and Xiaodong Zhang. The stretched exponential distribution of internet media access patterns. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 283–294, 2008.

[52] Syed Hasan, Sergey Gorinsky, Constantine Dovrolis, and Ramesh K Sitaraman. Trade-offs in optimizing the cache deployments of cdns. In *IEEE INFOCOM 2014-IEEE conference on computer communications*, pages 460–468. IEEE, 2014.

[53] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic modeling of data eviction in cache. In *2016 USENIX annual technical conference*, ATC'16, pages 351–364, Denver, CO, June 2016. USENIX Association.

[54] Xinyue Hu, Eman Ramadan, Wei Ye, Feng Tian, and Zhi-Li Zhang. Raven: belady-guided, predictive (deep) learning for in-memory and content caching. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*, CoNEXT '22, pages 72–90, New York, NY, USA, November 2022. Association for Computing Machinery.

[55] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 167–181, New York, NY, USA, November 2013. Association for Computing Machinery.

[56] Stratis Ioannidis, Laurent Massoulie, and Augustin Chaintreau. Distributed caching over heterogeneous mobile networks. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS'10, pages 311–322, 2010.

[57] Stratis Ioannidis and Edmund Yeh. Adaptive Caching Networks with Optimality Guarantees. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS'16, pages 113–124, Antibes Juan-les-Pins France, June 2016. ACM.

[58] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: an effective improvement of the CLOCK replacement. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATC'05, page 35, USA, April 2005. USENIX Association.

[59] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS Performance Evaluation Review*, volume 30 of *SIGMETRICS'02*, pages 31–42, June 2002.

[60] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB'94, pages 439–450, San Francisco, CA, USA, September 1994. Morgan Kaufmann Publishers Inc.

[61] R. Karedla, J.S. Love, and B.G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, March 1994.

[62] Cong Li. DLIRS: Improving Low Inter-Reference Recency Set Cache Replacement Policy with Dynamics. In *Proceedings of the 11th ACM International Systems and Storage Conference*, SYSTOR '18, pages 59–64, New York, NY, USA, June 2018. Association for Computing Machinery.

[63] Conglong Li and Alan L. Cox. GD-Wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys'15, pages 1–15, Bordeaux France, April 2015. ACM.

[64] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast In-Memory Key-Value storage. In *11th USENIX symposium on networked systems design and implementation*, NSDI'14, pages 429–444, Seattle, WA, April 2014. USENIX Association.

[65] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniel Campello, Andy Rudoff, and Raju Rangaswami. MULTI-CLOCK: Dynamic Tiering for Hybrid Memory Systems. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 925–937, April 2022. ISSN: 2378-203X.

[66] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Theory and practice of caching billions of tiny objects on flash. In *ACM Transactions on Storage*, volume 18 of *TOS'22*, August 2022.

[67] Nimrod Megiddo and Dharmendra S Modha. ARC: A self-tuning, low overhead replacement cache. In *2nd USENIX conference on file and storage technologies*, FAST'03, 2003.

[68] Sailesh Mukil. Cache warming: Leveraging ebs for moving petabytes of data. https://netflixtechblog.medium.com/cache-warming-leveraging-ebs-for-moving-petabytes-of-data-adcf7a4a78c3. Accessed: 2023-04-27.

[69] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, and others. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'13, pages 385–398, 2013.

[70] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, 22(2):297–306, June 1993.

[71] Sejin Park and Chanik Park. FRD: A filtering based buffer cache algorithm that considers both frequency and reuse distance. In *Proc. of the 33rd IEEE International Conference on Massive Storage Systems and Technology (MSST)*, 2017.

[72] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. Frozenhot cache: Rethinking cache management for modern software. In *Twenty-third EuroSys Conference*, EuroSys'23, New York, NY, USA, 2023. Association for Computing Machinery.

[73] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache:load-balanced,low-latency cluster caching with online erasure coding. In *12th USENIX symposium on operating systems design and implementation*, OSDI'16, pages 401–417, 2016.

[74] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS conference on measurement and modeling of computer systems*, SIGMETRICS'90, pages 134–142, New York, NY, USA, 1990. Association for Computing Machinery.

[75] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning Cache Replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies*, FAST'21, pages 341–354. USENIX Association, February 2021.

[76] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. CliqueMap: productionizing an RMA-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM'21, pages 93–105, Virtual Event USA, August 2021. ACM.

[77] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. EELRU: simple and effective adaptive page replacement. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):122–133, May 1999.

[78] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, September 1978.

[79] Zhenyu Song, Daniel S Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett

Witchel, and others. Learning relaxed belady for content distribution network caching. In *17th USENIX symposium on networked systems design and implementation*, NSDI'20, pages 529–544, 2020.

[80] Zhenyu Song, Kevin Chen, Nikhil Sarda, Deniz Altinbuken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi. Halp: Heuristic aided learned preference eviction policy for youtube content delivery network. In *20th USENIX Symposium on Networked Systems Design and Implementation*, pages 1149–1163, Boston, MA, April 2023. USENIX Association.

[81] Kunwadee Sripanidkulchai, Bruce Maggs, and Hui Zhang. An analysis of live streaming workloads on the internet. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 41–54, 2004.

[82] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K Sitaraman. Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 55–67, 2017.

[83] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 373–386, 2015.

[84] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ML-based LeCaR. In *10th USENIX workshop on hot topics in storage and file systems*, hotStorage'18, Boston, MA, July 2018. USENIX Association.

[85] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX annual technical conference*, ATC'17, pages 487–498, Santa Clara, CA, July 2017. USENIX Association.

[86] Hua Wang, Xinbo Yi, Ping Huang, Bin Cheng, and Ke Zhou. Efficient SSD Caching by Avoiding Unnecessary Writes using Machine Learning. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP'18, pages 1–10, Eugene OR USA, August 2018. ACM.

[87] Theodore M Wong and John Wilkes. My cache or yours?: Making storage more exclusive. In *USENIX Annual Technical Conference*, ATC'02, pages 161–175, 2002.

[88] Nan Wu and Pengcheng Li. Phoebe: Reuse-Aware Online Caching with Reinforcement Learning for Emerging Storage Models, November 2020.

[89] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. zExpander: a key-value cache with both high performance and fewer misses. In *Proceedings of the Eleventh European Conference on Computer Systems*, Eurosys'16, pages 1–15, London United Kingdom, April 2016. ACM.

[90] Gang Yan and Jian Li. RL-Bélády: A Unified Learning Framework for Content Caching. In *Proceedings of the 28th ACM International Conference on Multimedia*, pages 1009–1017, Seattle WA USA, October 2020. ACM.

[91] Gang Yan and Jian Li. Towards Latency Awareness for Content Delivery Network Caching. ATC'22, pages 789–804, 2022.

[92] Juncheng Yang. libcachesim: a high-performance library for building cache simulators. https://libcachesim.com/. Accessed: 2023-04-27.

[93] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. GL-Cache: Group-level learning for efficient and high-performance caching. FAST'23, pages 115–134, 2023.

[94] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and K. V. Rashmi. FIFO can be better than LRU: the power of lazy promotion and quick demotion. In *The 19th Workshop on Hot Topics in Operating Systems (HotOS 23)*, 2023.

[95] Juncheng Yang, Anirudh Sabnis, Daniel S. Berger, K. V. Rashmi, and Ramesh K. Sitaraman. C2DN: How to harness erasure codes at the edge for efficient content delivery. In *19th USENIX symposium on networked systems design and implementation*, NSDI'22, pages 1159–1177, Renton, WA, April 2022. USENIX Association.

[96] Juncheng Yang, Yao Yue, and K. V. Rashmi. Slides of a large scale analysis of hundreds of in-memory cache clusters at twitter. https://www.usenix.org/sites/default/files/conference/protected-files/osdi20_slides_yang.pdf. Accessed: 2023-04-27.

[97] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX symposium on operating systems design and implementation*, OSDI'20, pages 191–208. USENIX Association, November 2020.

[98] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'21, pages 503–518. USENIX Association, April 2021.

[99] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and K.V. Rashmi. Fifo queues are all you need for cache eviction. In *Symposium on Operating Systems Principles (SOSP'23)*, 2023.

[100] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission Optimization for Google Datacenter Flash Caches. In *2022 USENIX Annual Technical Conference*, ATC'22, pages 1021–1036, Carlsbad, CA, July 2022. USENIX Association.

[101] Yiying Zhang, Gokul Soundararajan, Mark W. Storer, Lakshmi N. Bairavasundaram, Sethuraman Subbiah, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Warming up storage-level caches with bonfire. In *Proceedings of the 11th USENIX conference on File and Storage Technologies*, FAST'13, pages 59–72, USA, February 2013. USENIX Association.

[102] Chen Zhong, Xingsheng Zhao, and Song Jiang. LIRS2: an improved LIRS replacement algorithm. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, SYSTOR'21, pages 1–12, Haifa Israel, June 2021. ACM.

[103] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenji Liu, and Tianming Yang. Tencent photo cache traces (SNIA IOTTA trace set 27476). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, February 2016.

[104] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. Demystifying cache policies for photo stores at scale: A tencent case study. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, page 284–294, New York, NY, USA, 2018. Association for Computing Machinery.

[105] Y. Zhou, Z. Chen, and K. Li. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):505–519, June 2004.

[106] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATC'01, pages 91–104, USA, 2001. USENIX Association.

Harvesting Idle Memory for Application-Managed Soft State with Midas

Yifan Qiao Zhenyuan Ruan‡ Haoran Ma Adam Belay‡ Miryung Kim Harry Xu

UCLA ‡*MIT CSAIL*

Abstract

Many applications can benefit from data that increases performance but is not required for correctness (commonly referred to as *soft state*). Examples include cached data from backend web servers and memoized computations in data analytics systems. Today's systems generally statically limit the amount of memory they use for storing soft state in order to prevent unbounded growth that could exhaust the server's memory. Static provisioning, however, makes it difficult to respond to shifts in application demand for soft state and can leave significant amounts of memory idle. Existing OS kernels can only spend idle memory on caching disk blocks—which may not have the most utility—because they do not provide the right abstractions to safely allow applications to store their own soft state.

To effectively manage and dynamically scale soft state, we propose *soft memory*, an elastic virtual memory abstraction with *unmap-and-reconstruct* semantics that makes it possible for applications to use idle memory to store whatever soft state they choose while guaranteeing both safety and efficiency. We present Midas, a soft memory management system that contains (1) a runtime that is linked to each application to manage soft memory objects and (2) OS kernel support that coordinates soft memory allocation between applications to maximize their performance. Our experiments with four real-world applications show that Midas can efficiently and safely harvest idle memory to store applications' soft state, delivering near-optimal application performance and responding to extreme memory pressure without running out of memory.

1 Introduction

A wide range of applications can benefit from storing *soft state* in memory, including web applications [43], databases [32], key-value stores [30], CDN services [12, 34], and model serving frameworks [9]. Data is considered soft state when it is helpful for efficiency, but discarding it does not impact correctness because it can easily be reconstructed if it is later needed. For example, caches and memoization are both common forms of soft state. Soft state enables applications to trade extra memory consumption for better performance, and these gains generally increase with the amount of memory available [45, 47]. A significant fraction of memory is left idle in today's datacenters [27, 48], suggesting there is a large untapped opportunity to improve overall efficiency by using idle memory to store soft state.

While spending memory on soft state can improve performance, it must not compete with the need to store regular application data. For example, if too much memory is spent on soft state, this could lead to swapping to disk or worse still, out-of-memory errors, which can result in failures. Because of this, developers often limit their storage of soft state to a small static amount, for fear that they may run out of memory. In other words, it is a challenge to allocate enough soft state to consume all available idle memory, but to not go beyond the point where it would cause performance issues or failures.

Existing OS abstractions for elastically responding to changes in available idle memory are too limited. For example, the Linux Kernel maintains a *page cache* that automatically fills idle memory but it can only be used to cache disk blocks. This constrains idle memory to storing just a single type of soft state which may or may not provide the most utility for applications.

An ideal abstraction would instead democratize access to idle memory so that each application could choose how to best spend it (*i.e.*, the type of soft state that is most beneficial). For example, suppose an application does not rely much on local storage, but frequently accesses objects stored in a key value store over the network. Instead of being limited to the page cache, idle memory could be spent on caching the key-value store's objects locally, resulting in a much greater benefit.

This problem is further complicated in today's multi-tenant cloud. It is common for each server to run multiple applications, and they may come from different users and exhibit dramatically different performance sensitivity to the amount of soft state. At the same time, adding memory to one application can lead to reductions in the performance of others. Consequently, determining how to dynamically balance the soft state needs of different applications in a way that maximizes overall memory utility/performance is a challenge.

Insight. In this paper, we aim to answer the following question: can we provide a new *virtual memory abstraction* for soft state (herein referred to as "soft memory") that developers can use to coordinate with the kernel so that they can take full advantage of all available memory? In other words, our goal is to no longer limit idle memory to the page cache, and to instead allow its use to be customized by each application in a way that maximizes overall utility.

Unlike existing systems that perform caching entirely in the user space [2, 9, 30], we propose *Midas*, a system that coordinates with the kernel to dynamically provision soft memory between applications. The advantage of this approach is two-fold. First, application developers can program with the illusion of an "unlimited cache", and are thus freed of the burden of manually managing their soft state. To avoid running out of memory, the kernel responds to memory pressure by rapidly *unmapping* soft memory pages. To transparently recover any lost soft state, later accesses will automatically trigger the application to *reconstruct* the missing soft state. Second, the kernel has global visibility of all applications, their memory usage, and the amount of idle memory, making it possible to understand each application's sensitivity to memory size and automatically coordinate soft memory allocation between applications. Midas also incorporates the page cache by treating it as another source of soft memory.

Challenges. Midas is a soft memory management system that achieves (1) programming flexibility and (2) dynamic memory provisioning, with *unmap-and-reconstruct* semantics, to guarantee both safety and efficiency. Realizing these benefits requires overcoming four major challenges:

First, what interfaces shall we expose to developers? To improve usability, Midas provides developers with a *soft memory pointer* abstraction (similar to C++ smart pointers) to access soft memory easily and safely (see §4.1). Midas offers a set of high-level key-value store APIs, which are similar to those of popular cache services (such as Memcached [30], Redis [2], CacheLib [9], *etc.*), but enhanced to allow the exposure of more semantics to the runtime. A critical interface we expose to developers is *data structure reconstruction*—developers not only register soft memory objects but also specify their (re)construction logic, so soft state can be transparently regenerated if it is later accessed after it was evicted.

Second, how shall soft memory be managed? Program data is allocated as objects on the heap but the kernel cannot recognize them, as it is only aware of memory pages. As a result, if we let the kernel manage soft memory alone, it could only reclaim space in coarse-grained units without knowledge of what objects the space contains. For example, reclaiming hot (*i.e.*, frequently accessed) objects in a soft-state cache can lead to significant slowdowns. In addition, it is undesirable to reclaim space from the programs that would benefit the most from soft state when others need it less, but such performance sensitivity information is invisible to the kernel.

To solve the problem, we propose a runtime library that can be linked into each application to recognize object behaviors, letting the runtime and the kernel *co-manage* soft memory. The Midas runtime offers a log-structured allocator [41] and a concurrent evacuator that continuously identifies and compacts hot objects into a small soft memory space. This information (of hot and cold regions) is shared with the kernel so that it can focus its reclamation on regions with cold objects (see §4.2).

Third, how can we coordinate soft memory allocation between applications? The runtime can only see each application's individual behavior without any global knowledge of the server's available memory and other applications' needs. Furthermore, the runtime can only manage objects in user space, but *cannot* dynamically add/remove memory between applications. To overcome this challenge, we propose a global coordinator inside the Linux kernel. The coordinator periodically probes each application by communicating with the runtime to request information regarding the application's sensitivity to cache size. Cold regions of soft memory from applications that are less sensitive to size changes will be reclaimed and memory will be given to those that are more sensitive (and hence benefit more from a larger cache) by the kernel (see §4.3).

Finally, how can the kernel quickly reclaim soft memory without disrupting a running application? Since the kernel operates at page granularity, a natural idea is to swap out pages that contain soft-state data. Unfortunately, swapping is disruptive—swapping out a page blocks all incoming memory allocations and hence all threads of the application; frequently swapping pages can introduce significant overheads that prevent applications from reaching service-level agreements (SLA) [40] (see §2).

To maintain high efficiency, Midas instead uses the kernel to unmap pages directly (which is much faster than swapping them to disk). When pages are unmapped, their underlying data is lost—this is acceptable for soft state because it can be regenerated. Without coordination, however, the kernel cannot distinguish soft state from application data, making unmapping potentially unsafe.

To solve this problem, our runtime is designed in a way that is resilient to data loss. A soft pointer-based interface detects data loss through segmentation faults that are triggered by the runtime's functions. These functions are carefully designed to capture faults and transparently invoke a *reconstruction* interface to regenerate the needed data (see §4.2.3).

Compared to paging, Midas does not freeze the execution when shrinking soft memory, resulting in less disruption to the application. Furthermore, reconstruction focuses on recovering the individual objects that are needed and hence is much more fine-grained and can be more efficient than swapping, which brings back entire pages. Reconstruction may require more computation than paging (the amount of computation depends on exact soft state data). Therefore, Midas provides a profiling tool that warns developers when reconstruction incurs a high cost (discussed in §4.4).

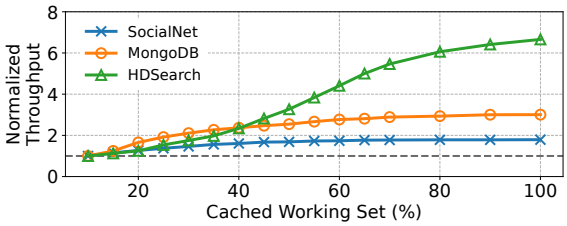


Figure 1: The throughput of all three applications increases by caching more soft state, but the benefit varies: SocialNet is 1.8× faster by caching 70% of its working set, while HDSearch, in contrast, achieves a 3.3× throughput increase by caching 50% state.

Results. With Midas, one can easily allow applications to take advantage of soft state that do not currently support it. It is also easy to port legacy code that uses an existing cache system to use Midas instead. Our evaluation shows that Midas can efficiently and safely harvest idle memory to store applications' soft state and achieve near-optimal performance while reacting to extreme memory pressure quickly enough to avoid running out of memory. By effectively granting soft memory to the applications that benefit the most, Midas achieves 1.34× higher overall throughput than Cliffhanger (a state-of-the-art caching system). Midas is available at https://github.com/uclasystem/midas.

2 Motivation

Many types of applications can benefit from soft state. For example, a web frontend could cache content locally after loading it from a backend to reduce network traffic and improve response times; a database could cache the results of user queries to reduce disk I/O and improve throughput; and a data analytic or machine learning system could memoize intermediate computation results to eliminate redundant computations.

To gain a high-level understanding of how much improvement can be achieved by storing soft state, we experimented with three datacenter applications: SocialNet (from DeathStarBench [18]), MongoDB [32], and HDSearch (from µSuite [46]). Each of these applications are capable of using soft state. SocialNet [18] is a web forum built using microservices; it employs Memcached and Redis to cache user data in its frontend services. MongoDB [32] is a NoSQL database; it has a built-in, in-memory caching engine that caches recently queried data. HDSearch is an image search service that memoizes the feature vectors of the images in its corpus, generated by a GPU-based DNN.

Figure 1 shows the throughput of each application with varying amounts of soft state. The *x*-axis represents the percentage of each application's working set cached in memory, and the *y*-axis shows the normalized throughput (to its performance without soft state). Soft state is helpful to all applications but the amount of benefit it provides varies. SocialNet is the least sensitive to its soft state size; however, it still sees a 1.8× speedup by storing 70% of its soft state. HDSearch, in contrast, is more sensitive to the soft state size—its throughput increases by more than 3× with only 50% of its soft state.

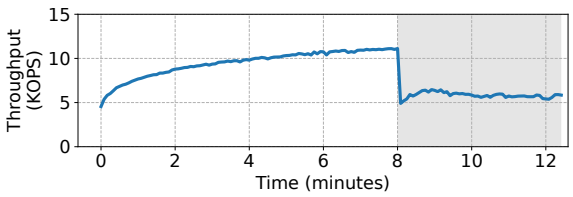


Figure 2: SocialNet starts to swap when it caches excessive data and exhausts all available memory at $t = 8$min and it experiences a throughput collapse.

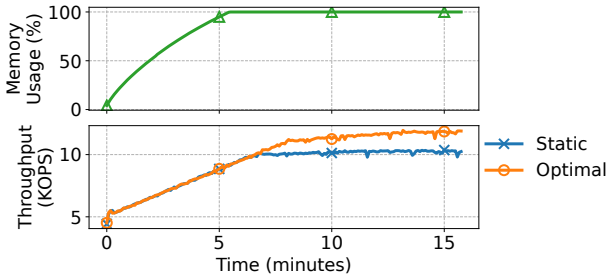


Figure 3: Statically provisioning the cache space for SocialNet is sub-optimal. During t =0min–5min, the cache is overprovisioned which wastes memory. After that, the cache becomes underprovisioned which limits performance.

Real-world datacenter applications can access a massive amount of data. For example, a web forum like Twitter generates petabytes of new data every day [49]. Thus, blindly storing soft state in memory without a proper limit can hurt application performance. An example of this problem is shown in Figure 2. Storing soft state increases the throughput of SocialNet up to a point. However, when idle memory becomes exhausted, the kernel begins to swap out pages (at $t = 8$min), leading to a severe collapse in throughput.

A simple strawman solution is to statically provision a limited memory capacity for storing soft state so that memory use does not grow unbounded. However, provisioning the right capacity is extremely challenging in practice.

First, for each application, we must find its sweet spot of cache capacity; underprovisioning limits performance while overprovisioning wastes memory. In addition, datacenter applications often have phased behaviors and load variability [7, 8], making it impossible to have a simple static configuration that is optimal at all times. For example, Figure 3 shows the results of SocialNet when statically provisioning it with 4 GiB for storing soft state. It takes about 5 minutes to fully fill this memory, leading to a suboptimal utilization during this period. Performance increases with more usage until it exhausts the soft state limit. After that, performance flattens out despite the possibility of higher throughput if additional soft state memory were available (the optimal line).

Second, as shown by Figure 1, different applications gain different amounts of benefits through caching. To achieve optimal overall performance with a limited amount of memory,

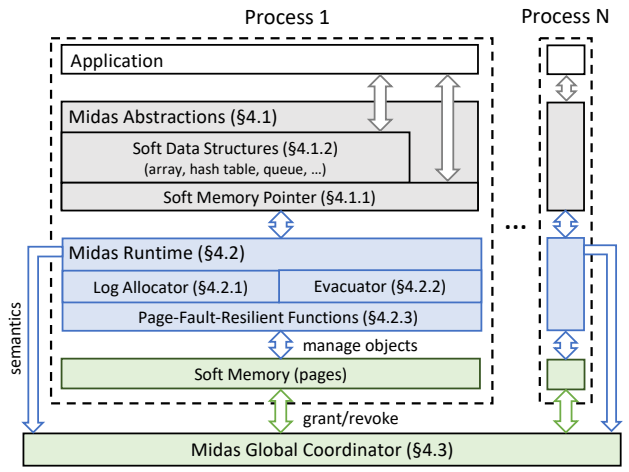


Figure 4: Midas enables developers to utilize soft memory easily and efficiently with three major components: a familiar programming abstraction, an application-integrated runtime, and a global soft-memory budget coordinator.

one must grant space correctly to the applications that benefit the most. For example, initially MongoDB's performance is most sensitive to the amount of soft state (the left side of Figure 1), and thus we should prioritize its need. However, the return diminishes quickly after caching 30% of its state. To make the best use of the remaining memory, we should respond by granting memory to HDSearch.

These problems call for a new system that can provide *elastic* access to soft state for applications and *dynamically coordinate* usage among applications in response to each one's execution phase and sensitivity to soft state size. To be efficient, soft state should be able to quickly scale up and down its capacity with little disruption. To be safe, the system should be resilient to data loss caused by scaling down. To be responsive, the system should conduct coordination among applications quickly. Finally, to be practical, the system should provide familiar programming abstractions for developers to store and access soft state.

3 Midas Overview

As shown by Figure 4, Midas consists of three main components: a programming abstraction for using soft memory (§4.1), an application-integrated runtime that manages soft-memory objects (§4.2), and a global coordinator that arbitrates soft memory usage across different applications (§4.3).

Midas provides programming abstractions that enable simple and efficient use of soft memory through familiar APIs. At a low-level, programmers can interact with Midas through *soft memory pointers*, an abstraction that provides object ownership similar to C++ smart pointers. However, it differs in that underlying objects can be forcibly released when under memory pressure, even if still in scope. If a released object is later accessed, a *reconstructor* function is invoked to regenerate the missing object (*e.g.*, by fetching it from a database over the network).

Building upon soft memory pointers, Midas provides a higher-level library of familiar STL-style *soft data structures*—including arrays, hash tables, and queues. These hide the complexity of managing individual soft memory pointers, and can be used as drop-in replacements for existing data structures. For example, a developer building a key-value store similar to Memcached could use a soft hash table to store soft memory objects. Midas's high-level interface is generally sufficient for most use cases, but developers are free to build their own custom soft data structures through use of soft memory pointers.

Midas manages soft memory objects through a runtime that is linked as a library with the application. It serves as an allocator for soft memory objects. It works cooperatively with the coordinator (discussed next) to determine the best memory to release (*i.e.*, idle memory first, then cold objects, and finally hot objects). To achieve this, the runtime provides a moving allocator that embraces the idea of log-structured memory [41] to organize soft memory into different segments. An evacuator thread scans and compacts logs to segregate hot objects, cold objects, and dead objects. This helps both to coordinate which memory should be freed and to reduce fragmentation.

However, the runtime is not trusted for correct operation. If it fails to respond quickly enough or if memory pressure becomes too severe, pages will be unmapped in an uncoordinated fashion to avoid swapping. In the event such forcible revocation happens, the runtime is designed to safely tolerate page faults when accessing unmapped memory. To achieve this, we developed a set of page-fault-resilient functions and used them as primitives to build our runtime.

Midas's global coordinator dynamically adjusts the soft memory budget among applications to optimize their overall performance. It periodically probes the marginal utility of soft memory for each application by granting a small amount of additional memory and observing the effect on performance. Using this information, the coordinator can optimize the allocation of soft memory by granting it to the applications that benefit the most. The coordinator defines the global utility function as the weighted average of all applications' performance and employs a hill-climbing algorithm to approach the global optimal point. Midas allows operators to specify the weight of each application to indicate relative significance, similar to the `nice` interface of Linux.

4 Design

4.1 Soft Memory Abstraction

Soft memory is a new type of memory that can be revoked under memory pressure. In Midas, soft memory is backed by physical pages that can be unilaterally unmapped and reclaimed by the OS kernel. Accessing reclaimed soft memory will trigger a *reconstruction event* to rebuild the missing data. Midas provides a smart-pointer-like API to enable developers to easily use soft memory, hiding the complex details of soft memory allocation/deallocation, page-fault handling, and data

```
template <typename T, typename... ReconArgs>
class SoftMemPool {
  SoftMemPool(std::function<T(ReconArgs...)>
    reconstructor);
  SoftUniquePtr<T, ReconArgs...> new_unique();
  SoftSharedPtr<T, ReconArgs...> new_shared();
};

template <typename T, typename... ReconArgs>
class SoftUniquePtr {
  ~SoftUniquePtr();
  T read(ReconArgs... args);
  void write(T newval);
  bool cmpxchg(const T &oldval, T newval);
};
```

Listing 1: Midas's soft memory pool and unique pointer interface.

```
1  template <typename T> class SoftArray {
2    SoftArray(size_t size, std::function<T(size_t)>
3      reconstructor);
4    T read(size_t idx);
5    void write(size_t idx, T t);
6    bool cmpxchg(size_t idx, const T &oldval, T newval);
7  };
8
9  class BlockCache {
10   BlockCache(size_t sz) : array_(sz, [](size_t idx) {
11     return read_from_storage(idx); }) {}
12   Block read(size_t idx) { return array_.read(idx); }
13   void write(size_t idx, Block block) {
14     array_.write(idx, block);
15     write_to_storage(idx, block);
16   }
17
18   SoftArray<Block> array_;
19 };
```

Listing 2: Midas's soft array interface and a simple user-level storage block cache (similar to Linux's page cache) built using soft array.

reconstruction (§4.1.1). Furthermore, Midas offers high-level data structure libraries as composable building blocks (§4.1.2).

4.1.1 Soft Memory Pointer

Listing 1 shows Midas's soft memory pool and pointer interface. To use soft memory, developers first need to create a soft memory pool which can later be used to allocate soft memory pointers. The pool abstraction conceptually groups together soft pointers whose objects can be reconstructed in a similar way. Midas exposes the pool as a C++ template class whose parameters consist of two parts: `T`, which is the object type of soft pointers to allocate, and `ReconArgs`, which are the types of arguments used for reconstructing a missing object. Developers can initialize a pool with a `reconstructor` function and then allocate pointers using `new_unique` (for soft unique pointers, similar to C++'s `std::unique_ptr`) and `new_shared` (for shared pointers).

Soft memory pointers support automatic lifetime management through reference counting. Developers can use its `read` API to get the value of the pointed object. In case the underlying soft memory has been reclaimed, Midas will automatically reconstruct the missing object using the reconstruction arguments passed into `read` (we will show a concrete example soon in §4.1.2). Midas hides the raw reference and returns the value by *copying*. This is critical as the underlying reference may become invalid any time when the soft memory gets reclaimed. With copying, Midas restricts potential faulting sites to stay inside Midas's internal code, thereby freeing developers from handling complicated page faults in the application code. The copying design incurs negligible performance overheads (only a few additional cache accesses). `write` enables developers to update the object value. However, different from `read`, `write` does not require reconstruction arguments as Midas can directly rebuild the object using the new value. Soft pointers also support atomic operations like compare-and-exchange, enabling developers to atomically update object values to support multi-threaded applications.

With its smart pointer design, Midas is able to capture rich application semantics for effectively managing soft memory. For example, since all soft object accesses go through the `read/write` API, Midas can accurately track the hotness information of each object which can be leveraged by Midas runtime for making intelligent object placement and eviction decisions (details in §4.2). Soft pointer's automatic lifetime management enables cascading eviction, improving the efficiency of using soft memory. For instance, in a web forum application, a forum post object may contain a soft unique pointer to an attached picture. Under memory pressure, Midas may decide to evict the post object in which case the reference count of the picture pointer will automatically drop to zero and trigger evicting the dangling picture object cascadingly.

4.1.2 Soft Data Structures

To further reduce the programming effort of using soft memory, Midas offers high-level data structures as convenient building blocks. Midas's built-in data structures include soft arrays, soft hash tables, and soft queues; developers can also easily build more based on the soft pointer abstraction.

Listing 2 presents the interface of soft array (lines 1-8). Developers can create a soft array by specifying its size and reconstructor (which rebuilds the array element of a given index). Soft array supports standard read, write, and atomic operations by index. Under the hood, a soft array is simply implemented via an ordinary array of soft pointers.

Lines 10-21 present a user-level storage block cache as a simple illustrative application, similar to Linux's page cache. `BlockCache` internally wraps a soft array whose elements are storage blocks (line 20). This enables it to efficiently leverage idle memory to cache storage blocks in a best-effort manner. For each block read request, it simply retrieves the result from the soft array (line 14). Upon an element miss, the array automatically reconstructs the element by reading the block back from the storage device (lines 12-13). For each block write request, `BlockCache` updates both the cache in array and the data in storage.

4.2 Application-Integrated Runtime

Midas runtime is the key component that manages soft objects to enable efficient use of soft memory. It includes a log-structured memory allocator that serves memory allocation

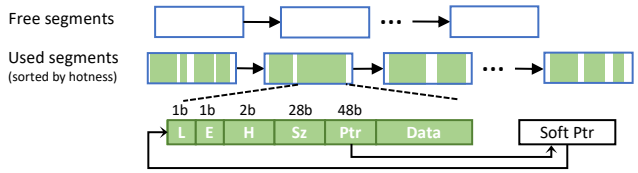


Figure 5: Midas organizes soft memory using a free segment list and a used segment list (sorted by segment's hotness, useful for Midas's evacuator in §4.2.2). It employs a log-structured allocator to serve memory allocation requests. Each object has a 10-byte header, which includes a liveness bit, an evacuating bit, hotness bits, an object size field, and a reversed pointer field.

requests and organizes objects into a list of segments (§4.2.1), a concurrent evacuator that constantly compacts hot objects and releases cold and dead objects (§4.2.2). Page faults can happen in Midas runtime when the soft memory it is accessing gets reclaimed and unmapped because of memory pressure. To ensure robustness, we carefully built the runtime using a set of page-fault-resilient functions which are able to capture page faults and gracefully recover from them (§4.2.3).

In Midas, the runtime as well as the soft memory it manages are linked directly into each application's address space. Compared to traditional cache services (*e.g.*, Memcached) that run in a separate process, our design offers several important advantages. First, it provides direct and efficient soft memory accesses for applications, eliminating the inter-process communication (IPC) overhead. Second, it enables our runtime to profile the application and collect semantics, greatly facilitating semantics-aware optimizations. Third, since each application has its own runtime, we can easily enforce soft memory isolation among applications and adaptively customize the memory management policy of each application.

4.2.1 Log-structured Soft Memory Allocator

Midas embraces the idea of log-structured memory [41] to manage soft memory; it reduces memory fragmentation through compaction, thereby achieving higher efficiency in utilizing soft memory.

Midas's log-structured allocator organizes soft memory using a free segment list and a used segment list, illustrated in Figure 5. Segments are the units for Midas to perform evacuation to compact objects and reclaim space (details in §4.2.2). The total size of all segments (used and free) equals the soft memory budget that the linked application receives from the global coordinator (§4.3). For each memory allocation request, the allocator allocates space from a free segment; if the current one is full, it will pop a new one from the free list. Midas backs each segment using a 2 MiB huge page; this reduces TLB pressure and page table walk cost. While small objects reside in only one segment (i.e., they do not cross the segment boundary), big objects whose sizes are larger than 2 MiB span across multiple segments. Since the free list does not provide any address contiguity guarantee for segments, Midas breaks the big object into smaller pieces—each one fits into a single segment—and chains them together using segment headers. The decomposition is transparent to application developers; upon object read, Midas automatically reads all segregated pieces and stitches them back. This is possible thanks to Midas's pass-by-copy interface (§4.1).

Each allocated object has a 10-byte header inlined with its data, used for tracking the object's runtime information. This includes 1) a liveness bit, indicating whether the object has been deallocated; 2) an evacuating bit, marked by the evacuator to synchronize evacuation with object accesses; 3) hotness bits, a counter that will be incremented (or unchanged when it has reached the maximum) each time the object gets accessed; 4) a size field, indicating the total size of the object; 5) a reverse pointer field, used by the evacuator, if it moves the object, to rewrite the soft pointer.

4.2.2 Soft Memory Evacuator

As the allocation goes on, the application may eventually deplete the free segment list. It is the responsibility of Midas's evacuator to constantly release cold and dead objects, ensuring the best use of soft memory by only storing hot objects. In addition, the evacuator tracks segments in order of hotness in a used list (see Figure 5), to simplify the design and improve the speed of memory reclamation, in which the kernel forcibly unmaps application's soft memory pages under intense memory pressure (§4.3).

Midas's background thread continuously monitors the free segment ratio and triggers evacuation if it falls below a configurable threshold (our default value is 90%). The evacuation mainly consists of three stages:

Scanning Stage. The evacuator first scans through all objects in the used segment list. For each scanned object, it decrements the embedded hotness counter (similar to the CLOCK algorithm [15]). The evacuator treats objects with a zero pre-scanning hotness value, in addition to deallocated objects, as *dead* objects; they will be released in the compaction stage. The evacuator calculates the *live ratio* of each segment (*i.e.*, the percentage of live bytes) during scanning, and then uses it to sort all scanned segments to decide their priority for compaction. The segment with the lowest live ratio will be compacted first as it yields the largest benefits (in terms of the reclaimed space).

Compaction Stage. The evacuator compacts one segment at a time. For each live object, it first relies on the evacuation bit to synchronize with application threads to avoid data race (similar to AIFM [40]). It then copies the object into a new segment and leverages the reversed pointer field to rewrite the address of the corresponding soft pointer. After evacuating all live objects, it moves the segment from the used list into the free list.

Sorting Stage. After compaction, the evacuator calculates the segment-level hotness value for all segments in the used list, defined as $\sum_{\forall obj \in seg} SIZE(obj) \cdot HOTNESS(obj)$. It finally sorts the used list by segment-level hotness in ascending order.

```
1  for each segment S to compact {
2    D = pick_destination_segment();
3    for each object O in S {
4      try {
5        // A wrapper around our PF-resilient memcpy
6        copy_object_into(O, D);
7      } catch (SoftMemUnmapped &exception) {
8        if (exception.fault_addr belongs to O)
9          break; // Skip S as it has gone
10       else // It must belong to D
11         goto line2; // Pick a new D and restart
12     }
13   }
14 }
```

Listing 3: Midas implements its evacuator's compaction code using a page-fault-resilient memory copy function.

4.2.3 Page-Fault-Resilient Functions

Midas runtime directly manipulates soft memory during allocation and evacuation. Since the kernel may unmap soft pages to reclaim memory under pressure (details in §4.3), the runtime has to be aware of page faults and be able to recover from them gracefully. We carefully built the runtime to achieve this goal. First, we stored the important metadata (e.g., the free and used segment lists) in normal memory instead of soft memory, therefore it will not be lost under memory pressure. This is viable as the metadata only consumes little memory (less than 10 MiB). Second, we introduced *page-fault-resilient functions* and used them as primitives to build the runtime.

A page-fault-resilient function is able to capture any internal page fault that stems from dereferencing unmapped soft memory and respond to it by reverting all side effects and throwing a `SoftMemUnmapped(fault_addr)` exception to the caller. As a concrete example, in Midas we internally implemented a page-fault-resilient memory copy function, which is used to build the evacuator's compaction code to withstand page faults (see Listing 3). Page faults can happen when copying objects from the old segment into the new segment. To deal with this case, Midas uses its resilient memory copy function (line 7) to capture and handle the potential exception (lines 8-13).

Midas registers its own signal handler to facilitate capturing and handling all soft-memory-related page faults. Additionally, a page-fault-resilient function satisfies the following requirements to ensure resilience:

- It embeds a fault recovery code block for aborting the partial execution and rolling back side effects. Midas runtime maintains a mapping from resilient functions to their recovery blocks so that when page fault happens the handler can invoke the corresponding recovery code.
- All of its inner non-resilient functions have to be inlined to prevent the control flow from jumping out of its scope. Otherwise, the page fault handler is unable to find the corresponding recovery code.
- It preserves its stack frame base pointer (by disabling the compiler optimization) so that the fault handler can easily unwind its stack and throw an exception back to its caller.

4.3 Global Soft Memory Coordinator

Midas's global coordinator is responsible for granting server's idle memory to applications as soft memory and coordinating the budget across applications to optimize the overall performance.

4.3.1 Soft Memory Management Mechanism

The coordinator maps idle memory pages directly into an application's address space as soft memory segments. For each application, the coordinator dynamically maps or unmaps pages to readjust its soft memory budget. To facilitate the management, the application's runtime shares its free segment list and used segment list with the coordinator.

To grant more soft memory to an application, the coordinator maps more pages to it and inserts them into the free segment list. Similarly, to reclaim memory from an application, the coordinator unmaps pages. The coordinator first tries to pop out and unmap the segments from the free list; since they do not hold any useful live objects, unmapping them does not incur any impact on the application's performance. Meanwhile, the runtime strives to avoid the exhaustion of the free list by triggering evacuation (§4.2.2).

The synergy between the runtime and the coordinator is able to handle moderate memory pressure (*i.e.*, the common case). However, under severe pressure, the evacuation may fall behind, leading to an empty free segment list. To avoid depletion, the coordinator reacts by unmapping used segments which may induce performance disruption in two folds. First, when the application later tries to access an unmapped object, the runtime will experience a page fault which incurs overhead. Second, the runtime has to spend additional time reconstructing the missing object. To alleviate this issue, the coordinator prioritizes cold segments over hot segments. Thanks to the evacuator, the segments in the used list have been ordered by their hotness (§4.2.2). Therefore, the coordinator can realize prioritization by simply unmapping segments based on their order in the list.

4.3.2 Coordination Policy

Midas continuously adjusts each application's soft memory budget by solving the following optimization problem:

$$\underset{m}{\text{maximize}} \sum_{\forall i \in APPS} w_i \Gamma_i(m_i) \text{ , subject to } \sum_{\forall i \in APPS} m_i = M$$

For each application i, w_i denotes its weight (which is either specified by the operator or uses the default value 1) and Γ_i denotes its performance utility when assigned soft memory of size m_i. The server-wide overall utility is defined as the weighted sum of all application's utilities. M denotes the server's total idle memory.

By default, the coordinator estimates Γ_i as $-RCOST_i$, where $RCOST_i$ is the application's CPU usage spent on reconstructing missing objects. Midas's runtime can easily

collect this per-application information and report it to the coordinator. Developers can also plug in the real performance metric reported by applications—which already exists in many datacenter applications [11]—for a more faithful Γ_i.

Midas solves the optimization problem using the hill climbing approach [42]. It periodically probes every application's marginal utility benefit $\frac{\partial \Gamma_i(m_i)}{\partial m_i}$ by additionally assigning a small portion of memory Δ_{m_i} and monitoring the change of utility Δ_{Γ_i}. Midas regrants the soft memory budget from the application with the lowest marginal utility benefit to the one with the highest benefit.

In contrast, Cliffhanger (a recent cache service) [13] adopts a coordination policy that optimizes for the overall cache hit rate, but this does not necessarily optimize the overall performance. For example, caching objects that are frequently accessed may not be helpful if they can be cheaply reconstructed. Midas avoids this issue by using both access frequency and reconstruction cost as metrics for optimization.

4.4 Discussion

Though Midas is mainly designed for caching hot data and memoizing intermediate computation results, developers have the freedom to put any data into soft memory as long as it is reconstructible. However, storing data that is expensive to reconstruct but infrequently accessed can lead to performance issues. Midas provides a profiling tool that generates runtime warnings if such cases are detected. In addition, Midas offers a debugging mode where we validate the reconstruction logic by calling the user-defined reconstruction function and comparing its result with the actual cached object using the object's comparison operator. Bugs are reported if these objects are not identical.

Midas also incorporates Linux's page cache by simply treating it as another per-application soft memory pool. For each application, Midas's shim layer intercepts all POSIX file operations and caches the file data using a soft hash table, whose keys are file inode numbers along with block-aligned offsets and values are file blocks. The reconstructor rebuilds the missing block by performing the actual file read.

5 Implementation

Midas is implemented in C++ and includes bindings for C. Our implementation has 2,814 LOC for the soft memory abstraction (§4.1), 3,866 LOC for the runtime (§4.2), and 1,029 LOC for the global coordinator (§4.3).

Soft data structures store their metadata (*e.g.*, a hash table's bucket array that stores indices) in normal memory and store their data payload (*e.g.*, a hash table's key-value pairs) in soft memory using soft pointers.

The log-structure allocator enforces 16-byte alignment for allocated data to make it GCC-compatible. The evacuator adopts a concurrent pauseless design similar to AIFM [40]. The evacuator ensures atomicity when evacuating or reconstructing large objects that span across multiple soft memory segments. Midas registers its own `SIGSEGV` handler. For each segmentation fault, the handler checks whether the faulting memory address belongs to a soft memory region and whether the faulting program counter (PC) belongs to a page-fault-resilient function; for faults that do not meet these conditions, the handler treats them as unrecoverable exceptions and aborts the program. To facilitate the PC check, Midas leverages a linker script to place all resilient functions into a separate code segment whose layout is known at compile time.

During each application's initialization, the runtime registers itself to the global coordinator using `ioctl` and uses `mmap` to create a shared memory region for exposing information—including its free segment list and used list (implemented as arrays) and the application's reconstruction cost (implemented as a counter)—to the coordinator.

We implemented the global coordinator as a user-space daemon (that runs the coordination policy) and a privileged kernel module (that executes the coordination decision by mapping/unmapping pages to/from user processes directly). Every 5 seconds, the coordinator probes the marginal utility of each application and makes a new adjustment to soft memory budgets. It probes an application by either granting or revoking 64 MiB soft memory and monitoring its performance change. In each adjustment, it regrants up to 256 MiB soft memory from the application with the lowest marginal utility to the one with the highest utility. To avoid oscillation, it refrains from granting more soft memory to the application until it has consumed the additional memory offered in the previous round.

6 Programming with Midas

We present general guidelines of programming with Midas (§6.1) followed by concrete examples of porting four real applications (§6.2).

6.1 Guidelines

When is it safe to use soft memory? Developers can generally use soft memory to store any application data that follows the unmap-and-reconstruct semantics. To support evacuation, developers have to implement copy constructors for objects stored in soft memory.

When is it beneficial to use soft memory? Developers should generally consider using soft memory when applications can opportunistically benefit from having additional memory. Typical use cases include caching in web applications and memoization in data analytics systems. They often have unknown marginal utility and unbounded memory footprint, making them hard to handle efficiently through static provisioning. Midas can benefit them by automatically rightsizing their soft memory budget and harvesting idle memory.

Applications	Abstractions used	Porting effort (LOC)	CPU cores	Normal mem. (GiB)	Peak soft mem. (GiB)	Reconstruction cost (µs/obj.)	Dataset
HDSearch [46]	Soft hash table	36	12	1.7	13.6	1244.2	`OpenImg` [25], 1.9M images
WiredTiger [33]	Soft pointer	332	12	3.7	21.3	20.6	Facebook `USR` [8], 50M KV
Storage Server [24]	Soft array	29	4	1.1	20.4	10.5	`multilate` [23], 16 GiB disk
SocialNet [18]	Soft hash table Soft queue	175	20	1.3	12.2	99.1–3227.7	Socfb-Penn94 [39], 41.5K nodes, 1.4M edges

Table 1: We ported four applications into Midas with low programming effort. All four applications extensively use soft memory while their data reconstruction costs vary drastically.

How to migrate from traditional cache services? Existing applications that employ local cache services (*e.g.*, Memcached [30] or Redis [2]) can directly use Midas as a drop-in replacement. Existing applications that employ distributed cache services (*e.g.*, AWS ElastiCache [1]) can use Midas as a fast local cache tier to reduce the overhead of accessing remote cache.

6.2 Application Case Studies

We ported four applications to Midas. They cover a range of CPU usage, normal and soft memory usage, data reconstruction cost, and Midas's abstraction usage (see Table 1).

HDSearch [46] is an image search service based on content similarity. For each query, a feature extraction backend transforms the input image into a feature vector via a DNN (running on GPU), and then caches the result along with a hash of the image (using Memcached for memoization). To port this application, we replaced Memcached with our soft hash table, which only involves 36 LOC changes. It has 1.7 GiB normal memory usage and 13.6 GiB peak soft memory usage. Reconstructing KV pairs is expensive (1244.2 µs per object) as it requires re-performing transformation on GPU.

WiredTiger [33] is a NoSQL key-value storage engine used by MongoDB [32]. It persists all key-value pairs in storage indexed via an in-memory B+ tree. It has a built-in in-memory caching engine that caches the data of B+ tree's internal nodes and leaf nodes to reduce expensive storage I/Os. To port WiredTiger, we implement its caching engine using Midas's soft memory pool and pointer abstractions; we created a soft memory pool with a reconstruction method that wraps WiredTiger's existing code for handling cache misses, and replaced ordinary B+ tree pointers with soft memory pointers allocated from the pool. This only involves 332 LOC changes. With our port, WiredTiger has 3.7 GiB normal memory usage and 21.3 GiB peak soft memory usage. Reconstructing a tree node object requires reading its content from the disk and rebuilding the index, which takes 20.6 µs.

Storage Service is an NVMe-based block storage service similar to Reflex [24]. It exposes a standard block I/O interface using RPC to support accessing 4KiB storage blocks remotely. Its original design uses SPDK [3] to communicate with the storage block device, which bypasses Linux's page cache. To port it, we cache the block data using a soft array, similar to the `BlockCache` design in Listing 2. This requires adding 29 LOC. With our port, it uses 1.1 GiB normal memory and 20.4 GiB peak soft memory. Reconstructing an array element requires a block I/O which takes 8.5 µs to finish.

SocialNet is a twitter-like latency-critical web service from DeathStarBench [18]. It is built using 12 microservices with sophisticated fan-out patterns and call dependencies. Its original design uses Memcached/Redis to cache users' data and memoize results of certain queries, and employs pools to cache TCP connections/RPC sessions. Since each microservice has its own binary and runs within its own process, Midas treats SocialNet as 12 different applications. To port it, for each microservice, we replace its Memcached/Redis usage with Midas's soft hash table and connection pool with Midas's soft queue; this involves 175 LOC changes. With our port, it uses 1.3 GiB normal memory and 12.2 GiB peak soft memory. It takes 99.1–3227.7 µs to reconstruct an object depending on its type; for example, it takes only 99.1 µs to re-establish an RPC session but requires 3227.7 µs to re-fetch a user's post.

7 Evaluation

Our evaluation seeks to answer the following questions:

1. Can Midas judiciously coordinate soft memory among applications to optimize overall performance? (§7.1)
2. Can Midas quickly and reactively harvest available idle memory to improve utilization and performance? (§7.2)
3. Can Midas quickly react to memory pressure to avoid out-of-memory killing while maintaining good performance? (§7.3 and §7.4)
4. How does the data reconstruction cost of an application affect its performance? (§7.4)

Setup. We conducted experiments on one server that equips a 48-core Intel Xeon Gold 6252 CPU and 128 GiB memory. The server ran Ubuntu 20.04 with Linux 5.14. In line with prior work [37], we enable hyperthreading, but disable dynamic CPU frequency scaling, transparent huge pages, and kernel mitigations for transient execution attacks. For interactive services (*e.g.*, SocialNet), we use a separate server to generate load, which connects to the application server via a 10 GbE network. For all four applications, we generated requests with Zipfian distribution, consistent with the study of real datacenter workloads [9].

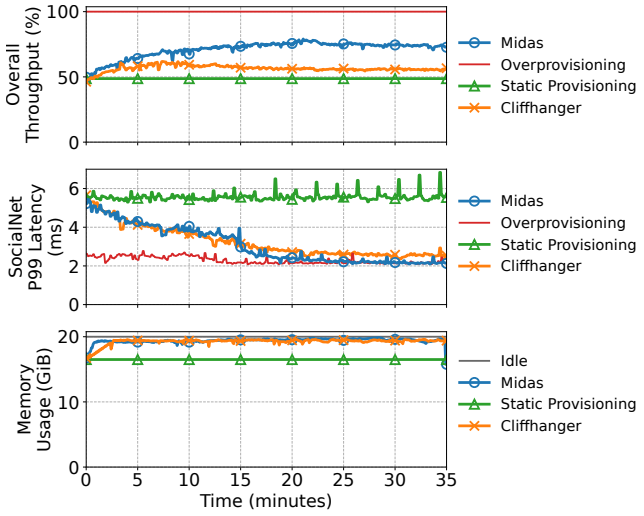


Figure 6: When co-running four applications with 20 GiB idle memory, Midas dynamically coordinates their soft memory budgets and reaches an equilibrium in around 20 minutes. Overall, it harvests 19.6 GiB idle memory as soft memory and achieves 75.0% of the ideal throughput (measured by overprovisioning soft memory for all applications regardless of the 20 GiB total budget constraint).

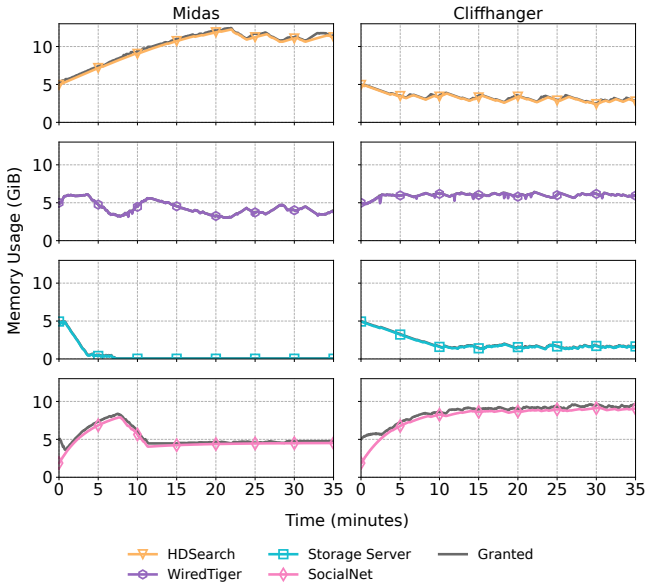


Figure 7: Midas and Cliffhanger converge to different allocations of soft memory between applications because of fundamental differences in their coordination policies.

7.1 Coordinating Soft Memory

In this experiment, we investigated whether Midas can judiciously coordinate soft memory usage among applications to optimize overall performance.

We provisioned the server with 20 GiB idle memory and co-ran all four applications (§6) using Midas. Initially, all applications start with the same amount of soft memory (*i.e.*, 5 GiB), but Midas will dynamically adjust it. SocialNet has 12 loosely-coupled microservices and we start by evenly splitting the 5 GiB budget across them. We measured the overall throughput (defined as the average of all applications' throughput normalized to their ideal throughput) and the soft memory usage. We compared Midas with three different baselines. The first baseline *overprovisions* soft memory for each application to cache all of possible soft state. This leads to a 67.5 GiB soft memory usage that is impossible to achieve under 20 GiB idle memory; thus, this represents the ideal throughput. The second baseline limits itself to the 20 GiB soft memory budget and *statically partitions* it across four applications in an even manner (*i.e.*, each application gets 5 GiB soft memory). The third baseline is *Cliffhanger* [13]. Similar to Midas, it dynamically coordinates soft memory among applications. However, it adopts a different coordination policy of maximizing the global cache hit rate as opposed to maximizing the overall performance utility. As the original version of Cliffhanger only supports Memcached, we emulated Cliffhanger by implementing its coordination policy atop Midas.

A good result for Midas would show that it quickly reaches an equilibrium by judiciously coordinating soft memory usage among applications and achieves good overall throughput close to the ideal throughput (of the overprovisioning baseline). In contrast, the overall throughput of the static provisioning baseline should be suboptimal, as it equally treats all applications and fails to prioritize the soft memory need of applications that can benefit the most. On the contrary, Cliffhanger does coordinate soft memory among applications, but it optimizes for the overall cache hit rate which does not guarantee optimal overall performance (§4.3.2). Therefore, we expect Cliffhanger to achieve overall throughput better than the static baseline but worse than Midas.

Figure 6 shows the results. The top figure presents the overall throughput of four systems normalized to the ideal value. The bottom figure presents soft memory usage; we leave out the usage of the overprovisioning baseline as it is much higher (67.5 GiB) than the amount of idle memory (20 GiB). Midas's overall throughput converges in around 20 minutes and achieves 75.0% of the ideal throughput by harvesting 98.0% idle memory. It also reduces SocialNet's 99th percentile latency by 58.4% from 5.5ms to 2.3ms. In contrast, the static provisioning baseline only achieves 48.7% of the ideal throughput and fails to improve SocialNet's tail latency due to the lack of coordination. It also uses 3.1 GiB less soft memory than Midas as some microservices of SocialNet fail to fully use their statically-provisioned soft memory budgets due to small soft memory footprints. Cliffhanger uses a similar amount of soft memory to Midas. Due to its coordination policy, it converges on the overall cache hit rate (not shown due to the space constraint) but oscillates in terms of the overall throughput. Therefore, it only achieves 56.0% throughput on average.

Figure 7 presents the per-application soft memory usage of Midas and Cliffhanger. For each application, the gray line

represents the soft memory budget it receives, while the colorful line represents the amount of soft memory it uses. Because of the difference in their coordination policies, Midas and Cliffhanger make very different allocations of soft memory between applications except for the storage server. For example, since it is time-consuming to reconstruct HDSearch's objects (as it involves recomputing the feature vectors of images), Midas scales up HDSearch's soft memory to cache more objects. However, since HDSearch has a relatively low request skewness (compared to other applications) and consequently a lower cache utility (in terms of hit rate), Cliffhanger deprioritizes it by scaling down its soft memory, significantly impacting its performance (and therefore the overall performance).

In summary, the experiment demonstrates that Midas can efficiently utilize available memory as soft memory and judiciously coordinate soft memory among applications, achieving high overall performance close to the ideal one that requires 3.4× more memory.

7.2 Harvesting Available Idle Memory

In this experiment, we investigated whether Midas can quickly and reactively harvest additional idle memory—whenever it is available—to improve memory utilization and application performance.

We ran an application using Midas and dynamically added idle memory to the server. A good result for Midas would show that it quickly detects any new idle memory and reactively grants it to SocialNet as additional soft memory to improve performance. Additionally, we expect that the marginal benefit decreases as SocialNet uses more soft memory and caches more hot items.

Figure 8 presents the results of SocialNet. The results of other applications show similar trends and can be found in Appendix A. Initially, the server has 2 GiB idle memory (the dark gray line). With Midas, SocialNet fully utilizes them as soft memory (the blue line) and achieves 13 MOPS throughput (the pink line). At $t = 5$min, we added 4 GiB more idle memory to the server. Midas immediately detects this change and rapidly ramps up its soft memory usage; it only takes around 3 minutes for SocialNet to reach a new steady state. Benefiting from more soft memory, SocialNet's throughput increases by 46% from 13 MOPS to 19 MOPS, and its 99th percentile latency decreases by 27% from 5.5ms to 4ms (the light brown line). At $t = 15$min, we again added 4 GiB more idle memory. This time we observed a reduced marginal benefit as SocialNet has already cached most hot items; it takes 15 minutes to reach a new equilibrium (*i.e.*, 8.5 GiB soft memory usage) and yields a 43% improvement of 99th percentile latency (from 4ms to 2.3ms).

In summary, these results highlight Midas can quickly detect idle memory and reactively scale up its soft memory usage to improve memory utilization and application performance.

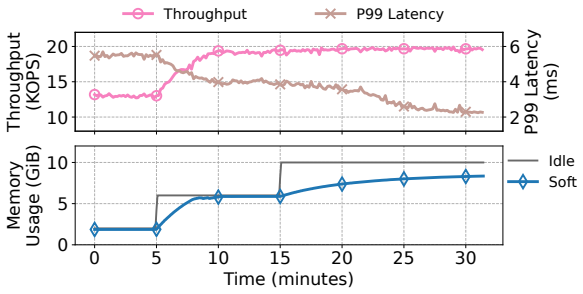


Figure 8: With Midas, SocialNet effectively harvests additional idle memory by scaling up its soft memory usage, improving both throughput and tail latency.

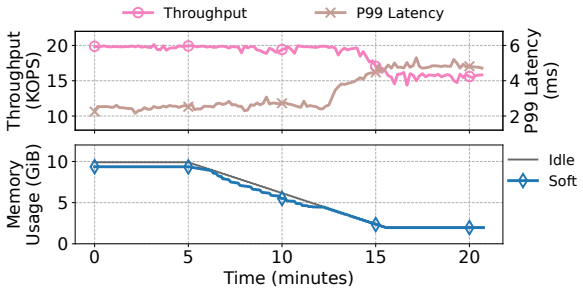


Figure 9: Under moderate memory pressure ($t = 5$min-15min), Midas is able to reactively scale down SocialNet's soft memory usage to avoid running out of memory with moderate performance impact.

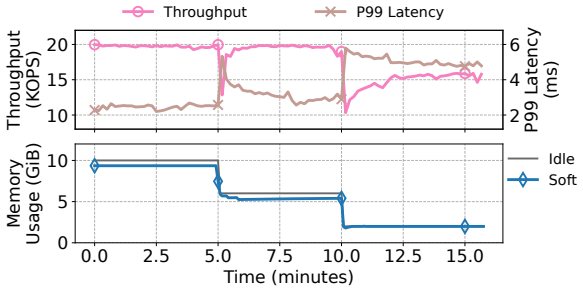


Figure 10: Midas is able to avoid out-of-memory killing even under extreme memory pressure ($t = 5$min and $t = 10$min). SocialNet experiences brief throughput collapses and tail latency spikes but quickly recovers to normal once the pressure is finished.

7.3 Reacting to Memory Pressure

In this experiment, we investigated whether Midas can quickly react to memory pressure to avoid out-of-memory killing and studied its impact on application performance.

Similar to §7.2, we ran SocialNet using Midas, but dynamically decreased the server's idle memory with a colocated memory antagonist. We measured the impact on SocialNet's soft memory usage and performance.

Under moderate memory pressure, ideally, Midas's global coordinator should reactively unmap free soft memory segments while Midas's evacuator should be able to replenish them (by evicting cold objects and evacuating hot objects) to match the coordinator's unmapping rate. A good result for Midas would show that SocialNet's throughput degrades gradu-

ally and mildly as the pressure persists, since Midas prioritizes the eviction of cold and dead objects over hot objects.

Under intense memory pressure, we expect the coordinator to also unmap the used soft memory segments as the evacuator cannot keep up with the high unmapping rate. In this case, SocialNet may experience a sudden throughput collapse due to the loss of hot objects. However, a good result for Midas would show that SocialNet is still able to operate without experiencing any out-of-memory killing. In addition, immediately after the pressure is finished, SocialNet's performance should be able to recover to normal by reconstructing back hotter objects and evicting colder objects.

Figure 9 presents the results under moderate memory pressure. Initially, the server has 10 GiB soft memory. The application uses around 9.6 GiB of it as soft memory and achieves around 20 MOPS throughput and 2.3ms 99th percentile latency. At $t = 5$min, the memory antagonist starts to allocate 8 GiB more memory at a moderate rate of 0.8 GiB/min, resulting in the decrease of idle memory until $t = 15$min. As shown by the bottom figure, Midas is able to reactively scale down SocialNet's soft memory usage through reclamation to avoid running out of memory. As shown by the top figure, SocialNet's throughput and 99th percentile latency remain unaffected in the beginning, as Midas prioritizes the reclamation of cold soft memory. After running below 5 GiB idle memory, SocialNet experiences a mild throughput drop and latency increase, as Midas starts to reclaim hotter soft memory.

Figure 10 presents the results under intense memory pressure. In this case, the antagonist allocates memory as fast as Linux permits (7 GiB/s), making it an extremely challenging case to handle. Despite the high rate, Midas is still able to avoid out-of-memory killing by rapidly scaling down SocialNet's soft memory usage. In this case, Midas has to unmap the used soft memory segments, inevitably causing brief throughput collapses and latency spikes (at $t = 5$min and $t = 10$min). However, once the memory pressure is finished, SocialNet's throughput and latency quickly recovers to the normal level, consistent with the numbers reported in Figure 8 and 9.

The results of other applications also show similar trends (see Appendix B). In summary, these results demonstrate that Midas can always quickly react to memory pressure to avoid out-of-memory killing while maintaining good application performance whenever it is possible.

7.4 Design Drill-Down

Soft Pointer Dereference Cost. We measured the latency of dereferencing a soft pointer and compared it to the latency of dereferencing an ordinary C++ `unique_ptr`, when the pointer and data pointed to are originally in memory (*i.e.*, not in CPU's cache). Table 2 shows the results of small objects (32 B) and large objects (4 MiB), and Appendix D has more results of other object sizes.

For small objects that fit into CPU's cache line (Table 2a), Midas is able to deliver comparable read latency as its extra object copying overhead is negligible. Midas achieves higher write latency (< 200 cycles) as it has to additionally update the metadata in the object header.

[*read*|*write*] Latency (cycles)	Average *read/write*	Median *read/write*	P90 *read/write*
C++ `unique_ptr`	367 / 199	382 / 176	510 / 332
`SoftUniquePtr`	400 / 393	370 / 368	516 / 500

(a) Small objects (32 B).

[*read*|*write*] Latency (Mcycles)	Average *read/write*	Median *read/write*	P90 *read/write*
C++ `unique_ptr`	0.97 / 1.39	0.94 / 1.36	0.99 / 1.37
`SoftUniquePtr`	1.77 / 1.15	1.75 / 1.14	1.77 / 1.18

(b) Large objects (4 MiB).

Table 2: Midas' soft pointer only adds moderate dereferencing cost compared to C++'s ordinary smart pointer.

Live Object Ratio		10%	30%	50%	70%	90%
Reclamation Tput. (MiB/s)	Cooperative	312.5	243.1	173.6	104.2	34.7
	Direct	8268.5				

Table 3: Midas's cooperative reclamation reclaims memory at the throughput of 35 MiB/s-313 MiB/s, depending on the live object ratio of soft memory. Midas's direct reclamation trades off reclamation quality for faster speed; it achieves a throughput of 8269 MiB/s, exceeding the rate at which the Linux kernel can allocate memory.

For large objects (Table 2b), Midas achieves ≈800K cycles (82%) higher read latency since now the additional object copying happens in memory (rather than in CPU's cache). However, Midas achieves lower write latency than `unique_ptr` thanks to its optimized memory copy implementation.

Memory Reclamation Speed. We measured Midas's memory reclamation throughput using a synthetic microbenchmark. Under moderate memory pressure, the coordinator reclaims memory with the cooperation from the runtime (Figure 9); we refer to it as *cooperative reclamation*. Under severe memory pressure, the coordinator directly unmaps soft memory segments (Figure 10); we refer to it as *direct reclamation*.

Table 3 presents the throughput of both reclamation methods. The speed of cooperative reclamation depends on the live object ratio of soft memory; the lower the live ratio, the easier to make room by compacting hot objects, thereby yielding faster reclamation speed. It achieves a throughput of 313 MiB/s under 10% live ratio and 35 MiB/s under 90% live ratio. To handle intense memory pressure, direct reclamation trades off reclamation quality for faster reclamation speed; it achieves a significantly higher throughput of 8269 MiB/s, unrelated to the live object ratio. This exceeds the rate at which the Linux kernel can allocate memory (7-8 GiB/s measured in our machine), therefore Midas can always safely harvest server's idle memory without leading to OOM killing.

Performance Impact of Data Reconstruction. To examine the performance impact of using soft memory, we conducted

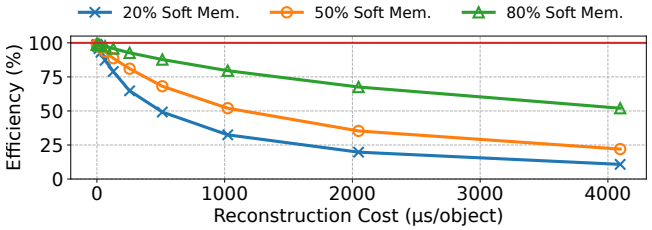


Figure 11: Midas's efficiency (*y*-axis) as a function of data reconstruction cost normalized to the ideal throughput of caching all soft state. Midas's efficiency increases as the reconstruction cost decreases, delivering >80% efficiency for applications with <1024 µs/object reconstruction cost when caching 80% of soft state.

an experiment using a synthetic application; we measured its performance with varying data reconstruction costs under different soft memory ratios (*i.e.*, the ratio of cached soft state). Intuitively, the cheaper the data reconstruction, the lighter the performance impact it incurs.

Figure 11 shows the result. When the soft memory ratio is 20%, Midas is able to deliver >80% efficiency when the reconstruction cost is <128 µs/object. When the soft memory ratio is higher, Midas can tolerate a higher cost as reconstruction happens less frequently; thus, it is able to provide >80% efficiency for applications with <256 µs/object reconstruction cost under 50% memory ratio and <1024 µs/object under 80% memory. This suggests that Midas can still achieve high performance with moderate data reconstruction costs.

8 Related Work

Resource Harvesting and Deflation. Datacenters today suffer from low resource utilization [6, 17, 50]. To make use of vacant resources, major cloud providers now offer spot VMs [5, 20, 31], which run at a low priority and get evicted under resource pressure. Others propose new VM designs to gracefully adjust VMs' resource usage. Harvest VM is a new type of VM that grows and shrinks according to the amount of unallocated resources at its underlying server, including CPU [6], memory [17], and storage [38]. Similarly, deflatable VM [45] codesigns the hypervisor, VM, and the application to reclaim resources from applications under memory pressure. These approaches focus on VMs only, and take minutes to re-configure a VM to release resources.

Resource Disaggregation and Remote Memory. Resource disaggregation and remote memory systems are another trending approach for improving utilization, thanks to faster datacenter networking [19, 26, 29]. Their key idea is to break the server hardware boundary with a fast network interconnection to exploit stranded resources on a remote server. Various systems have established the viability of disaggregated storage [22, 24], accelerators [35, 51], and memory [4, 21, 44, 54]. While some provide remote memory transparently via OS paging, it is also possible to use a library-based approach that modifies the application to bypass the OS. AIFM [40] proposes remote-able data structures to build remote-memory-aware applications. Semeru [52], Mako [28], and MemLiner [53] co-design the JVM with the kernel to offer transparent remote memory for Java programs. Like Midas, these systems adopt customized pointer formats for their remote-able objects. Unlike Midas, they do not consider the unmap-and-reconstruct semantics and suffer from swapping or out-of-memory killing under intense memory pressure.

Cache Services. Improving cache performance is important to datacenter applications, especially in a shared setting [10, 36]. Fairride [36] and RobinHood [10] provide fair and latency-aware cache-sharing policies, and CliffHanger [13] uses a hill climbing method to incrementally optimize cache allocation across applications. Memshare [14] further improves the cache partitioning with a log-structured allocator for higher hit rates. However, existing cache service systems still rely on static memory allocation, and cannot efficiently use idle memory. CacheLib [9] provides a library-based approach for caching, but it again relies on static provisioning and lacks global coordination, hindering its ability to manage memory across multiple applications.

Cooperative Memory Revocation. In parallel with our work, researchers are also exploring the benefits of soft state by managing it at the application level [16]. Midas instead uses kernel coordination and unmap-and-reconstruct semantics, which enables it to reclaim pages even if applications do not cooperate or are slow to respond. This makes it possible to react to severe memory pressure without running out of memory.

9 Conclusion

In this paper, we presented Midas, a system that efficiently and safely harvests idle memory to store the soft state that is most beneficial to each application, improving both memory utilization and application performance. Midas provides familiar high-level programming abstractions and maximizes overall performance through coordination between an application-integrated runtime and a global coordinator. Our evaluation demonstrates that Midas is able to effectively use soft memory to achieve near-optimal performance and can respond to extreme memory pressure fast enough to avoid running out of memory.

Acknowledgement

We thank the anonymous reviewers for their valuable and thorough comments. We are grateful to our shepherd Sanidhya Kashyap for his feedback. This work is supported by NSF grants CNS-1907352, CNS-2007737, CNS-2006437, CNS-2128653, CNS-2106838, CCF-1764077, CHS-1956322, CCF-1460325, CCF-2106404, and CNS-2104398, an Amazon Ph.D. fellowship, a gift from Amazon, a contract from Samsung, and a contract from Cisco.

References

[1] Amazon elasticache. https://aws.amazon.com/elasticache/, 2023.

[2] The redis database. https://redis.com/, 2023.

[3] Storage performance development kit. https://spdk.io, 2023.

[4] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can far memory improve job throughput? In *EuroSys*, 2020.

[5] Amazon Elastic Compute Cloud. Amazon ec2 spot instances. https://aws.amazon.com/ec2/spot, 2022.

[6] P. Ambati, I. Goiri, F. Frujeri, A. Gun, K. Wang, B. Dolan, B. Corell, S. Pasupuleti, T. Moscibroda, S. Elnikety, M. Fontoura, and R. Bianchini. Providing SLOs for Resource-Harvesting VMs in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751. USENIX Association, Nov. 2020.

[7] D. Ardelean, A. Diwan, and C. Erdman. Performance analysis of cloud applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 405–417, Renton, WA, Apr. 2018. USENIX Association.

[8] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[9] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768. USENIX Association, Nov. 2020.

[10] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter. RobinHood: Tail latency aware caching – dynamic reallocation from Cache-Rich to Cache-Poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 195–212, Carlsbad, CA, Oct. 2018. USENIX Association.

[11] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site reliability engineering: How Google runs production systems*. " O'Reilly Media, Inc.", 2016.

[12] M. Calder, R. Gao, M. Schröder, R. Stewart, J. Padhye, R. Mahajan, G. Ananthanarayanan, and E. Katz-Bassett. Odin: Microsoft's scalable Fault-Tolerant CDN measurement system. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 501–517, Renton, WA, Apr. 2018. USENIX Association.

[13] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, Santa Clara, CA, Mar. 2016. USENIX Association.

[14] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, Santa Clara, CA, July 2017. USENIX Association.

[15] F. Corbató. *A Paging Experiment With the MULTICS System*. Project MAC. Massachusetts Institute of Technology, 1968.

[16] M. Frisella, S. L. Sanchez, and M. Schwarzkopf. Towards increased datacenter efficiency with soft memory. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 127–134, New York, NY, USA, 2023. Association for Computing Machinery.

[17] A. Fuerst, S. Novaković, I. n. Goiri, G. I. Chaudhry, P. Sharma, K. Arya, K. Broas, E. Bak, M. Iyigun, and R. Bianchini. Memory-harvesting vms in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 583–594, New York, NY, USA, 2022. Association for Computing Machinery.

[18] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[19] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, pages 249–264, 2016.

[20] Google Cloud. Preemptible vm instances. https://cloud.google.com/compute/docs/instances/preemptible, 2022.

[21] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *NSDI*, pages 649–667, 2017.

[22] J. Hwang, Q. Cai, A. Tang, and R. Agarwal. TCP ≈ RDMA: CPU-efficient remote storage access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 127–140, Santa Clara, CA, Feb. 2020. USENIX Association.

[23] Jacob Leverich. Mutilate: High-performance memcached load generator. https://github.com/leverich/mutilate, 2023.

[24] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote flash ≈ local flash. In *ASPLOS*, pages 345–359, 2017.

[25] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, T. Duerig, and V. Ferrari. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *IJCV*, 2020.

[26] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. First-generation memory disaggregation for cloud platforms, 2022.

[27] C. Lu, K. Ye, G. Xu, C. Xu, and T. Bai. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *Big Data*, pages 2884 – 2892, 2017.

[28] H. Ma, S. Liu, C. Wang, Y. Qiao, M. D. Bond, S. M. Blackburn, M. Kim, and G. H. Xu. Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *PLDI*, 2022.

[29] Mellanox. RDMA aware programming manual (rev. 1.7). https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[30] Memcached. A distributed memory object caching system. http://memcached.org, 2020.

[31] Microsoft Azure. Azure spot virtual machines. https://azure.microsoft.com/en-us/pricing/spot, 2022.

[32] MongoDB. https://www.mongodb.com/, 2022.

[33] MongoDB. Wiredtiger storage engine. https://www.mongodb.com/docs/manual/core/wiredtiger/, 2023.

[34] U. Naseer and T. A. Benson. Configanator: A data-driven approach to improving CDN performance. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1135–1158, Renton, WA, Apr. 2022. USENIX Association.

[35] Nvidia. Virtual gpu (vgpu) | nvidia. https://www.nvidia.com/en-us/data-center/virtual-solutions/.

[36] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica. FairRide: Near-Optimal, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 393–406, Santa Clara, CA, Mar. 2016. USENIX Association.

[37] Y. Qiao, C. Wang, Z. Ruan, A. Beley, Y. Zhang, M. Kim, and G. H. Xu. Hermit: Low-latency, high-throughput, and transparent remote memory via feedback-directed asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Apr. 2023.

[38] B. Reidys, J. Sun, A. Badam, S. Noghabi, and J. Huang. BlockFlex: Enabling storage harvesting with Software-Defined flash in modern cloud platforms. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 17–33, Carlsbad, CA, July 2022. USENIX Association.

[39] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[40] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332. USENIX Association, Nov. 2020.

[41] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured memory for DRAM-based storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 1–16, Santa Clara, CA, Feb. 2014. USENIX Association.

[42] S. J. Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[43] A. Shalita, B. Karrer, I. Kabiljo, A. Sharma, A. Presta, A. Adcock, H. Kllapi, and M. Stumm. Social hash: an assignment framework for optimizing distributed systems operations on social networks. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 455–468, 2016.

[44] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, pages 69–87, 2018.

[45] P. Sharma, A. Ali-Eldin, and P. Shenoy. Resource deflation: A new approach for transient resource reclamation. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[46] A. Sriraman and T. F. Wenisch. μsuite: A benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12, 2018.

[47] K. Tati and G. M. Voelker. Shortcuts: Using soft state to improve dht routing. In C.-H. Chi, M. van Steen, and C. Wills, editors, *Web Content Caching and Distribution*, page 44–62, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[48] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. Borg: The next generation. In *EuroSys*, 2020.

[49] Twitter Inc. Processing billions of events in real time at twitter. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2021/processing-billions-of-events-in-real-time-at-twitter-, 2021.

[50] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[51] L. Vilanova, L. Maudlej, S. Bergman, T. Miemietz, M. Hille, N. Asmussen, M. Roitzsch, H. Härtig, and M. Silberstein. Slashing the disaggregation tax in heterogeneous data centers with fractos. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 352–367, New York, NY, USA, 2022. Association for Computing Machinery.

[52] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280. USENIX Association, Nov. 2020.

[53] C. Wang, H. Ma, S. Liu, Y. Qiao, J. Eyolfson, C. Navasca, S. Lu, and G. H. Xu. MemLiner: Lining up tracing and application for a far-memory-friendly runtime. In *OSDI*, 2022.

[54] C. Wang, Y. Qiao, H. Ma, S. Liu, Y. Zhang, W. Chen, R. Netravali, M. Kim, and G. H. Xu. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. In *NSDI*, 2023.

A Harvesting Available Idle Memory

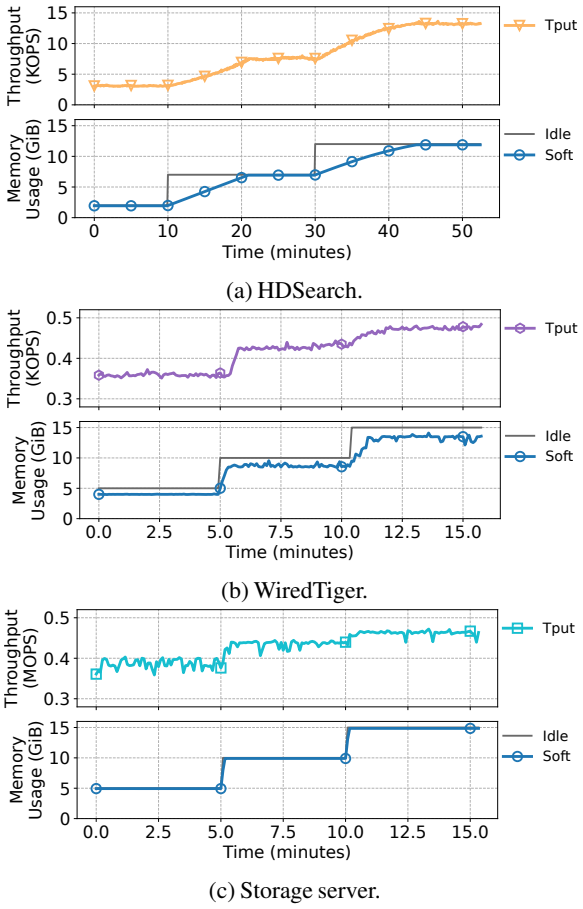


Figure 12: With Midas, applications can reactively scale up their soft memory usage to harvest additional idle memory and improve performance.

In this section, we evaluated the other three applications individually under the same setting as in §7.2 to show how Midas can harvest available idle memory to improve memory utilization and application performance. Figure 12 presents the results. Similar to Figure 8, Midas can quickly detect any idle memory and reactively grant it to the application to improve its performance. As applications expose different allocation speeds and utilities of their soft state, the average time to scale up the soft memory usage as well as the performance gain also varies across applications. HDSearch takes longer to fully utilize the additional soft memory because it needs expensive GPU computations to re-construct a cache-missed object. It also enjoys higher throughput increases by memoizing computation results with additional soft memory. On the contrary, both WiredTiger and storage server can quickly utilize all additional soft memory, but they only get marginal performance improvement after caching most hot blocks at $t = 10$min.

B Reacting to Memory Pressure

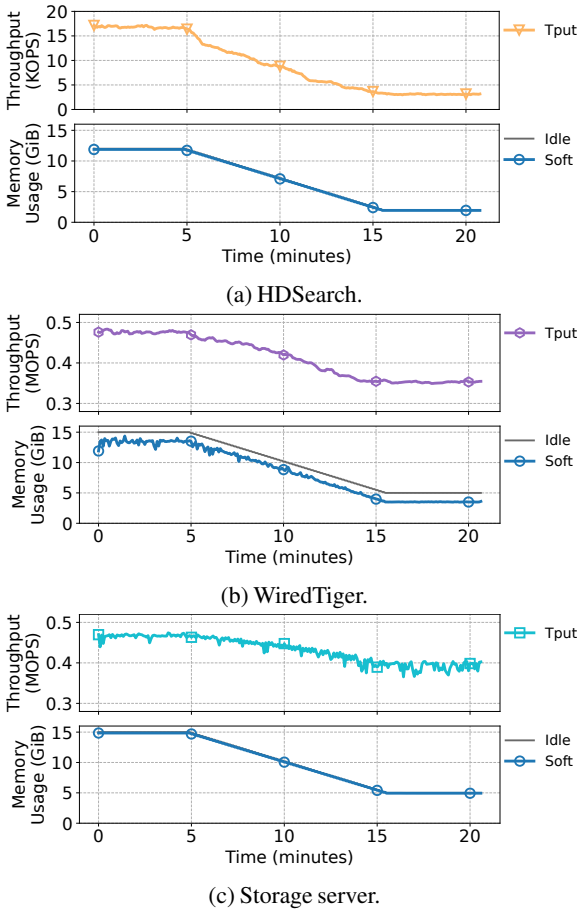
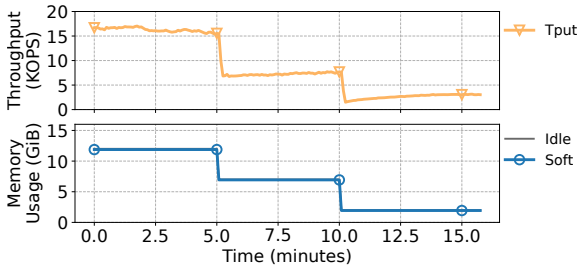


Figure 13: Under moderate memory pressure (t = 5min-15min), Midas is able to reactively scale down applications' soft memory usage to avoid running out of memory while minimizing its impact on their throughput.

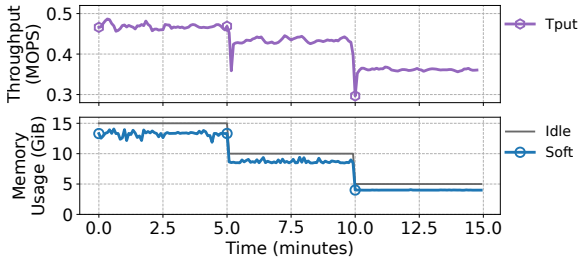
In this section, we further investigated whether Midas can quickly react to memory pressure by running the other three applications individually under the same setting as in §7.3. Similarly, we measured memory utilization and application throughput under moderate memory pressure and intense memory pressure, respectively.

Figure 13 presents the results of each individual application under moderate memory pressure. Similar to Figure 9, At t = 5min, the memory antagonist starts to allocate 10 GiB more memory with a moderate rate of 1.0 GiB/min, leading to the decrease of idle memory until t = 15min. As shown by the bottom figure, for all three applications, Midas reactively scaled down their soft memory usage and avoided out-of-memory killing. As shown by the top figure, application throughput drops gradually and mildly as the reclamation goes on and never experiences any severe disruption.

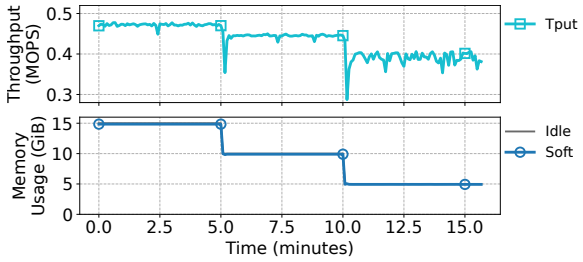
Figure 14 shows the results of each individual application under intense memory pressure. Similar to Figure 10, the



(a) HDSearch.



(b) WiredTiger.



(c) Storage server.

Figure 14: Midas is able to avoid out-of-memory killing even under extreme memory pressure (t = 5min and t = 10min). The victim application experiences brief throughput collapses but quickly recovers to normal once the pressure is finished.

memory antagonist intensely allocated 5 GiB memory at $t = 5$min and $t = 10$min. Despite the high memory allocation rate, Midas is still able to rapidly reclaim application's soft memory and avoid running out of memory. Because Midas has to unmap the used soft memory segments in this case, both WiredTiger and Storage server experience brief throughput collapses. However, once the memory pressure is finished, their throughput can quickly recover to the normal level, consistent with the numbers reported in Figure 12 and 13. HDSearch has a relatively lower request rate, therefore it is more tolerable to the enforced soft memory unmapping and does not experience severe throughput collapse at all.

C SocialNet Microservices Memory Usage

We have reported the overall soft memory usage of SocialNet in Figure 7. Among SocialNet's 12 microservices, two microservices used the most soft memory, namely UserTimeline and PostStorage. Figure 15 reported the detailed memory usage for each of them.

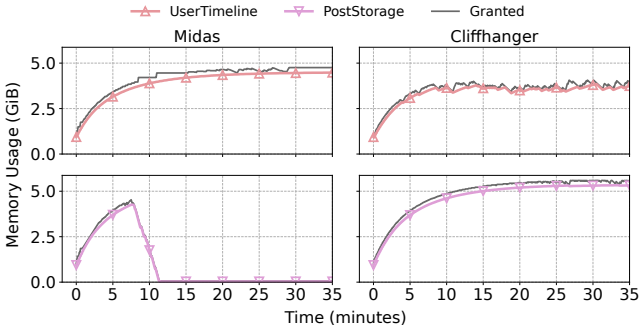


Figure 15: Memory usage of two major microservices in SocialNet. Midas dynamically coordinates memory between the microservices to achieve high memory utilization and optimal performance for SocialNet.

UserTimeline is the frontend microservice that handles user requests. It fetches a group of user posts from the storage backend and composes them as a timeline webpage. It caches composed user timelines in soft memory to reduce backend storage accesses. PostStorage is the backend database microservice that stores user posts. It handles post requests from UserTimeline with MongoDB and caches hot posts using soft memory. As shown in 15, at first, Midas reactively grants soft memory to both microservices to quickly recover SocialNet's throughput and latency. As UserTimeline gets more soft memory, it caches more hot timelines and consequently reduces its request rate to PostStorage. At $t = 8$min, Midas's profiling reveals that PostStorage is no longer frequently accessed and therefore has relatively low cache utility, so Midas reactively scales down PostStorage's soft memory. At $t = 20$min, SocialNet reaches a new equilibrium, where UserTimeline consumes most of the soft memory budget and PostStorage only keeps a small portion of soft memory. Cliffhanger, in contrast, only profiles the cache hit rate of each microservice regardless of their cache access rate and performance sensitivity. Therefore, it continuously grants soft memory to PostStorage, resulting in overprovisioning soft memory to SocialNet.

D Soft Pointer Dereference Cost

In this section, we reported the detailed results of soft pointer dereference cost when reading and writing large objects in various sizes and compared them with the cost of dereferencing an ordinary C++ `unique_ptr`. Similar to Table 2, we measured the P90 latency and throughput of accessing large objects (Figure 16) in various sizes.

As shown in Figure 16, reading a large object whose size is smaller than 512 KiB with Midas soft pointer has similar latency and throughput compared to dereferencing a C++ `unique_ptr`, although dereferencing a soft pointer incurs an additional memory copy. This is because the object and its

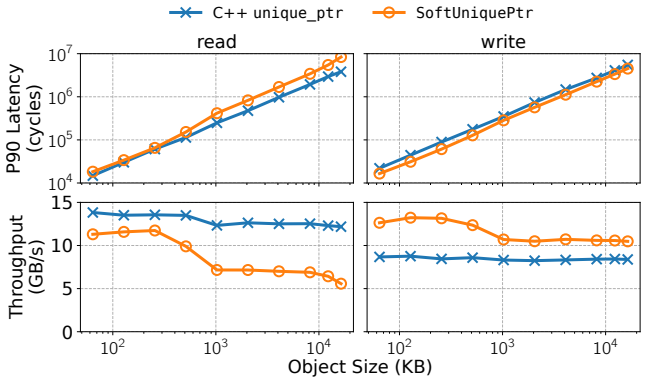


Figure 16: Midas's soft pointer achieves similar performance compared to C++'s ordinary smart pointer when objects can fit into CPU L2 cache, and it only adds moderate dereferencing cost otherwise.

copy can both fit into the CPU L2 cache and hence the second copy is fast. For all object sizes, soft pointer offers lower write latency and higher write throughput than `unique_ptr` thanks to Midas's optimized memory copy implementation.

Efficient Exposure of Partial Failure Bugs in Distributed Systems with Inferred Abstract States

Haoze Wu[†] Jia Pan[†] Peng Huang[‡]

Johns Hopkins University[†] *University of Michigan*[‡]

Abstract

Many distributed system failures, especially the notorious partial service failures, are caused by bugs that are only triggered by subtle faults at rare timing. Existing testing is inefficient in exposing such bugs. This paper presents LEGOLAS, a fault injection testing framework designed to address this gap. To precisely simulate subtle faults, LEGOLAS statically analyzes the system code and instruments hooks within a system. To efficiently explore numerous faults, LEGOLAS introduces a novel notion of *abstract states* and *automatically* infers abstract states from code. During testing, LEGOLAS designs an algorithm that leverages the inferred abstract states to make careful fault injection decisions. We applied LEGOLAS on the latest releases of six popular, extensively tested distributed systems. LEGOLAS found 20 new bugs that result in partial service failures.

1 Introduction

Deployed distributed systems frequently encounter faults in the underlying hardware and dependent software. While these systems are generally fault-tolerant, an unexpected fault can still expose bugs. Indeed, real-world distributed system outages are often triggered by some fault events [6, 14, 16, 34].

Fault injection testing, also known as chaos engineering [48], has gained popularity to find fault-induced bugs early. Various solutions are developed to inject common faults such as crashes [3,15,40], disk faults [13,26], and network partitions [2, 3,27]. Despite the progress, many complex fault-induced bugs remain hidden in existing testing and cause failures after deployment. These bugs share several characteristics.

First, they cause puzzling symptoms where the services seem to work but are partially broken, which are notorious in production distributed systems [10, 12, 22, 23, 36]. Figure 1 shows a real failure from a ZooKeeper deployment. The clients experienced timeouts in `create` requests, but `get` requests still succeeded. Pinging the leader also showed that it was alive. As another example, users reported [5, 9, 25] that their Kafka cluster occasionally experienced partial breakdown, and one broker could not return to an *in-sync* status.

Second, these bugs are triggered under subtle faulty conditions, such as a network error that only affects some operations but not others [1, 36], transient slowness [17, 21], or microburst [28]. In the aforementioned ZooKeeper example,

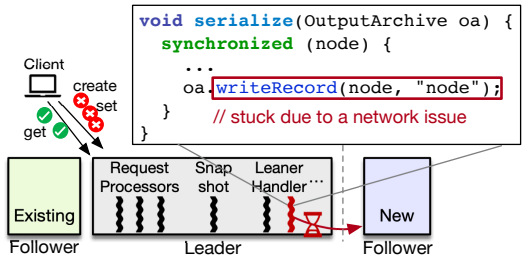


Figure 1: A real ZooKeeper production incident [46] triggered by a partial network fault, which caused the `writeRecord` operation to be stuck while holding a lock.

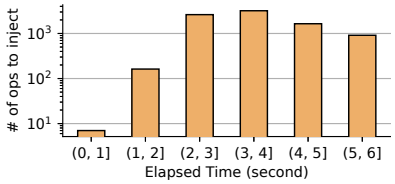


Figure 2: Hundreds to thousands of operations per second are candidates for injecting `IOException` during ZooKeeper's execution.

the buggy code works properly in normal conditions. The failure was only exposed by a partial network fault between a leader and a new follower, and the fault only affects a specific operation (`writeRecord`). The faults can also originate from software and be system-specific, such as a custom exception from an RPC to a remote component (*e.g.*, when the database is overloaded). The Kafka failure example was caused by a custom exception returned from an RPC to the dependent ZooKeeper service. Simulating these fault conditions in testing requires precise control of the fault types and locations.

Third, these bugs require careful choices of when and where the fault occurs. Distributed systems have a large number of fault points (Figure 2), but since these systems are robust, most faults would be tolerated or result in an *expected* failure (*e.g.*, abort on error in reading a file). To expose the ZooKeeper failure, a transient network latency increase must be injected while a new follower is requesting a snapshot from the leader. Injecting the fault at other times or other locations is ineffective. A random injection choice, which is commonly used in existing solutions, will have a high chance of missing the buggy point.

We present LEGOLAS, a fault injection framework designed to efficiently expose the above class of complex fault-induced bugs in large distributed systems. Unlike the practice of

injecting system-agnostic faults externally in the environment or libraries, LEGOLAS uses program analysis to perform fine-grained and system-specific fault injection. It analyzes the fault conditions for each instruction in the code and instruments hooks to precisely simulate subtle faults *within* the system.

With more faults to consider, the problem of a large fault injection space becomes more pronounced.

Our insight is that production bugs occur in unusual conditions—otherwise, existing testing likely has exposed it. Thus, we can *selectively* inject faults by checking if the system reaches an unusual condition. Unfortunately, we do not know beforehand whether a program point is unusual or not. Using *ad-hoc* heuristics, such as only injecting faults when the program is inside a critical section, can miss many failure-inducing conditions. We should still systematically explore the injection choices for generality and completeness.

Based on this insight, LEGOLAS introduces a novel notion of *abstract state* to guide systematic but efficient exploration of the fault injection space. The basic idea is to use system states to *group* injection points. A system's state can be represented by its variables and concrete values. This representation, however, is massive for large systems, and it would make almost all injection points appear in unique groups. For fault injection, we need a more high-level state representation, in which multiple injections likely yield similar effect.

LEGOLAS uses a simple yet novel static analysis that *automatically infers* abstract states from the target system code. The automation is feasible because developers usually leave clear hints in the code about abstract states: the system checks one or more state variables' concrete values in a branch to see if an important condition occurs; if so, it performs some significantly different action. An abstract state thus can indicate that a system enters a unique stage of service, *e.g.*, request parsing, snapshotting, and leader election.

Specifically, LEGOLAS first infers concrete state variables in a system. It then identifies code blocks that are control-dependent on some concrete state variable. It finally creates a mnemonic abstract state variable for each such block, which will be set when the program execution reaches that point.

Leveraging the inferred abstract states, LEGOLAS can efficiently explore the injection space. During testing, the injection hooks LEGOLAS instruments dispatch queries to the LEGOLAS controller. The controller checks the system's current abstract states and decides whether to grant an injection or not. Essentially, LEGOLAS enables *stateful* fault injection.

We design a stateful injection decision algorithm called *budgeted-state-round-robin* (bsrr). Other stateful policies are also feasible, and it is easy to add and switch policies in LEGOLAS. Compared to the straightforward *new-state-only* policy, *bsrr* is robust to tolerate potential inaccuracies in the abstract state analysis. It also reduces biases in injections.

We have built an end-to-end prototype for the LEGOLAS framework, including the static analyzer, fault injection controller, workload driver, and failure checkers.

We apply LEGOLAS to six large distributed systems: Kafka, ZooKeeper, HDFS, HBase, Cassandra, and Flink. LEGOLAS automatically instruments these systems and extracts abstract states without special tuning. We run fault injection experiments on these systems' recent releases. Using the *bsrr* algorithm, LEGOLAS finds 20 new bugs with a median time of 58 minutes. These bugs all cause partial service failure symptoms. We report the bugs to developers. Four reports are marked as critical, fourteen reports are marked as major, and two are marked as normal. Eleven reports have been explicitly confirmed by developers so far. We also compare LEGOLAS (*bsrr*) with state-of-the-art solutions and other policies. The best performing baseline is the *new-state-only* policy with LEGOLAS, which exposes eight bugs. The random injection policy only exposes three bugs in a median of 362 minutes.

In summary, this paper makes the following contributions:

- We propose an approach that uses program analysis to enable customized and fine-grained fault injection.
- We introduce a novel concept of abstract state and a method that automatically infers abstract states from a given system's code. We design a new decision algorithm that leverages the inferred abstracts to guide efficient fault injections for exposing complex bugs that cause partial service failures.
- We build a fault injection framework LEGOLAS and evaluate LEGOLAS on large distributed systems.

2 Overview of LEGOLAS

LEGOLAS is an end-to-end fault injection testing framework for large distributed systems. It aims to efficiently expose fault-induced bugs like the motivating examples.

Scope. Consider a distributed system $\mathcal{S}$ that consists of multiple processes P and provides a range of services R. One definition of a partial failure is that a subset of P are faulty (crash, Byzantine, or gray faults [23]), which may be tolerated and not affect the functionalities of $\mathcal{S}$.

LEGOLAS focuses on exposing partial failures with respect to services, where some $R_f \subset R$ fail to maintain their safety or liveness properties, while other services $R \setminus R_f$ behave as expected. In contrast, in a total failure, all services in R break. An intuitive strategy to uncover partial failures is thus to perturb each service based on their specifications, but this strategy can be difficult to apply with large concrete codebases.

Owing to the modular designs prevalent in large distributed systems, each service is typically implemented by a specific component made up of threads; each process π encompasses disjoint sets of components that provide different services for $\mathcal{S}$. For example, a leader process in ZooKeeper has dedicated request handlers, snapshot manager, quorum messenger, *etc.* If one thread fails, the recovery mechanisms will try to avoid interruptions to the corresponding service. Thus, LEGOLAS is designed to perturb each component (some thread) within π—instead of crashing π outright—by inducing faults to the instructions executed by the component. This allows for a deeper exploration of the potential partial failures in $\mathcal{S}$.

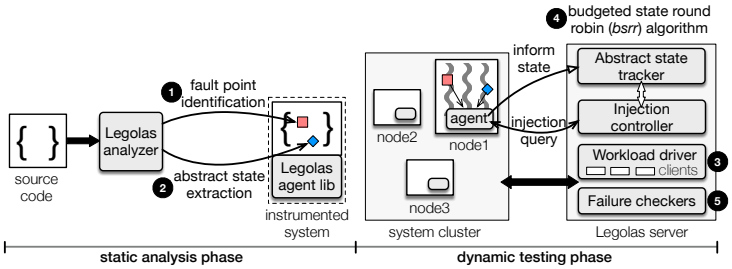


Figure 3: Fault injection workflow with LEGOLAS.

```
InjectionQuery query = new InjectionQuery(serverId,
      threadId, ..., invokedMethodSig, faultIds);
InjectionCommand command = stub.inject(query);
if (command.id == -1) return;  // no injection
/* simulate the decided fault */
...
+ LegolasAgent.inject("org.apache.zookeeper.server.DataTree",
        "serializeNode", 1115,"<org.apache.jute.OutputArchive:
         void writeRecord(...)>", 268, 0, 3);
  outputArchive.writeRecord(node, "node");
```

Figure 4: Injection hook instrumented for code in Figure 1.

Workflow. Figure 3 shows the workflow with LEGOLAS. It takes a system's code as an input, uses static analysis to identify faulty conditions unique to the system, and instruments injection hooks directly within the system (❶) to simulate subtle faults.

In addition, LEGOLAS runs a novel analysis that *automatically* extracts abstract states from the system code (❷). It identifies code locations that may represent an important change in the system's service status, and inserts mnemonic abstract state variables at these locations. LEGOLAS then links a thin *agent* library with the target system.

During testing, LEGOLAS starts a cluster of the instrumented system and runs the workload driver (❸). When a node reaches an injection hook, the embedded agent dispatches an RPC query to the LEGOLAS controller, which decides whether to grant the injection or not. If the agent receives a positive reply, it will simulate the fault inside the node directly.

Importantly, the LEGOLAS server tracks abstract states for each node. When a node enters a new abstract state, the LEGOLAS agent informs the state tracker in the server. The controller leverages the abstract state to make injection decisions. In particular, we design an algorithm called budgeted state round-robin (*bsrr*) (❹). To determine the injection outcome, LEGOLAS runs the failure checkers (❺).

3 Identify and Instrument Injection Points

A widely-used fault injection approach is to introduce node-level faults such as process crashes and network disconnections *externally* in the environment. This approach is suitable for exposing distributed protocol bugs or crash-recovery bugs. However, partial failures are often triggered by subtle faults in the implementations. Existing solutions that inject fine-grained faults focus on the boundaries between an application and libraries or services, and take an *interception* approach. For example, LFI [43] intercepts libc API such as `recv` and returns error codes to applications. Although their injection is more fine-grained (library APIs or service requests) than node-level faults, they miss *internal* errors in a system. Moreover, it is difficult for them to precisely simulate partial faults because they do not directly control the program execution.

To address these issues, LEGOLAS goes deeper—to the program statements inside a system—and takes an *instrumentation* approach. It uses static program analysis to deduce potential faulty conditions for each statement and add injection control points to directly simulate faults *in situ*.

Identify Faulty Conditions. The LEGOLAS analyzer locates each call instruction in the system and examines its invocation target to extract potential fault conditions. A straightforward way is to leverage the method signature. However, two challenges arise. First, a method may internally throw an exception that is not declared in the signature. Some languages also do not enforce or support exception specification in method signature. Second, due to polymorphism and interface, a call site of a method may be *impossible* to encounter an exception declared in the method signature. This is especially problematic with I/O related exceptions. Consider `dump(OutputStream out) throws IOException`, which is declared this way because the argument `out` is an abstract class with `IOException` in its methods' signatures. However, if a call site of `dump` passes a `ByteArrayOutputStream` as an argument, injecting an `IOException` causes an invalid scenario.

To handle the first issue, LEGOLAS inspects the method body and deduces the exceptions. However, it cannot simply collect the exceptions in the throw instructions. This is because the method may have an exception handler that catches the exception. LEGOLAS analyzes the error handlers in the function to determine if an exception may be (i) caught and handled; (ii) caught and re-thrown; (iii) uncaught. Only (ii) and (iii) are treated as the true method-level exceptions.

To address the issue of invalid injection, LEGOLAS designs an inter-procedural, context-sensitive analysis to check call instructions with potential `IOException`. It deduces whether the objects (argument, return, field, class) associated with a call site may come from definition points with known in-memory object types, and ignores the fault if so.

Besides exceptions, the fault condition could also be a delay. All operations could in theory experience some delay. In practice, mild delays are benign and developers need evidence to explain the delay. LEGOLAS by default only considers function calls that involve I/O as delay injection candidates.

Instrument Injection Hooks. For each program point with potential faulty conditions, LEGOLAS instruments an injection hook. LEGOLAS also emits a thin *agent* to link with the target system. At runtime, when an injection hook is reached, the agent creates a query to a controller (Figure 4), which includes the interposed operation, possible fault ids, node id, name and

id of the current thread. If the injection is granted, the agent looks up the fault id, which can be either a delay or some exception. For delay, the agent simply invokes the thread sleep function. We focus on worst-case situations and use 1 minute as the default delay (the `writeRecord` call in Figure 1 can hang for over 15 minutes under default Linux TCP settings). For exception, the agent constructs an exception instance and throws it before the injection hook returns.

Automatically creating an exception instance to throw is a non-trivial task. Some custom exception type includes complex arguments and compositions. LEGOLAS analyzes the constructor and recursively reduces complex arguments to primitive types. It then creates an exception instance by assigning the primitive fields with default values.

Where to add the injection hooks also requires careful considerations. The straightforward way is to instrument each call instruction that LEGOLAS analyzes to possibly encounter a fault. Suppose `foo()` is analyzed to possibly throw `MyError`, but that is only because `foo()` internally calls `bar()`, which can throw that error. If `MyError` is injected at the call sites of `foo()`, we need further explanation of why this exception occurs. For deep call chains, such injections make the exception reasoning difficult and may turn out to be invalid.

To address this issue, LEGOLAS instruments as deep as possible. It identifies faults that *originate* from a method through explicit `throw` statement. It then only instruments calls to either a method that has a non-empty list of such faults, or an external function. If the called method is from an interface or abstract class, LEGOLAS injects at the caller's call sites to handle potential invalid injections.

Benefits. LEGOLAS's approach eases fault simulation without requiring a special environment (e.g., a FUSE-based file system [13, 49]). It also gives precise control to simulate partial faults, *e.g.*, a partial disk failure that only affects a subset of file operations; only some RPCs within certain code region are delayed. It also supports simulating custom errors. While a custom error may be caused by some environment fault, it can be difficult to simulate them with external injection. For example, a method may throw an exception only when all three retries of a connection fail.

4 Abstract State Guided Fault Injection

A key challenge in fault injection for distributed systems is the enormous injection choices (Figure 2). Moreover, only few choices can expose bugs. This is because production distributed systems have extensive fault resilience mechanisms.

Insight. Our insight is that many fault injection attempts are unnecessary because they are testing the same or similar scenarios. Take a ZooKeeper code snippet in Figure 5 as an example. The `SyncRequestProcessor` component is responsible for synchronizing the requests to log files on disks. Suppose we are injecting faults on I/O operations. There are numerous injection points here, including operations inside the called

```
1  public class SyncRequestProcessor extends Thread {
2    public void run() {
3      int logCount = 0;                                   s0
4      while (true) {                                      s1
5        Request si = queuedRequests.take();
6        if (zks.getDB().append(si)) {
7          logCount++;
8          if (logCount > snapCount) {
9            if (snapThd != null && snapThd.isAlive()) {
10             LOG.warn("Too busy to snap, skipping");
11           } else {
12             (snapThd = new Thread(() -> {
13                 zks.takeSnapshot();                     s2
14             })).start();
15           }
16           logCount = 0;
17         }                                               s3
18       }
19       ...
20     }
21   }
22 }
```

Figure 5: The grayed areas are code regions containing I/O operations. The bug in Figure 1 occurs inside a call chain from line 13.

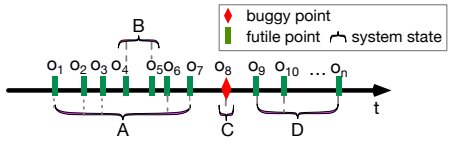


Figure 6: Group the injection points by the state they appear in.

functions. Line 6, which syncs requests to logs, gets executed at each loop iteration, while line 13 only occurs occasionally. With limited testing resources, we may only inject faults on operations inside line because of their frequent occurrences.

Idea. Inspecting the system state for each component can help us make better decisions. For the previous example, we could realize that the system enters a rare state (snapshotting) when it reaches line 13. Operations in this state could be of high interest. Our basic idea is thus to group the injection points based on the underlying system state (Figure 6). Grouping helps avoid being indiscriminate when making injection decisions. The injection points that lie in the same group of state are hypothesized to yield similar outcomes if injected, while the injection points in different groups may yield different outcomes.

However, we do not just focus on rare states, as defining them is subjective. Moreover, the presence of an injection point in a rare state does not imply a bug. Neither does an injection point in a common state guarantee the absence of bugs. For example, a bug may be exposed with a fault occurring inside the `append` call in Figure 5—which is in a common state—when it is executed for the fourth time.

We thus explore the injection space *systematically*. That is, if there were four chances, we try to explore injections in all four states, instead of spending them only in one state.

4.1 State Representation Choices

The next question is *how to define the system state for effective grouping?* Unlike distributed protocols that have specifications, determining the state representation for complex system implementation is not easy. The complete execution states—the program counter, stack traces, and memory snapshot—are

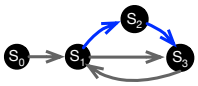


Figure 7: State machine with abstract states for Figure 5. Each state is an abstraction over the concrete state variable `logCount`.

obviously too excessive. A more reasonable representation is to use some key state variables (SV). A value change of these variables then could indicate the system is in a different state. This representation, however, can still be excessive.

Take Figure 5 as an example. If we treat the `logCount` as a state variable (SV), the value is incremented for n times, and each increment is counted as a new state. Using such a representation not only requires frequent state tracking, but also degrades the injection point grouping to be useless.

The key reason is that some values in a concrete state variable do not imply a significant change, at least for fault injection purposes. All the increment-by-one value transitions of `logCount` while `n<=snapCount` indicate the same information about the system, while only the transition of `n>snapCount` indicates something new (starting to snapshot).

Essentially we need a more high-level representation than concrete state variables (SV), which we define as **abstract state variable** (ASV). The intuition behind ASVs is that they represent different *stages of service* in a system. For the example in Figure 5, a natural way to define the abstract state is to divide the execution into four stages—S_0 to S_3. Figure 7 shows the corresponding abstract state machine. This simpler representation can recognize when the system starts to do snapshot (state S_2). In turn, they can effectively group the injection points to make fault injection efficient.

4.2 Infer Abstract State Variables

The LEGOLAS analyzer uses a simple yet novel method to automatically infer ASVs in a system. The feasibility of the automation is based on our insight that developers usually already encode sufficient hints about ASVs. In particular, developers checks one or more state variables (SV) in a branch, and if certain condition occurs, the system performs some action, *i.e.*, `if (func(state_var1, var2, ...)) { do_action1(); }`. From our inspection, in long-lived components, it is a common practice to utilize SV to designate different functionalities at different iterations. For example, the `QuorumPeer` component in ZooKeeper uses a static variable `state` to indicate the node status, which could be `LEADING`, `FOLLOWING`, *etc.* The `QuorumPeer` component then has a `while` loop that does a switch case on this SV to select different functionalities over time.

LEGOLAS first locates all the task-unit classes in the system. These classes are generally threads or workers, such as classes that extend `Thread` or `Runnable` in Java. The analyzer then runs ASV inference on each task-unit class.

Algorithm 1 lists the core algorithm. It starts by inferring the SVs in the code (Line 2). `InferCSV` simply treats all *non-static*, *non-constant* fields defined in a task unit to be SVs.

LEGOLAS then analyzes the main task method of the task unit class, such as the `run()` method of a `Thread`. It finds the

Algorithm 1: Infer abstract state variables

```
1  Function InferASV(task_class):
2      csv_list ← InferCSV(task_class);
3      task_method ← getTaskMethod(task_class);
4      dep_graph ← buildDependence(task_method, csv_list);
5      asv_locations ← [task_method.body().getFirst()];
6      Process(task_method.body(), dep_graph, false);
7  Function Process(instructions, dep_graph, flag):
8      inst ← instructions.begin();
9      hasAction ← false;
10     while inst ≠ instructions.end() do
11         if isBranch(inst) then
12             <cond, blocks, next> ← parseBranch(inst);
13             if dep_graph.contains(cond) then
14                 for block ← blocks do
15                     Process(block.body(), dep_graph, true);
16                 end
17             end
18             inst ← next;
19         else
20             hasAction ← hasAction | isAction(inst);
21             inst ← inst.next();
22         end
23     end
24     if hasAction and flag then
25         asv_locations.add(instructions.begin());
```

basic blocks in the task method that are control dependent on some SV and treats each such basic block as a new ASV.

Specifically, the analyzer iterates through instructions in the task method. Upon a branch instruction, it checks if the branch condition is dependent on some SV (Line 13). This check considers not only direct usage of SV but also indirect data dependence, *i.e.*, a branch condition involving a local variable that gets its value from an SV. Accordingly, the analyzer builds a data dependence graph of the SVs (Line 4). The algorithm then recursively processes the basic blocks control dependent on this branch instruction (Line 15). A system should perform non-trivial actions in an abstract state. Thus, we check if the basic block contains at least one function invocation or an operation that could change a state variable (Line 20).

Once the proper basic blocks are located, the analyzer assigns indexes for them, asv_0, ..., asv_n. The indexes are local to the task class. For each inferred ASV, the analyzer instruments a call to the LEGOLAS agent. At runtime, the agent notifies the LEGOLAS state tracker of the asv_i that is entered, along with the node id, the name and id of the current task.

Note that our ASV is *not* equivalent to conventional control-flow path. We make program paths collapse into more meaningful ones (service stages) that guide fault injection.

Example. Figure 8 shows the ASVs LEGOLAS infers and inserts for the code in Figure 5. The inferred ASV is slightly different from S_2 in the simplified snippet in Figure 7. This is because the `logCount` is a local variable, thus the LEGOLAS analyzer does not treat it as an SV. Another variable `snapThd` is

```
1  LegolasAgent.inform(identityHashCode, ..., 0);          asv0
2  while (true) {
3    Request si = queuedRequests.take();
4    if (request == requestOfDeath) break;                 asv1
5    LegolasAgent.inform(identityHashCode, ..., 1);
6    if (zks.getDB().append(si)) {
7      logCount++;
8      if (logCount > snapCount) {                         asv2
9        if (snapThd != null && snapThd.isAlive()) {
10         LegolasAgent.inform(identityHashCode, ..., 2);
11         LOG.warn("Too busy to snap, skipping");         asv3
12       } else {
13         LegolasAgent.inform(identityHashCode, ..., 3);
14         (snapThd = new Thread(...
15       }
16       logCount = 0;
17 ...
```

Figure 8: The ASVs Legolas infers for Figure 5.

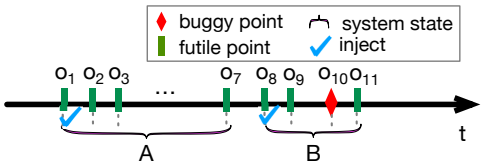


Figure 9: The buggy point may not be the first request in a state.

a non-static field of `SynchRequestProcessor`. Legolas treats it as an SV and infers the asv_3 that represents the snapshot stage. This result is in fact more accurate than using `logCount`, because it infers an additional state (asv_2)—a previous snapshot is ongoing while the snapshot threshold is reached.

Alternative. We also explored other ASV inference methods. For example, we observe that although some function only uses local variables, it can still represent an important system service stage, *e.g.*, handling an event. Defining an ASV at the function entry can be useful. We chose our above inference method for its simplicity. As Section 7 later show, it is general enough to apply on all the popular distributed systems we evaluate and achieve significant performance. It is also feasible to extend Algorithm 1 and analyze the functions called in the task method to extract more thorough stages. However, only analyzing the main task method already provides a good generalization to capture key stages in a component that match the system modularity and design documentations.

4.3 Injection Decision Algorithm

With the inferred abstract states, Legolas enables *stateful* decision policies for efficient fault injection. When the controller receives an injection request from the Legolas agent, the controller checks which abstract state the target system is in at the time of the injection request to make a decision.

A straightforward stateful policy is to grant an injection request only if the system is in a new state, which we call a *new-state-only* policy. While this policy matches the intuition that complex bugs are often only triggered when the system enters an unusual condition, it has several drawbacks. As Figure 9 shows, there can be multiple injection requests from one state, and a buggy point may not be the first request. Indeed, for the ZooKeeper example, even though the bug only appears in the snapshot state, the snapshot function performs several write operations before it reaches the buggy point. This policy also

Algorithm 2: Budgeted state round robin (bsrr) policy

```
Global Vars: Queue<State> rrl, Map<State,Info> visit
/* invoked at start of a fault injection trial                    */
1  Function setupNewTrial():
2    resetIfAllUsed(rrl, visit);
3    while !rrl.empty() do
4      s ← rrl.pop();
5      info ← vist.get(s);
6      if info = nil or info.budget > 0 then
7        rrl.append(s);
8        break;
9      end
10   end
11   while !rrl.empty() and visit.get(rrl.front()).budget = 0 do
12     rrl.pop();
13   end
14   updateProbabilities(visit);
/* invoked for each injection request                             */
15 Function shouldInject(request):
16   curr ← getCurrentState(request);
17   if not visit.contains(curr) then
18     visit.put(curr, new Info());
19     rrl.append(curr);
20   end
21   info ← visit.get(curr);
22   info.occur ← info.occur + 1;
23   if rrl.front() ≠ curr then return false;
24   if info.budget > 0 and rand() < info.prob then
25     info.budget ← info.budget - 1;
26     return true;
27   end
28   return false;
```

relies on the abstract state analysis to be precise. If the static analysis misses instrumenting an ASV close to the buggy point, the buggy point will likely be treated as in a seen state. In addition, the system can take a long time to enter a new state. If we only wait for new states, we may not inject anything when the workload finishes and waste an experiment trial.

To address these drawbacks, we design a **budgeted-state-round-robin** (*bsrr*) policy. Algorithm 2 lists its algorithm.

The algorithm allocates a budget (default 5) for each state to be potentially granted injection more than once. This relaxes the stringent new state requirement. After *all* states use up their budgets, the budgets are reset (Line 2).

It keeps a round-robin list of the abstract state tuples (`rrl` in Algorithm 2). Suppose the list has $s_1, s_2, \ldots, s_n$. The algorithm intends to grant injection requests from state s_1 for the first trial, grant requests from s_2 for the second trial, and so on. In other words, it **focuses on** one state in one trial.

Specifically, before each trial, *bsrr* rotates the state focused in the last trial to the end of the round-robin list (Line 7). If a state's budget is used up, it is removed from the list.

The round-robin design addresses the imbalanced injections problem illustrated by Figure 10: in all three experiment trials,

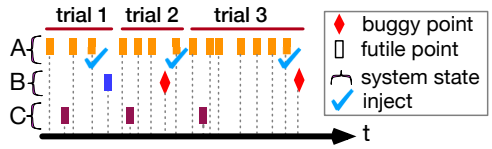


Figure 10: All injection chances are given to operations in state A.

we would inject operations in the frequent state A (within its budget), while no operation in state B or C is injected.

The algorithm also applies randomization to allow exploring different choices when a state has multiple injection requests. The probability p should be set properly. If it is too large, we would always grant the first (few) requests in a state. If it is too small, we may waste the injection trial.

We calculate p for each state tuple based on c—the times this state appears in injection requests. We set $p = 1 - e^{ln(0.01)/(c+1)}$. This formula's rationale is that we want to (i) grant at least one injection among the c requests to avoid wasting the trial; (ii) let the injection occur neither too early nor too late among the c requests. The probability that all c injections are *not* granted is $(1-p)^c$. Because of (i), this probability should be close to 0. Suppose $(1-p)^c = \epsilon$. With (ii), ϵ should not be too small; otherwise p is too close to 1 and the injection would be too early. We set ϵ to 0.01, and solve this equation, which gives $p = 1 - e^{ln(0.01)/c}$. We use $1 - e^{ln(0.01)/(c+1)}$ instead to handle corner cases of $c = 2$ or 3.

The *bsrr* policy is adaptive to leverage information from prior trials. Upon each injection request, the algorithm dynamically updates the parameter c (Line 22). Before a trial starts, it uses the occurrences from previous trials to re-calculate the probabilities for the visited states (Line 14). Similarly, *bsrr* updates the round-robin list dynamically (initially empty). If a state in an injection request is not visited before, it is added to the round-robin list for later exploration (Line 19).

5 Testing Experiment

LEGOLAS starts fault injection testing after the analyzer finishes instrumentations (§3, § 4.2) on the target system. LEGOLAS uses a client-server architecture to manage the testing (Figure 3), where the LEGOLAS agents embedded in the system send RPC requests to a LEGOLAS server that is composed of an abstract state tracker, injection controller, workload driver, and failure checkers. The testing proceeds in continuous trials.

5.1 Injection Trial

In each trial, LEGOLAS starts a cluster of the target system and then invokes the workload driver (Section 5.2). The trial ends when the workload finishes (successfully or not).

To support stateful injection decision algorithm (Section 4.3), while the target system is restarted in each trial, the LEGOLAS server will live throughout the experiment. Thus, it carries information such as the round-robin list across trials.

When a node enters a new abstract state, the LEGOLAS agent notifies the state tracker, which maintains one Abstract State Machine (ASM) per task-unit (usually a thread) for each system node. Each state update event is a tuple of node id, ASM-name (class name), ASM-instance (class instance), and ASV. The tracker records the current ASV and transitions for each ASM.

When a node reaches an injection hook, the LEGOLAS agent sends an injection query to the controller, which is a tuple of node id, ASM-instance, operation, and fault ids. The controller runs the *bsrr* algorithm to decide whether to grant the injection or not. Notice, however, that the injection query does not carry the ASV information. The controller obtains the associated ASV by indexing the node id and ASM-instance from the injection query to the ASM map in the state tracker.

LEGOLAS by default grants at most one injection in one trial. Allowing multiple injections in a trial only requires a simple change. While it seems more attractive to keep injecting faults in a trial, that choice has several disadvantages. Although distributed systems are designed to be fault-tolerant, each system has a limited tolerance level. If we keep injecting in a single run, the system may likely break as expected. Moreover, each injected fault can alter the system state and leave side effects. With continuously injected faults, it becomes very difficult to judge the system behavior and tell which fault is responsible for the symptom. Also importantly, if injections are performed non-stop, we may go deeper in an execution path, but we will not inject earlier, skipped operations or explore other paths, sacrificing completeness.

5.2 Workload Driver

LEGOLAS uses a workload driver to exercise the target system. For each system, we select several existing, representative test cases to create the workload driver.

To better suit our objectives, we make a few adaptations to the test cases. First, the driver creates multiple clients and each client is typically dedicated to interacting with one node. In this way, LEGOLAS can observe the status of every system node without mixing signals. Second, the driver divides workloads into phases, *e.g.*, create, read and write. Only when the current workload phase finishes successfully will the next phase starts. This is to localize the failed system functionalities and avoid unnecessary errors that mislead the results. Third, in one workload phase, each client is expected to send a series of requests and will report its progress to LEGOLAS server after one request completes. The LEGOLAS server also tracks when a client timeouts or encounters exceptions. This allows LEGOLAS to more accurately assess the failure impact. Lastly, we use a small workload scale, such that a trial does not take long and LEGOLAS can explore more trials.

5.3 Failure Checkers

To determine the testing results, LEGOLAS currently provides three failure checkers:

- *Crash checker*: it monitors the OS signals to check if a system node crashes, aborts, or exits with a non-zero status.
- *Client checker*: it approximates Panorama [22], a state-of-the-art gray failure detector, to identify whether differential observability exists. In particular, it marks a trial suspicious

System	Release	SLOC	Type
ZooKeeper	3.6.2	95K	Coordination service
HDFS	3.2.2	689K	Distributed file system
Kafka	2.8.0	322K	Event streaming system
HBase	2.4.2	728K	Distributed database
Cassandra	3.11.10	210K	Distributed database
Flink	1.14.0	78K	Stateful streaming system

Table 1: Evaluated distributed systems in latest releases.

if (1) a fault is injected in one node, but *only* another node's clients report errors; (2) the system's own detector indicates a node is active, but the node's clients report errors; (3) only a subset of clients fail to complete their workloads.

- *Log checker*: it scans the logs of each system node to identify whether there are log entries at warning or error level.

As a testing framework, LEGOLAS is extensible to add more checkers. For example, users can add checkers about inconsistency [38], semantic failures [37], or transaction isolation [30].

When a fault is injected, LEGOLAS records the stack trace of the originating operation. With the stack traces, LEGOLAS further clusters the trials by stack trace similarities so that similar symptoms are investigated together.

6 Implementation

We implement LEGOLAS with around 7,500 SLOC for the core components, and 100–300 SLOC for the workload driver for each evaluated target system. The LEGOLAS static analyzer is built on top of the Soot [52] framework, so it supports systems in JVM bytecode, including Java and Scala. Its core analysis algorithms are based on universal programming language constructs such as thread classes, member variables, branches, and conditionals. Thus, they are language-independent. The controller and orchestrator are designed in a client-server architecture using Java RMI for local RPCs.

7 Evaluation

Our evaluation aims to answer several key questions: (1) does LEGOLAS work on large distributed systems? (2) can LEGOLAS expose new complex fault-triggered bugs? (3) does the abstract states LEGOLAS infers significantly help the fault injection efficacy? (4) how does LEGOLAS using the *bsrr* policy compare to other policies and state-of-the-art solutions?

Evaluated Systems. We evaluate LEGOLAS on the recent stable releases of six popular, large-scale distributed systems (Table 7). These systems have different functionalities, written with various programming paradigms. Our appendix lists the workloads we use in the testing.

Measure. For a testing tool, its ability to exposes new bugs is a key measure. Our evaluation thus centers around this aspect (Section 7.2). Since our target systems are widely deployed in production and have been extensively tested for years, finding new bugs in their latest releases is not an easy task.

Additionally, we apply LEGOLAS on a number of randomly sampled known bugs in old releases of the systems (Section 7.5), including the running example in Figure 1.

System	Class	ASM	ASV				Static. Injected	
			Total	Mean	Min	Max	Methods	Points
ZK	708	36	226	6	1	31	484	1947
HDFS	4636	104	390	4	1	16	2127	3913
Kafka	5829	51	220	4	1	15	343	754
HBase	10462	96	312	3	1	17	5874	11051
CSD	4636	104	390	4	1	18	2127	3913
Flink	4852	48	110	2	1	6	997	2299

Table 2: Statistics of applying LEGOLAS static analyzer. *Class*: analyzed Java classes; *ASM*: classes analyzed as abstract state machines; Mean, min, and max of *ASV* are abstract state variables in each ASM.

Setup. We run experiments on servers with a 20-core 2.20GHz CPU and 64 GB memory running Ubuntu 18.04.

Each system's fault injection experiment consists of 2000 trials. A trial's time is dominated by the system startup and workload execution. The trials' durations vary depending on how the system reacts to the injected faults and whether it fails early or not. The experiment time for the six systems is 2.7 hrs, 10.7 hrs, 23.5 hrs, 8.4 hrs, 54.6 hrs, and 26.5 hrs respectively.

We use the *bsrr* policy (Section 4.3), and set the state budget to the default value of 5 *for all systems*.

Due to the large scale of experiments and time constraints, our testing focuses on the following faults: (1) I/O related exceptions, e.g., `IOException`, `ClosedChannelException`; (2) custom exception types that inherit from `IOException`; (3) delays to function calls that involve disk or network I/O. We run two separate experiments (exception and delay) for each system. We observe that `IOException` is widely used to represent more than hardware issues. For example, developers add `throw new IOException` statements for situations such as "unreasonable length", "missing signature", "current epoch is less than accepted epoch", and "snapshot already exists", which are difficult to simulate by external fault injection tools.

7.1 Injection Points and Abstract States

LEGOLAS successfully applies on the six systems. Besides scaffolding information (*e.g.*, class paths, task class types), the analyzer does not require additional input for a new system. The injection policies are also not specially tuned.

As Table 2 shows, the number of task classes (ASMs) LEGOLAS extracts is much smaller compared to the number of classes in the system. It also varies across different systems due to their design choices. For example, ZooKeeper has a relatively small number, while Cassandra has over 100; yet, ZooKeeper has the largest ASVs per ASM. This is because ZooKeeper uses long-running threads, while Cassandra adopts an event-driven architecture that uses many short-lived runnables. The mean ASVs per ASM is moderate, because currently LEGOLAS only analyzes the direct ASM classes and task entry methods.

We further manually inspect the 36 ASMs and 226 ASVs LEGOLAS generates for ZooKeeper. We find that they can represent the state transitions in ZooKeeper at different granularities. For example, in the `QuorumPeer` ASM, the ASVs exactly match the states of a node in the quorum: for the states such as

Looking, Observing, Leading, and Following, there exists exactly one ASV for each. In the `SyncRequestProcessor` ASM, the ASVs capture local state transitions: there is one ASV in which the transaction log is written, one when flushing in-memory log, and one when the snapshot is taken.

Dynamically, 24 ASMs and 76 ASVs are traversed during our testing. For the 150 non-traversed ASVs, 46 are from the 12 unutilized ASMs. We check the rest 104 ASVs to see if they encode meaningful execution states. In 20 of them, the code blocks have at least one I/O operation. In another 58 ASVs, they are in the exception handlers or shutdown blocks. In 8 ASVs, they only print a log. The remaining 18 ASVs do not contain significant operations and are introduced due to code optimizations by Soot. For the 20 unvisited ASVs that contain I/O operations, we tried to enlarge our workload (more reads/writes, reconfiguration), which did not help. However, our small workload achieves decent utilization of the ASVs.

7.2 Finding New Bugs

Our overall experience in the fault injection experiments is that the evaluated systems are robust to tolerate or at least cleanly abort various faults in most places. Take ZooKeeper as an example. If a thread is doing a socket write and a network delay is injected, this thread will get stuck. In general, ZooKeeper can handle the fault correctly even though this thread hangs. For example, if the `LearnerHandler` thread hangs in this way, the `QuorumPeer` is able to confirm the stale `PING` state and abandon the problematic `QuorumPeer`.

Despite the robustness, LEGOLAS finds new bugs in *all* tested systems. It finds 20 unique bugs (Table 8 in Appendix). All bugs are non-trivial and require domain knowledge to understand, such as mishandling of errors, design flaws, and synchronization issues. They all trigger partial failure symptoms, such as some requests get stuck while others succeed.

We reported the bugs to developers. Four reports are marked as critical, fourteen as major, and two as normal. Eleven reports have been explicitly confirmed by developers so far. Our bug reports generate substantial discussions with developers, with a median of 21 comments and a maximum of 42 comments. Our reports to ZooKeeper inspired the developers to adopt fault injection testing practice.

Case Studies. *HDFS-15925* In one trial, LEGOLAS injects an `IOException` in the `BlockReceiver` module while one datanode is forwarding the data blocks to a mirror (another datanode). One client gets stuck without any error log, and its workload progress is partial (1/5), while other clients finish the workload (5/5). After investigation, we find that normally the datanode in such a situation will inform the client of this error state *immediately*. Then the client will resend the blocks. This process would be fast. Through code analysis, we find the root cause is a complex timing bug. In particular, when the datanode encounters the `IOException` it sets the `mirrorError` flag (Figure 11). However, a concurrency condition exists in which the `mirrorError` flag set could be shortly

```
class BlockReceiver {
  private int receivePacket() {
    if (mirrorOut != null && !mirrorError) {
      try {                          IOException injected inside
        ...                          ↘
        packetReceiver.mirrorPacketTo(mirrorOut);
        ...
      } catch (IOException e) {      → set flag mirrorError
        handleMirrorOutError(e);
      }
    }
    return lastPacketInBlock?-1:len;
  }
}
class PacketResponder {
  public void run() {
    while (isRunning() && !lastPacketInBlock) {
      PipelineAck ack = new PipelineAck();
      try {
        if (type != LAST_IN_PL && !mirrorError) {
          ack.readFields(downstreamIn);
        }                            ← gets stuck
      } catch (IOException ioe) {
        ...
      }
    }
  }
}
```

Figure 11: A timing bug that causes the packet responder to get blocked when the datanode encounters an IOException.

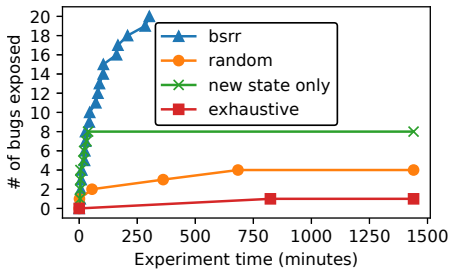


Figure 12: Efficacy of decision policies in LEGOLAS on detecting new bugs. *bsrr*: our budgeted-state-round-robin algorithm.

after the `PacketResponder` thread checks this flag, causing `PacketResponder` to not notice this status and get blocked, and the `ACK` packet will not be sent by the mirror datanode.

LEGOLAS exposes the bug five times in the experiment, with the first time in trial 124 at around 43 minutes.

HDFS-15869 The HDFS namenode uses the `EditLog` to maintain a transaction log of the namespace modifications. In one trial, LEGOLAS injects a delay to a remote write in the `FSEditLogAsync` thread. The injection occurs when the thread sends a response to the client and other servers, after it commits a transaction. This causes the whole `FSEditLogAsync` to be unable to proceed. The critical `logSync` function cannot be executed for incoming transactions. This is undesirable because `FSEditLogAsync`'s key feature is asynchronous edit logging that is supposed to tolerate slow I/O.

7.3 Impact of Abstract States and BSRR

This paper's thesis is that our inferred abstract states can enable efficient fault injection. Section 7.2 shows that LEGOLAS finds complex new bugs with our *bsrr* algorithm. To further validate our thesis, we compare the *bsrr* algorithm with alternative decision policies on the 20 new bugs.

	Detected Bugs	Median Detection Time
FATE	1	1057.9 minutes
CrashTuner	4	20.4 minutes
CORDS	0	N/A

Table 3: Effectiveness of existing work on the 20 new bugs.

For each policy, we run a 2000-trial experiment and measure the number of bugs it exposes, as well as the time it takes to expose the bugs. The latter is an important metric. If a solution cannot expose a bug within a reasonable time, developers in practice likely will not use it even if the solution in theory may expose the bug after a long time.

Figure 12 shows the result. The exhaustive policy only exposes one bug. The random policy only exposes three bugs. It is also inefficient. It takes a median of 208 minutes and a max of 994 minutes to find the three bugs. After finding the third bug, it fails to find more bugs in 24 hours.

The *new-state-only* policy (§ 4.3) exposes eight bugs in a median of 11.4 minutes. It is the best among the baseline policies, showing the advantages of our inferred abstract states.

The *bsrr* significantly outperforms all alternatives. It exposes 20 bugs in a median of 58.2 minutes (min 4.0 minutes, max 302.0 minutes). Compared to *new-state-only*, it is more robust in leveraging the imperfectly inferred abstract states, exposing much more bugs while achieving good efficiency.

The inferred ASVs help improve the fault injection efficacy in two ways. First, they can capture the unusual system service stages, allowing LEGOLAS to inject in places that are not well tested and buggy. For example, in HDFS-15957, one of the ASVs LEGOLAS infers for `FSEditLogAsync` represents the state of sending a response to client while `FSEditLogAsync` is committing transactions. The ASVs for the running ZooKeeper example also belong to this category. Second, the inferred ASVs can help make progress in skipping uninteresting injection points. For example, in HDFS-15925, the injection (Figure 11) occurs inside the `DataXceiver` thread in the datanode. The relevant ASV that LEGOLAS infers corresponds to the `processOp` stage. Although this ASV is just the main stage of the thread, the other ASVs LEGOLAS infers in other threads help avoid wasting too much time in injecting in other places.

7.4 Comparing with Other Solutions

Research Baselines. We compare LEGOLAS with three state-of-the-art fault injection research projects, FATE [15] CrashTuner [40] and CORDS [13]. FATE tests *multiple failures* by using a concept of *failure IDs* to efficiently enumerate the combinations of failures. CrashTuner uses *meta-info variable accesses* to decide the timing of injecting node crashes for exposing crash recovery bugs. CORDS uses a FUSE file system to inject a single corruption or read/write error to one file-system block at a time, and enumerate all possible faults.

The first two works focus on node-level faults, making them not directly comparable to LEGOLAS. We apply their key ideas to attempt meaningful comparisons. We define the *failure IDs* as described in the FATE paper and implement a policy in LEGOLAS to grant an injection request when its associated failure ID has not been visited. For CrashTuner, because its analyzer component is not available, we re-implement its static analysis to identify all meta-info variable accesses and assign each access point a global ID. We instrument each access point to record the accesses at runtime. Then we grant an injection request when some meta-info variable access occurs within the past 5 ms and the access ID has not been seen. The latter is needed because a system may access the meta-info variable in a deterministic order, leading to only one injection being always granted if the access ID is not checked.

For CORDS, we utilize similar procedures as described in the paper to enumerate file system level errors on the requests to FUSE. For the experiment, we use the same workloads as in LEGOLAS but use the injection algorithm in CORDS.

Table 3 shows the result. FATE only detects one of the 20 new bugs in 1057.9 minutes. CrashTuner only detects four bugs. LEGOLAS significantly outperforms both solutions. CORDS does not detect any of the 20 new bugs despite enumerating all of its injection choices during the experiment. Although CORDS is a fine-grained fault injection tool, its fault scope is limited. It only injects corruption or error of a file block. Only 2 of the 20 bugs' root causes are related with that. For the two cases, they require a transient error and special timing, while CORDS injects persistent corruption or error that more likely leads to a total failure (node crash).

Popular Tool Baselines. We further compare LEGOLAS with three fault injection tools that are popular among developers: `CharybdeFS` [49] (a fault-injection filesystem), `tcconfig` [19] (a network fault injection tool based on Linux Traffic Control), and `byte-monkey` [53]. Byte-monkey is closer to LEGOLAS in that it also performs bytecode-level fault injection.

These tools rely on user-provided parameters to configure the injection, such as the packet loss rate and probability of returning an error code. Settings that are too large or too small produce meaningless results. We choose one moderate setting and one mild setting for each tool. We exercise the target systems using the same workloads from LEGOLAS.

Most injections lead to either a high percentage of successful trials or a high percentage of early exits (shown in appendix). For the small percentage of *partial_progress* injection trials, the failed client requests either happen directly because of the injected fault (*e.g.*, the server logs that it is unable to read data from client) or the system is in the middle of fault handling and successfully recovers. We verify that none of these trials expose any of the 20 new bugs LEGOLAS finds. We also vary the parameters, but the conclusions remain the same.

In the LEGOLAS decision policy comparison experiments (§ 7.3), its random policy exposes three bugs. In comparison, the evaluated popular tools do not expose any bugs with their random strategies. The discrepancies are due to the probability factor and the fact that LEGOLAS's in-situ injection mechanism has more precise control—it instruments operations that are possible to throw `IOException` (or its subtype) errors, which

System	Bug Id (Exposure Time)
ZooKeeper	ZK-2029 (15.4 min), ZK-2201 (30.6 min), ZK-2247 (52.1 min), ZK-2325 (2.6 min), ZK-2982 (18.5 min)
Cassandra	CA-6364 (10.0 min), CA-6415 (330.6 min), CA-8485 (25.3 min), CA-13833 (86.7 min)
HDFS	HDFS-11608 (29.2 min), HDFS-12157 (39.9 min)

Table 4: LEGOLAS exposes *known* bugs in old releases.

ZooKeeper	HDFS	Kafka	Cassandra	HBase	Flink
8.9 s	31.6 s	36.9 s	20.9 s	77.6 s	63.9 s

Table 5: Time of static analysis and instrumentation.

may be caused by complex environment errors. For instance, while `tcconfig` injects packet loss, it fails to trigger errors for function calls that internally throw `IOException` only upon a sequence of network errors. Also, its injection has a low chance of affecting only a special subset of operations.

7.5 Exposing Known Bugs

Besides finding new bugs, we further evaluate LEGOLAS's capabilities on exposing known partial failure bugs. We sample 11 *real-world* partial-failure issues from older system releases. In particular, we first use keywords matching on Jira to collect all cases whose root cause is related to I/O exception or delay. Then we randomly sample 50 of them, and select all the cases that satisfy our definition of partial failures and require one fault, which result in 11 cases. As Table 4 shows, LEGOLAS can relatively quickly reproduce these known bugs.

7.6 Performance

We measure the fault injection trial duration for LEGOLAS. Figure 13 shows the results. Although LEGOLAS tracks abstract states and dispatches the fault injection queries through RPCs, the injection duration is still acceptable, with a maximum of around 70 seconds. LEGOLAS uses local RPCs based on Java RMIs. Our microbenchmark shows the RMI latency is between 10 μs–50 μs. The *bsrr* policy function for one injection request has a median latency of 3 ms.

Table 5 shows the performance of the LEGOLAS static analysis and instrumentation. The analysis is fast, with the longest time being (73 s) in analyzing HBase.

7.7 Effort and False Positive

The static analysis and instrumentation steps in LEGOLAS are fully automated. The fault injection experiment does not require manual tuning. Our workload drivers are adapted from existing test cases and re-usable across versions. The main effort in using LEGOLAS is to confirm a bug after testing. However, this is common for testing tools, not a unique requirement of LEGOLAS. For the new bug finding exercise in Section 7.2, it roughly takes a graduate student author one to two weeks per system to examine the testing results, read and understand the relevant source code, and confirm where the bug is.

A false positive occurs if the fault being injected turns out to be impossible in reality. We observe such false positives

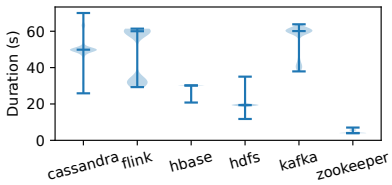


Figure 13: Distribution of one fault injection trial duration.

	ZooKeeper	HDFS	Kafka	Cassandra	HBase	Flink
w/ i.i.a	0	0	0	0	0	0
w/o i.i.a	45 (6)	20 (9)	0	894 (10)	86 (10)	0

Table 6: Number of trials that have invalid injections, with and without the invalid injection analysis in Section 3. The numbers in parentheses are the unique locations of these invalid injections.

in initial development of LEGOLAS and they are caused by the same problem: it injects an `IOException` to a function that declares `IOException` in its signature but cannot possibly have I/O errors. For example, we inject an `IOException` to the `writeBoolean` call inside the Cassandra `serialize` method. This injection triggers a buggy symptom. However, this method uses a memory buffer for the `writeBoolean` call.

As described in Section 3, we introduce an inter-procedural, context-sensitive analysis to eliminate such false injections. Table 6 shows the number of false injection trials for each system with and without this analysis. The result shows that the analysis successfully eliminates all of these invalid injections.

Another potential source of false injection is the automatic exception instance creation. Our method may create some invalid exception instances. However, we did not observe such false positives in our experiments.

Our failure checkers (Section 5.3) can make mistakes. When they mark a trial as being suspicious, it may turn out not to be a bug. For example, in ZooKeeper, the write requests to followers are forwarded to the leader. When the client checker flags a trial where a fault is injected in node 1 and the clients connected to node 2 experience write timeouts, it is possible that the node 1 is the leader and it is temporarily unavailable because of the delay injection. Developers may consider this to be a false positive, but the false positive is from the checker rather than the fault injection—the injected fault is legitimate. Our current checkers are basic. From our experience, to enhance checkers with more comprehensive bug rules, the knowledge of system-specific protocols and modularity practices must be exploited. After the enhancement, developers may still disagree on the definition of a bug. However, LEGOLAS's core contribution is on fault injection methodology.

Our bug confirmation process is as follows. We first utilize the basic checkers to highlight suspicious trials. For each suspicious trial, we analyze how the fault propagates its effect from one component to another and leads to the suspicious symptom. We then diagnose whether that is a bug and check if the design documentation describes the expected behavior. Finally, we will report it to developers to discuss with them.

8 Discussion and Limitations

Our ASV inference currently only analyzes states inside the task classes and the task entry functions. Thus, the inferred ASVs for each task are relatively coarse-grained. The analysis can be extended to other task functions and states that are passed to other classes through function calls, which will extract more detailed ASVs. However, it is not always the more detailed the better. A large system typically has tens to hundreds of task classes, so too fine-grained ASVs can lose the benefits of effectively grouping injection requests.

Our workload drivers use workloads on a small scale to exercise the target system. More workloads can be added to the LEGOLAS workload drivers, which is not a difficult task and would allow LEGOLAS to expose fault-induced bugs that require large workloads (*e.g.*, performance bugs).

LEGOLAS injects a single fault in one fault injection trial. It would miss bugs triggered by multiple faults. Supporting injection of multiple faults in LEGOLAS only requires a simple change. However, the decision algorithm and failure checkers would likely require significant changes. Indeed, we tried enabling multi-fault injection for the ZooKeeper experiment, but it only improved the efficiency for one bug.

LEGOLAS does not explicitly control non-determinism in the target system, such as thread schedules, which is the focus of concurrency testing tools. Thus, while LEGOLAS can expose a concurrency bug, it may not expose it reliably or efficiently. LEGOLAS can be combined with concurrency testing tools.

9 Related Work

Fault Injection. Early fault injection work targets standalone software. Faults are injected into hardware, simulated environment [20], or libraries (LFI [43]). Fault injection testing becomes popular in distributed systems with much research [2–4, 7, 13, 15, 24, 26, 27, 40, 45]. Many inject coarse-grained faults externally such as node crashes to expose protocol or crash recovery bugs. The injection is done randomly, or exhaustively, or based on user specifications. Several solutions proposed more advanced techniques. For example, LDFI [3] leverages data lineage to inject crashes or network faults if the faults could prevent correct outcomes; CrashTuner [40] injects crashes when meta-info variables are accessed.

LEGOLAS focuses on complex partial failure bugs. It uses an instrumentation approach to inject system-specific, instruction-level faults within a target system. It designs a novel static analysis method that automatically infers abstract states from distributed system code. Its decision algorithm leverages the abstract states to efficiently explore the fault space.

Recent fault injection research addresses other applications, such as multi-threaded programs [32], cluster-management controllers [51], microservices [44, 57], and REST applications [8]. LEGOLAS is orthogonal to these efforts. It targets large-scale distributed systems and aims to expose partial failure bugs triggered by exceptions or delays in the operations of a component within a distributed system node.

Model Checking. Model checking enumerates the possible interleaving of non-deterministic events such as messages. It has been applied to distributed systems [18, 29, 31, 50, 54]. Distributed system model checkers (dmcks) including MODIST [54], SAMC [31], and FlyMC [42] also explore the interleaving of crash/reboot failure events. LEGOLAS shares high-level similarity with these solutions in that it systematically explores the fault injection space. However, LEGOLAS is a complementary effort. Existing dmcks target protocol bugs caused by complex interleaving of node-level events, while LEGOLAS targets implementation-level bugs triggered by diverse faults in fine-grained program instructions. LEGOLAS can leverage a dmck to drive the target system into unexplored states, allowing LEGOLAS to try more injections.

Distributed Concurrency Bug Detection. Several projects [33, 35, 39, 56] aim to detect concurrency bugs in distributed systems. FCatch [35] applies happens-before analysis on correct execution traces to identify unprotected conflicting operations. LEGOLAS is a general fault injection framework aiming to expose diverse bugs.

Partial Failure Detection. Failure detectors are part of a running production distributed system to determine whether the system is faulty or not. Recent works [22, 36, 41, 47] explore advanced detectors for the notorious partial failures. LEGOLAS is an offline testing tool. It can leverage these advanced techniques in its checkers to find more bugs in testing.

Error Handling Bug Detection. Error handling code is known to be buggy. Studies [36, 55] have shown that this is also true for distributed systems. Aspirator [55] uses rules to statically find simple error handling bugs such as empty handlers. LEGOLAS focuses on fault injection to systematically test distributed systems and uncover diverse types of bugs.

10 Conclusion

This paper presents LEGOLAS, a fault injection testing framework that aims to catch complex partial failure bugs in large distributed systems. LEGOLAS uses static analysis to enable fine-grained, system-specific fault injection. It designs a novel method to extract abstract states from system code and uses them to efficiently explore the fault injection space. We apply LEGOLAS on six distributed systems and find 20 new bugs. LEGOLAS is available at https://github.com/OrderLab/Legolas.

Acknowledgments

We thank our shepherd, Peter Alvaro, and the anonymous reviewers for their valuable and detailed feedback that improved our work. We appreciate the help from the developers of the open-source distributed systems we evaluated. We thank CloudLab [11] for providing the resources to run our experiments. This work was supported in part by NSF grants CNS-2317698, CNS-2317751, and CCF-2318937.

References

[1] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. Toward a generic fault tolerance technique for partial network partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 351–368. USENIX Association, November 2020.

[2] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, pages 51–68, Berkeley, CA, USA, 2018. USENIX Association.

[3] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. Lineage-driven fault injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 331–346, New York, NY, USA, 2015. ACM.

[4] Cory Bennett and Ariel Tseitlin. Chaos monkey released into the wild. http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html, 2009.

[5] Dmitry Bugaychenko. Kafka production failure because of BadVersionException. https://issues.apache.org/jira/browse/KAFKA-1407, 2014.

[6] Haicheng Chen, Wensheng Dou, Yanyan Jiang, and Feng Qin. Understanding exception-related bugs in large-scale cloud systems. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 339–351. IEEE Press, 2020.

[7] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. CoFI: Consistency-guided fault injection for cloud systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 536–547, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Yinfang Chen, Xudong Sun, Suman Nath, Ze Yang, and Tianyin Xu. Push-Button reliability testing for Cloud-Backed applications with rainmaker. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1701–1716, Boston, MA, April 2023. USENIX Association.

[9] Kim Christensen. Kafka partial cluster breakdown. https://issues.apache.org/jira/browse/KAFKA-3577, 2016.

[10] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 14:1–14:14, New York, NY, USA, 2013. ACM.

[11] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, jul 2019. USENIX Association.

[12] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. Amazon DynamoDB: A scalable, predictably performant, and fully managed NoSQL database service. In *Proceedings of the 2022 USENIX Annual Technical Conference*, USENIX ATC '22, pages 1037–1048, Carlsbad, CA, July 2022. USENIX Association.

[13] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST '17, page 149–165, USA, 2017. USENIX Association.

[14] Supriyo Ghosh, Manish Shetty, Chetan Bansal, and Suman Nath. How to fight production incidents? an empirical study on a large-scale cloud service. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 126–141, New York, NY, USA, 2022. Association for Computing Machinery.

[15] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A framework for cloud recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 238–252, Berkeley, CA, USA, 2011. USENIX Association.

[16] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, pages 1–16, October 2016.

[17] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 1–14, Berkeley, CA, USA, 2018. USENIX Association.

[18] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, October 2011.

[19] Tsuyoshi Hombashi. tcconfig: A tc command wrapper. https://github.com/thombashi/tcconfig, 2022.

[20] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, April 1997.

[21] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 73–90, Carlsbad, CA, July 2022. USENIX Association.

[22] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 1–16. USENIX Association, October 2018.

[23] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS XVI, British Columbia, Canada, May 2017. ACM.

[24] LLC. Jepsen. Jepsen: a framework for distributed systems verification, with fault injection. https://github.com/jepsen-io/jepsen, 2023.

[25] Jiahongchao. updateisr should stop after failed several times due to zkVersion issue. https://issues.apache.org/jira/browse/KAFKA-3042, 2015.

[26] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. PREFAIL: A programmable tool for multiple-failure injection. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 171–188, New York, NY, USA, 2011. ACM.

[27] Xiaoen Ju, Livio Soares, Kang G. Shin, Kyung Dong Ryu, and Dilma Da Silva. On fault resilience of OpenStack. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SoCC '13, pages 2:1–2:16, New York, NY, USA, 2013. ACM.

[28] Pravein Govindan Kannan, Nishant Budhdev, Raj Joshi, and Mun Choon Chan. Debugging transient faults in data centers using synchronized network-wide packet histories. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '21, pages 253–268. USENIX Association, April 2021.

[29] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI 07)*, Cambridge, MA, April 2007. USENIX Association.

[30] Kyle Kingsbury and Peter Alvaro. Elle: Inferring isolation anomalies from experimental observations. *Proc. VLDB Endow.*, 14(3):268–280, November 2020.

[31] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, page 399–414, USA, 2014. USENIX Association.

[32] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 162–180, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 677–691. ACM, April 2017.

[34] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. What bugs cause production cloud incidents? In *Proceedings of the Workshop on Hot Topics in Operating*

Systems, HotOS '19, page 155–162, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. FCatch: Automatically detecting time-of-fault bugs in cloud systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 419–431. ACM, 2018.

[36] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 559–574, Santa Clara, CA, February 2020. USENIX Association.

[37] Chang Lou, Yuzhuo Jing, and Peng Huang. Demystifying and checking silent semantic violations in large distributed systems. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 91–107, Carlsbad, CA, USA, July 2022. USENIX Association.

[38] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 295–310, New York, NY, USA, 2015. Association for Computing Machinery.

[39] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. CloudRaid: hunting concurrency bugs in the cloud via log-mining. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, FSE '18, pages 3–14. ACM, November 2018.

[40] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. CrashTuner: Detecting crash-recovery bugs in cloud systems via meta-info analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 114–130, New York, NY, USA, 2019. Association for Computing Machinery.

[41] Ruiming Lu, Erci Xu, Yiming Zhang, Fengyi Zhu, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Jiwu Shu, Minglu Li, and Jiesheng Wu. PERSEUS: A fail-slow detection framework for cloud storage systems. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies*, FAST '23, USA, 2023. USENIX Association.

[42] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. FlyMC: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[43] Paul D. Marinescu and George Candea. LFI: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, DSN '09, pages 379–388. IEEE, June 2009.

[44] Christopher S. Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. Service-level fault injection testing. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 388–402, New York, NY, USA, 2021. Association for Computing Machinery.

[45] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, page 33–50, USA, 2018. USENIX Association.

[46] Donny Nadolny. Debugging distributed systems. In *SREcon 2016*, Santa Clara, CA, April 7-8 2016.

[47] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S. Gunawi. IASO: A fail-slow detection and mitigation framework for distributed storage services. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 47–61, USA, 2019. USENIX Association.

[48] Casey Rosenthal, Lorin Hochstein, Aaron Blohowiak, Nora Jones, and Ali Basiri. *Chaos Engineering*. O'Reilly Media, Inc., 2017.

[49] ScyllaDB. CharybdeFS: A fuse based fault injection filesystem. https://github.com/scylladb/charybdefs, 2021.

[50] Jiri Simsa, Randy Bryant, and Garth Gibson. dbug: Systematic evaluation of distributed systems. In *5th International Workshop on Systems Software Verification (SSV 10)*, Vancouver, BC, October 2010. USENIX Association.

[51] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. Automatic reliability testing for cluster management controllers. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 143–159, Carlsbad, CA, July 2022. USENIX Association.

[52] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–, Mississauga, Ontario, Canada, 1999. IBM Press.

[53] Alex Wilson. Bytecode-level fault injection for the JVM. https://github.com/mrwilson/byte-monkey, 2019.

[54] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '09, page 213–228, USA, 2009. USENIX Association.

[55] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 249–265, Berkeley, CA, USA, 2014. USENIX Association.

[56] XinHao Yuan and Junfeng Yang. Effective concurrency testing for distributed systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1141–1156. ACM, March 2020.

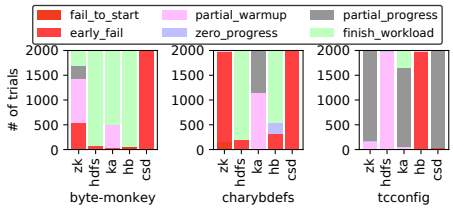
[57] Jun Zhang, Robert Ferydouni, Aldrin Montana, Daniel Bittman, and Peter Alvaro. 3MileBeach: A tracer with teeth. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 458–472, New York, NY, USA, 2021. Association for Computing Machinery.

	CharybdeFS	tcconfig	Byte-monkey
	FS syscalls error, delay	packet loss, delay	exceptions
moderate	10%, 8 s	20%, 300 ms	10%
mild	1%, 1 s	10%, 80 ms	1%

Table 9: Configurations of the baseline tools.

System	Release	SLOC	Workload
ZK	3.6.2	95K	3 clients create 10 entries in total, each writes and reads a random entry 40 times.
HDFS	3.2.2	689K	5 clients each writes a 10KB file 5 times, 5 clients each reads 5 files 5 times.
Kafka	2.8.0	322K	3 clients creates 15 topics, 15 clients each produces a topic, 30 clients consume a topic.
HBase	2.4.2	728K	creates 5 tables, each with 5 columns, each with 5 rows, 5 clients r/w a random row 100 times.
Cassandra	3.11.10	210K	creates 1 table, inserts 5 rows, 3 clients select and update a random table field 100 times
Flink	1.14.0	78K	1 client submit batch workload, 1 client submit streaming workload, 1 Kafka consumer to receive 170 messages from the batch workload, 1 Kafka producer to send 5 messages to the streaming workload, 1 Kafka consumer to receive 10 messages from the streaming workload

Table 7: Evaluated distributed systems in latest releases, and the workload LEGOLAS uses in the fault injection experiments.



(a) w/ moderate fault injection config.

(b) w/ mild fault injection config.

Figure 14: 2000 fault injection trials with existing tools: Byte-monkey, CharybdeFS, tcconfig.

Appendix A Evaluation Details

Table 8 lists the injected fault, symptom, and root cause for each of the 20 new bugs that LEGOLAS finds.

Table 9 shows the configuration settings we use for the baseline fault injection tools (Byte-monkey, CharybdeFS, and tcconfig) in the evaluation.

Figure 14 shows the aggregated fault injection results using the baseline tools.

Bug Id	Injected Fault	Symptom	Root Cause
ZK-4074	delay when `Learner` is executing `writePacket`	requests to one follower get stuck and the follower cannot rejoin the quorum for a long time	write in a critical section and prevents `QuorumPeer` from entering the receiving stage
ZK-4203	an exception during accepting a connection from the second follower	one follower keeps trying to join the quorum but keeps failing, even though the other 2 nodes get the request	the `ERROR` state set by learner is not discovered by the leader in some condition
ZK-4419	an exception when a learner creates a socket to connect to leader	one follower takes a long time to join the quorum and causes temporary service unavailability	the server state is prematurely changed, which triggers unnecessary re-election
ZK-4424	an exception when leader is configuring socket options for a follower	causes re-election and partial service unavailability	error is unnecessarily re-thrown, which causes handler exit and costly re-creations
KA-13457	an exception when a broker is accepting a connection	some client requests to create topics experience InvalidReplicationFactorException errors for a long time	error is swalloed without closing the socket channel
KA-13468	an exception when the log manager initalize a log	a broker proceeds but consumers hang for more than 3 minutes without any error log	the log manager should handle the error but instead let it propagate to the request handler
KA-13538	an exception when a broker is accessing checkpoint file	some clients get unexpected TopicExistsException even though they have never created the topic before	a design flaw in the client library for handling broker change
KA-14882	a delay when a broker sends request to ZooKeeper	some retry of topic creation requests gives the client TopicExistsException	broker controllers do not roll back the metadata in ZooKeeper when topic creation fails
KA-14886	a delay when handling a request from consumer and storing data to disk	a critical thread pool in broker gets full soon after the delay of a single request from client	the delayed thread blocks multiple threads and causes the thread pool to be used up
HD-15925	IOException when a datanode is forwarding a packet to the mirror	normally a client is immediately notified of the error, but now the client hangs for 1min	race condition causes PacketResponder to be blocked without notifying the client
HD-15957	exception when namenode finishes sync edit log and notifies journalnode	some client hangs forever without any log and the expected file does not exist in HDFS	namenode dismisses one client RPC, adding retry of the notification resolves the issue
HD-15869	delay when namenode sends the edit log notifications	namenode hangs even with the async edit logging	the notification sending is performed synchronously and blocks queued edit logs
HA-17552	error after the namenode accepts a socket before creating a reader	some client hangs instead of timing out after ping interval	read method does not re-throw the socket timeout exception
HA-18024	an exception when namenode configures socket options	some client hangs for a long time	socket connection is not closed when error happens
HB-26256	a delay when the region server tries to open a region using HDFS RPC	table creation command hangs for a long time without any error but list command shows the table exists	region sequence id file write operation is blocking without any timeout
HB-26955	an exception when the master tries to do an update operation	some table create requests experience a long delay	retry code misses the case when a server is quickly reinitialized
CS-16603	a delay in serializing a mutation to commit log in node 2	clients to node 1 experience sporadic CQL operation timeout due to unconfigured table	the add method is not protected with a timeout
CS-17564	exception when deleting file during a compaction task	node continues after erroreous startup state and later causes client failures	missing sync. to wait for compaction completion before setting node startup flag
FL-30032	an IOException when sending a watermark to Kafka	synchronous batch processing request from client finishes without errors while the job is actually not finished	the exception is not handled properly and then a few messages do not get sent to Kafka
FL-31746	an IOException when task managers finish a job and commit some data	when the job is finished, the client throws confusing errors due to a fault in commit phase	The commit phase does not affect the correctness of output but its fault is propogated to the client with confusing messages

Table 8: New bugs found by LEGOLAS. All issues cause partial failure symptoms. The root causes are diverse. ZK: ZooKeeper; KA: Kafka; HD: HDFS; HA: Hadoop; HB: HBase; CS: Cassandra; FL: Flink.

Load is not what you should balance: Introducing Prequal

Bartek Wydrowski
Google Research

Robert Kleinberg
Google Research and Cornell

Stephen M. Rumble
Google (YouTube)

Aaron Archer
Google Research

Abstract

We present Prequal (*Probing to Reduce Queuing and Latency*), a load balancer for distributed multi-tenant systems. Prequal aims to minimize real-time request latency in the presence of heterogeneous server capacities and non-uniform, time-varying antagonist load. It actively probes server load to leverage the *power of d choices* paradigm, extending it with asynchronous and reusable probes. Cutting against received wisdom, Prequal does not balance CPU load, but instead selects servers according to estimated latency and active requests-in-flight (RIF). We explore its major design features on a testbed system and evaluate it on YouTube, where it has been deployed for more than two years. Prequal has dramatically decreased tail latency, error rates, and resource use, enabling YouTube and other production systems at Google to run at much higher utilization.

1 Introduction

We report our experience deploying the power of d choices (PodC) load balancing paradigm [1, 2, 18] to run multiple large-scale web services at Google, focusing on YouTube, where it has run successfully for several years. To our knowledge, this is the first public report of PodC being used successfully for load balancing at this scale.

The PodC paradigm involves sampling $d \geq 2$ servers for their load and sending the next request to the least loaded one. Two key questions for any implementation are: what signal is used to represent the load, and how is the sampling done? We answer those questions with the name of our load balancing system: Probing to Reduce Queuing and Latency (Prequal). Namely, our two signals are requests-in-flight (RIF) and latency, and we sample servers by actively probing them.

Many existing load balancing systems (e.g., NGINX [19], Envoy [7], Finagle [8], YARP [26], C3 [23]) offer some variant of PodC, and most of them offer RIF, latency, or some combination as the load balancing signal. We introduce two primary new innovations. First, we combine RIF and latency in a new way that works especially well, called the *hot-cold lexicographic* (HCL) rule. Second, we introduce a novel asynchronous probing mechanism that reduces probing overheads (CPU + critical-path latency) while retaining the freshness of the load signal offered by synchronous probing.

For purposes of this paper, we refer to Prequal as a load balancing *system*, but technically, it is load a balancing *policy* implemented within Google's Stubby RPC framework (externally, gRPC [24]). As such, we expect that some of our innovations could be integrated into these other existing load balancers. Our main thrust is the two innovations above (HCL, async probing); other implementation details of our system are incidental to our message.

Prequal has been deployed in a diverse collection of 20+ large-scale services at Google over the past several years. In addition to driving most of the YouTube serving stack, it has found success in a wide variety of other applications, including search ads, logs processing, and serving ML models. In the typical application, query processing times are tens to hundreds of milliseconds, but in one system the queries take around ten minutes, and in another they range from seconds to hours. Most of these services are part of a complex tangle of services calling each other, and we have seen benefits from deploying Prequal for the most critical services, regardless of whether others in the tangle are also using Prequal.

We care about CPU and latency overheads from probing, but in our deployment experience we have found them to be small and more than pay for themselves. Probe response times within a data center are well below 1 millisecond. For some applications, fractions of a millisecond matter, so we used async probing to take it off the critical path. We created parameters that can shrink the CPU overheads arbitrarily. Moreover, better balance saves CPU by reducing resource contention. On net, we find that CPU utilization usually *dips* slightly when deploying Prequal. More importantly, systems are provisioned to control tail latency at peak load, so even an *increase* in *mean* CPU usage can result in provisioning wins when combined with decreases in tail latency.

Our evaluation of Prequal consists of two parts: a holistic

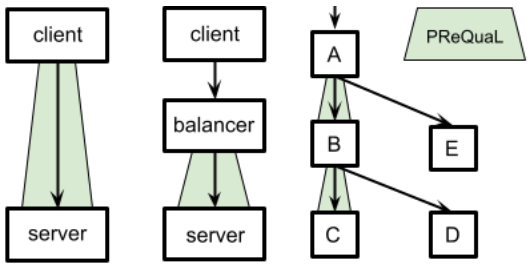


Figure 1: Queries flow down the tree, responses flow up. A separate balancing job is optional on each link. Load balancing policies may vary by link: A → B uses Prequal, while B → D does not.

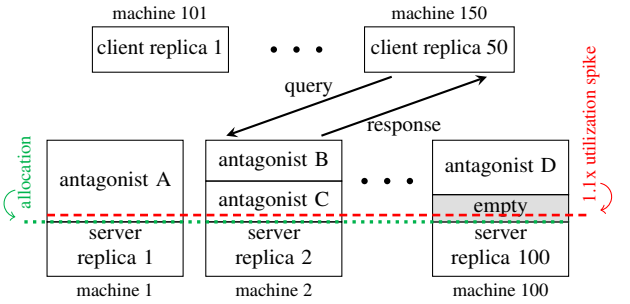


Figure 2: Zooming in on any single link from Fig. 1: server replicas respond to queries from client replicas while their machines run antagonist processes. Aggregate utilization (dashed line) may exceed allocation (dotted line) during spikes in demand.

evaluation in the wild on YouTube (§2 and §3), and benchmarks on a testbed to evaluate each of the major design choices independently (§5). The default incumbent load balancing policy that we displaced at YouTube and other parts of Google is (dynamic) weighted round robin (WRR), which focuses on balancing CPU utilization across distributed servers in a single job. Thus, the evaluation in §3 focuses on comparing Prequal to WRR. §2 explains why CPU-balancing policies like WRR are a formidable competitor in this environment, but also why it pays to rethink that approach, motivating our paper title. §5 and §6 include quantitative and qualitative comparisons against well-known load balancing policies from the literature, including other variants of PodC.

2 Environment and motivation

We describe the operating environment for YouTube and other large cloud services, motivating why balancing CPU load would seem to be a self-evidently superior approach. Next, we examine a scenario in which balancing CPU load backfires spectacularly, motivating the Prequal approach. Finally, we show data from YouTube illustrating that our motivating circumstance occurs commonly.

Large-scale services like YouTube are typically composed of many distributed jobs issuing queries among themselves via RPC. Fig. 1 depicts a vastly simplified version with only five jobs, queries flowing down a tree, and responses flowing back up. In any of these interactions, the job sending the query is called the *client job*, while the job returning the response is the *server job*. For instance, on edge A → B, A is the client and B is the server, while B is the client on edges B → C and B → D. To ensure scalability and redundancy, each job is distributed across a large number of machines, called *replicas* (Fig. 2). To avoid overloading any server replica, a load balancing policy must be used between each pair of communicating jobs, to determine which server replica should receive each query. Some policies require proxying the queries through a separate load balancing job whose only function is to decide where to forward each query, while others can operate directly between client replicas and server replicas (Fig. 1).[1] Prequal is well-suited to both modes of operation, and in fact we use it both ways in different systems at Google.

Each job runs within a single data center, although different jobs might run in different data centers. Each replica runs inside its own virtual machine (VM), which has access to a guaranteed portion of the CPU cycles on its host machine; this amount is called its *(CPU) allocation*. The fraction of its allocation that a replica uses over any given time period is its *(CPU) utilization*, which could be more or less than 100%, since the allocation is just a guaranteed minimum. The server job's *aggregate (job) utilization* is the fraction of its overall CPU allocation that it uses, across all replicas. In the usual case, each replica has the same allocation, so aggregate job utilization equals average replica utilization.

A server replica typically shares its machine with many other VMs, which we call *antagonists*. The *machine (CPU) utilization* accounts for both our server replica and all antagonists, and is distinct from both replica and job utilizations. In order to ensure consistent performance for all users of this multi-tenant system, *isolation* mechanisms attempt to prevent the behavior of one VM on a machine from adversely affecting the behavior of another. The philosophy here is: "If your usage stays within your allocation, you will be fine." However, if a VM overflows its CPU allocation, it may suffer when the isolation system kicks in.

Given this isolation regime and design philosophy, it *seems* self-evident that trying to balance CPU utilization across replicas within one job is the best approach.[2] Indeed, the WRR policy works well at Google. It uses smoothed historical statistics on each replica's goodput, CPU utilization, and error rate to periodically compute individual per-replica weights. Clients then route queries to replicas in proportion to these

[1] At this scale, even the balancing job must be distributed across machines.

[2] Trying to balance machine CPU utilization across these entire heterogeneous machines would work at cross purposes to the isolation mechanism.

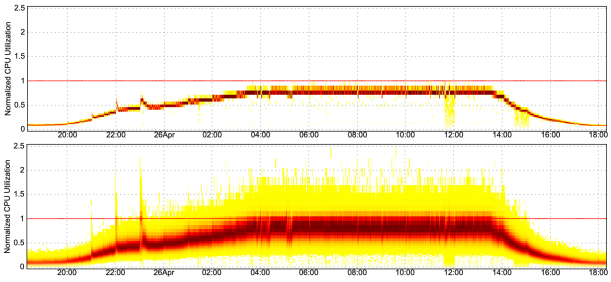


Figure 3: Normalized CPU usage heatmap with 1-minute samples (top) and 1-second samples (bottom) for the YouTube homepage service running across hundreds of machines in a single datacenter. Darker colors represent denser regions. The usage limit, represented by 1.0 (red line) is satisfied in all of the 1m intervals but on 1s intervals it is frequently violated at peak load — sometimes by more than a factor of two!

weights. In the absence of errors, each replica weight w_i is calculated as q_i/u_i, where q_i and u_i represent the recent query-per-second (QPS) rate and CPU utilization of replica i. Historically, WRR originated in networking [11] but has also been adapted for replica selection as we just described [4, 9].

We now describe a scenario where this approach backfires, even if we could balance CPU load *perfectly*. Suppose that our server job has 100 replicas running on identical machines, each replica allocated 40% of the CPU on its machine. Moreover, suppose that antagonist load is soaking up the full remaining 60% on machines 1 and 2, whereas the other 98 machines have ample spare capacity, safely above 4%. Finally, suppose that our job experiences a temporary spike in demand that would raise its aggregate utilization to 1.1x its allocation, i.e., 44% of each machine, on average. A load balancer that attempts to equalize CPU utilization across replicas will aim to peg each replica at this average. All 100 replicas will exceed their CPU allocation, but the last 98 will be okay because the system will let them momentarily spill outside their allocation to soak up the unused CPU cycles. On the heavily contended machines 1 and 2, CPU isolation mechanisms will typically kick in and hobble those replicas, sometimes in ways that affect *all* queries served by them. Thus, although the problematic load is only the extra 10% on only 2% of our replicas (i.e., ~0.18% of our 1.1x load), it could degrade performance on a full 2% of our queries. Tail latency at p99[3] is likely to spike. Fig. 6 in §5.1 exhibits this behavior on a testbed.

In this overload case, equalizing replica utilizations was exactly the wrong load-balancing policy because the available machines differed greatly in their capacity to absorb additional load. Furthermore, since the difference in available capacity is due to antagonist processes, it cannot be predicted by our job in advance but can potentially be detected at runtime.

Equalizing replica utilizations can be a great idea if all replicas always stay within their allocation, as this maximizes the amount of traffic we can serve, subject to the allocation. Unfortunately, Fig. 3 shows it is easy to trick ourselves. Plotting CPU utilization over 1m time intervals leads us to believe that the replicas are all respecting their allocations, but using 1s intervals reveals greater underlying variability in the signal, with frequent bursts up to nearly twice the limit! In other words, overload is not really a special case; at sufficiently small timescales, there is nearly always *some* replica in overload, even if our aggregate load fits within our job allocation. The only questions are whether this replica is unlucky enough to be on a highly contended machine, and whether the spike lasts long enough for isolation mechanisms to kick in.

Thus, avoiding high tail latency requires some mechanism to alert clients quickly about highly-loaded replicas in real-time. The mechanism should use load signals that are as current as possible, and are highly predictive of high latency when serving future requests. CPU utilization fails to meet these criteria, partly because it is a trailing signal. It must be averaged over a time window to be meaningful, which automatically imposes a lower bound on its staleness. This signal also overlooks other factors that contribute to latency, like contention for locks, memory bandwidth, or other shared resources that are sometimes tough to measure or isolate.

Prequal instead uses two load signals: RIF and latency. RIF is an instantaneous signal (its precise value is available at the time of a probe), and our latency estimates are near-instantaneous as well (§4). Here are our main design goals.

1. Minimize probing overheads. The number of probes per query should be a small constant, and latency estimation (running on server replicas) must be lightweight, taking $O(1)$ or $\tilde{O}(1)$ update time per query.
2. Probing should not add significant latency to the query's critical path. Prequal accomplishes this via asynchronous probing: the current query is assigned using probe responses initiated by previous queries.
3. Minimize tail latency. Prequal removes the worst probes (those with the highest RIF and/or estimated latency) before they can be used for replica selection, in a process inspired by the theory of balanced allocations with memory [14, 16].
4. Limit RAM footprint of query processing on server replicas.[4] Prequal avoids assigning queries to replicas that have an anomalously large RIF, even if their estimated latency is low. The RIF signal does double duty, since it is also a strong leading indicator of future load.

Recall that Prequal can be used either with or without a dedicated load balancing layer (Fig. 1). Advantages of the dedicated layer include (1) keeping probes local when clients

[3] We use expressions like p99 to denote percentiles of a distribution.

[4] RIF has a material impact on RAM because most in-flight queries are in some stage of processing, *not* sitting in a queue. The RPCs themselves are often small.

query server jobs in a distant data center, (2) the balancer often has fewer replicas than the client does, so each one sees a larger fraction of the query stream, hence its probes are fresher (as measured by number of queries landing on a server replica since the most recent probe), (3) software upgrades to the balancer need not touch the clients. Disadvantages include (1) further RPC overhead (latency, CPU, network), (2) it is an extra job to manage, and (3) one still needs to balance the load arriving at the balancers, although that is usually easier because the balancer's work is fairly uniform, and it is usually cheaper to overprovision the balancer than the server.

3 Evaluation of deployment in YouTube

A couple of years ago, we identified load imbalance as the cause of persistent SLO[5] violations in YouTube, prompting us to switch some key services to Prequal. These included YouTube's Homepage service, which is responsible for user-tailored recommendations. With Prequal we saw dramatic improvements across all metrics, including reductions of 2x in tail latency, 5-10x in tail RIF, 10-20% in tail memory usage, 2x in tail CPU utilization, and a near-elimination of errors due to load imbalance. In addition to meeting SLOs, this also allowed us to significantly raise the utilization targets that govern how much traffic we are willing to send to each datacenter, thereby saving significant resources. Based on this success, we rolled out Prequal to many other major services that comprise YouTube, and subsequently to a wide variety of other services at Google, as detailed in §1.

The YouTube service is composed of many subsidiary jobs — answering search queries, providing recommendations, and serving ads, to name a few. Queries from some of these jobs trigger a cascade of subsidiary queries, as illustrated in Fig. 1. Since these jobs are so diverse, their query processing requirements (CPU, RAM, latency, etc.) vary widely as well. In this section we show the results of switching from WRR to Prequal using a small subset of live, user-facing traffic for the Homepage job itself, with the load balancing policies in other parts of the system held constant. Historically, the Homepage was the first server job where we deployed Prequal, while most other jobs still used WRR. At the time of this experiment, most but not all of these were using Prequal. In both cases, the impact of the switchover was similar. We state Prequal's impact here, but defer the details of its design to §4. We then explore this design space via experiments on a testbed (not YouTube) in §5.

The first result to notice is that explicitly balancing on RIF really works, bringing down the tail from ~225 to ~50 (Fig. 4). Since Homepage query processing carries a large amount of per-query state, this reduces our tail RAM usage by 10-20%, allowing us to reduce our RAM footprint accordingly. As shown in Fig. 3, WRR was very effective at maintaining a tight CPU load distribution at the 1m time scale, but much poorer when measuring tail CPU utilization at 1s resolution. Prequal fixes that, dropping the tail by ~2x. As a consequence, Prequal decreased occasional server replica error spikes of 0.01-0.1% down to nearly zero and reduced latency by 10-20% at the median and 40-50% at the tail (Fig. 5).

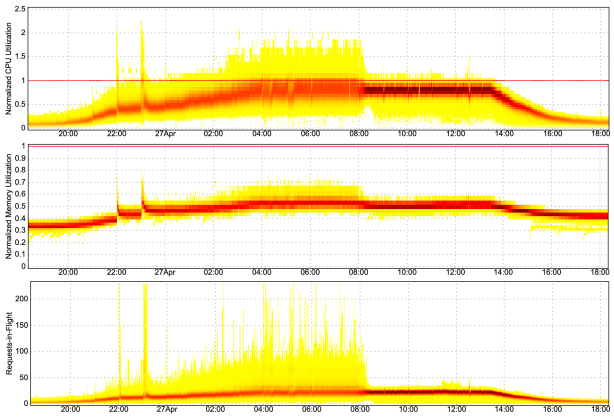


Figure 4: Heatmaps of normalized cpu usage, normalized memory usage, and the number of requests in flight on each YouTube Homepage server replica, first using WRR to balance load before transitioning to Prequal shortly after 08:00.

Fig. 5 normalizes each latency quantile (p50, p99, p99.9) separately, based on a typical value for *that quantile* at the daily traffic trough. This reveals that Prequal does such a good job of pulling in the tail latency at peak load that the p99 and p99.9 actually suffers *less* at peak (in a multiplicative sense) than p50 does. This is the opposite of the behavior one would normally expect, and that we indeed see for WRR.

For these experiments, we configured Prequal to send 5 probes per query, meaning the total number of RPCs is multiplied by 6. Moreover, as an early adopter of Prequal, this service still uses the synchronous probing mode (§4), which adds latency to the critical path. While we intend to migrate to asynchronous probing, our experience with YouTube shows that the improvements we get by pulling in the tails more than compensates for these overheads. This is particularly true for services with relatively heavyweight requests, e.g., tens or hundreds of milliseconds of processing time per query. We set a 3ms timeout for our probes, but most return far quicker than that.[6] The probing delay and the extra CPU spent on processing the probes are both in the noise compared to the savings.

4 System design

When load-balancing a torrent of queries, every millisecond matters. Prequal is designed so that clients can select a replica

[5] SLO = service level objective [10]

[6] Elsewhere at Google, we successfully use a 1ms timeout, and could probably do so here too.

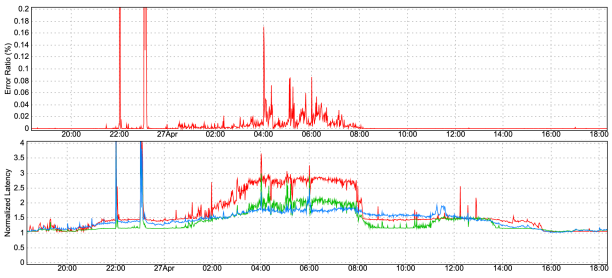


Figure 5: Normalized YouTube Homepage server request error rate and latency p99.9 (red), p99 (green), and p50 (blue). Failures were due to timeouts or load shedding stemming from imbalance. Cutting over from WRR to Prequal shortly after 08:00 eliminated most errors and reduced tail latency by 40-50% and median latency by 5-10%. Each latency quantile is normalized separately to a typical value at daily trough *for that quantile*, which is why it is possible for the p50 line to lie above the other two after the cutover. The spikes near 22:00 and 23:00 are unrelated to the experiment.

based on up-to-date information (ideally no more than a few milliseconds old) while keeping the process of sending and receiving probes out of the critical serving path. Hence, the most consequential design decisions center around how to maintain a pool of high-quality probes and how to assign queries to replicas based on the information in the pool. The following are some key questions.

1. How often should a client probe replicas, and which ones should it probe?
2. What information should replicas report in probe responses, and how should they compute it?
3. When selecting a replica to handle a query, how should a client make use of the information in its probe pool?
4. How should a client manage its pool of probe responses? In particular, when should elements be removed?

Probing rate Clients issue a specified number of probes, r_{probe}, triggered by each query. In addition, they can be configured to issue probes after a specified maximum idle time has been exceeded, to ensure the availability of recent probe responses in the pool even when no queries have arrived in the recent past. The probing rate (per unit time) is therefore the product of the query arrival rate with r_{probe} (or the minimum probing rate, whichever is greater). We link the probing rate to the query rate because that is the rate at which the client must make decisions. It is also strongly tied to how rapidly the load statistics on the replicas (particularly RIF) change over time, although that also depends on the ratio of client replicas to server replicas. Probing at a rate that greatly exceeds the query arrival rate would thus yield many redundant probes, whereas probing at too low a rate runs the risk that probe results will be "stale" (only weakly correlated with the replica's current state) by the time they are used.

Probe destinations are sampled uniformly at random without replacement from the set of available replicas. This design choice is motivated by the theory literature on balanced allocations, which advocates sampling a uniformly random set of probe targets. It also helps to avoid the *thundering herd* phenomenon, when a replica with low estimated latency is inundated with queries from many clients simultaneously seeing it as the best choice, leading to queueing and high latency [15]. This is unlikely to happen in Prequal, where each client's probe pool represents only a small random subset of replicas, and conversely, each replica belongs to the probe pools of only a small (in expectation) random subset of clients.

We allow r_{probe} to be fractional [7], even less than one. Probing consumes a small but nonzero amount of server CPU and network bandwidth, so it is desirable to lower the probing rate to the extent that doing so has only negligible impact on tail latency. Our experiments in §5.3 reveal that $r_{\text{probe}} > 1$ is quite safe from the standpoint of tail latency.

Load signals Prequal includes a server-side module for tracking RIF and latency statistics and responding to probes, as follows. We say that the query *arrives* at the server when the application logic receives the RPC from Stubby, and *finishes* when the application logic hands the response RPC back to Stubby. We define the latency of the query to be the length of this interval, during which the query contributes to this server's RIF count. If there is any application-level queueing, then the latency includes the sojourn time in the queue. However, it is more common for the application to eschew queueing and rely on thread or fiber scheduling instead. We do not attempt to capture the network latency, because all server replicas reside in the same datacenter.

When responding to a probe, the RIF comes from simply checking a counter. The latency estimate is more subtle. When a query finishes, we record its latency, tagged by the value of the RIF counter when it arrived. When a probe prompts us to estimate latency, we consult a set of recent latency values at (or near) the current RIF, and report the median[8]. The per-query overhead of maintaining these data structures is small. At moderate-to-high query arrival rates, the samples are plentiful enough that we base the latency estimates entirely on queries that finished in the last few hundredths of a second.

The probe pool Prequal clients maintain a pool of probe responses to be used in replica selection. Each pool element indicates the replica that responded, the response receipt time,[9] and the load signals discussed above. The pool is capped at a maximum size: we have found that a pool size of 16 suffices to achieve the benefits of Prequal, and the gains from

[7] Each query triggers either $\lfloor r_{\text{probe}} \rfloor$ or $\lceil r_{\text{probe}} \rceil$ probes, rounding deterministically so as to guarantee r_{probe} probes per query in the limit.

[8] The median was chosen as a summary statistic robust to outliers.

[9] The sent time would be ideal, but could introduce clock skew.

increasing beyond 16 are modest. New probe responses are added to the pool upon receipt, evicting a probe if necessary to respect the maximum pool size.

Replica selection Prequal could access many load signals, and we have chosen to focus on RIF and latency. The latency signal is obvious: since we want to minimize latency, why not route queries to the replicas exhibiting minimum latency? The RIF part is important because when the replica must store significant per-query state (as in YouTube), its RAM allocation must be a constant offset plus a term that is linear in its max anticipated RIF. In addition, RIF is an instantaneous signal that is a leading indicator of future load, since it represents the queries processing *right now*. The latency signal is based on observed latencies of queries that have completed *recently*. Both signals are more up-to-date than CPU utilization, which must be averaged over a long time period to be meaningful.

From the RAM perspective, RIF matters only if our choice of replica raises the max RIF. In principle, we would prefer to balance on latency at all other times, although we must temper this impulse because of the leading versus trailing indicator consideration described above.

To minimize both latency and RIF, Prequal selects replicas using a *hot-cold lexicographic* (HCL) rule that labels probes in the pool as either *hot* or *cold* based on their RIF value. Prequal clients maintain an estimate of the distribution of RIF across replicas, based on recent probe responses. They classify pool elements as hot if their RIF exceeds a specified quantile (Q_{RIF}) of the estimated distribution, otherwise cold. In replica selection, if all probes in the pool are hot, then the one with lowest RIF is chosen; otherwise, the cold probe with the lowest latency is chosen. Our experiments (§5.2) suggest that $Q_{\text{RIF}} \in [0.6, 0.9]$ is a good choice, although even 0 is effective (i.e., RIF-only control).

An exception arises if the pool is empty. Then, Prequal simply falls back to selecting a uniformly random replica. In fact, our experience suggests it is useful to invoke this fallback whenever the pool occupancy drops below 2.

One popular approach to combining two signals is to use a linear combination of them with a scaling constant to put them into the same units. In our experiments, HCL outperforms RIF-only (§5.3), which outperforms every non-trivial linear combination of RIF and latency (§5.2 and Appendix A). The HCL approach also nicely captures our hierarchy of concerns: containing latency is nice, but replicas obeying their RAM allocations is often a hard constraint.

Probe reuse and removal Prequal manages the probe pool to avoid three conditions: *staleness*, when the load signals are too old to be accurate; *depletion*, when the pool becomes empty; and *degradation*, when the loads represented in the pool exhibit a selection bias towards replicas with higher load. These are intertwined and the discussion will necessarily be somewhat circular, so we start with the simplest.

Depletion: Prequal employs several probe removal processes (detailed below) to control staleness and degradation, including removing probes when they are used. In order to stave off pool depletion without increasing the probing rate, we can extend the life of each probe by reusing it up to b_{reuse} times. This reuse limit is set according to the formula

$$b_{\text{reuse}} = \max\left\{1, \frac{1+\delta}{(1-m/n)\cdot r_{\text{probe}} - r_{\text{remove}}}\right\}, \qquad (1)$$

where $\delta > 0$ is a configuration parameter that governs the net rate at which probes accumulate in the pool, m is the maximum pool size, n is the number of replicas, r_{probe} is the probing rate discussed above, and r_{remove} is the rate of probe removal discussed below. We always set $b_{\text{reuse}} \geq 1$, and when it is fractional, we randomly round it to its floor or ceiling so as to preserve the expectation.

Staleness occurs for two reasons: aging and overuse. As a probe ages, its load data becomes less accurate because the replica receives new queries and finishes ones it already had. Overuse is a special case of aging that we can partially mitigate; namely, when the client *itself* sends a query to that replica, it can compensate by incrementing the RIF value on that probe. Ideally, we would also increase its latency estimate, but currently we do not. Part of Prequal's solution for general staleness is to set a time limit on probes and remove them from the pool when their age exceeds this limit. In addition, whenever a new probe arrives that would increase the pool beyond its size limit, we drop the oldest probe.

Degradation is the subtlest of these phenomena. If the probes corresponding to lightly-loaded replicas are continually being selected and removed from the pool, then the probes that remain after many rounds of replica selection correspond to highly-loaded replicas. To avoid this, Prequal periodically removes the worst probe from the pool. This is analogous to — but much more permissive than — the use of only the least-loaded of d randomly selected bins in the standard power-of-d-choices model. It turns out that broadening the bin-selection policy from "use only the best of d random probes" to "avoid using the worst of d random probes" qualitatively preserves the theoretical guarantees for the power-of-d-choices model [21]. When removing the worst probe, Prequal alternates between two rules: removing the oldest probe (i.e., worst age) and removing the probe deemed worst according to the same ranking used for replica selection (but in reverse): if at least one probe in the pool is hot, the hot probe with highest RIF is removed; otherwise, the cold probe with highest latency is removed.

We define a r_{remove} parameter, and delete that many probes from the pool with each query. As with the probing rate, this number may be fractional, in which case the actual number of probes removed is always either the floor or the ceiling, deterministically rounded to achieve the configured rate of probe removal on average. By alternating our removals between oldest and most loaded, we achieve a unified approach to avoid both staleness and degradation (on top of the probe timeouts).

To summarize, Prequal removes probes from the pool for four reasons. It evicts the oldest probe in the pool when necessary to avoid exceeding the maximum pool size. Probes are removed once they reach their reuse budget or when their age exceeds a timeout threshold. Additionally, they are removed at a configurable rate per query, alternating between worst and oldest.

Synchronous mode So far, we described Prequal with asynchronous probing (i.e., async mode), which maintains a probe pool to take probing off the critical serving path. The asynchrony is helpful in many use cases, such as when the probe latency is non-negligible or the CPU overhead of probing is high compared to the queries. However, we have also implemented a sync mode of Prequal in which there is no probe pool. Instead, when a query arrives, the client issues some d probes (at least 2, typically 3-5) to random replicas, waits to receive a sufficient number of responses (typically $d-1$) and then chooses among those using the same replica selection rule described above. We describe the sync mode here because it has been used in YouTube services as reported in §3, but all of the experiments reported in §5 use async Prequal because that mode is better for most uses.

One significant use case that requires sync mode is when replicas hold state that influences the cost of query execution, e.g., a replica may cache certain data in memory to prevent reading from a slower storage layer. Sync probing allows us to include relevant information from the query in the probe. If the replica then determines that it can execute that query more efficiently because of data it already has in the cache, then it can manipulate its reported load so as to attract the query, e.g., by scaling down its reported load by 10x. We have used sync Prequal in this way for part of YouTube.

Error aversion to avoid sinkholing Suppose a certain replica has a problem (such as misconfiguration) that causes it to process queries very quickly by immediately returning errors for a non-trivial fraction of its queries. Then its latency on the remaining successful queries, RIF, CPU utilization, and other metrics will make it appear less loaded than it normally would, given the amount of traffic sent its way. If the load balancer is not smart about this, this replica can attract more and more traffic, in a phenomenon known as *sinkholing*. Prequal includes some heuristics to avoid sinkholing, but since they are not central to our contribution, we have chosen to simplify our exposition by omitting details.

5 Testbed evaluation

In §3 we reported on our experience deploying Prequal in YouTube. The comparison between Prequal and WRR in that section constitutes one type of evaluation, under conditions that arguably represent the ground-truth definition of a realistic workload. However, evaluating the deployed version of Prequal in production does not shed light on how individual aspects of its design affect its performance. In this section we report on a set of tests performed on a testbed environment that allows for more controlled experiments with Prequal and baselines, while preserving key features of the production environment for which Prequal was designed: variable query costs and unpredictable time-varying antagonist load.

We first describe our testbed and the baseline system parameters. Within each experiment, we explain which parameters we changed from the baseline.

Our testbed consists of one client job and one server job, each comprising multiple replicas running on distinct but identical physical machines colocated in the same datacenter. The queries represent a very simple CPU-intensive workload: they simply iterate an expensive hash function. In order to simulate variability in query costs, we vary the number of iterations, drawing it from a normal distribution whose standard deviation equals its mean (then truncated at zero). The jobs are running within a standard Google datacenter, on commodity multicore machines, using Google's standard isolation mechanisms, and the antagonist traffic is just whatever we happen to encounter in the wild.

The system configuration in our experiments, as well as Prequal's configuration parameters, exemplify a representative operating environment in which Prequal performs as intended and its benefits become evident. All experiments use 100 client replicas and 100 server replicas, and each of the server replicas is allocated 10% of the machine's CPU. In addition, the probe pool size is 16, probes age out of the pool after one second, and the net probe-pool drift rate in Equation (1) is $\delta = 1$. Unless otherwise specified, we set $Q_{\text{RIF}} = 2^{-0.25} \approx 0.84$ and $r_{\text{remove}} = 1$. We use 3 probes per query as our baseline probe rate to stay safely away from probe rates low enough to impact performance. We target different aggregate utilizations in each experiment as necessary to induce the behaviors we wish to illustrate. Wherever we plot CPU utilization, it is scaled as a percentage of the allocation.

Some of the plots in these experiments show quantiles of RIF values. Since this data is all integer, the quantiles should be integer as well. However, when our monitoring system builds histograms, all instances of an integer k are uniformly smeared across the interval $[k-\frac{1}{2}, k+\frac{1}{2})$. For this reason, the plots of our RIF quantiles contain fractional values.

5.1 Robustness to variable antagonist load

In this experiment, we start with the aggregate CPU load at about 75% of our allocation, and ramp it up in 8 multiplicative steps of 10/9, yielding 0.75x, 0.83x, 0.93x, 1.03x, 1.14x, 1.27x, 1.41x, 1.57x, and 1.74x our allocation. We increased the load by increasing the aggregate query rate from about 5.6k queries per second (henceforth, qps) to around 13k qps, while holding the mean work per query constant. Within each load level, we use WRR for the first half of the period, then Prequal for the second half.

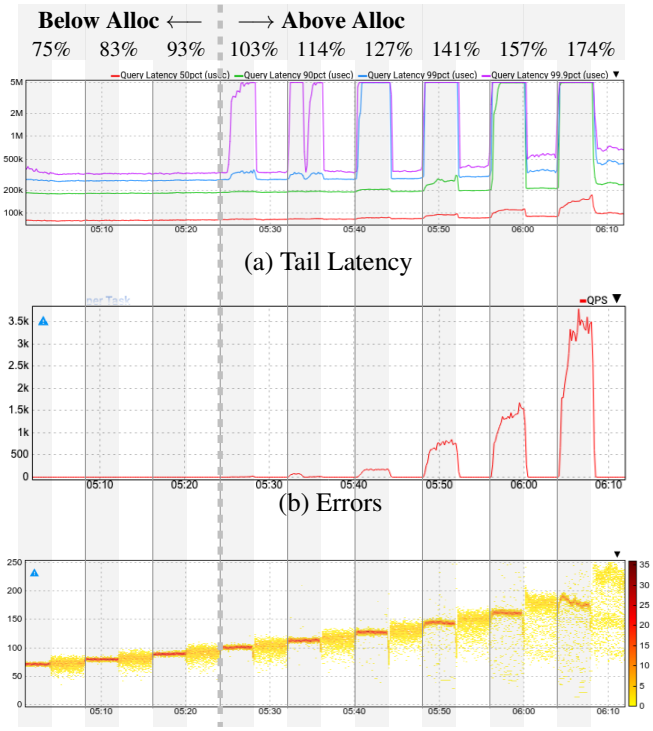


Figure 6: Load ramp experiment. Gray background denotes WRR policy, white denotes Prequal. Note that tail latency is plotted on a log scale.

The top plot in Fig. 6 shows the effect on median and tail latency, on a log scale, measured in microseconds. We employ a 5s timeout for each query, so the graph tops out at 5s. In the first three steps (below allocation) the latency distribution stays steady for both WRR and Prequal at roughly 80ms (p50), 182ms (p90), 265ms (p99), and 325ms (p99.9). Then, as soon as we first exceed our CPU allocation (by 1.03x) in step 4, the behaviors of WRR and Prequal rapidly diverge.

For WRR, the p99.9 latency maxes out at the 5s limit, while p99 jumps about 26% to 335ms, slightly above the previous p99.9. By step 6 (1.27x allocation), p99 has maxed out, and p90 is starting to show a visible bump to ~203ms (+11%). By step 8 (1.57x allocation), p90 has maxed out, and even p50 has risen to ~111ms (+39%). By step 9, even p50 has risen to ~150ms (+88%).

In contrast, Prequal suffers almost no visible change at p99.9 in load steps 4 and 5, and by step 6, p99.9 rises only to ~350ms (+8%). It does begin to rise appreciably at load step 7 (1.41x allocation), but even at step 9 it is still contained around 700ms, far from the 5s timeout. Meanwhile p99 latency does not visibly degrade until load step 7, p90 does so around step 8, and p50 does so around step 9.

The second plot shows error rates, which are zero for both WRR and Prequal up through step 3. Note that this is the *absolute* number of errors per second, not a percentage. WRR starts showing a very small error rate (5 to 20 errors/s) at load step 4. By load step 5, the error rate rises visibly, increasing much faster than the incoming qps. By step 9, more than a quarter of all queries are returning errors. Essentially all of these are "deadline exceeded" errors, from queries that hit their 5s deadline. In contrast, Prequal returns *zero errors* at all load levels in this experiment.

The bottom plot shows a heatmap of the CPU utilization distributions for WRR and Prequal. Here, one can see that WRR is doing a superb job at accomplishing what it was designed to do: balance CPU load. In contrast, the CPU distribution for Prequal is substantially looser. So why are the latency and error metrics for WRR so bad? It is precisely the explanation that we offered as motivation in §2. Digging further into the server-level data for WRR (not pictured), we see a strong correlation between the server replicas with high antagonist load and ones where our latency suffers. Meanwhile, Prequal shifts load away from those server replicas to avoid the worst of the latency impact.

This collection of results explains the title of our paper. Although it is counterintuitive for most people, in this case the load balancer that achieves near-perfect load balance (WRR) is clearly worse (higher errors and latency). That is because *the real goal of a load balancer is not to balance load:* it is to direct load where capacity is available.

These results are possible only because many server replicas in our system are *not* fully allocated with antagonists, and at any given moment, many of these antagonists are not using their full CPU allocations. Prequal allows our job to shift load among replicas so as to fit into these cracks of temporary spare capacity. This has two implications for service provisioning: (1) We can provision much more aggressively, because Prequal can deal with transient load spikes. (2) In theory, this could even enable overcommitment, because it enlarges the relevant resource pool from the level of one machine to a much larger set of machines, enabling more effective statistical multiplexing.

5.2 Replica selection rule

In modern datacenters with thousands of machines, it is common for the machines to span multiple hardware generations and processor architectures. When provisioning a job across multiple machines, we can try to account for these differences by applying a performance multiplier to different machine types, e.g., 1 CPU core of type A is equivalent to 1.2 CPU cores of type B. However, it gets tricky to do this well at scale because the performance skew is often highly dependent on the particular workload, e.g., YouTube may be more suited to one of the hardware generations, while Google Maps is better suited to the other. As a result, when our replicas are assigned to machines with different hardware, their effective throughput becomes heterogeneous, and it is the faster

machines that exhibit lower latency when unloaded (by definition). In this case, the way to minimize mean latency is to fill the fast machines until their latency degrades to match the slow machines, and then fill both fast and slow together.

There are various ways that a replica selection rule could be designed to fulfill this goal. We experimented with nine replica selection rules ranging from very simple baselines to sophisticated rules used in popular open-source load balancers (e.g. [26]) and in the research literature (e.g. [23]).

In describing the replica selection rules we evaluated, we distinguish between *client-local* and *server-local* quantities. The former are measured at the client (or load balancer) itself, the latter are measured at the server replica and reported back to the client (or load balancer). For example, client-local RIF refers to the number of queries the client has sent to the replica that have not yet yielded responses. Server-local RIF refers to the total number of queries the replica has received (from all clients) that it has not yet finished processing. In the literature on load balancing of HTTP traffic there is also a distinction between individual requests and connections; the latter may be long-lived, comprising multiple requests. Since the queries in our work represent short-lived connections, we ignore this distinction in our experiment, treating connections and requests as synonymous.

We evaluated the following replica selection rules.

- **Random** selects a uniformly random replica.
- **Round Robin (RR)** cycles through the replicas, keeping track of the most recently chosen one and always selecting the next available replica in cyclic order.
- **Weighted Round Robin (WRR)** is as described in §2.
- **Least Loaded (LL)** represents the LeastLoaded policy implemented in the NGINX and Envoy reverse proxies [6, 19]. It chooses the available replica with the least client-local RIF, breaking ties in favor of one nearest to the most-recently-chosen replica in cyclic order.
- **Least Loaded with Power of Two Choices (LL-Po2C)** samples two available replicas uniformly at random and selects the one with the least client-local RIF. This modification of LL is also implemented in NGINX and Envoy.
- **YARP-Po2C** is a replica selection rule from Microsoft's YARP reverse proxy library [26] based on power-of-two-choices. All replicas are periodically polled to report their (server-local) RIF. Replica selection is performed by randomly sampling two replicas and selecting the one with lower reported RIF. In our experiments we set the polling interval to 500ms, a 30x faster rate of polling than in the standard YARP-Po2C implementation. We chose the 500ms interval to approximately equalize the total number of RIF reports each client receives, per second, with the number of probe responses received per second by Prequal clients in our experiment.
- **Linear, C3, and Prequal** all use the asynchronous probing method described in §4, but they differ in the scoring rule used to select a replica from the pool of probe responses.
- **Linear** uses a linear combination of RIF and latency. To represent RIF and latency in comparable units, we scale RIF by the median query processing time measured on replicas with one request in flight. A replica's score is defined to be an equally weighted average[10] of latency and scaled RIF.
- **C3** in this paper uses the replica scoring function described in [23] with Prequal's probing logic. It computes a RIF estimate for each replica as $\hat{q} = 1 + os \cdot n + \bar{q}$, where os is the client-local RIF, n is the number of clients participating in the job, and $\bar{q}$ is an exponentially weighted moving average of the server-local RIF. It then computes a score for each replica as $\Psi = (R - \mu^{-1}) + \hat{q}^3 \cdot \mu^{-1}$, where R and μ^{-1} are exponentially weighted moving averages of the client-local and server-local response time, respectively. See [23] for intuitions justifying the formulae for $\hat{q}$ and Ψ.
- **Prequal** uses the HCL replica selection rule described in §4, with the RIF limit quantile set to $Q_{\mathrm{RIF}} = 0.75$.

The results of our experiment are depicted in Fig. 7. We tested the replica selection rules at two different levels of aggregate CPU load — 70% and 90% of our allocation — and reported two quantiles of tail latency, the 90th percentile (p90) and 99th percentile (p99).

The replica selection rules that performed best at all load levels and latency quantiles were C3 and Prequal. What these rules have in common is that they incorporate server-local quantities, they penalize high (server-local) RIF severely, and they favor low-latency replicas when there are multiple replicas with low RIF. In the case of Prequal this behavior is evident in the way it distinguishes between hot and cold replicas. In the case of C3 it is implicit in the scoring function's cubic dependence on estimated queue size, $\hat{q}$: when $\hat{q}$ is near zero it contributes negligibly to the score, but it rapidly grows to predominate the score as $\hat{q}$ moves away from zero.

Comparing C3 and Prequal, one sees a small quantitative advantage for Prequal (3-8%) at all load levels and latency quantiles we tested. Another convenient feature of Prequal's replica selection rule is that it has a tunable parameter, Q_{RIF}, that allows it to span the full range from RIF-only to latency-only control, depending on the needs of the application. Additionally, Prequal uses fewer signals than C3, simplifying the implementation and monitoring.

It is noteworthy that the LL policy, which bases replica selection on client-local rather than server-local RIF, experiences high p99 latency when load is at 70% of allocated

[10] In the supplementary material, we report on an experiment showing that the performance of the Linear rule is equally poor for most weightings, except when it degenerates to RIF-only control.

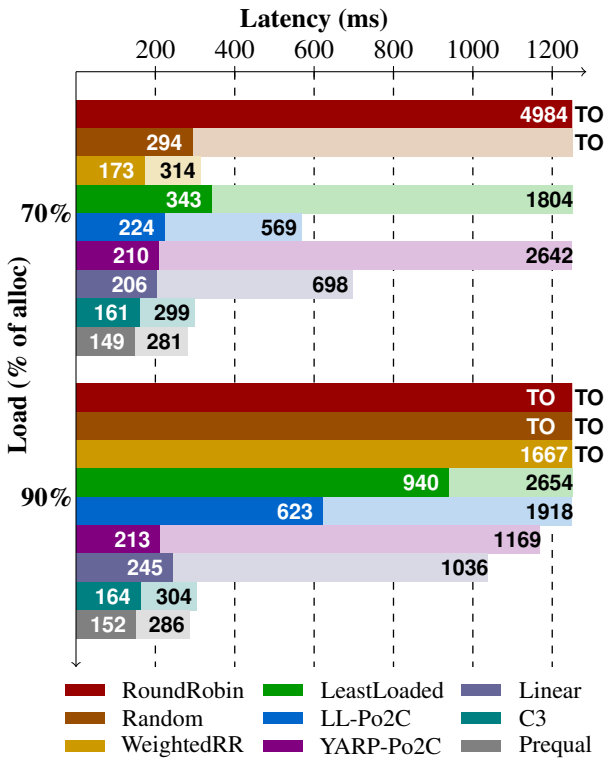
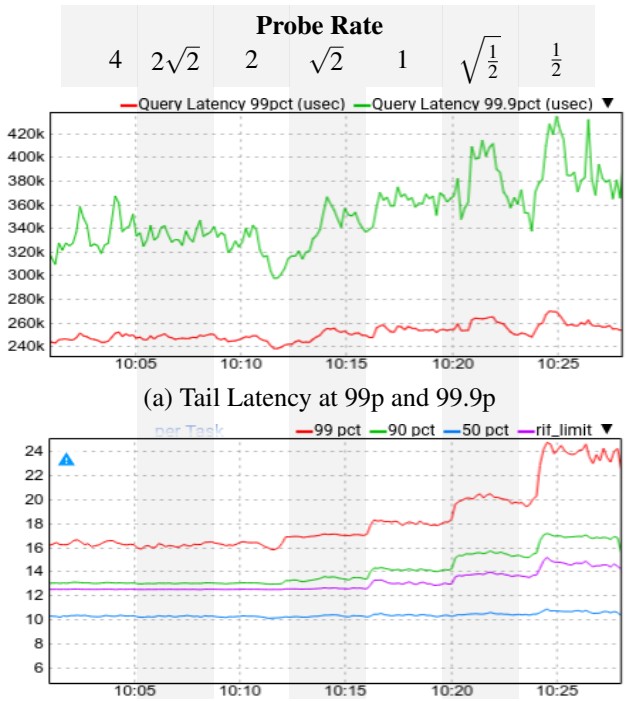


Figure 7: Comparison of replica selection rules. Dark portions of bars represent 90th percentile, light portions are 99th percentile. Bars representing latencies exceeding 1.25sec are truncated. Timeouts are represented by 'TO'.



(a) Tail Latency at 99p and 99.9p

(b) RIF Quantiles

Figure 8: Probing rate experiment. Rates (top) are expressed in probes/query.

capacity. Even when a server has no active connections from a given client, it can be highly loaded with queries from other clients. If this possibility happens more than 1% of the time, that is enough to impact p99 latency. When load is 90% of allocation, we see that even p90 latency suffers heavily under the LL policy. Combining LL with Po2C improves latency at all load levels and quantiles in our experiments, but the LL-Po2C rule still lags far behind Prequal and C3.

YARP-Po2C's selection rule using server-local RIF fares better at high load than LL-Po2C which uses client-local RIF, as one might expect. However, its decisions are often based on stale information due to the infrequent polling of replicas, and this adversely affects latency.

Interestingly, the replica selection rule based on a 50-50 linear combination of latency and RIF performed much worse than Prequal's and C3's scoring rules. This indicates that a linear function of RIF doesn't penalize high RIF severely enough compared to C3's cubic function or Prequal's strict prioritization of cold replicas over hot ones.

Finally, the WRR policy performs quite well when load was at 70% of allocation, but its p99 latency suffers greatly at 90% load. This is consistent with the results in §5.1, where the tail latency experienced by WRR increases sharply in response to a modest increase in aggregate CPU load. (WRR experienced this crossover at different load levels in the two experiments, probably because of differing amounts of antagonist load.)

5.3 Tunable parameters

Prequal has a few tunable parameters that influence its behavior and performance. In this section we evaluate the impact of two of those parameters, probe rate and RIF limit threshold. The first influences the freshness of the load signals, while the second influences the propensity for steering queries toward low estimated latency versus avoiding high RIF.

Probing Rate In the YouTube application, the probes are so light compared to the queries themselves that the probing overhead is negligible. Unfortunately, this is not true for all applications. Thus, we seek to identify the minimum probing rate we can tolerate while retaining the benefits of Prequal.

In this experiment, we ramp down the probing rate from 4x to $\frac{1}{2}$x the query rate, in 6 multiplicative steps of $\sqrt{2}$ each, while keeping the probe removal rate steady at 0.25 per query, and b_{reuse} increases to compensate, guided by (1). In order to magnify the effects, we ran the system very hot, at roughly 1.5x our CPU allocation throughout.

Fig. 8 shows the resulting latency and RIF, as well as the θ_{RIF} control parameter. The take-home point is that Prequal is fairly insensitive to the probing rate until we drop below

one probe per query, at which point the negative effects become significant. At probing rates of $\frac{1}{\sqrt{2}}$x and $\frac{1}{2}$x, the tail RIF distributions jump visibly, and this change is echoed by both latency quantiles. Anecdotally, we have observed this phenomenon across many similar experiments, always around 1 probe per query.

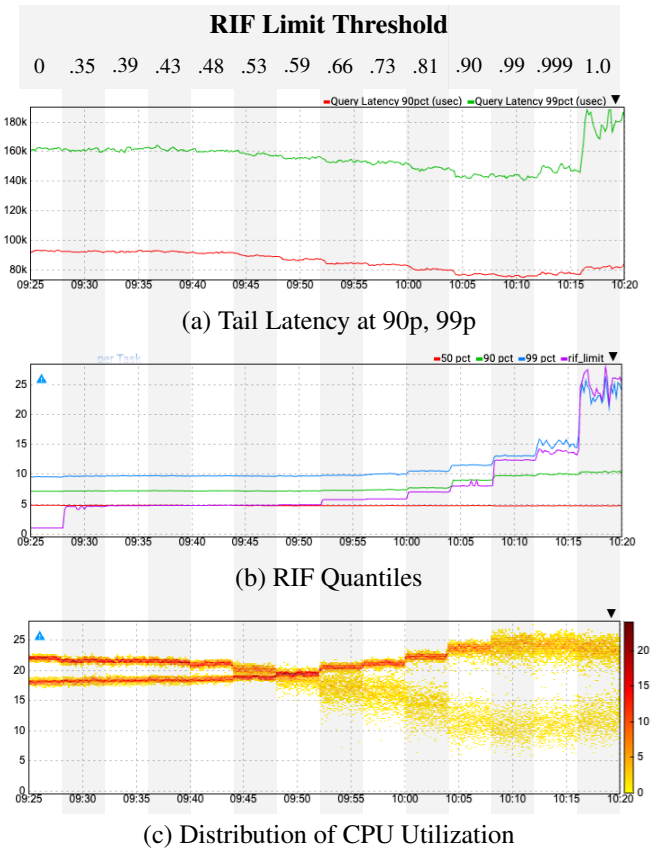
RIF Quantile We now present an experiment to explore the RIF vs latency tradeoff expressed by the quantile that distinguishes hot from cold replicas in the HCL rule. To do this, we partition our 100 server replicas into 50 fast and 50 slow replicas, the latter group standing in for servers from earlier hardware generations. We artificially inflate the query work by a factor of 2 on the slow replicas, thereby causing them to burn 2x the CPU cycles and act as if they were 2x slower. We vary the Q_{RIF} parameter over the course of the experiment, starting at 0 (meaning RIF-only control), and ramping it up from $0.9^{10} \approx 0.35$ to 0.9, in steps of $\frac{10}{9}$, then to 0.99, 0.999, and 1 (meaning latency-only control). Throughout, the mean load is held steady at about 75% of the CPU allocation.

Note that there is actually a discontinuity in the behavior of Prequal between the last two steps. At $Q_{\text{RIF}} = 0.999$, the RIF limit is effectively the maximum RIF: any replica tied for the max is considered hot. In contrast, when $Q_{\text{RIF}} = 1$, the RIF limit is ∞, and every replica is considered cold.

Fig. 9 shows the results. The RIF limit threshold Q_{RIF} increases from 0 on the left (pure RIF control) to 1 on the right (pure latency control). The top two plots show the p99, p90, and p50 latencies. As we would expect, all three latency quantiles go down as we shift more towards latency-based control, up through step 12 ($Q_{\text{RIF}} = 0.99$). Specifically, p99 drops from ~162ms to ~142ms (-12%), p90 drops from 93ms to 75ms (-19%) and p50 drops from ~34.5ms to ~31ms (-10%). At step 13, all three quantiles begin to edge up slightly. When we switch to full latency control in step 14, all quantiles move up sharply, especially p99, which jumps up from ~148ms (step 13) to about 178ms (+20%). Even more dramatic is p99.9 (not shown), which falls from 210ms in step 0 to 198ms in step 12, rises back to 210ms in step 13, and then fluctuates chaotically in the 337ms to 502ms range in step 14 (1.6x to 2.4x, compared to step 13).

It appears that even a tiny bit of RIF control goes a long way. Why is it that pure latency control results in such worse latency than $Q_{\text{RIF}} = 0.999$? We hinted at our interpretation earlier: RIF is a valuable leading signal of load, so ignoring it entirely is a bad idea.

The bottom plot shows the CPU utilization distribution. Notice the two crossing bands, which correspond to the "slow" (i.e., even) and "fast" (i.e., odd) replicas. The slow (respectively, fast) replicas correspond to the band that is decreasing (respectively, increasing) to the right. This is exactly as we would expect, as increasing Q_{RIF} means we balance more often based on latency, which favors the fast replicas. In addition, the CPU distributions are tighter to the left, where RIF control is higher. This is because RIF is quite a good predictor of future CPU utilization.



(a) Tail Latency at 90p, 99p

(b) RIF Quantiles

(c) Distribution of CPU Utilization

Figure 9: RIF Limit experiment. Q_{RIF} varies from 0 at left (pure RIF control) to 1 at right (pure latency control).

This experiment justifies the HCL rule, since turning the dial towards more latency-based control does indeed decrease latency, and turning it up even as high as 0.73 (step 9) has next to no impact on RIF. At step 7 ($Q_{\text{RIF}} = 0.59$), all three RIF quantiles are just as good as with RIF-only control, despite the fact that the vast majority of queries are being routed based on latency. In this experiment, the probe pool has size 16 and is nearly always full. Because of the use-best and remove-worst processes, the probe pool is not uniformly random. But if it were, then the probability of all 16 probes being hot would be roughly $2^{-16} \approx 1.5 \times 10^{-5}$, since θ_{RIF} and p50 RIF are both 5 during step 7. Otherwise, the query is routed to a cold replica based on latency. This is what allows us to have the best of both worlds, most of the time: simultaneously routing based on latency *and* avoiding the highest-RIF replicas.

6 Related work

The "power of two choices" paradigm for randomized load balancing was first analyzed in the influential work of Broder et al. [2] and Mitzenmacher [18]. Over the years, these ideas

made their way from the theory literature into a wide variety of applications in industry, as noted in the prize citation for the 2020 ACM Paris Kanellakis Theory and Practice Award [1]. Theoretical results on the power of two choices initially pertained to the case of throwing n balls into n bins, but they were soon generalized to the "heavily loaded" setting, when balls exceed bins [3]. Qualitatively, these results show that load-balancing processes based on randomly sampling multiple choices and selecting the best ones vastly outperform simpler procedures based on sampling a single bin for each ball. Subsequent theoretical research affirmed that this finding is surprisingly robust to varying the modeling assumptions or the specific rule for selecting among the sampled bins; see, for example, the surveys [17, 25]. Of particular relevance to the design of Prequal is the generalization to load-balancing processes with memory [13, 14, 16], the closest analogue in the theory literature to our use of asynchronous probing.

The need for datacenter task scheduling with sub-second latencies has prompted a great deal of research into decentralized load balancers. Several proposed systems in this branch of the load balancing literature leverage the power-of-d-choices (PodC) paradigm. One prominent example is Sparrow [20], a scheduler for highly parallel jobs that each consist of a large number of tasks. Sparrow aims to minimize the job response time, i.e. the time when the last constituent task of a job finishes executing. This goal motivates a different set of design choices, combining batch sampling and late binding: a client whose job comprises m tasks places $d \cdot m$ reservations on randomly selected server replicas; the m replicas that become available soonest request to run the m tasks, and the other $(d-1) \cdot m$ reservations are canceled. This achieves a similar effect to probing $d \cdot m$ replicas and selecting the m best probe responses, with reservations playing the role of probes. Our setting differs in that jobs are not batched, which allows us to substitute a simpler and more efficient probing mechanism, avoiding the use of reservations and late binding which would incur undesirable overheads in our environment. Building upon Sparrow's batch sampling idea, Eagle [5] is a hybrid scheduler that incorporates a distributed probe-based scheduler using batch sampling to handle short-lived tasks, alongside a centralized scheduler that places long tasks on replicas. The hybrid scheduler design makes sense in settings where a wide disparity in job sizes makes head-of-line blocking a potentially serious problem; in our setting the jobs (queries) are short-lived, allowing the fully distributed load-balancing solution offered by Prequal to perform well in the production environment of YouTube and in our testbed.

A centralized PodC-based load-balancer is incorporated into RackSched [27], where it runs on a top-of-rack switch to make microsecond-scale inter-server scheduling decisions enabling rack-scale computing. Client-based probing solutions such as ours are inapplicable in their setting, where the resource cost of probing is comparable to the resource cost of serving requests. Instead of probing, their servers piggyback load signals into normal traffic and the scheduler treats these load signals as implicit probes. Another rack-scale load balancing system using a RIF-based queueing policy is R2P2 [12], whose JBSQ(n) policy is a variation on the LeastLoaded policy discussed in §5.2. Both RackSched and R2P2 rely on all connections passing through a single router or switch; as explained in §2, at the scale of a service such as YouTube using a single load balancer is not a viable option.

C3 [23] is another distributed load balancer with the same design goal as Prequal— minimizing tail latency by adaptively selecting replicas in response to real-time load feedback — but a very different methodology using much more state. C3 uses distributed rate control and backpressure, with every client maintaining exponentially weighted moving average estimates of latency and RIF for every server replica, and using a token-bucket based rate limiter for each server replica. Prequal's design, in which the only state clients maintain is a probe pool of bounded size, is better suited to load balancing at the scale of a system such as YouTube.

Many existing load balancing systems (e.g., NGINX [19], Envoy [7], Finagle [8], YARP [26]) offer some variant of PodC, and most of them offer (client-local) RIF, latency, or some combination as the load balancing signal. As detailed in §5.2, the use of signals local to the client or load-balancer adversely affects performance at scale, when many clients or balancers might be sharing the same pool of replicas.

A number of other production systems leverage PodC load balancing. Netflix's Zuul [22] incorporates decentralized routing of requests from load balancers to backends by randomly considering two candidates and using a combination of local and piggybacked RIF statistics from previous responses to choose the least-loaded server. Unlike Prequal, Zuul eschews active probing and does not incorporate latency information. NGINX's implementation [19], like Zuul, is probe-less, but its selection criterion is configurable: "least-connections" uses the lowest RIF from a given balancer to each backend and "least-time" uses estimated latency based on prior requests and the number of current connections. Given our experience with Prequal, we believe it is likely that the performance of the load-balancing mechanisms described in [19, 22] could be improved by supplementing the piggybacked data with active probing.

Acknowledgments

We wish to recognize the contributions of our current and former Google colleagues Melika Abolhassani, Doug Rohde, and Aliaksei Kandratsenka, who built and experimented extensively with an earlier prototype of Prequal. Their previous yeoman's work gave us a large head start on the current iteration of the project. We have also benefitted from helpful discussions with David Applegate, Thomas Adamcik, Pratik Worah, and Soheil Yeganeh, along with excellent suggestions from Philip Fisher-Ogden on the exposition.

References

[1] 2020 ACM Paris Kanellakis Theory and Practice Award. https://www.acm.org/media-center/2021/may/technical-awards-2020, 2020. Accessed: 2023-05-02.

[2] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999.

[3] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced allocations: The heavily loaded case. *SIAM J. Comput.*, 35(6):1350–1385, jun 2006.

[4] Alejandro Forero Cuervo. Load balancing in the datacenter. In Jennifer Petoff, Niall Murphy, Betsy Beyer, and Chris Jones, editors, *Site Reliability Engineering: How Google Runs Production Systems*, chapter 20, pages 231–247. O'Reilly, Sebastopol, 2016.

[5] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In Marcos K. Aguilera, Brian Cooper, and Yanlei Diao, editors, *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*, pages 497–509. ACM, 2016.

[6] Envoy github repository. https://github.com/envoyproxy/envoy. Accessed: 2023-09-21.

[7] Envoy homepage. https://www.envoyproxy.io. Accessed: 2023-09-21.

[8] Deterministic Aperture: A distributed, load balancing algorithm. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2019/daperture-load-balancer, 2019. Accessed: 2023-09-21.

[9] gRPC Weighted Round Robin load balancing component. https://github.com/grpc/grpc/tree/master/src/core/ext/filters/client_channel/lb_policy/weighted_round_robin. Accessed: 2023-05-03.

[10] Chris Jones, John Wilkes, Niall Murphy, and Cody Smith. Service level objectives. In Jennifer Petoff, Niall Murphy, Betsy Beyer, and Chris Jones, editors, *Site Reliability Engineering: How Google Runs Production Systems*, chapter 4, pages 37–48. O'Reilly, Sebastopol, 2016.

[11] Manolis Katevenis, Stefanos Sidiropoulos, and Costas Courcoubetis. Weighted round-robin cell multiplexing in a general-purpose ATM switch chip. *IEEE Journal on Selected Areas in Communications*, 9(8):1265–1279, 1991.

[12] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. {R2P2}: Making {RPCs} first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, 2019.

[13] Dimitrios Los, Thomas Sauerwald, and John Sylvester. Balanced allocations: Caching and packing, twinning and thinning. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1847–1874, 2022.

[14] Dimitrios Los, Thomas Sauerwald, and John Sylvester. Balanced allocations with heterogeneous bins: The power of memory. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 4448–4477, 2023.

[15] Michael Mitzenmacher. How useful is old information? *IEEE Trans. Parallel Distributed Syst.*, 11(1):6–20, 2000.

[16] Michael Mitzenmacher, Balaji Prabhakar, and Devavrat Shah. Load balancing with memory. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 799–808. IEEE, 2002.

[17] Michael Mitzenmacher, Andréa W. Richa, and Ramesh Sitaraman. The power of two random choices: A survey of techniques and results. In Sanguthevar Rajasekaran, Panos M. Pardalos, and José Rolim, editors, *Handbook of Randomized Computing*, volume 1, pages 255–312. Springer Science & Business Media, 2001.

[18] Michael David Mitzenmacher. *The Power of Two Choices in Randomized Load Balancing*. PhD thesis, UC Berkeley, 1996.

[19] Nginx and the "Power of Two Choices" load-balancing algorithm. https://www.nginx.com/blog/nginx-power-of-two-choices-load-balancing-algorithm/, 2018. Accessed: 2023-05-04.

[20] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 69–84, 2013.

[21] Gahyun Park. A generalization of multiple choice balls-into-bins. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 297–298, 2011.

[22] Rethinking Netflix's edge load balancing. https://netflixtechblog.com/netflix-edge-load-balancing-695308b5548c, 2018. Accessed: 2023-05-04.

[23] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2015)*, pages 513–527, 2015.

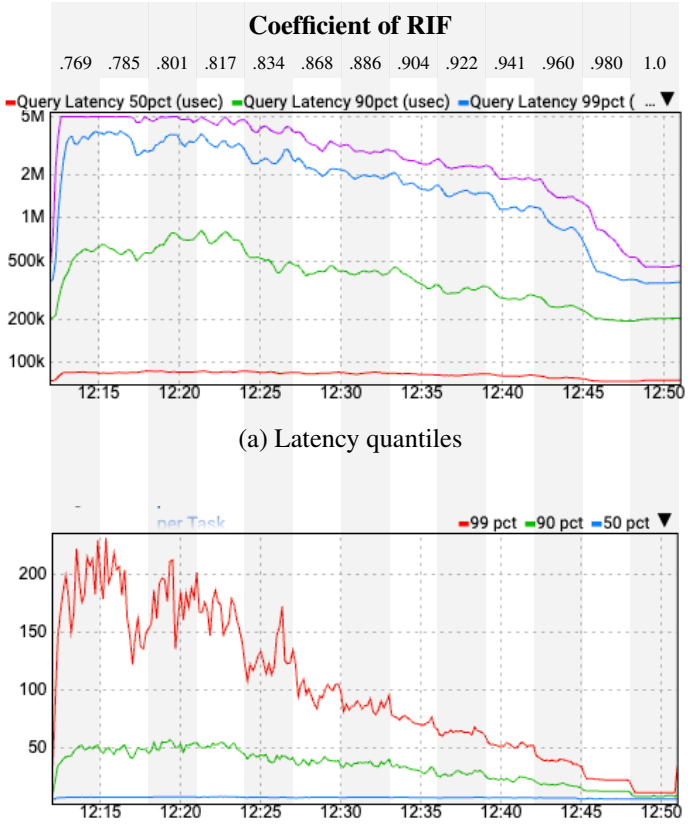
[24] Varun Talwar. gRPC: a true internet-scale RPC framework is now 1.0 and ready for production deployments. https://cloud.google.com/blog/products/gcp/grpc-a-true-internet-scale-rpc-framework-is-now-1-and-ready-for-production-deployments, 2016. Accessed: 2023-09-21.

[25] Udi Wieder. Hashing, load balancing and multiple choice. *Foundations and Trends® in Theoretical Computer Science*, 12(3–4):275–379, 2017.

[26] YARP: Yet Another Reverse Proxy. https://microsoft.github.io/reverse-proxy/. Accessed: 2023-09-21.

[27] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020*, pages 1225–1240. USENIX Association, 2020.

A Linear combinations of latency and RIF



(a) Latency quantiles

(b) RIF quantiles

Figure 10: Evaluating replica selection rules based on linear combination of latency and RIF.

To evaluate the effectiveness of replica selection rules that use a linear combination of latency and RIF, we used our testbed[11] to experiment with a variant of Prequal that uses the some asynchronous probing method detailed in §4, modified to select replicas using a linear combination of latency and RIF. In other words, the HCL replica selection rule was replaced with one that chooses among the replicas represented in the probe pool by minimizing the score

$$\text{score}_i^{\lambda} = (1-\lambda)\cdot \text{latency}_i + \lambda \cdot \alpha \cdot \text{RIF}_i \qquad (2)$$

where latency_i, RIF_i, respectively, denote the latency and RIF in the probe response from replica i, α is a scale factor applied to RIF to convert it into the same units as latency, and $\lambda \in [0,1]$ is a tunable parameter that adjusts the relative weight given to latency and RIF. Setting $\lambda = 0$ corresponds to latency-only control, whereas $\lambda = 1$ corresponds to RIF-only control.

To set the scale factor α, we used the approximate median query response time for server replicas with one request in

[11] See the start of §5 for a general description of the testbed environment.

flight. This value turned out to be 75 milliseconds. Note that for any $\alpha > 0$, the set of scoring rules $\{\text{score}^{\lambda} \mid 0 \leq \lambda \leq 1\}$ obtained by varying λ over the interval $[0, 1]$ is always equal to the set of convex combinations of latency and RIF. In other words our choice of $\alpha = 75$ms affects the way that this set of scoring rules is parameterized by λ, but doesn't affect the set itself.

We evaluated linear combination scoring rules by measuring quantiles of latency and RIF in our testbed at a constant level of aggregate CPU utilization (equal to 94% of our allocation) while varying λ. Replicas were partitioned into an equal number of fast (odd-numbered) and slow (even-numbered) ones, with a 2x difference in query processing speed, as in the RIF quantile experiment described in §5.3. We initially experimented with varying λ over the full range $[0, 1]$ in increments of 0.1. It was evident from this initial experiment that linear combination rules with $\lambda \leq 0.7$ performed poorly compared to larger values of λ, which prompted us to examine the range depicted in Fig. 10 at a finer resolution. Of the 13 linear combinations tested in the experiment, all quantiles of latency and RIF improved monotonically as λ increased, with $\lambda = 1$ (i.e., RIF-only control) dominating all other linear-combination feedback control rules, in most cases by a wide margin.

Recall from Fig. 9 in §5.3 that RIF-only control (represented in that figure by RIF limit threshold 1.0, the rightmost configuration) performs strictly *worse* than Prequal at all quantiles of latency and RIF. Since the results of the experiment reported here show that RIF-only control performs strictly *better* than any other linear combination of latency and RIF, it follows by transitivity that Prequal strictly dominates all linear combinations of latency and RIF.

Orthcatter: High-throughput In-band OFDM Backscatter with Over-the-Air Code Division

Caihui Du
Beijing Institution of Technology

Jihong Yu *
Beijing Institution of Technology

Rongrong Zhang
Capital Normal University

Ju Ren
Tsinghua University

Jianping An
Beijing Institution of Technology

Abstract

The existing ambient backscatter systems suffer from either more spectrum utilization or low throughput. we propose Orthcatter, the first in-band OFDM backscatter system that provides a higher throughput while consuming fewer spectrum resources. Our key innovation is the designed over-the-air code division technique that enables the cancellation of the co-channel interferences, solving the core challenge of the in-band backscatter communication. Unlike the common code-division systems that generate orthogonal codewords locally, we construct the quasi-orthogonal backscatter codewords by swapping the subcarriers of each excitation OFDM symbol and concrete this design passively with a double side-band symbol construction method. Armed with these quasi-orthogonal codewords, we design a two-step interference cancellation scheme, significantly improving reliability. We prototype and test Orthcatter. The results show that Orthcatter can achieve throughput of 248kbps and a BER of 10^{-4} under OFDM WiFi exciter, improving by over 4.6× and 300× compared with the state-of-the-art in-band backscatter system. Our throughput and BER can even be 11kbps higher and 59× better than the prior side-band backscatter systems, and the exciter-to-tag communication range is 3× of prior OFDM backscatter systems.

1 Introduction

The proliferation of wireless applications brings about the ever-increasing deployments of wireless devices, which emphasizes the importance of lower power consumption and saving spectrum resources. Since ambient backscatter systems utilize the existing exciters for communication rather than requiring a dedicated carrier emitter, they are ultra-low power and play an important role in future IoT applications. Therefore, many novel backscatter systems have been proposed [7, 10, 11, 14, 19–23, 27, 29, 33–38]. Among them, the OFDM backscatter has received the most attention due to the wide deployment of OFDM exciters. Inspired by them, we envision that a ready-to-use OFDM backscatter system should satisfy the three requirements:

Table 1: Summary of OFDM backscatter systems.

Technology	High throughput	Little spectrum occupation	Single receiver
LScatter [9]	✔	✖	✖
SyncScatter [12]	✔	✖	✖
HitchHike [35]	✔	✖	✖
STScatter [33]	✔	✖	✔
RapidRider [31]	✔	✖	✖
TScatter [27]	✔	✖	✖
WiFi Backscatter [20]	✖	✔	✔
WiTAG [7]	✖	✔	✔
Study in [22] [23]	✖	✔	✔
Orthcatter	✔	✔	✔

• **High throughput.** Its data rate should be at least hundreds of kbps to support various high data rates applications such as telecommuting and live streaming.

• **Little spectrum occupation.** Due to the scarce spectrum resources, its transmission should leverage the excitation spectrum rather than occupying additional frequencies.

• **Single receiver.** It should support one-radio demodulation. That is, only one receiver is required to decode the tag data, boosting the potential for working with such popular single-receiver mobile devices as laptops and smartphones.

If these requirements were satisfied, OFDM backscatter systems could be pervasively adopted for ultra-low power communication and thus radically change our life. For example, even in smart cities with a crowded radio spectrum, backscatter devices can still provide efficient data transmission by reusing the signal and spectrum of the existing exciters. Passive wearable sensors can also interact with one-receiver devices like smartphones or smartwatches without consuming extra spectrum or power resources. Unfortunately, to our knowledge, no existing systems satisfy these requirements simultaneously (c.f. Table 1).

The reason preventing this lies in that prior works are

*Corresponding author: Jihong Yu.

forced to tradeoff between the less spectrum occupation and the higher throughput. Specifically, studies in [9, 12, 27, 31, 33, 34, 36] provide side-band backscatter systems. They show high throughput yet consuming significant spectrum resources because they must shift the backscatter signal into a new channel, referred to as the backscatter channel, to avoid the original signal. The frequency shift doubles the channel occupancy and requires an empty channel which may not exist in today's crowded wireless environment. Moreover, their backscatter receivers do not known the excitation data a priori, they have to use two physically-separated receivers and synchronize them for decoding the tag data [12, 27, 31, 34–36]. This significantly complicate their deployment and limit their usage.

Studies in [7, 11, 20, 22, 23] design in-band backscatter systems. Unlike the side-band systems, they do not frequency shift the backscatter signal, so they would not occupy extra spectrum and can decode tag data by reusing the receiver that is already deployed for the exciter's transmission. However, the backscatter signal is strongly interfered by the coexisting original signal. These works cannot eliminate such interference effectively, suffering from poor BER (over 10^{-1}) and low throughput (tens of kbps). Therefore, the existing backscatter systems transmit the tag data either in an independent backscatter channel for a higher data rate (namely side-band systems), or in the original channel with poor reliability for saving the spectrum resources (namely in-band systems).

Solving the trade-off problem, we present Orthcatter, the first **in-band** OFDM backscatter system that has a communication ability comparable to the side-band systems. Similar to prior works, Orthcatter reflects the ambient excitation signal to generate the backscatter signal and embeds every 1-bit tag data over an excitation data segment in the backscatter signal. We define such excitation data segment carrying tag data as the **backscatter codeword** and define the excitation data segment in the original signal overlapping with the backscatter codeword in the time and frequency domain as the **original codeword**. These codewords possess two features. 1) As Orthcatter transmits data at the single-symbol rate, the length of the backscatter codeword equals that of an OFDM symbol. We emphasize that the backscatter codeword is not an OFDM symbol; instead, it is constructed by combining halves of two successive OFDM symbols (c.f. §3.1). 2) Consider in-band Orthcatter, the original codeword would interfere with the backscatter codeword, posing great challenge for tag data decoding. We would make the original and backscatter codeword quasi-orthogonal during backscatter modulation and design the interference cancellation scheme in §3.2.

As shown in Figure 1, Tag in Orthcatter not only changes the phase of the backscatter codewords to embed his data but also varies their content to make them so different from the original codewords that the receiver Bob can perform interference cancellation. Our key observation is that the randomization process at the scrambler of an OFDM exciter ensures any non-overlapped data segments are non-identical. Inspired

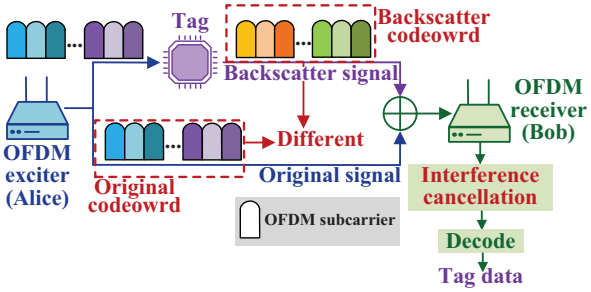


Figure 1: Concept of Orthcatter.

by this, we design the **over-the-air code division** technique which makes the original and backscatter codewords contain non-overlapping excitation data segments. This way, the correlation result between such codewords is small. We find such correlation results less than 0.5 under the 802.11g WiFi and less than 0.1 under the 20MHz LTE. Since the correlation results are not equal to 0 (i.e., orthogonal), we define the relation between our original and backscatter codewords as **quasi-orthogonal** for distinction. The quasi-orthogonality provides a chance for the receiver Bob to effectively cancel the interference from the original codewords and decode the tag data. To passively generate the backscatter codewords, we design the **double side-band symbol construction method** with a passive RF switch. During backscattering, this switch creates two OFDM symbols whose in-band parts would splice over the air into a new symbol that contains the backscatter codeword quasi-orthogonal with the original one.

In the design of Orthcatter, we answer three questions.

1) How to design the quasi-orthogonal codewords utilized for backscattering? To enable interference cancellation on the receiver side, it is crucial to ensure that the backscatter codewords are quasi-orthogonal to the original ones. This is challenging because the backscatter codewords are constructed from the uncontrolled excitation data segments rather than the Pseudo-Noise (PN) sequences that can be locally generated by conventional code-division systems. Furthermore, modifying each excitation data in the original codeword with such PN sequences to create the backscatter codeword is also not feasible. This is because Tag can only modulate the original signal in the time domain where the subcarriers carrying the excitation data would overlap. Hence, we must design the backscatter without local PN sequences.
Solution. We propose the over-the-air code division in §3.1 which makes the backscatter and original codewords quasi-orthogonal by building the former to carry non-overlapping excitation data segments with the latter. As shown in Figure 2, we generate the quasi-orthogonal backscatter codeword via two steps: subcarrier swapping and tag data mapping. In the first step, we exchange the position of the first and second half of each backscattered OFDM symbol in frequency domain. In the second step, we maps every bit of tag data into a manipulated excitation data segment. Note that we utilize the second half of a backscattered OFDM symbol and the

first half of its following one to carry every bit of tag data. This way, the original and backscatter codewords would carry non-overlapping data segments and are thus quasi-orthogonal.

2) How to passively generate the backscatter codewords? The over-the-air code division technique requires to swap frequency-domain subcarriers. However, a passive tag can only modulate the excitation signal in the time domain where the subcarriers overlap. This scheme also needs to embed the first and second half of each backscatter symbol with different tag data, requiring the modulation of the tag at the **half-symbol** level. However, the state-of-the-art works like RapidRider and STScatter [31, 33] just support single-symbol modulation for the overlapped subcarriers in the time domain.
Solution. We propose a double side-band symbol construction method in §3.1, which gets the desired backscatter OFDM symbol with a passive RF switch. This switch, toggling at fHz, can create two copies of an excitation OFDM symbol with frequency deviation of $\pm f$ from its center frequency during backscattering. We set $f = BW/2$ where BW is the bandwidth of the excitation signal so that just one half of each copy would fall inside the excitation channel and they can splice into a new OFDM symbol over the air which conveys 1-bit tag data, referred to as a backscatter symbol. This realizes the subcarrier swapping because the backscatter symbol is built by reversing the order of the first and second half of the two neighbour original excitation symbols, respectively. In addition, we vary the phase of these copies individually, making each in-band half of the copies modulated independently and achieving the half-symbol level modulation.

3) How to cope with the interference from the original signal? Since Orthcatter is an in-band system that conducts the backscatter communication in the same channel as the original one, the original signal would greatly interfere with its communication. Prior in-band works [22,23] use the redundant coding which embeds 1-bit tag data into multiple OFDM symbols or packets to cope with this interference. This significantly degrades the throughput to tens of kbps, e.g., 40kbps in [22]. In addition, their communication distance is still short, e.g., 5m in [22], because the original signal is not diminished at all and still hurts the tag data decoding. Therefore, we should cancel the interference of the original signal before decoding tag data for better decoding performance.
Solution. Given that the over-the-air code division ensures the quasi-orthogonality, we solve this through the proposed quasi-orthogonal interference cancellation and decoding scheme in §3.2. Bob can cancel the original signal via coarse and accurate cancellation. The coarse cancellation process roughly computes the original channel state from the inner product of the received OFDM symbol and the quasi-orthogonal codewords, and the accurate cancellation process adopts an adaptive filter to eliminate the residual original signal after the coarse cancellation process. Superior to prior works, most of the original signal is canceled before tag data decoding. Hence, our Orthcatter improves communication performance.

By answering these questions, we prototype Orthcatter and conduct experiments with typical OFDM exciters like WiFi and LTE. Our main results are summarized as follows.

- Orthcatter has an impressive throughput of 248kbps. It is 6.2× and 4.6× higher than the in-band backscatter systems [22] and [23], respectively. It is 11kbps higher than the side-band backscatter system RapidRider [31].
- Orthcatter's minimal BER under WiFi exciters is 3.4×10^{-4}, which is 59× and over 300× better than the side-band system RapidRider and the in-band system [23], respectively.
- Orthcatter is generic because it does not involve any upper-layer coding schemes of the exciter. It can work with different types of OFDM exciters, such as WiFi and LTE. Moreover, our tag can be put 3× further from the exciter Alice than prior OFDM backscatter works.

In summary, we design and build Orthcatter, the first in-band OFDM backscatter system that saves spectrum resources while providing considerable throughput. Our design makes the following technical contributions:

1) Over-the-air code division. This technique makes the original and backscatter codewords quasi-orthogonal. It enables the interference cancellation that is not supported in existing in-band works [22, 23].

2) Double side-band symbol construction. This method concretes the backscatter codewords and embeds tag data while keeping the power consumption similar to existing backscatter systems [27, 35, 36].

3) Quasi-orthogonal interference cancellation and decoding. This method separates the backscatter signal and the original one on the receiver side. It decodes tag data under much smaller interference, enabling better decoding performance than the state-of-the-art in-band works [22, 23].

2 Motivation and Design Overview

2.1 What makes Orthcatter outstanding?

Before introducing our Orthcatter, we first show what makes it outstanding. To explain this, we classify the current OFDM backscatter systems into two types: side-band and in-band. Tags in side-band backscatter systems would frequency shift the backscatter signal to another channel to avoid the original signal [12, 27, 31, 33, 35, 36]. Since the original signal would not interfere with backscatter transmission, they provide high throughput, e.g., hundreds of kbps, and good BER performance, e.g., around 10^{-3}. However, they share some severe drawbacks: **1) Large spectrum resources occupation.** The frequency shift here consumes significant spectrum resources, e.g., 20MHz under OFDM WiFi exciter. **2) Vulnerable transmission.** Since a tag cannot sense or occupy an empty channel for communication, the transmission suffers from strong interference in today's crowded wireless environment. **3) Complex deployment.** Most of these works [12, 27, 31, 35, 36] have to use and synchronize two physically-separated receivers since

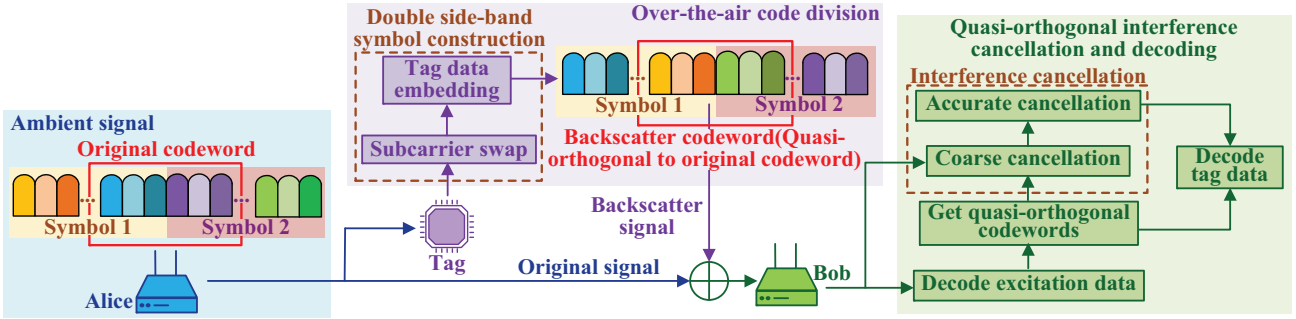


Figure 2: Orthcatter overview. Symbols 1 and 2 are two OFDM symbols, the backscatter codeword carrying 1-bit tag data is built from two excitation symbols by Tag. It is interfered by the original codeword.

they need to receive the original and backscatter signals in two frequency channels for decoding tag data.

Tags in the in-band backscatter systems transmit his data in the original channel [7,20,22,23]. These works saves the spectrum resources and can get tag data by reusing the receiver deployed for Alice's transmission. Moreover, other ambient signals would not interfere with their transmission because commercial systems can occupy the wireless channel through channel access methods like RTS/CTS scheme [13]. However, the received signal is a superposition of the original and backscatter signals. Worse still, the receiver cannot cancel the original signal effectively. To address this problem, redundant coding is used to make the superposed signal decodable on the receiver side. WiFi Backscatter [20] and WiTag [7] embeds 1-bit tag data over a WiFi frame with tens of OFDM symbols. Both [22] and [23] encode 1-bit tag data over multiple OFDM symbols. Unfortunately, they still experience poor BER. For example, although [23] encodes 1-bit tag data over four WiFi symbols (16 μs), its minimal BER is over 10^{-1}.

Observation. We observe that the root cause preventing the interference cancellation is the extreme similarity between the received original signal and the backscatter signal. For example, a received OFDM symbol Y_r in [23] is

$$Y_r = Y_o + Y_b = \sum_{i=1}^{N} H_o(i) S_i e^{-j2\pi f_i t} + \sum_{i=1}^{N} H_b(i) B S_i e^{-j2\pi f_i t} \quad (1)$$

where Y_o is the original symbol, Y_b is the backscatter symbol, N is the number of subcarriers in a symbol, and B is the tag data. Consider the i-th subcarrier, S_i is its excitation data, f_i is its center frequency, $H_o(i)$ is the direct-link channel state, and $H_b(i)$ is the backscatter channel state.

The differences between Y_o and Y_b just lie in the channel state and an initial phase offset that exists only when the transmitted tag data is '1'. Consequently, the receiver cannot distinguish Y_o and Y_b, and has to decode with strong interference. Denote by $S = \{S_i, \cdots, S_N\}$ the data carried by the original codeword, the observation inspires us to design Orthcatter that enables interference cancellation by making Y_o and Y_b carry different S. The problem hindering our design is whether we can achieve this without consuming extra power. Our answer is YES, and our insight is that two non-overlapping excitation data segments would be quasi-orthogonal thanks to the scrambler at the OFDM exciter. Specifically, we make Y_b carry the backscatter codeword that is quasi-orthogonal to the original codeword carried by Y_o (c.f. Figure 2). Y_r here hence becomes

$$Y_r = Y_o + Y_b = \sum_{i=1}^{N} H_o(i) S_i e^{-j2\pi f_i t} + H_b(i) B \hat{S}_i e^{-j2\pi f_i t} \quad (2)$$

where $\hat{S}_i$ is the excitation data carried by the i-th subcarrier in the backscatter symbol. Since the backscatter codeword $\hat{S} = \{\hat{S}_1, \cdots, \hat{S}_N\}$ and S are quasi-orthogonal, Bob can separate Y_o and Y_b, and decodes tag data under much smaller interference. Therefore, Orthcatter achieves better performance.

2.2 What is Orthcatter?

Orthcatter is an in-band OFDM backscatter system that outperforms the prior works. We first give a brief description of the architecture of OFDM technology. Communication systems like WiFi and LTE utilize OFDM to embed their data into different subcarriers. For simplicity, we denote an OFDM symbol by $S = \sum_{i=1}^{N} S_i e^{-j2\pi f_i t}$ where f_i is the frequency of the subcarrier i and S_i is the carried excitation data. A scrambler would randomize S_i to avoid the all-zero or all-one sequences. That said, two segments of S_i would be nonidentical if they do not overlap. We refer to such feature as quasi-orthogonal and employ it to design Orthcatter. As shown in Figure 2, Orthcatter has three parts: Alice, Tag, and Bob.

- **Alice.** Alice is an OFDM-based exciter and is beyond control. In our experiments, we choose two typical OFDM exciters for Alice: 802.11g WiFi [6], and LTE [4].
- **Tag.** Tag could piggyback his data over the ambient OFDM excitation signals. He can naturally support generic OFDM exciters including WiFi and LTE without hardware modification. During transmission, Tag adopts the over-the-air code division technique (c.f. §3.1) which embeds and spreads his data with quasi-orthogonal backscatter codewords. As shown in Figure 2, the backscatter codeword conveys 1-bit tag data and is quasi-orthogonal to its counterpart in the original excitation signal, i.e., the original codeword in Figure

2. Such a backscatter codeword is generated passively with the double side-band symbol construction method. Through this design, Tag can embed his data at the single-symbol rate same as existing side-band systems like RapidRider [31] and thus has an impressive throughput. We emphasize that: 1) As an in-band backscatter system, Tag's transmission takes place in the same channel as the excitation signal. 2) Tag is passive with around $63.3\mu W$ power consumption (c.f.§4).

• **Bob.** Bob is an OFDM receiver that decomposes the incoming signal into frequency-domain OFDM symbols through FFT. He decodes the excitation and tag data from this symbol through the quasi-orthogonal interference cancellation and decoding method explained in §3.2. To achieve this for a strong or weak backscatter signal, he would roughly compute the ratio between the original and backscatter signal strength, and uses a ratio-aware scheme to decode the excitation data. He then eliminates the original signal through a two-step cancellation process and decodes tag data. Bob needs only **one** receiver staying at the original excitation channel.

Benefits. Orthcatter enables tag data transmission over ambient OFDM signals with two main advantages: 1) Tag data is transmitted while being decoded in the same channel as the original excitation signal with only one receiver needed. 2) Interference from the excitation signal is cancelled. With these advantages, Orthcatter has the potential to be deployed in noisy and signal-rich environments like office buildings and smart cities. Orthcatter achieves even better throughput and BER than the state-of-the-art side-band backscatter system like RapidRider [31], meaning that it consumes few spectrum resources without harming the communication ability. We believe that our design benefits a wide range of applications.

3 Design

3.1 Encoding: Over-the-air code division

As shown in Figure 2, we construct the backscatter codeword from two adjacent excitation OFDM symbols, i.e., symbols 1 and 2, and utilize its quasi-orthogonality with the original codeword to achieve reliable in-band backscatter communication. To enable this, we design the over-the-air code division technique. We first introduce the challenges in our design.

Challenges. Facing the signal collision problem, the traditional systems would carefully design their transmitted codewords for signal separation. For instance, the CDMA [30] system spreads their messages with orthogonal codewords so that the receiver can decode them independently. [25] introduces structured permutation when generating its transmitted codewords and utilizes the data carried by the collision-free subcarriers to infer the data carried by the collided subcarriers. That said, it works only when two channels partially overlap. These schemes cannot be applied directly to in-band backscatter systems for two reasons.

1) Tag cannot build the quasi-orthogonal codeword by utilizing the PN sequence to spread tag data like existing CDMA systems. In a generic backscatter system, the tag data is spread by the uncontrolled excitation data instead of the locally generated PN sequences adopted by existing CDMA systems. If Tag tries to modulate the excitation data with a PN sequence, it would modulate all the excitation data in one OFDM symbol instead of any individual one [33]. For instance, denote the phase shift to convey data in the PN sequnece by $e^{j\phi_t}$ and an OFDM symbol by $S = \{S_1, S_2, \ldots, S_N\}$. Embedding this data via backscatter yields the symbol $\hat{S} = \{S_1 e^{j\phi_t}, S_2 e^{j\phi_t}, \ldots, S_N e^{j\phi_t}\}$ instead of the desired $\{S_1 e^{j\phi_t}, S_2, \ldots, S_N\}$. Here the value of the normalized correlation result between S and $\hat{S}$ is still one, and the quasi-orthogonality is not established.

2) Constructing the desired quasi-orthogonal codewords must consume similar power as prior backscatter systems. As shown in Figure 2, generating the backscatter codeword requires changing the subcarrier pattern and modulating at the half-symbol level. This causes trouble because excitation data of an OFDM symbol are independent in the frequency domain and coexist in the time domain. However, without energy-consuming components like ADCs and mixers, Tag cannot convert his received time-domain signal to the frequency domain to manipulate the excitation codewords individually. Hence, we must keep time-domain modulation while changing the subcarrier pattern to generate the desired backscatter codewords without rising energy consumption.

Solutions: over-the-air code division. Recall that an excitation data segment that is embedded with 1-bit tag data is called a backscatter codeword, we aim at making it quasi-orthogonal to its counterpart in the original excitation signal, i.e., the original codeword. To this end, we design the over-the-air code division technique, which builds quasi-orthogonal codewords without relying on local PN sequences. As shown in Figure 3, this method makes the original and backscatter codewords carry non-overlapping excitation data segments. This way, since the randomization at Alice ensures that two non-overlapping data segments are different, the original and backscatter codewords are quasi-orthogonal. Specifically, the backscatter codeword is built by two steps: subcarrier swapping and tag data mapping.

To explain this, we take the symbol 1 (denoted as $S1$) and symbol 2 (denoted as $S2$) as an example, each containing N subcarriers which are denoted by $S1_1 \sim S1_N$ and $S2_1 \sim S2_N$, respectively. **In the subcarrier swapping step**, Tag reverses the order of the first and second half of each OFDM symbol. This changes $S1$ and $S2$ into the backscatter symbols $\hat{S1} = \{S1_{N/2+1} \cdots S1_N, S1_1 \cdots S1_{N/2}\}$ and $\hat{S2} = \{S2_{N/2+1} \cdots S2_N, S2_1 \cdots S2_{N/2}\}$. **In the tag data mapping step**, he pieces the second half of $\hat{S1}$, i.e., $S1_{N/2+1} \sim S1_N$, and the first half of $\hat{S2}$, i.e., $S2_{N/2+1} \sim S2_N$, to form the backscatter codeword $\{S1_1 \ldots S1_{N/2}, S2_{N/2+1} \ldots S2_N\}$ conveying 1-bit tag data, i.e., the backscatter codeword in Figure 3. This is completely different from prior works [31, 35, 36] that em-

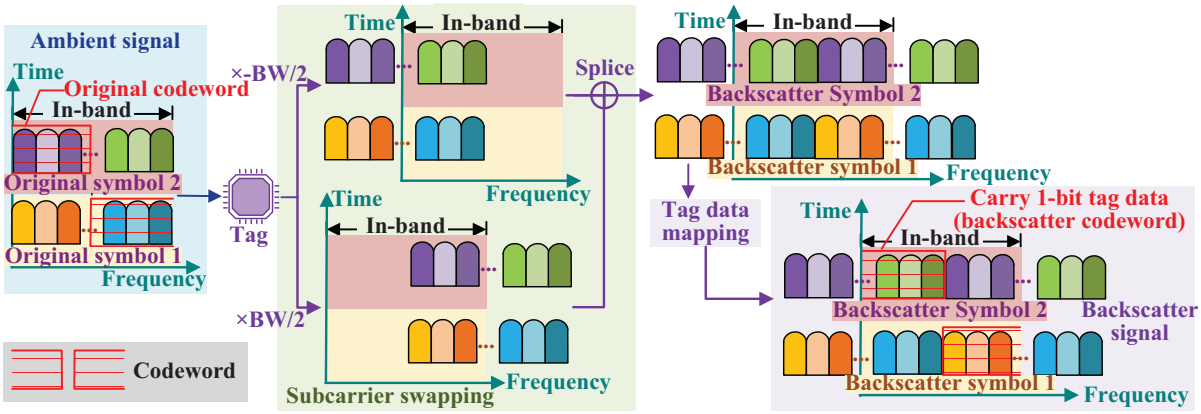


Figure 3: Overview of the over-the-air code division. The backscatter codeword carries 1-bit tag data and is interfered with by the original codeword. We make them quasi-orthogonal for interference cancellation.

beds 1-bit data over the whole $S1$ or $S2$. This way, the original codeword coexisting with such backscatter codeword is $\{S1_{N/2+1}\dots S1_N, S2_1\dots S2_{N/2}\}$. They are made from non-overlapping excitation data and hence are quasi-orthogonal. Note that the rest of the in-band subcarriers, i.e., $S1_{N/2+1}\sim S1_N$ and $S2_{N/2+1}\sim S2_N$, would not be wasted, they are also utilized to form other quasi-orthogonal codewords. Specifically, $S1_{N/2+1}\sim S1_N$ is pieced with the second half of the preceding backscatter symbol and $S2_{N/2+1}\sim S2_N$ is pieced with the first half of the backscatter symbol following $\hat{S}2$.

We emphasize that these backscatter symbols $\hat{S}_1$ and $\hat{S}_2$ would hardly be affected by the impact brought by propagation delays, echoes, and reflections. The reason is that the time domain guard signal, i.e., CP, in an OFDM system is utilized to add immunity to these effects. During our generation process, the tag would change the CP and the data part of an OFDM symbol based on the same scheme and tag data, making it still hold that the CP is the same as the end of the OFDM backscatter symbol, that said, the receiver still receives the backscatter codeword $S_b=\{S1_{N/2+1}\dots S1_N, S2_1\dots S2_{N/2}\}$.

Passive quasi-orthogonal codeword generation. The key left is to arm a tag with the ability of modulating subcarriers of an OFDM symbol, i.e., conducting half-symbol level modulation. To this end, Orthcatter leverages the fact that an fHz square wave generated by the tag can shift the original signal by $\pm f$Hz during backscattering. Such a double side-band characteristic is unwanted in prior side-band backscatter systems because it introduces interference in the adjacent channels. In contrast, we turn it into profits and design the **double side-band symbol reconstruction** method, which passively builds the backscatter codeword. Specifically, Tag adopts a passive switch and toggles it at $f=BW/2$ to create the square wave. As shown in Figure 3, this square wave shifts the original OFDM symbols $S1$ and $S2$ by $\pm BW/2$. This way, the in-band parts of these frequency-shift subcarriers would splice over the air forming the backscatter OFDM symbols $\hat{S}_1$ and $\hat{S}_2$. Since $S1$ and $S2$ are shifted by the same process, we take $S1$ as an example to explain this in details. Suppose Tag toggles the switch at $f_t=BW/2$, his antenna reflection coefficient $\Gamma(t)$ would be changed following a square wave $y_t(t)$. From the Fourier analysis [17], $y_t(t)$ is

$$\begin{aligned} y_t(t) &= \frac{4}{\pi}\sum_{n=1}^{\infty}\frac{\bar{\Gamma}\sin\left((2n-1)2\pi f_t t\right)}{2n-1} \\ &\approx \frac{4\bar{\Gamma}}{\pi}\sin(2\pi f_t t) = \frac{2j\bar{\Gamma}}{\pi}\left(e^{-j2\pi f_t t}-e^{j2\pi f_t t}\right) \end{aligned} \tag{3}$$

where $\bar{\Gamma}$ denotes the amplitude and phase of $\Gamma(t)$. For example, $\bar{\Gamma}=j$ when Tag alters $\Gamma(t)$ between $\pm j$.

Let f_c and $y_b(t)$ respectively be the OFDM excitation signal's center frequency and baseband waveform. The backscatter signal output from the switch is

$$\begin{aligned} y_{out}(t) &= A_1 y_b(t) e^{-j2\pi f_c t} y_t(t) \\ &= \frac{2A_1 y_b(t)\bar{\Gamma} j}{\pi}\left(e^{-j2\pi(f_c+f_t)t}-e^{-j2\pi(f_c-f_t)t}\right) \end{aligned} \tag{4}$$

where A_1 is the antenna gain. This equation shows that the in-band subcarriers of $y_{out}(t)$ is $\{S1_{N/2+1}\dots S1_N, S1_1\dots S1_{N/2}\}$, which is the desired backscatter OFDM symbol $\hat{S}_1$. Consequently, we passively achieve the subcarrier swapping. After this, Tag should embed his data over the built quasi-orthogonal codewords, e.g., the backscatter codeword in Figure 3. This requires embedding different tag data over the first and second half of $\hat{S}1$ because they belong to the different backscatter codewords. That said, Tag needs to take two adjacent data during modulation and change the phases of the subcarriers at a time. $\hat{S}1$ after embedding tag data (td) is

$$\hat{S}1=\begin{cases}\{S1_{N/2+1},\dots,S1_N,S1_1,\dots,S1_{N/2}\}, & td=\{0,0\}\\ \{-S1_{N/2+1},\dots,-S1_N,S1_1,\dots,S1_{N/2}\}, & td=\{1,0\}\\ \{S1_{N/2+1},\dots,S1_N,-S1_1,\dots,-S1_{N/2}\}, & td=\{0,1\}\\ \{-S1_{N/2+1},\dots,-S1_N,-S1_1,\dots,-S1_{N/2}\}, & td=\{1,1\}\end{cases} \tag{5}$$

This indicates that the modulation is performed at the **half-symbol level**. In traditional backscatter systems, this is

unattainable because the subcarriers are inseparable in the time domain where a passive tag conducts his modulation [33]. In contrast, our tag would create two copies of backscatter signals at two side-bands, whose in-band parts form the backscatter symbol. Employing such double side-band nature, we can thus modulate the first and second half of the backscatter symbol individually. To further explain this, we distinguish (5) into two cases based on whether the phases of the first and second half of $\hat{S}1$ are opposite.

• **Case 1: Tag data is $\{1,0\}$ or $\{0,1\}$.** Recall (5), the phases of the first and second half of $\hat{S}1$ are opposite, so we alter $\Gamma(t)$ between $\pm j$ to transmit these tag data. We consider an antenna with impedance $Z_A = 50$. Since $\Gamma(t) = \frac{Z_L(t)-Z_A}{Z_L(t)+Z_A}$, altering the load impedance $Z_L(t)$ between $\pm j50$ could change $\Gamma(t)$. The output signal $y_{out1}(t)$ is shown in (6), and we reverse it for tag data of $\{1,0\}$ and do nothing for $\{0,1\}$.

$$y_{out1}(t) = \frac{2A_1 y_b(t)}{\pi}\left(-e^{-j2\pi(f_c+f_t)t} + e^{-j2\pi(f_c-f_t)t}\right) \quad (6)$$

• **Case 2: Tag data is $\{1,1\}$ or $\{0,0\}$.** The phases of the first and second half of these subcarriers are identical, making $y_t(t)$ in Case 1 unsuitable. We here delay $y_t(t)$ by $\frac{f_t}{4}$ to create an additional $\pi/2$ phase offset, reformulating (3) into

$$\begin{aligned} y_t(t) &= \frac{2j\bar{\Gamma}}{\pi}\left(e^{-j2\pi f_t(t-\frac{1}{4f_t})} - e^{j2\pi f_t(t-\frac{1}{4f_t})}\right) \\ &= -\frac{2\bar{\Gamma}}{\pi}\left(e^{-j2\pi f_t t} + e^{j2\pi f_t t}\right) \end{aligned} \quad (7)$$

We change the load impedance $Z_L(t)$ between 0 and $+\infty$ to alter $\Gamma(t)$ between ± 1 and transmit the tag data. Such $y_{out}(t)$ is stated in (8) and is reversed when tag data is $\{0,0\}$.

$$y_{out}(t) = \frac{-2A_1}{\pi}\left(e^{-j2\pi(f_c+f_t)t} + e^{-j2\pi(f_c-f_t)t}\right) \quad (8)$$

As analysed above, we change $Z_L(t)$ among $\pm j50$, 0 and $+\infty$ to get (5), which enables the half-symbol level modulation. Compared with the prior works [27, 31, 35, 36] that changes $Z_L(t)$ between 0 and $+\infty$, our $Z_L(t)$ has two additional values. This would hardly increase the hardware complexity or the power consumption because we only need to arm Tag with a passive RF switch that has more states. As confirmed in §4, such a passive RF switch has a less than 3.3μW power consumption which is similar to the power consumption of the RF switch adopted by the prior backscatter works [32].

3.2 Decoding: Quasi-orthogonal interference cancellation

We have presented how to generate the backscatter signal carrying the quasi-orthogonal backscatter codewords where tag data are embedded. Here we focus on decoding the tag data at Bob. As illustrated in Figure 2, we need the quasi-orthogonal codewords to perform interference cancellation and get tag data. To this end, we first decode the excitation data from the received superposition signal. Specifically, we would roughly compute the signal strength ratio. If the excitation signal is 10dB stronger than the backscatter one, we employ the capture effect to decode it. Otherwise, we utilize the null subcarriers to create the linear equations with a unique solution and decode the excitation data by solving these equations. Second, we infer the quasi-orthogonal codewords and conduct a two-step interference cancellation scheme to extract the backscatter signal. In the step 1, we regard the original channel state as constant among different subcarriers and roughly subtract the extracted excitation signal from the received one. In the step 2, we utilize the adaptive filter to further eliminate the residual excitation signal, and decode tag data from the left signal. Our decoding process is detailed as follows.

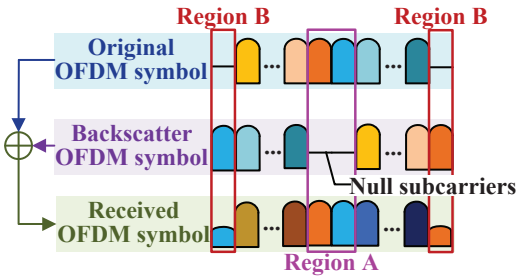


Figure 4: Illustration of the received OFDM symbol. Region A contains only the original excitation OFDM subcarriers and Region B contains only the backscatter OFDM subcarriers due to the existence of null subcarriers.

Decoding the excitation data. We need to decode the excitation data first to infer the quasi-orthogonal codewords. Since the original signal is much stronger than the backscatter one, we decode the excitation data from it based on the capture effect. However, such decoding would result in a higher BER unless the original signal is 10dB higher than the backscatter one [24]. That said, to get the excitation data correctly, Bob must first sense the ratio of the excitation signal strength to the backscatter one before employing the capture effect.

We achieve this through an empty part of the OFDM spectrum, i.e., the null subcarriers. These subcarriers usually lie on the sides of an OFDM symbol spectrum as the frequency-domain guard interval. However, these subcarriers would be moved to to the center of the backscatter symbol by the over-the-air code division scheme. This makes some subcarriers in the received OFDM symbol come from either the original or backscatter symbol, which can be utilized for computing the signal strength ratio. As shown in Figure 4, the received subcarriers in Region A only come from the original OFDM symbol, and those in Region B are only from the backscatter OFDM symbol. Therefore, we calculate the original and backscatter signal strength from the subcarriers strength in these regions. If the original subcarriers is 10dB stronger than the backscatter ones, we adopt the capture effect to get the excitation data; otherwise, we design the following ap-

proach. Take the OFDM symbol $S1$ as an example. Denote the received symbol of $S1$ by $\tilde{S1}$

$$\tilde{S1} = h_o S1 + h_b \hat{S1} \tag{9}$$

where h_o and h_b mean the original and backscatter channel, and $S1$ and $\hat{S1}$ are the corresponding symbols. Consider all the subcarriers, we have

$$\begin{bmatrix} \tilde{S1}_1 \\ \tilde{S1}_2 \\ \vdots \\ \tilde{S1}_N \end{bmatrix} = h_o \begin{bmatrix} S1_1 \\ S1_2 \\ \vdots \\ S1_N \end{bmatrix} + h_b \begin{bmatrix} e^{j\phi_{1t}} S1_{N/2+1} \\ \vdots \\ e^{j\phi_{1t}} S1_N \\ e^{j\phi_{2t}} S1_1 \\ \vdots \\ e^{j\phi_{2t}} S1_{N/2} \end{bmatrix} \tag{10}$$

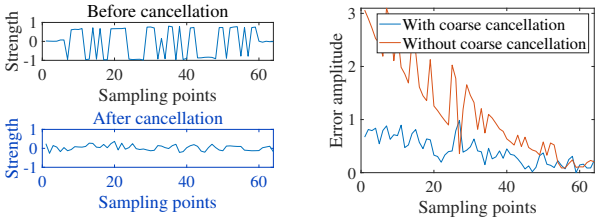
where ϕ_{1t} and ϕ_{2t} are the phase offsets carrying different tag data. We take a deeper look at (10). It has N equations and less than N variables because the pilot and null subcarriers would make some among $S1_1 \sim S1_N$ known a priori. For example, an 802.11g WiFi symbol has 4 pilot and 12 null subcarriers, i.e., solving 52 variables in (10) with 64 equations. Therefore, it has a unique solution and we can solve it to get the channel states, i.e., h_o and h_b, and the excitation data. In this way, we get the excitation data under either strong or weak backscatter signals. We can exam this decoding result through the check code like cyclic redundancy check (CRC) adopted by the exciter. The task here hence moves to decode tag data.

Decoding the tag data. Since the quasi-orthogonal codewords are inferred after decoding the excitation data, Bob tends to decode the tag data from the inner product of the received OFDM symbol and the quasi-orthogonal codewords. There are still two problems to be solved. First, unlike the standard CDMA systems whose received signals from different transmitters have similar strength, our backscatter signal here is much weaker than the original one. Second, the relation between the original and backscatter codewords is quasi-orthogonal instead of orthogonal, meaning that the inner product between these codewords would not be zero. Consequently, the original signal greatly interferes with decoding the tag data. To address these problems, we design a two-step interference cancellation scheme. Our goal is to compute the original channel state h_o and subtract $h_o S1$ from $\tilde{S1}$.

1) Coarse cancellation. To this end, we first assume that h_o in (9) is constant across different subcarriers to compute its rough value. Specifically, Bob computes the normalized inner product among $\tilde{S1}$, $S1$, and $\hat{S1}$, and gets

$$\begin{aligned} \sum_{i=1}^{N} \frac{\tilde{S1}_i S1_i^*}{|\tilde{S1}_i||S1_i|} =& N h_o + h_b e^{j\phi_{1t}} \sum_{i=1}^{N/2} \frac{S1_{N/2+i} S1_i^*}{|S1_{N/2+i}||S1_i|} \\ &+ h_b e^{\phi_{2t}} \sum_{i=1}^{N/2} \frac{S1_i S1_{N/2+i}^*}{|S1_i||S1_{N/2+i}|} \end{aligned} \tag{11}$$

$$\sum_{i=1}^{N/2} \frac{\tilde{S1}_i S1_{N/2+i}^*}{|\tilde{S1}_i||S1_{N/2+i}|} = h_o \sum_{i=1}^{N/2} \frac{S1_i S1_{N/2+i}^*}{|S1_i||S1_{N/2+i}|} + \frac{N}{2} h_b e^{j\phi_{1t}} \tag{12}$$



(a) Received OFDM symbol (b) Adaptive filter's convergence time

Figure 5: Empirically measured convergence time and interference cancellation performance.

$$\sum_{i=1}^{N/2} \frac{\tilde{S1}_{N/2+i} S1_i^*}{|\tilde{S1}_{N/2+i}||S1_i|} = h_o \sum_{i=1}^{N/2} \frac{S1_{N/2+i} S1_i^*}{|S1_{N/2+i}||S1_i|} + \frac{N}{2} h_b e^{j\phi_{2t}} \tag{13}$$

where $S1_i^*$ is the complex conjugate of $S1_i$.

Solving these equations yields the h_o under the hypothesis that it remains constant among different subcarriers. However, the frequency selective fading resulting from the multi-path effect would break such hypothesis. For distinction, we denote such computed h_o as $\hat{h}_o$. Since subtracting $\hat{h}_o S1$ from the received symbol $\tilde{S1}$ cannot remove all the original excitation symbol, we further conduct the accurate cancellation process.

2) Accurate cancellation. In this process, we adopt the adaptive filtering algorithm to output the remaining $S1$ after the coarse cancellation [18]. Specifically, given that the backscatter codewords are quasi-orthogonal to the original ones, we regard them as noise and utilize the least-mean-square (LMS) adaptive filter to reconstruct the remained $S1$ in $\tilde{S1}$. Denote that $\bar{S1} = \tilde{S1} - S_1 \hat{h}_o$, the LMS filter is

$$\begin{aligned} e(n) &= d(n) - \omega^H(n) u(n) \\ \omega(n+1) &= \omega(n) + \mu u(n) e^*(n) \end{aligned} \tag{14}$$

where $u(n) = S1_n/|S1_n|$ is the input variable, $d(n) = \bar{S1}_n/|S1_n|$ is the optimal output, $\omega(n)$ is the LMS filter, and $e(n)$ is the estimation error with $n = 1, 2, \ldots N$. μ is the step size not smaller than $2/R$, and R equals the square root of the self-correlation of the input variable $u(n)$.

Since $u(n)$ is very limited, e.g., 64 when the excitation signal follows 802.11g WiFi, we set μ to its maximum value for a shorter convergence time. We further subtract the output from $\bar{S1}$ to decrease the interference caused by the residual original signal. As shown in Figure 5(a), the original signal is eliminated after our two-step interference cancellation.

Although the LMS filter can distinct the original symbol from the received superposition one, we emphasize that the coarse cancellation is still necessary because the adaptive filter would waste sampling points before convergence [18]. This can be ignored in common wireless systems where the received signal with countless sampling points is input into the filter. In contrast, in our Orthcatter where the input signal is an OFDM symbol with limited sampling points, such a waste would greatly degrade the performance and hence

should be avoided. The empirically measured convergence time is shown in Figure 5(b), confirming that the coarse cancellation process significantly reduces the error and decreases the number of required sampling points.

We then decode the tag data by correlating the left signal after the interference cancellation with the inferred backscatter codeword. Since each backscatter codeword crosses two OFDM symbols, we take two OFDM symbols $S1$ and $S2$ to explain this. According to the modulation scheme, the halves of the received symbols containing a 1-bit tag data is $\check{S} = \{\check{S}1_{N/2+1}, \ldots, \check{S}1_N, \check{S}2_1, \ldots, \check{S}2_{N/2}\} \approx e^{j\phi_{2t}} h_b \{S1_1, \ldots, S1_{N/2}, S2_{N/2+1}, \ldots, S2_N\}$, and the backscatter codeword is $S_b = \{S1_1, \ldots, S1_{N/2}, S2_{N/2+1}, \ldots, S2_N\}$. Therefore, the inner product between $\check{S}$ and S_b is

$$Z_i = \left(\sum_{i=1}^{N/2} \frac{\check{S}1_{i+N/2} S1_i^*}{|\check{S}1_{i+N/2}||S1_i|} + \sum_{i=1}^{N/2} \frac{\check{S}2_i S2_{i+N/2}^*}{|\check{S}2_i||S2_{i+N/2}|} \right) = e^{j\phi_{2t}} h_b N$$

Since the phase of Z_i is $\arctan\left(\frac{h_b N \sin\phi_{2t}}{h_b N \cos\phi_{2t}}\right) = \phi_{2t}$, Bob finally decodes tag data from the phase of Z_i. The operations above benefit our Orthcatter in two ways: First, it decodes tag data under a much smaller interference, improving the throughput and communication distance. Second, since $|Z_i|$ is the sum of all the subcarrier amplitudes, it is distinguishable under a small SNR. Consequently, Orthcatter has an even smaller BER than side-band systems as confirmed in §5.

4 Orthcatter Implementation

Challenge & solution. Synchronization accuracy poses great challenge for OFDM backscatter. Specifically, in the backscatter systems that adopts a dedicated single-tone RF source as their exciter, the synchronization with the excitation signal is not required, so they achieves a further Alice-to-Tag distance, e.g., 20m in RF-transformer [14] and 9m in DigiScatter [39]. In contrast, in the OFDM backscatter systems, Tag must achieve symbol synchronization because his data is transmitted at the single-symbol rate for higher throughput [12]. Consequently, most of them [9, 27, 28, 31, 37] maintain Alice-to-Tag distance within 1m for better BER. Consider such limitation, we try to break it in Orthcatter.

The prior works usually utilize an envelope average and a comparator to determine whether tag receives the ambient OFDM signal and hence synchronize tag's transmission with this signal [37]. While such an energy detection method shows a small power consumption, it has a very limited synchronization distance because it cannot detect the weak signal. To address this limitation, we propose a matching scheme instead of the energy detection method used in works [9, 27, 28, 31, 37] to detect the weaker ambient OFDM signal.

Our **sliding window matching** scheme (c.f. Figure 6) enhances synchronization with the cyclic prefix (CP) of an OFDM symbol. Given that the CP is a periodic extension

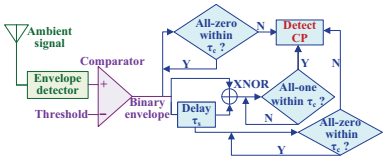


Figure 6: The sliding window matching scheme.

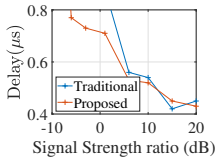
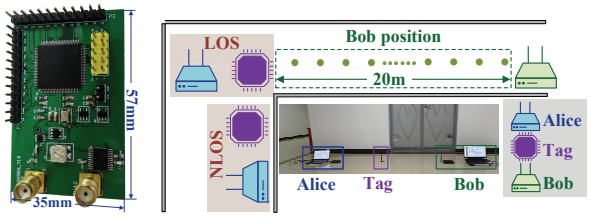


Figure 7: Delay of synchronization.

of the OFDM symbol, it holds that $y(t) = y(t+\tau_s), t \in [0, \tau_c]$ where τ_c and τ_s are the duration of the CP and the data part of an OFDM symbol, respectively. Our basic idea is to use a sliding window that matches every τ_c of the OFDM signal with the OFDM signal delayed by τ_s. As shown in Figure 6, we would first utilize an envelope detector to extract the envelope of the OFDM signal and then utilize a comparator with a predetermined threshold to binarize the envelope. We divide the binary envelope sequence into two paths and delay one path by τ_s. Finally, we compute the XNOR result between these paths which would remain one for τ_c if the CP is detected or the noise keeps below the threshold. Therefore, if the XNOR result remains one within τ_c while neither of these sequences is all-zero, the CP is detected and Tag can synchronize with the excitation signal. Note that we set the threshold to 1V based on the empirically measured noise floor.

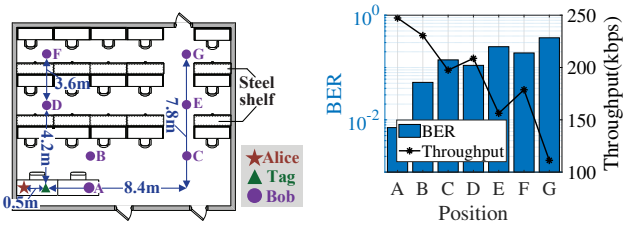
To show the advantage of this scheme, we test the synchronization accuracy under 802.11g WiFi exciter. We put Alice 0.1m away from Tag. As shown in Figure. 7, When the excitation signal strength is over 1dBm, our approach and the traditional one has similar synchronization accuracy. This is because we digitize the ambient signal's envelope with the same approach as the traditional one. When the signal strength drops below -6dBm, our approach maintains a synchronization accuracy within 750ns while the traditional approach fail to detect the ambient signal. Different from the traditional 1-bit detection scheme, our approach is a correlation-based synchronization scheme that considers multiple bits of the digitized envelope sequence at a time and thereby achieves synchronization under weaker signals. Therefore, our proposed scheme enables accurate synchronization within a longer distance. The experiment results in §5 confirm this.

Backscatter tag. We implement Orthcatter following an open-source backscatter platform [32] that is widely used [9, 27, 35, 36] (c.f. Figure 8(a)). The power consumption of its simulated ASIC design is merely 33μW. However, due to the high tap out cost, it utilizes COTS components to build its hardware prototype, resulting in power consumption of 40mW. The increased power consumption mainly results from the utilization of an FPGA and a COTS envelope detector LT5534 [3]. Both of these components consumes tens of microwatts energy. We only replace the SPST ADG902 RF switch [1] with an SP4T ADG904 RF switch [2] to get the required antenna coefficient for generating the quasi-orthogonal



(a) Tag prototype (b) LOS & NLOS experiment setup

Figure 8: Tag prototype and experiment setup.



(a) Deployment (b) Performance

Figure 9: Testing Orthcatter in laboratory.

codewords. According to the data sheets, these two types of RF switches have the same maximum quiescent power supply current of 1μA under the 3.3V voltage supply. Therefore, our prototype also has about 40mW power consumption. By conducting ASIC design, the modulation component of Orthscatter has 56μW power consumption, and the detection component implemented by LT5534 is passive, so the simulated overall power consumption of Orthscatter is 63.3μW.

Alice & Bob. We adopt two USRPs for Alice and Bob, and choose two typical commercial OFDM signals as exciters: 802.11g WiFi and LTE. The WiFi signal is implemented with the open-source 802.11g WiFi transceiver on Ubuntu [8]. The LTE signal is generated on MATLAB by the LTE toolbox [5]. They are compatible with the commercial standards [8] [5].

5 Orthcatter Evaluation

We evaluate Orthcatter's performance under diverse scenarios. Our experiment confirms that it outperforms the state-of-the-art in-band backscatter systems [22, 23] in terms of BER, throughput, and communication distance. 1) The maximal throughput of Orthcatter is 248kbps, which is 6.2× higher than [22], and 4.6× higher than [23]. 2) The minimal BER of Orthcatter under WiFi is 3.4×10^{-4}, which is over 300× better than [23]. 3) When the communication distance is 20m, the throughput of [22] and [23] drops to zero, yet Orthcatter still experiences throughput of over 50kbps. Orthcatter also outperforms the side-band backscatter system like RapidRider [31]. Under the same settings, Orthcatter increases the maximum throughput by 11kbps and reduces the minimal BER by 59×.

5.1 Experiment Setup

Settings. As shown in Figure 8(b), we test Orthcatter in the LOS and NLOS scenarios. We use an 802.11g OFDM WiFi exciter. The packet rate is 500pkt/s, and the transmit power equals the common value of WiFi products, i.e., 20dBm [16].

Competitions. We compare Orthcatter with in-band [22, 23] and side-band OFDM backscatter systems [31]. Specifically, we choose RapidRider to signify the state-of-the-art side-band OFDM backscatter system. This work is known for its throughput and symbol-level modulation. However, it needs two receivers deployed in the original and backscatter channels respectively, occupying more spectrum resources. [22, 23] are in-band OFDM backscatter systems and have to use redundant coding schemes to cope with the interference caused by the excitation signal. Consequently, they show much lower throughput and a much worse BER.

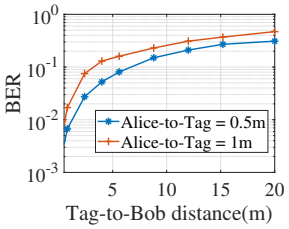
5.2 End-to-End Performance

We set the Alice-to-Tag distance same as RapidRider and [23], i.e, 0.5m and 1m, and choose the same type of the exciter. We vary the Tag-to-Bob distance in LOS & NLOS to assess end-to-end performance. Our results are shown in Figure 10 and Figure 11, confirming that Orthcatter performs best.

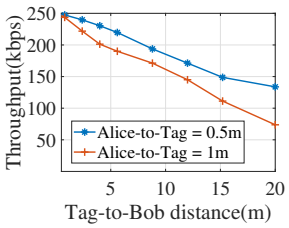
BER. We vary the communication distance from 0.5m to 20m and show the BER under LOS&NLOS deployments in Figure 10(a) and Figure 11(a). As shown in Figure 10(a), our BER is around 0.1 even when the Tag-to-Bob distance is 6m. It is outstanding because the BER of [22] increases to 1 under the same setting, and that of [23] is over 0.1 even for 2m Tag-to-Bob distance. The side-band RapidRider can maintain its BER below 0.1 only when Bob is less than 7m from Tag. Orthcatter also outperforms them in NLOS.

Throughput. In terms of throughput, neither [22], [23], nor [31] can match our Orthcatter (c.f. Figure 10(b) and Figure 11(b)). Specifically, the maximum throughput in Orthcatter is 248kbps, bigger than 237kbps in RapidRider, and 40kbps in [22], and 54kbps in [23]. There are two main reasons for these results. First, the interference cancellation is conducted before decoding, avoiding the redundant coding method employed by prior in-band works [22, 23]. Second, although Orthcatter and RapidRider conduct symbol-level modulation, Orthcatter experiences a smaller BER and thus better throughput.

Laboratory deployment. We further test Orthcatter in a laboratory to show how it works in a multipath-rich environment shown in Figure 9(a). Figure 9(b) shows that Orthcatter is robust to the multipath effect. Specifically, Orthcatter can maintain its throughput over 100kbps within the room of 65m^2. In position C, the BER is 0.14, similar to the LOS performance. In position D where the steel shelf blocks the direct link between Tag and Bob, our BER is 0.12 similar to the NLOS one. The reason is that our backscatter signal is still an OFDM signal armed with the cyclic prefix that can

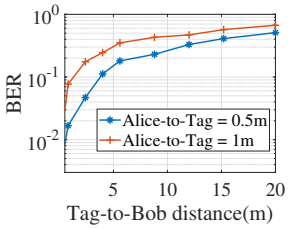


(a) BER

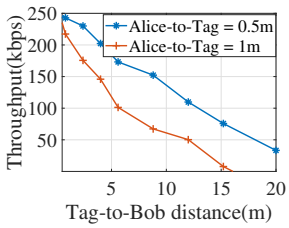


(b) Throughput

Figure 10: Orthcatter's performance in LOS.

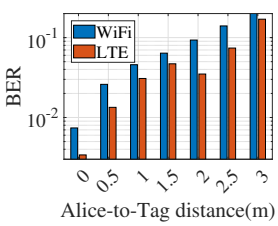


(a) BER

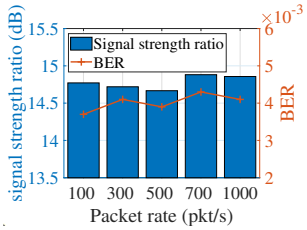


(b) Throughput

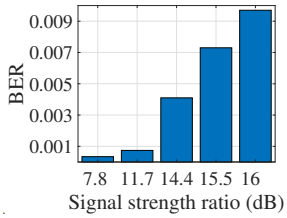
Figure 11: Orthcatter's performance in NLOS.



(a) Different exciter types



(b) Different packet rates



(c) Different signal strength ratios

Figure 12: Orthcatter's performance under different excitation types and packet rates and signal strength ratios.

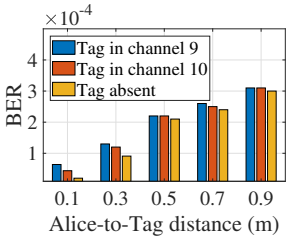


Figure 13: Impact of Orthcatter on the ambient WiFi.

cope with this effect. In addition, the extracted backscatter OFDM symbol after the interference cancellation process also has the pilot subcarriers, which can correct the phase error introduced by the wireless channel.

5.3 Micro Benchmarks

In order to confirm the applicability of Orthcatter to various exciters and deployments, we change Alice's signal type and packet rate and the spatial deployment of Tag. The results show that when the signal strength ratio between the original and backscatter signal is 7.8dB, its BER goes to as low as 3.4×10^{-4}, which is 59× better than the minimal BER of the side-band backscatter system RapidRider [31].

Impact of Alice-to-Tag distance and exciter type. We demonstrate that Orthcatter can work with longer Alice-to-Tag distance and other types of OFDM exciters besides WiFi. In our experiment, the exciter is the down-link LTE signal with a bandwidth of 1.4MHz and 128 subcarriers (72 data subcarriers and 56 null subcarriers). We set the Alice-to-Bob distance to 3m and put Tag in between. As shown in 12(a), the BER is smaller under the LTE exciter. This is because an LTE symbol contains more subcarriers, making it easier for Bob to perform the quasi-orthogonal interference cancellation and decoding. Furthermore, since the CP of an LTE OFDM symbol is far longer than that of WiFi, the tag working with LTE exciters can synchronize with Alice at a farther position. It is worth mentioning that the BER under the LTE exciter of a 1m Alice-to-Tag distance is similar to such BER of a 2m Alice-to-Tag distance. That said, when the Tag is within 2m from Alice, the synchronization error would hardly impact the tag data decoding. This is much better than prior works whose Alice-to-Tag distance is restricted to within 1m [9, 27, 28, 31, 37]. Besides, even with 3m Alice-to-Tag distance, the backscatter communication is still feasible.

Impact of Alice's packet rate. We here increase Alice's packet rate from 100pkt/s to 1000pkt/s and depict our results in Figure 12(b). We set both the Alice-to-Tag distance and the Tag-to-Bob distance at 0.5m in this experiment. The results show that changing excitation rates hardly has impact on the signal strength ratio and BER.

Impact of signal strength ratio. Since the original and backscatter signals are mixed on the receiver side, an important factor affecting our decoding is the signal strength ratio between these signals. We test its influence here and show the results in Figure 12(c). To this end, we should measure the signal strength ratio. This is actual value that would be shown in the x axis. In order to accurately measure the ratio, we replace the wide-band OFDM excitation signal with a narrow-band sine wave, enabling measurement of the narrow-band excitation signal and the narrow-band backscatter signal in the WiFi band. Note that we do not use this ratio for decoding, but use the estimation from null subcarriers (recall Figure 4) instead. We first change the relative positions among Alice, Tag, and Bob to get different signal strength ratios. Second, for each deployment, we ask Alice to generate a sine wave whose frequency spectrum has no overlap with the side-band backscatter one. Hence, we can receive the original and backscatter signals separately and compute their received signal strength. For the testing, we ask Alice to emit the OFDM WiFi signal to test Orthcatter's performance under the measured signal strength ratios. From Figure 12(c),

we have two observations: 1) When the signal strength ratio is 7.8dB, i.e., the original signal strength is 6× higher than backscatter one, the BER of Orthcatter is 3.4×10^{-4}, which is 59× better than the minimal BER of RapidRider. 2) Even when the ratio is 16dB, i.e., 40×, Orthcatter can still keep its BER below 10^{-2}. This demonstrates that Orthcatter can be deployed under diverse settings.

5.4 Coexist with ambient WiFi

We finally evaluate whether our backscatter transmission would impact the ambient WiFi traffics including both the in-band excitation signal (Alice) and an adjacent WiFi signal. We fix the Tag-to-Bob distance at 0.5m and vary the Alice-to-Tag distance. We here use two adjacent channels for the excitation signal (channel 9) and another WiFi signal (channel 10), respectively. Note that we focus on the encoded data before passing through the convolutional decoder because we need to acquire the quasi-orthogonal codewords utilized for the decoding. That said, the actual BER of the excitation signal would be lower due to the convolutional decoder. As shown in Figure 13, Tag would slightly degrade the in-band and side-band WiFi transmission. Consider the worst case where Tag is 0.1m from the exciter, the BER is increased by 4.4×10^{-5}. This increment is small and would be reduced by the convolutional decoder. Moreover, the impact nearly disappears when the distance exceeds 0.5m because of the greatly dropped backscatter signal strength. Therefore, our backscatter transmission hardly harms the ambient WiFi.

6 Discussion

Supporting Multiple tags. In this paper, we primarily focus on improving the point-to-point physical-layer backscatter transmission performance. However, Orthcatter can be expanded to support the access of multiple tags by adopting the MAC layer protocols like TDMA or Aloha.

Commercial receiver. Since tag data is decoded from the frequency-domain OFDM symbol, any OFDM receiver that converts the received signal to the frequency domain can work with Orthcatter. This means that our work has the potential for the commercial applications. However, Orthcatter is not fully compatible with commercial devices that would treat the weaker backscatter signal as noise and neglect it and do not provide access to the received encoded OFDM symbol. We hope that future commercial devices could grant more access to PHY information to make our Orthcatter applicable.

7 Related work

The related work can be classified into the side-band and in-band backscatter systems.

Side-band backscatter systems. Hitchhike [35], FreeRider [36], and MOXcatter [37] are side-band WiFi backscatter systems. They utilize the codeword translation, which embeds tag data by transforming the original codeword into another valid codeword. By avoiding the original signal, these works provide tens of kbps throughput. RapidRider [31] and STScatter [33] design OFDM backscatter and utilize the phase of the backscatter symbol to convey their data. This way, their data is embedded at the single-symbol rate, and their throughput is hundreds of kbps. In addition, PLoRa [29] focuses on the LoRa exciter and embed tag data over the initial frequency of the LoRa signal. LScatter [9] frequency shifts the LTE signal and embeds tag data over its phase.

In-band backscatter systems. Here the backscatter communication is completed in the same channel as the excitation to save the spectrum resource. [22] and [23] embed their data through phase modulation like RapidRider [31] and STScatter [33], yet only has a tens of kbps throughput. [26] and [20] respectively embed their message over ambient TV and WiFi signal amplitude and only provide throughput of several kbps. Aloba [15] conducts LoRa modulation through OOK instead of frequency modulation and has a limited communication range of only $1/3$ of PLoRa [29]. The performance of these in-band backscatter systems is incomparable to the side-band ones because they cannot cope with interference of the excitation signal effectively. Superior to them, our Orthcatter can cancel the interference and achieve even better performance than some side-band works.

8 Conclusion

We have designed and implemented Orthcatter, a novel in-band backscatter system that achieves reliable and high throughput communication while saving spectrum resources. The key innovation lies in the designed over-the-air code division technique that constructs the quasi-orthogonal codewords in the backscatter signal from the original excitation signal passively. This technique enables the in-band interference cancellation on the receiver. Our extensive field studies show that Orthcatter experiences 10^{-4} BER and 248kbps throughput, respectively 300× and 4.6× better than the state-of-the-art in-band system [23]. And it outperforms the side-band system like RapidRider [31] in some settings. Orthcatter can efficiently work under diverse practical scenarios, and would benefit a wide range of future backscatter applications.

9 Acknowledgements

This work was supported in part by National Natural Science Foundation of China under Grant No. 62271055, 62171296, 62122095, 62341201 and 62072472, and by Beijing Natural Science Foundation under Grant No. 4242002, and by a grant from the Guoqiang Institute, Tsinghua University.

References

[1] Adg902 rf switch. https://www.analog.com/media/en/technical-documentation/data-sheets/ADG901-902.pdf.

[2] Adg904 rf switch. https://www.analog.com/media/en/technical-documentation/data-sheets/ADG904.pdf.

[3] Lt5534 detector. https://www.analog.com/media/en/technical-documentation/data-sheets/5534fc.pdf/.

[4] Lte protocol. https://www.3gpp.org/dynareport?code=36-series.html/.

[5] Lte toolbox. https://ww2.mathworks.cn/help/lte/.

[6] Wifi protocol. https://www.tutorialspoint.com/wi-fi/wifi_access_protocols.html/.

[7] Ali Abedi, Farzan Dehbashi, Mohammad Hossein Mazaheri, Omid Abari, and Tim Brecht. Witag: Seamless wifi backscatter communication. In *Proceedings of the ACM SIGCOMM*, page 240–252, 2020.

[8] Bastian Bloessl, Michele Segata, Christoph Sommer, and Falko Dressler. An ieee 802.11a/g/p ofdm receiver for gnu radio. page 9–16, 2013.

[9] Zicheng Chi, Xin Liu, Wei Wang, Yao Yao, and Ting Zhu. Leveraging ambient lte traffic for ubiquitous passive communication. In *Proceedings of the ACM SIGCOMM*, pages 172–185, 2020.

[10] Huixin Dong, Yirong Xie, Xianan Zhang, Wei Wang, Xinyu Zhang, and He Jianhua. Gpsmirror: Expanding accurate gps positioning to shadowed and indoor regions with backscatter. In *Proceedings of the ACM MobiCom*, 2023.

[11] Caihui Du, Jiahao Liu, Shuai Wang, Rongrong Zhang, Wei Gong, and Jihong Yu. Timespan-based backscatter using a single cots receiver. In *Proceeding of the ACM MobiSys*, pages 450–461, 2023.

[12] Manideep Dunna, Miao Meng, Po-Han Wang, Chi Zhang, Patrick Mercier, and Dinesh Bharadia. SyncScatter: Enabling WiFi like synchronization and range for WiFi backscatter communication. In *Proceeding of the USENIX NSDI*, pages 923–937, 2021.

[13] Yalda Edalat, Katia Obraczka, and Bahador Amiri. A machine learning approach for dynamic control of rts/cts in wlans. In *proceedings of the EAI MobiQuitous*, pages 432–442, 2018.

[14] Xiuzhen Guo, Yuan He, Zihao Yu, Jiacheng Zhang, Yunhao Liu, and Longfei Shangguan. Rf-transformer: A unified backscatter radio hardware abstraction. In *Proceedings of the ACM MobiCom*, page 446–458, 2022.

[15] Xiuzhen Guo, Longfei Shangguan, Yuan He, Jia Zhang, Haotian Jiang, Awais Ahmad Siddiqi, and Yunhao Liu. Efficient ambient lora backscatter with on-off keying modulation. *IEEE/ACM Transactions on Networking*, 30(2):641–654, 2021.

[16] Roger Pierre Fabris Hoefel. Ieee wlans: 802.11, 802.11e mac and 802.11a, 802.11b, 802.11g phy cross layer link budget model for cell coverage estimation. In *Proceeding of the IEEE CCECE*, pages 1877–1882, 2008.

[17] Youjun Hu. Fourier analysis. 1970.

[18] Andrew H Jazwinski. Adaptive filtering. *Automatica*, 5(4):475–485, 1969.

[19] Jinyan Jiang, Zhenqiang Xu, Fan Dang, and Jiliang Wang. Long-range ambient lora backscatter with parallel decoding. In *Proceedings of the ACM MobiCom*, pages 684–696, 2021.

[20] Bryce Kellogg, Aaron Parks, Shyamnath Gollakota, Joshua R Smith, and David Wetherall. Wi-fi backscatter: Internet connectivity for rf-powered devices. In *Proceedings of the ACM SIGCOMM*, pages 607–618, 2014.

[21] Bryce Kellogg, Vamsi Talla, Shyamnath Gollakota, and Joshua R Smith. Passive wi-fi: Bringing low power to wi-fi transmissions. In *Proceeding of tbe USENIX NSDI*, pages 151–164, 2016.

[22] Taekyung Kim and Wonjun Lee. Exploiting residual channel for implicit wi-fi backscatter networks. In *Proceeding of the IEEE INFOCOM*, pages 1268–1276, 2018.

[23] Taekyung Kim and Wonjun Lee. Channel independent wi-fi backscatter networks. In *Proceeding of the IEEE INFOCOM*, pages 262–270, 2019.

[24] Jeongkeun Lee, Wonho Kim, Sung-Ju Lee, Daehyung Jo, Jiho Ryu, Taekyoung Kwon, and Yanghee Choi. An experimental study on the capture effect in 802.11a networks. In *Proceedings of the ACM WiNTECH*, page 19–26, 2007.

[25] Li Erran Li, Kun Tan, Harish Viswanathan, Ying Xu, and Yang Richard Yang. Retransmission $\neq$ repeat: simple retransmission permutation can resolve overlapping channel collisions. In *Proceedings of the ACM MobiCom*, page 281–292, 2010.

[26] Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Gollakota, David Wetherall, and Joshua R Smith. Ambient backscatter: Wireless communication out of thin air. In *Proceedings of the ACM SIGCOMM*, pages 39–50, 2013.

[27] Xin Liu, Zicheng Chi, Wei Wang, Yao Yao, Pei Hao, and Ting Zhu. Verification and redesign of ofdm backscatter. In *Proceeding of the USENIX NSDI*, pages 939–953, 2021.

[28] Xin Na, Xiuzhen Guo, Zihao Yu, Jia Zhang, Yuan He, and Yunhao Liu. Leggiero: Analog wifi backscatter with payload transparency. In *Proceedings of the ACM MobiSys*, page 436–449, New York, NY, USA, 2023.

[29] Yao Peng, Longfei Shangguan, Yue Hu, Yujie Qian, Xianshang Lin, Xiaojiang Chen, Dingyi Fang, and Kyle Jamieson. Plora: A passive long-range data network from ambient lora transmissions. In *Proceedings of the ACM SIGCOMM*, pages 147–160, 2018.

[30] Mohammed Saad Talib, Burairah Hussin, and Aslinda Hassan. Converging vanet with vehicular cloud networks to reduce the traffic congestions: A review. *Int. J. Appl. Eng. Res*, 12(21):10646–10654, 2017.

[31] Qiwei Wang, Si Chen, Jia Zhao, and Wei Gong. Rapidrider: Efficient wifi backscatter with uncontrolled ambient signals. In *Proceeding of the IEEE INFOCOM*, pages 1–10, 2021.

[32] Chenren Xu and Pengyu Zhang. Open-source software and hardware platforms for building backscatter systems. *GetMobile: Mobile Computing and Communications*, 23(1):16–20, 2019.

[33] Yifan Yang and Wei Gong. Universal space-time stream backscatter with ambient wifi. In *proceeding of the IEEE PerCom*, pages 101–110. IEEE, 2022.

[34] Jihong Yu, Caihui Du, Jiahao Liu, Rongrong Zhang, and Shuai Wang. Subscatter: Subcarrier-level ofdm backscatter. In *Proceeding of the IEEE INFOCOM*, pages 1–10, 2023.

[35] Pengyu Zhang, Dinesh Bharadia, Kiran Joshi, and Sachin Katti. Hitchhike: Practical backscatter using commodity wifi. In *Proceedings of the ACM SenSys*, pages 259–271, 2016.

[36] Pengyu Zhang, Colleen Josephson, Dinesh Bharadia, and Sachin Katti. Freerider: Backscatter communication using commodity radios. In *Proceedings of the ACM CoNEXT*, pages 389–401, 2017.

[37] Jia Zhao, Wei Gong, and Jiangchuan Liu. Spatial stream backscatter using commodity wifi. In *Proceedings of the ACM MobiSys*, pages 191–203, 2018.

[38] Renjie Zhao, Fengyuan Zhu, Yuda Feng, Siyuan Peng, Xiaohua Tian, Hui Yu, and Xinbing Wang. Ofdma-enabled wi-fi backscatter. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–15, 2019.

[39] Fengyuan Zhu, Yuda Feng, Qianru Li, Xiaohua Tian, and Xinbing Wang. Digiscatter: Efficiently prototyping large-scale ofdma backscatter networks. In *Proceedings of the ACM MobiSys*, page 42–53, New York, NY, USA, 2020.

EdgeRIC: Empowering Real-time Intelligent Optimization and Control in NextG Cellular Networks

Woo-Hyun Ko[*1], *Ushasi Ghosh*[*2], *Ujwal Dinesha*[1], *Raini Wu*[2], *Srinivas Shakkottai*[1] *and Dinesh Bharadia*[2]

[1] *Texas A&M University, TX, USA,* [2] *UC San Diego, CA, USA*

{whko, ujwald36, sshakkot}@tamu.edu *{ughosh, rainiwu, dineshb}@ucsd.edu*

Abstract

Radio Access Networks (RAN) are increasingly softwarized and accessible via data-collection and control interfaces. RAN intelligent control (RIC) is an approach to manage these interfaces at different timescales. In this paper, we introduce EdgeRIC, a real-time RIC co-located with the Distributed Unit (DU). It is decoupled from the RAN stack, and operates at the RAN timescale. EdgeRIC serves as the seat of real-time AI-in-the-loop for decision and control. It can access RAN and application-level information to execute AI-optimized and other policies in real-time (sub-millisecond). We demonstrate that EdgeRIC operates as if embedded within the RAN stack. We showcase RT applications called μApps over EdgeRIC that significantly outperforms a cloud-based near real-time RIC (> 15 ms latency) in terms of attained system throughput. Further, our over-the-air experiments with AI-based policies showcase their resilience to channel dynamics. Remarkable, these AI policies outperform model-based strategies by 5% to 25% in both system throughput and end user application-level benchmarks across diverse mobile scenarios.

1 Introduction

As we move into the age of NextG applications, cellular networks need to be versatile and must cater to a wide array of application-specific requirements concerning throughput, latency, and reliability. The "one size fits all" cellular network approach is fading, raising the need to be replaced by an adaptive, application-specific model. Modern applications can often furnish granular details about their operational context to aid such adaptation. A streaming app can reveal buffer status, AR/VR applications share viewing angles, robotic controllers offer positional data, and industrial IoT devices indicate data freshness. These capabilities can enable tailored network responses. For instance, in a video streaming network environment, rather than merely maximizing system throughput, it might be paramount to prioritize users nearing buffer exhaustion to prevent video playback halts. Different apps have varied response time needs; while a video might handle some delay, a VR game demands near-instant reactions.

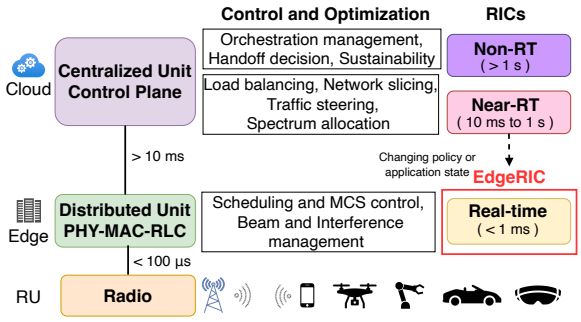


Figure 1: Timescales of RAN Intelligent Control for O-RAN. We desire real-time control at a latency < 1 ms.

While application awareness has become critical to the decision-making process of network functions in both wired and wireless (cellular) communications, the problem is far more complex with rapidly changing wireless channels for cellular communications. More specifically, the channel evolves at the timescale of milliseconds. Such changes in the channel make the control and optimization of the link to meet application requirements far more challenging.

The need for cellular networks to adapt to different applications, all while keeping up with the rapidly changing wireless channel, is supported by the cellular industry's pivot towards the standardization of open interfaces for RANs, encapsulated by the term O-RAN. By harnessing softwarization and disaggregation at every layer, O-RAN provides the flexibility to operate the RAN stack across diverse distributed computing platforms and offers enhanced monitoring and control through novel interfaces. Alongside this evolution, the RAN Intelligent Control (RIC) concept has gained prominence. RICs, distinct from the time-critical RAN stack, are designed to seamlessly access both the application layer and RAN-level

[*]These authors contributed equally to this work.

The source code for this project is available at https://github.com/ushasigh/EdgeRIC-A-real-time-RIC.git

data. This dual access facilitates cross-layer decision-making and control, bolstered by AI/ML enhancements, to serve a plethora of high-demand applications.

A defining attribute of the RIC is the time scales at which it operates, necessitating control decisions and information access at those time scales. Current RIC architectures, visualized predominantly as centralized control microservices, are placed in the cloud, as depicted in Figure 1. These can be classified into: (i) Near Real-time RIC (near-RT RIC): Provides a feedback loop to the RAN stack in a range of 10ms to 1s. (ii) Non Real-time RIC (non-RT RIC): Operates with a feedback loop timescale exceeding 1s.

Unfortunately, both these RICs inspired by SDN controllers, adopt a centralized, cloud-based control approach, minimizing risk to the RAN stack's essential operations within each TTI. This approach prevents potential disruptions, such as PHY-MAC task latency breaches leading to UE detachment. However, the inherent delay in decision-making, exceeding 10ms due to the wireless channel's rapid variability, creates bottlenecks. Such delays result in broad decision-making strategies like resource slicing, where, for example, a streaming app is allocated to a high-throughput RAN slice. Yet, as network demands grow, this coarse strategy struggles to maintain efficiency, given the fast-paced fluctuations in wireless channels that require TTI-scale responsive decisions for optimal performance.

While O-RAN has enabled unprecedented macro network optimizations, a vast reservoir of potential remains untapped in granular, *real-time* control. To that end, we propose a disaggregated real-time RIC platform called EdgeRIC. EdgeRIC is positioned on edge-compute close to the radio head, synchronizes intelligence with the granularity of RAN functions, but is decoupled with the RAN stack, which enables resilient operation of the RAN stack. EdgeRIC's careful design enables it to synchronize with RAN events and provide decision-control, while ensuring functioning of the tight-constrained RAN stack, even if decisions are not received from EdgeRIC. EdgeRIC facilitates control decisions and network telemetry at the TTI timescale (smallest unit of decision-making available at RAN), which is faster than underlying channel variations. The philosophy of EdgeRIC is to revolutionize the algorithmic control of lower-layer RAN functionalities (PHY-MAC-RLC), integrated with application awareness, thereby realizing the true potential of an AI-driven air interface. To provide an overview, Figure 1 showcases the myriad RAN enhancements that can be made feasible through intelligent control across diverse timescales.

Main Contributions

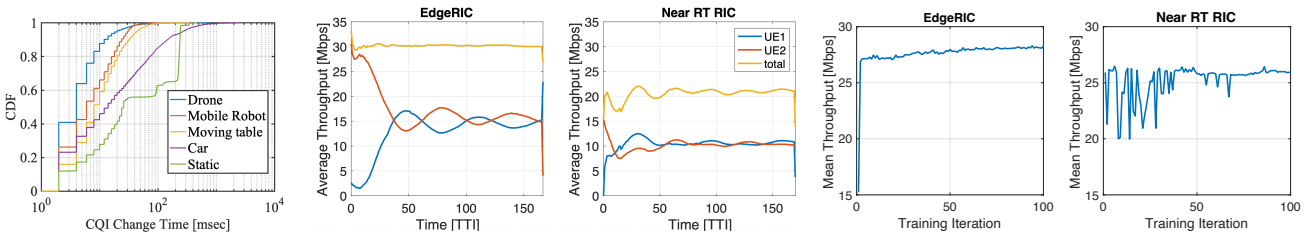
Our contributions are twofold: (i) EdgeRIC: a real-time RIC module, which facilitates real-time RAN telemetry and control. EdgeRIC is driven by the basic observation that real-time control for the cellular stack implies TTI-level sync with the RAN stack—all events happen between TTIs. Our principal contribution is rooted in the architectural and engineering decisions that ensure real-time performance of telemetry and AI-optimized control policies, without ever violating the TTI boundaries. EdgeRIC is strategically situated on edge compute, independent from the RAN stack, and interfaces with it via an O-RAN-like standard. (ii) EdgeRIC emulator: We further introduce the EdgeRIC training module, a cloud-compute-based emulator. This module is instrumental in the design, offline training, and deployment phases of AI-optimized algorithms. These features are enabled by providing a comprehensive full-stack, trace-driven (network and channel) emulation environment that accommodates multiple users and diverse applications.

Finally, we showcase potential benefits of real-time control with an AI-optimized μApp to provide resource allocation decisions at each TTI. We demonstrate (i) throughput increases of 5-10% using RT resource scheduling over near-RT approaches, (ii) up to 15% throughput increases through robust Reinforcement Learning (RL) based RT scheduling, and (iii) up to 30% enhancement in Quality of Experience (QoE) for video streaming via an application-aware RL-based RT scheduling policy over application-agnostic approaches. We also benchmark its performance against a near RT RIC. To the best of our knowledge, no existing RIC platform has demonstrated the benefits of real-time AI-in-the-loop-based RAN control while leveraging cross-layer application information in over-the-air experiments.

2 Motivation for Real-Time RIC

A fundamental value proposition of RIC is that it would enable the RAN to adapt to support heterogeneous applications over a variety of end devices ranging from smartphones, drones, cars, headsets, to sensors. What timescale of monitoring and control would enable application and environment responsive intelligent configuration and control?

Wireless Environment: Our channel measurements at 2.5 GHz across various mobile environments—drones, cars, indoor robots, and human movement—reveal that wireless channels fluctuate within milliseconds, highlighting the need for real-time control in mobile scenarios, illustrated in Figure 2(a). For instance, drone channels showed quality changes within 3-4 ms in over half the cases. Even in low-mobility scenarios, like a robot moving indoors, we observed significant changes within less than 10ms. This demonstrates the importance of designing systems like ours to adapt quickly to the dynamic nature of wireless channels. Additionally, even in stable channel conditions, real-time capabilities enhance RAN functions, such as enabling more aggressive Modulation and Coding Scheme (MCS) selections in the absence of packet drops. As cellular networks evolve towards supporting highly mobile applications with brief channel coherence times, achieving RIC latencies that match the sub-millisecond TTI timescale becomes crucial.



a) Channel qualities change as low as every 3 to 4ms under mobility

b) Realtime scheduling can support a higher system throughput

c) Near RT RL training yields lower throughput with delayed states

Figure 2: NextG networks need real-time intelligence and control.

RAN Control: We next verified the thesis that channel variations require real-time control by implementing a rule-based scheduler that prioritizes users with the largest channel quality index. We ran the experiment over a mix of channels gathered from our mobile experiments and show the throughput obtained in Figure 2(b). The latency experienced by near-real-time RIC with a round-trip latency of 30ms, corresponding to accessing cloud-based compute resources causes the throughput to drop by a third as compared to real-time control using the same scheduler located at edge compute accessed with a round-trip latency of 1ms.

AI Training: Can the RIC can support data collection, training and adaptation of AI/ML policies? Since many RAN management tasks involve feedback control, we are specifically interested in whether reinforcement learning (RL) training and execution can be supported. We conducted experiments on training an RL-based scheduler with a near-RT round-trip delay between state-action-reward of 60ms, versus one that has a round-trip delay of 1ms. As illustrated in Figure 2(c), the correlations between state-action-reward break down at near-RT, leading to lower throughput.

The wireless community is focusing on data-driven approaches for the air interface, enabling intelligent decisions at the TTI timescale. These encompass various tasks such as beam-forming decisions [14] [13] [26], interference management [18] [40], localization [17], channel estimation [21] [23], Modulation and Coding Scheme (MCS) selection [44] [15], power allocation [31], retransmissions and more. As illustrated above, none of these will be possible without real-time RIC co-located with the RAN on edge compute.

3 Related Work

Recent advancements in the development of near real-time (near-RT) RAN Intelligent Controller (RIC) frameworks and xApps have gained significant attention. Scope [4] introduces a containerized method for deploying network elements, supporting real-world emulation, AI/ML data collection, and network control APIs. [5] demonstrates the the integration of deep RL agents for near-RT RIC-based control, built with ColO-RAN [32], an AI/ML framework built upon the Colosseum network emulator [24]. These works focus on RAN resource slicing using xApps and RAN-embedded schedulers assigned to each slice. ORAN E2 [42] presents a software-defined radio testbed featuring an open-source 5G system that interacts with the O-RAN near-RT RIC through standard interfaces, utilizing xApps for RAN slicing. FlexRAN [9] offers a software-defined RAN platform where a master controller communicates with agents embedded in the LTE stack. However, FlexRAN lacks the ability to train or utilize AI-optimized policies while maintaining real-time constraints. FlexRIC [35] addresses this by providing a more modular variant of FlexRAN. It simplifies the 5G near-RT RIC architecture, adhering to the agent-controller approach.

Closer to the TTI-timescale, ChARM [1] presents spectrum selection based on supervised learning over IQ samples collected in real-time, but with control at near-RT RIC. For even finer control, an architecture for integrating distributed applications (dApps) into O-RAN has been proposed, with simulation results on the possible benefits that might be realized via network intelligence at real-time (<10 ms) [7].

Our initial work on EdgeRIC provided an architecture and messaging scheme for enabling real-time RIC (<1 ms), along with a feasibility study and demonstrations of the approach [8, 16]. More recently, an approach entitled Janus [10] also recognizes the significance of real-time intelligence. While both EdgeRIC and Janus aim at real-time measurement and control of the RAN, their architectural choices are fundamentally different. Janus is integrated directly with each Distributed Unit (DU), making it vendor-specific. Thus, Janus requires updates to its hooks and codelets within the RAN software at each site or DU, contingent upon the purchase of Janus by that specific DU. EdgeRIC avoids such tight coupling by following O-RAN compliant messaging interfaces, which decouples EdgeRIC from the RAN stack. EdgeRIC then uses the open source Gym-class interface to connect with an in-memory database and AI/ML algorithms. EdgeRIC's decoupled design allows for robustness by never interfering with time-critical RAN tasks, ease of runtime data gathering, standardized training and runtime updating of AI/ML models, and support across different vendors' equipment via compatibility with universally accepted O-RAN service models. Finally, since EdgeRIC is implemented over the open source

Table 1: Comparison of RIC frameworks

Framework	Connectivity to RAN stack	Monitoring and control	Application awareness	Adaptability to channel fluctuations	Full stack AI training support with real traces	Real World OTA evaluations
FlexRIC [35]	Disaggregated	10ms-1s	✓	×	×	×
ColO-RAN [32]	Disaggregated	10ms-1s	✓	×	×	×
dApps [7]	Disaggregated	6-10ms	×	✓	×	×
Janus [10]	Integrated	<1ms	×	✓	×	✓
EdgeRIC	Disaggregated	<1ms	✓	✓	✓	✓

software srsRAN stack and uses open messaging standards and interfaces, it is available for unfettered experimentation by the research community.

In the applications domain, streaming media has received much attention for AI-optimized control. For instance, AI/ML for choosing video streaming rate selection is considered in [12, 22, 30, 43, 45] from the server's perspective. In contrast, [3] studies optimal policies when the network can be controlled in the context of WiFi-based access. Here, reconfiguration of WiFi flow priorities using AI-optimized policies is shown to improve streaming performance.

In contrast to the above works, EdgeRIC is a simple, lightweight, disaggregated approach towards ensuring that TTI-level synchronized policies can be trained in non-RT and executed in real-time. Specifically, we show that our approach provides the ability to train robust cross-layer optimized policies in non-RT and a guarantee of completing the full feedback loop from sensing, AI-based policy execution and control within each TTI, and are the first to verify our claims while running full stack over-the-air experiments on mobile nodes. Table 1 summarizes comparable frameworks.

4 EdgeRIC Concept Architecture

The EdgeRIC design is primarily driven by the objective of infusing real-time intelligence into network functions and decisions at the network's edge. This approach is particularly important for making informed decisions based on instantaneous channel conditions. By situating decision-making processes closer to the RAN edge, we ensure that the channel metrics utilized are as current and relevant as possible, as opposed to being relayed from a distant cloud infrastructure.

EdgeRIC's architecture extends beyond edge operations to foster a cooperative relationship with cloud systems, enabling smooth data exchange and access to shared databases. It adopts a dual strategy, utilizing real-time edge data alongside cloud analytics to optimize decision-making. This approach aims to merge the promptness of edge processing with the extensive insights of cloud computing, emphasizing the synergy between cloud and edge to enhance cellular network capabilities. This integration effectively combines local responsiveness with global intelligence.

Our architecture design is motivated by two fundamental considerations, namely *(i) Disaggregated Programming Model:* O-RAN is driven by the desire to disaggregate the cellular stack into functional components that can be created by independent developers and instantiated on distributed compute resources. Consequently, EdgeRIC must be modular and decoupled from the RAN components. This will permit simple models of application development, run-time updates, robustness to errors, and bi-directional information sharing with user-defined applications, and *(ii) Real-time RAN Connectivity and Control:* While functionally decoupled from the RAN stack, EdgeRIC must enable messaging with TTI-level sync (< 1ms) with RAN events. This will enable real-time observability of RAN state, such as channel quality or backlog buffers, and decision making and control of the RAN stack each TTI to optimize performance. Consequently, EdgeRIC must be slaved to the TTI clock at the RAN, and messaging and decision making must be lightweight.

In order to realize these goals, EdgeRIC is composed of two modules: (i) the EdgeRIC execution module, which is the seat of μApps for real-time monitoring control of the RAN, as well as information aggregation from user applications, and and (ii) EdgeRIC emulation module, which is used as a full-stack emulator used for training of AI-based and other algorithms prior to instantiating them as μApps in the EdgeRIC execution module. We discuss the architecture and workflow of these modules below.

4.1 Disaggregated EdgeRIC Architecture

The real-time EdgeRIC execution module is illustrated in Figure 3, where we have shown it within the O-RAN architecture. O-RAN consists of a radio unit along with disaggregated microservices that perform the RAN functions. These microservices are divided across edge compute (near the radio) and cloud compute resources, based on the required latency targets. The components of the O-RAN stack are as follows: (i) RF Frontend: Open Radio Unit (O-RU), (ii) Edge Compute: Real-time components at the Open Distributed Unit (O-DU) supporting High-PHY, MAC and radio link control, and (iii) Cloud Compute: Open Centralized Unit (O-CU) with control and management functions. The final element is (iv) Cloud Compute: 5G Core, supporting management, billing

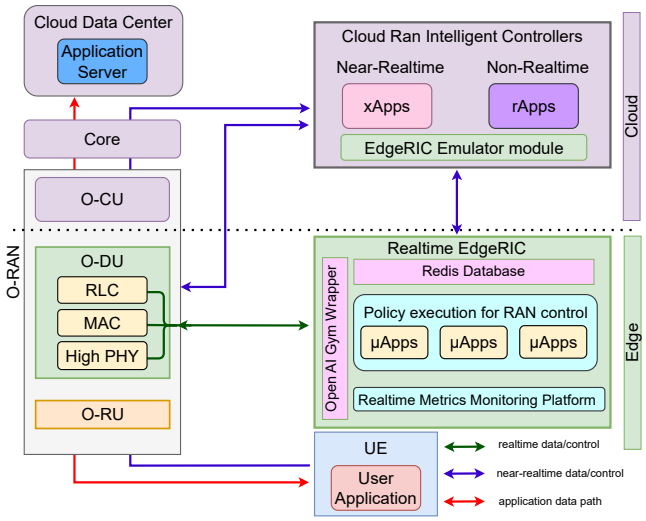


Figure 3: EdgeRIC concept architecture, showing its integration into O-RAN.

and Internet gateway functions.

O-RAN also provides specifications for two cloud-hosted microservices for (i) near-RT RIC, which supports xApps for policy adaptation at near-RT (10ms - 1s), and (ii) non-RT RIC, which supports rApps providing large timescale microservices management and data analytics. The standard provides protocols for the near-RT and the non-RT RIC to communicate with the O-RAN stack and with each other. In particular, the E2 Application Protocol (E2AP) operates over SCTP and provides pub-sub and on-demand messaging between RAN and near-RT RIC at near-RT latency.

EdgeRIC is designed as a microservice for the O-DU, closely integrated with the O-RAN architecture to enhance PHY-MAC level RAN functionalities through real-time. It utilizes μApps for executing real-time policies (with TTI latency), allowing for immediate RAN state adjustments and control. Connection to the O-RAN stack is achieved through a specialized real-time-E2 protocol (RT-E2), aligned with the TTI clock and leveraging IPC for microservice communication, ensuring a latency around 100 μs to meet stringent real-time requirements.

EdgeRIC operates in real-time, running on separate CPU cores at the edge compute cluster to ensure low latency without interfering with the O-RAN PHY-MAC microservices. It allows μApps to dynamically use RAN and application data for decision-making. Supporting integration with protocols like OpenFlow and ROS, EdgeRIC enables cross-layer policies for PHY-MAC control, enhancing system performance without disruption.

4.2 EdgeRIC Functional Components

The EdgeRIC real-time execution module, depicted in Figure 3, features interfaces for real-time communication with the RAN stack and near-real-time interaction with cloud modules. These interfaces connect to an Open AI Gym Wrapper, which abstracts them into a Gym-compatible environment, allowing components to interact with the RAN using Gym methods. This design enables μApp developers to employ either custom or AI/ML-based strategies, benefiting from the compatibility with standardized reinforcement learning frameworks. Consequently, μApps can seamlessly integrate these standardized codeblocks, facilitating efficient development and deployment within the EdgeRIC ecosystem.

The EdgeRIC execution module incorporates an in-memory Redis database for managing real-time RAN metrics and application data. This data supports μApps directly or through cloud processing, enabling cloud-hosted xApps to refine policies or ML models based on the data. These enhancements are fed back to the Redis database for μApps integration, allowing them to start with basic policies and improve them over time with cloud-derived insights, optimizing their performance dynamically.

4.3 EdgeRIC Emulator Module

The EdgeRIC execution module directly enables support for optimization based approaches to modulation, coding and queuing that are designed around well studied, substantiated, and tractable models. For instance, we can immediately instantiate approaches such as proportionally fair [37] or max-weight [41] scheduling across UEs on a per TTI basis with execution in RT, as if embedded within the RAN stack.

Our architecture also supports AI/ML approaches such as Reinforcement Learning (RL), a branch of ML that is explicitly tailored towards learning feedback-control policies. Training such policies is often hard in a real-world system, where user satisfaction is paramount at all times. Hence, we endow EdgeRIC with a full-stack emulator module for training, which can support user applications over trace-based or synthetic channels. The RL workflow is well aligned with the modality of a emulator based non-RT training of a base policy using data collected offline or via an emulator. Such polices can then undergo near-RT adaptation to the current environment, culminating in RT policy execution. Simultaneously, data is gathered at the edge, which is shared with the non and near-RT RIC for accurate training and adaptation. This three-timescale workflow is illustrated in Figure 4.

5 EdgeRIC Implementation

We now describe the implementation of EdgeRIC to satisfy the goals of our concept architecture. The experimental results that we present in this section were collected on two servers: Intel Xeon Gold 5218R CPU @ 2.10GHz, 20 cores and Intel i9 CPU @ 2.4GHz, 12 cores, without using GPUs. We chose the open source Software Radio Systems srsRAN stack [39] as the experimental RAN system for EdgeRIC integration due to its simple, modular codebase, its stability and compatibility with various core networks, 4G and 5G versions, and

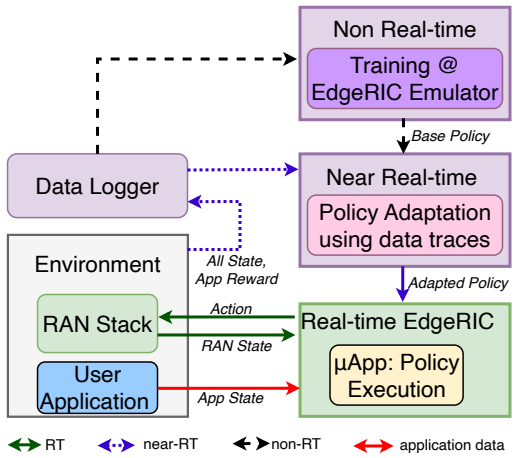


Figure 4: Non-RT policy training on EdgeRIC Emulator, Near-RT policy adaptation and RT policy execution.

the availability of the srsUE codebase. srsRAN runs on the general-purpose Ubuntu OS, which does not provide real-time guarantees.

5.1 EdgeRIC Execution Module

5.1.1 Real-time Connectivity to RAN and messaging

O-RAN specifications provide the E2 interface between the near-RT RIC and the O-DU or O-CU. Specifically, the E2 application protocol (E2AP) operates over SCTP and provides near-RT services for RAN monitoring and control. E2 does not support a real-time connectivity service, i.e., it is not synchronized with the TTI clock at the RAN. Hence, we extend the specification to create a real-time, RAN-synchronized variant that we call RT-E2. RT-E2 supports the following to connect EdgeRIC with the PHY-MAC stack at the O-DU.

RT-E2 TTI-Sync: Our system is synchronized to the TTI-level clock tick from the RAN stack, ensuring that EdgeRIC and the RAN maintain TTI-by-TTI alignment for accurate real-time control actions and reward feedback. The RAN stack uses a TTI counter, referred to as *RANtime*, included in all RT-E2 messages to EdgeRIC. Correspondingly, RT-E2 messages from EdgeRIC to RAN specify the TTI for policy application, ideally set to $RANtime + 1$, to ensure actions match the current RAN state. To prevent asynchrony caused by EdgeRIC's compute time exceeding one TTI, termed "Lazy RIC," EdgeRIC's RAN subscriber only retains the most recent RAN message, tagging policy messages with the latest $RANtime + 1$. The RAN disregards any EdgeRIC message not matching the current *RANtime*. Additionally, the RAN is equipped with a default mechanism to manage any potential Lazy RIC scenarios, ensuring stability despite possible invalid inputs from EdgeRIC.

RT-E2 Report: This is a periodic pub-sub procedure under which a module at the O-DU, such as radio link control may publish information at a given rate. Our default periodicity is one TTI, i.e., information may be generated in real-time. μApps at EdgeRIC may subscribe to the RT-E2 Report service and utilize it for inference and control. Subscription may be blocking in that the μApp will proceed only when new information is available from the RAN.

RT-E2 Policy: This is an event-driven pub-sub procedure under which a μApp at EdgeRIC may publish information to one of the O-DU modules such as UE priorities for resource allocation at the MAC layer. This information is used directly for real-time control at the O-DU. Subscription is non-blocking in that the O-DU subscriber will move on if no new information is available on this procedure, without breaking the tight TTI deadlines required by PHY-MAC.

RT-E2 API Support: RT-E2, synchronized with the RAN, supports messaging over TCP/UDP/SCTP or IPC, adapting to EdgeRIC's hosting environment. It necessitates both blocking and non-blocking pub-sub capabilities and must manage diverse messages, including sync, state, action, reward fields, and UE identities. Opting against modifying the O-RAN E2's limited SCTP-based messaging, we use ZMQ for RT-E2 due to its low latency, minimal overhead, versatile pub-sub modes, and compatibility with IPC or TCP, accommodating our required message formats.

TTI-Level Events: The sequence of events, occurring every TTI is shown in Figure 5, where we see (i) state measurement and transmission from RAN, (ii) reception, processing and response at EdgeRIC, and (iii) final resource allocation at RAN. Note that srsRAN receives each EdgeRIC message well before the TTI boundary, but only reads it in a non-blocking manner at the beginning of each TTI.

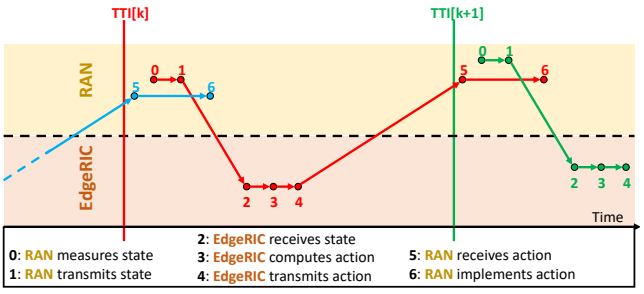


Figure 5: TTI-level events for EdgeRIC to RAN loop.

Real-time operation evaluations: We aim to demonstrate the effectiveness of our synchronization and messaging techniques between the RAN and RT-RIC, crucial for feedback control and RL training success. Through tests with the srsRAN stack and RT-RIC on a server, operating at 10 MHz downlink load and achieving about 37.5 Mbps throughput, we found our system maintains a median round trip latency of just 100 μs. This performance is satisfactory for managing

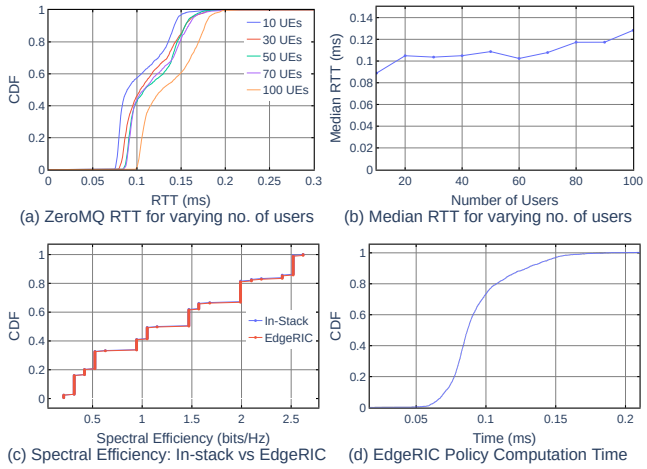


Figure 6: EdgeRIC Feedback latency, spectral efficiency and AI-policy execution times.

information dissemination and control commands for approximately a hundred UEs, showcasing ZeroMQ's suitability for real-time operations as illustrated in Figure 6(a) and (b).

In our study on MAC layer resource block (RB) allocation, the RAN stack communicates UE state information, including RNTIs, CQIs, buffer states, and previous downlink bitrates, to the RT-RIC, which then decides on downlink RB allocations per UE. Figure 6(c) shows that the spectral efficiency achieved by implementing policies through EdgeRIC is on par with integrating them directly into the RAN stack, indicating that our modular approach maintains efficiency while ensuring RAN stack stability. This underscores the effectiveness of our decoupled architecture in balancing policy enforcement with architectural integrity.

5.1.2 Cross-Layer Connectivity and Logging

Our choice of ZMQ for inter-process communication between RAN and EdgeRIC is also extendable to cross-layer application awareness, shown in Figure 4. Since ZMQ can operate over TCP or UDP on an IP network, applications can simply use ZMQ to publish their state information to EdgeRIC. Apart from being lightweight and having APIs in most programming languages, ZMQ also permits client authentication and encryption via CurveZMQ [6] for security.

We also enable EdgeRIC with an in-memory Redis database for data logging and sharing, shown in Figure 4. Redis is a fast, lightweight, key-value store, in which we log data digests, as well as trained models for sharing across the elements of EdgeRIC. An added advantage of Redis is that we can save all traces to drive at experiment conclusion, which allows for post processing and performance analysis.

5.1.3 Integration with OpenAIGym

OpenAIGym is an open source python library that provides a framework for developing an interface to interact with and query the environment by any given algorithm. While it is typically used for developing and comparing RL algorithms, it can be used as a standard approach for realtime policy execution, regardless of whether the policy in question is based on RL. This openness motivates us to develop an OpenAIGym interface connecting the RAN stack, EdgeRIC and the control algorithms in the form of μApps that it hosts. Our OpenAIGym interface allows for swift policy development and freedom of execution of algorithms as desired. Figure 6(d) shows the time taken by EdgeRIC to execute a forward pass of a trained policy network using only CPU, while running a fully loaded RAN. The mean value is less than 100 μs, which implies that TTI-scale execution is straightforward.

5.2 EdgeRIC Emulator Module

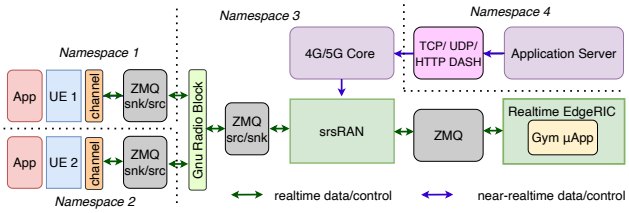


Figure 7: EdgeRIC emulation environment

To bridge the "sim-to-real" gap in RL-optimized control, we've developed the EdgeRIC training module, which forgoes the need for a complex Python simulator for simulating RAN, RIC, and application dynamics. This module leverages actual RAN and RIC codebases with virtual radios and channel simulation techniques, using ZeroMQ for virtual radio interfaces in srsRAN, enabling accurate real-world emulation. It routes complex-valued samples, typical for software-defined radios, through ZeroMQ sockets, incorporating simulated channel effects. A GNU Radio flowgraph with ZMQ Source and Sink blocks allows distribution of these samples to multiple srsUEs, simulating real user equipment. Furthermore, separate IP namespaces for UEs and application servers facilitate running real-world TCP or UDP based applications end-to-end, closely mimicking actual deployment scenarios as shown in Figure 7. This setup ensures minimal sim-to-real disparity, making the EdgeRIC emulator highly effective for real-world applicable policy training and testing.

6 Case Study: An RL based scheduling μApp

In this paper, our illustrative use-case is on realtime downlink resource block allocation (scheduling), which utilizes (i) channel quality information (wireless state), (ii) downlink backlog buffers (RAN state) and (iii) media buffer length for video streaming (application state). We develop an RL-trained realtime scheduling application operating on EdgeRIC, which we refer to as a μApp. We study performance, both from the

perspective of throughout maximization under a variety of channel conditions, as well as stall minimization in a video streaming application case. We present the results of real-world over-the-air experiments to validate our approach. In this section, we establish the performance gains achievable with real time RAN control on simulated channel traces.

We use a downlink resource block (RB) scheduling μApp to illustrate an EdgeRIC application, since (i) scheduling has to be done each TTI in realtime, (ii) it requires channel quality information from the wireless link, downlink backlog buffer on a per-UE basis from the RAN, and can utilize application-level information such as the media buffer length in a video streaming for prioritization, and (iii) being such a fundamental problem, has a variety of baseline approaches to compare against RL-based scheduling algorithms. We now describe the design and training of a scheduling μApp.

Weight Based Abstraction of Control: Optimal queueing and wireless resource management often employ structures like threshold [11, 25], index [29, 33, 34], and linear policies [2, 19] for their simplicity and learnability, with some showing properties like monotonicity or concavity [2]. Systems such as [27] develop weights to prioritize flows, ensuring max-min fairness among them. All these structured policies can effectively be represented by assigning relative priorities to the different connected UEs. For example, the so-called Whittle index is a scalar parameter corresponding to the value of resources allocated to a given UE, which can quickly be learned independently of other UEs [28]. Resource allocation may also be done with a fairness metric in mind, such as proportional fairness, where RBs are assigned to a UE based on the ratio of its current as compared to its average channel quality, or max-min fairness [38]. Motivated by these ideas, our general approach for downlink RB allocation is for EdgeRIC to provide values w_i for each connected UE i over realtime information exchange at each TTI. The 5G MAC will then allocate an number of RBs in a manner proportional to w_i over the next TTI. Such an abstraction provides simplicity of actions for the resource allocation policy, while maintaining its ability to attain near-optimal allocations in realtime.

6.1 Training RL on emulator

We utilize the model-free RL algorithm, Proximal Policy Optimization (PPO) [36], for training an agent on optimal resource allocation due to its straightforward implementation and efficiency. Training involves collecting 5,000 samples per iteration, updating the agent's policy neural network via backpropagation, and then using the updated agent to collect another 5,000 samples to evaluate performance and track progress. Each sample corresponds to a transmission time interval (TTI) and includes the environment's current state, the agent's action, and the resulting reward and next state. The RL policies, trainable on the emulator, can focus on RAN-specific scenarios or incorporate application-level data for broader optimizations. We specifically explore downlink throughput enhancement (Section 7.3) and video streaming stall reduction (Section 7.4) as two key use cases.

Table 2: RL Specifications: Throughput Maximization

State ($s[t]$)	$B_i[t], CQI_i[t]\ \forall i$
Action ($a[t]$)	$w_i[t]\ \forall i$
Reward ($r[t]$)	*total throughput*

For the throughput-maximization goal, we utilize RAN-level CQI and UE backlog buffer lengths as state information, with action being the allocation weights for UEs and the reward as total throughput. Training typically reaches reward saturation after 100 iterations, equivalent to 500,000 TTI samples. The RL setup, detailed in Table 2, includes CQI ($CQI_i[t]$) and backlog data ($B_i[t]$) per UE i and the allocation weight for UE i ($w_i[t]$) as actions. Factoring in data collection/transfer to the RL agent and time for actor-critic policy updates, total training completes in about ten minutes.

6.2 Evaluations on emulator

We conducted emulations using synthetic channel traces to assess the potential gains achievable by a real-time agent for policy computation and control. Two metrics we use throughout our study is the downlink system throughput and the downlink backlog buffer lengths at the RAN. We desire to evaluate two basic questions, *(i) How much does performance improve with realtime control as opposed to near real time control* and *(ii) How does RL-based control policy perform compared to basic algorithms?*

We consider three basic algorithms for weight-based resource allocation. In all the below approaches, each UE is assigned a weight $w_i[t]$ at TTI t. The weights are then normalized over all UEs as $\tilde{w}_i[t] = w_i[t]/\sum_j w_j[t]$. The RAN receives the normalized weights $\tilde{w}_i[t]$ from the RIC, and performs an allocation of resource block groups (RBGs) in proportion to the weights, i.e., $R_i[t] = \tilde{w}_i[t] R_{total}[t]$, where $R_i[t]$ is the assignment to UE i, and $R_{total}[t]$ is the number of RBGs available in TTI t. While some approaches call for an absolute prioritization of UEs that have a maximum weight [41], we find in practice that a proportional division based on weight leads to better overall performances.

CQI-Fair Allocation: Here, the weight of UE is equal to its realized CQI. Hence, $w_i[t] = CQI_i[t]$, where $CQI_i[t]$ is the realized CQI of UE i at time t. This approach effectively tries to obtain a large total throughput by prioritizing these UEs that have a large CQI in the current timeslot.

Proportionally-Fair Allocation: The allocation weight for each UE is determined by the ratio of its current CQI to its average CQI, aiming to prioritize UEs with better-than-average channel conditions. The average CQI for UE i, represented as $AvgCQI_i[t]$, is computed using an exponentially weighted moving average up to time t. Therefore, the weight $w_i[t]$ is calculated as $CQI_i[t]/AvgCQI_i[t]$.

Max-weight Allocation: Here, the weight of a UE is the product of its current CQI and the backlogged bytes in the downlink queue corresponding to that UE. The max-weight policy is known to be throughput optimal [41], in that it can achieve the capacity region of the system. Thus, we have, $w_i[t] = CQI_i[t]B_i[t]$, where $B_i[t]$ is the number of backlogged bytes in the downlink queue of UE i.

Our first question, the performance comparison between downlink RB allocation algorithms as μApps on EdgeRIC versus as xApps on a cloud-based RIC focuses on the impact of latency. μApps on EdgeRIC benefit from low round-trip latencies of mere tens of microseconds for state information reception and action generation from the RAN. In contrast, xApps in the cloud suffer from significant forward and reverse network latencies, leading to round-trip times in the tens of milliseconds. To illustrate cloud latency effects, we simulate appropriate delays within EdgeRIC.

Our experiment with synthetic channel traces shows that using the CQI-Fair allocation algorithm as a μApp on EdgeRIC, compared to a cloud-based RIC with 30ms latency, results in a 50% throughput increase with stable backlog buffers (Figure 2(c)). This is further detailed in Table 3. The throughput improvement is also demonstrated in a 4-user scenario, Figure 14 and Table 7 in appendix, thus underscoring the benefits of real-time control in enhancing performance metrics.

Table 3: Load: 35Mbps, Channel: 2 UE synthetic channel

		EdgeRIC	15ms	30ms
Max CQI	Avg. Thrpt.	**32.6**	24.2	18.0
	BL[MB]	0.61	0.64	**0.57**
Prop. Fair.	Avg. Thrpt.	**30.7**	25.7	21.9
	BL[MB]	**0.65**	0.67	0.68
Max Weight	Avg. Thrpt.	**30.0**	23.3	20.9
	BL[MB]	**0.60**	0.62	0.65

To answer our second question, we demonstrate the training and evaluation of an RL algorithm. Scenario 1 is based on a 2UE environment. both connected users have uniform variation in CQI values over time, ranging from 1 to 15. In scenario 2, one user experiences good channel conditions (CQI values between 8 and 15), while the other user experiences poor channel conditions (CQI values between 1 and 7). In Scenario 3, the CQI values are randomly generated. Figure 8 shows training and evaluation on Scenario 2 and Table 8 (appendix) summarizes the performance of RL algorithms on synthetic channel traces.

We show the CDF of end-to-end latency of the entire event chain from RAN to EdgeRIC and back (including policy execution via a forward pass on the policy network), culminating in resource allocation at each TTI in Figure 9. We observe that the end-to-end latency is always less than 1 ms, i.e., the RT-E2 procedures (Section 5.1.1) successfully meet the target of event completion within each TTI.

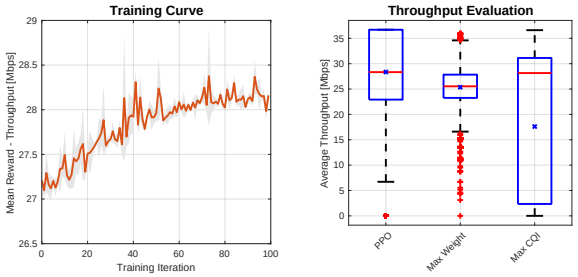


Figure 8: RL training and evaluation on emulator

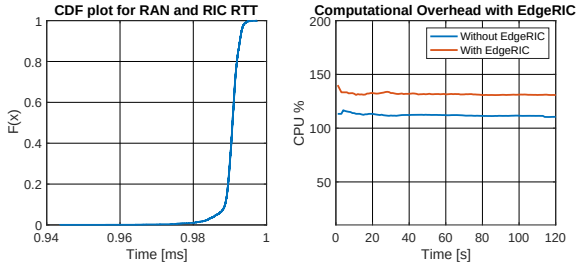


Figure 9: CDF of End-to-end RTT between RAN and EdgeRIC, showing that TTI timings are always met.

Finally, we measure the computational overhead of running EdgeRIC, as compared to running the vanilla srsRAN stack under a full traffic load. We see that the difference in CPU utilization is only about 20%, which means that EdgeRIC is fairly lightweight and does not need execessive additional compute resources. Hence, co-locating EdgeRIC at the O-DU level seems quite feasible.

6.3 Scalability Study

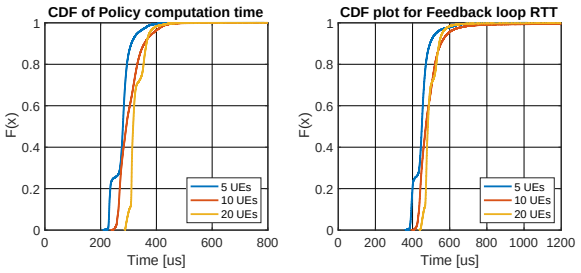
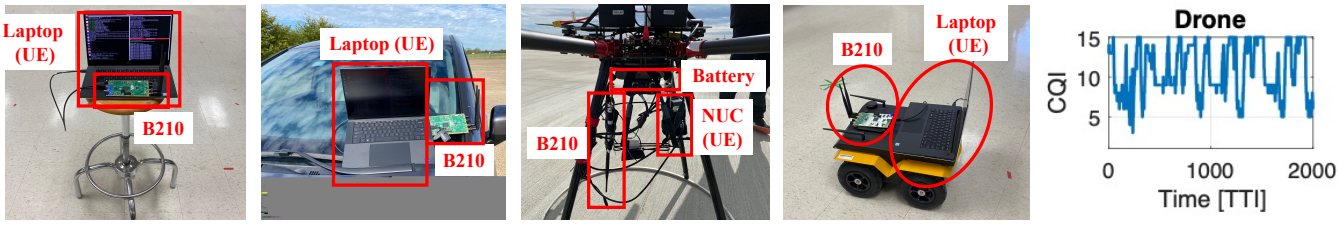


Figure 10: EdgeRIC is able to handle a large number of UEs

We show that our system is stable and is feasible to operate as we scale the number of UEs. For the scheduling μapp considered in this case study, we introduce additional, simulated UEs (which have the same states and application behavior as the real UEs) into the system in order to capture the system performance under an increased number of users. Figure 10 shows that the feedback loop RTT remains well below 1ms and the ML model inference time is under 400μs.



(a) Turntable (b) Car (c) Drone (d) Mobile Robot (e) CQI-evolution

Figure 11: Experimental setups for collecting CQI data of various mobility and evolution of their CQI traces

7 EdgeRIC Evaluation

In this section, we evaluate EdgeRIC by addressing three fundamental questions. These questions are: *(i) Does real-time RAN control through µApps on EdgeRIC outperform near-real-time control through a cloud-based RIC? (ii) Is it feasible to implement real-world AI optimization (RL training and feedback) for resource allocation over EdgeRIC* ? and *(iii) Does application state feedback to EdgeRIC provide significant improvement to the end-user Quality of Experience (QoE)*? All results presented in this section are based on real world channel traces and over the air experiments.

7.1 Experimental Setup

EdgeRIC extends the srsRAN codebase, supporting both software-defined radios and commercial UEs. Built on srsRAN version 21.10, our setup includes USRP B210 SDRs, an edge DU, and an embedded 20ms delay in the CU stack for realism. The srsRAN station features channel trace logging and operates in the 2.5 GHz EBS band under an experimental FCC license to prevent interference with local networks. Experiments utilized a 10 MHz bandwidth, involving a stationary base station and mobile users.

7.1.1 User Devices and Channel Traces

We use a static base-station with different UE types to collect channel traces as shown in Fig. 11. We considered various mobility models, both within the laboratory and in outdoor environments. We extract the channel quality indicator (CQI) values sampled at each transmission time interval (TTI) for each user at the srsRAN for runs of approximately 8 minutes each. Next, we summarize the UEs considered in this study.

TurnTable UE: Figure 11(a) shows the setup of a UE on a movable turntable, similar to a user sitting on a chair and rotating. The base station equipped with a B210 is on a static table, while a UE node with a B210 is mounted on the turntable. The UE node is moved away from or towards the base station at about 1 m/s and rotated to produce significant variations in its CQI values. The distance between the UE node and the base station ranges from 0.5 meter to 4 meter.

Car UE: Figure 11(b) shows a UE setup in a car, using a USRP B210 and laptop powered by the car. CQI data was collected while driving along three paths: 1) a 6-meter-radius circle with a maximum speed of 4.5 m/s, 2) a 30 meter x 3 meter rectangle with a maximum speed of 5.4 m/s, and 3) a 50-meter straight line with a maximum speed of 9 m/s.

Drone UE: We used a drone experiment to demonstrate performance with faster channel variations in 3D space. Figure 11(c) shows a B210 and a NUC (a small and portable computer) mounted on a Big-Hexy hexa-copter drone, while the base station, equipped with another B210, was placed on the ground. The drone was flown over the base station along two paths: 1) a 10-meter straight line at a height of 15 meters and with a speed of 2.2 m/s, and 2) a 15-meter straight line at a height of 20 meters and with a speed of 3 m/s.

Indoor robotic UE: Mobile robots were used to for indoor mobility experiments. In Figure 11(d), a B210 and laptop were mounted on a Jackal robot, while the base station with an X310 was placed on a table. The robot moved along a 1.6 m x 1.6 m square path and rotated at each corner, causing CQI values to drop due to signal blockage. The working area was limited to 4 m from the base station, similar to robot control or industrial IoT with Private 5G.

We show the drone trace in Figure 11(e), and the overall CDF was earlier shown in Figure 2(a). We see that several scenarios yield CQI changes in the sub 10 ms range, with the drone trace showing this effect for almost 90% of samples.

7.1.2 Evaluation Scenarios

To generate realistic scenarios, we collected channel traces from various environments and utilized them in our emulation setup. This setup includes the same core, radio access network (RAN), and user equipment (UE) modules as our over-the-air experiments that generated the CQI traces. By replaying the CQI traces with a desired number of UEs, algorithmic methods can be compared while maintaining the same end-to-end applications and channel conditions. For end-to-end over-the-air experiments, we used a X310 as a base station and two B210s as UEs. One of the UEs had lower channel quality than the other by placing them at different distances from the base station. Table 4 summarizes the scenarios considered.

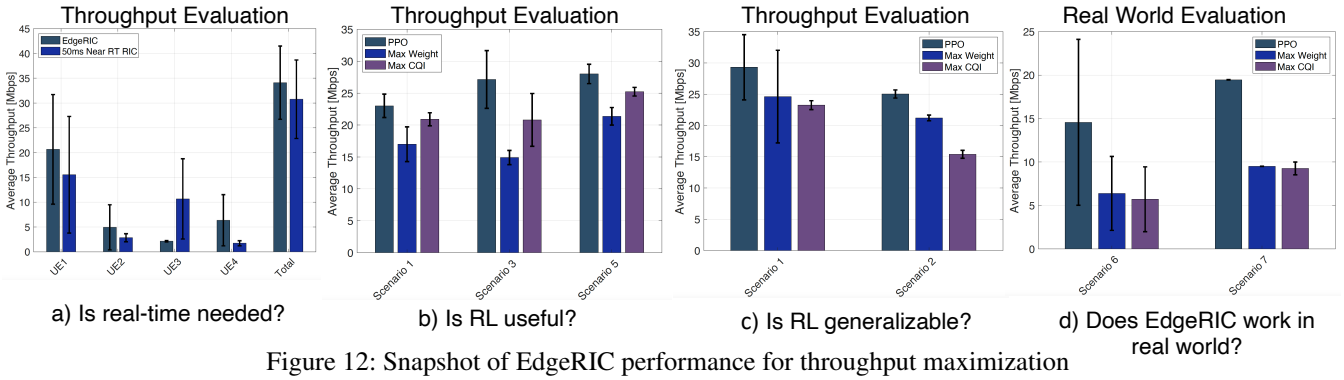


Figure 12: Snapshot of EdgeRIC performance for throughput maximization

Table 4: Summary of all scenarios

Scenario	Channel Description
Channel Traces from Experiments	
Scenario 1	2 Drone UEs
Scenario 2	2 Turntable UEs
Scenario 3	2 Car UEs and 2 Drone UEs
Scenario 4	2 Car UEs and 2 Indoor Robotic UEs
Scenario 5	2 Random Walk UEs and 2 Turntable UEs
Complete Over-the-Air Experiments	
Scenario 6	2 UEs on indoor mobile robots
Scenario 7	2 UEs on indoor stationary robots

7.2 Micro-benchmarks: Edge vs. Cloud

We use iPerf for measuring throughput and generate microbenchmarks. We test our scenarios on different UEs, each with varying traffic loads, and report the throughput and backlogs observed with different scheduling algorithms presented in Section 6.2. The validity of our results presented in Section 5.4 is confirmed using realistic channel traces (4 Turntable UEs), shown in Table 5 and Figure 12(a). This provides evidence in support of the hypothesis that real-time control can significantly improve the system throughput.

Table 5: Load: 35Mbps, Channel Trace: 4 Turntable UEs

		EdgeRIC	50ms	100ms
Max CQI	Avg. Thrpt.	**33.4**	21.2	29.5
	BL[MB]	1.34	**0.84**	1.12
Prop. Fair.	Avg. Thrpt.	**28.6**	26.6	23.5
	BL[MB]	1.20	1.29	**0.93**
Max Weight	Avg. Thrpt.	**33.2**	28.8	31.0
	BL[MB]	1.14	1.30	**1.12**

7.3 Impact of RL on Micro-benchmarks

We present compelling evidence for the feasibility, impact, and robustness of RL in executing real-time control policies. We begin by addressing the question of RL training in an emulation environment, which is answered by the set of curves in Figure 16 (appendix). These curves demonstrate that RL training with realistic channel traces is highly efficient, with an agent typically converging in just 20 to 40 iterations. The potential of RL is evident from a series of figures that answer important questions about its application.

Figure 12(b) shows us that RL can achieve higher throughput than traditional algorithms, bringing us to answer: is RL truly useful? This is further summarized in Table 6, which displays the total system throughput and the mean of the total backlog buffer for various scenarios, offering a snapshot of the RL performance in real-time. The values displayed in the table represent the throughput in Mbps (left) and the backlog buffer in MBytes (right).

Reinforcement learning (RL) schemes can surpass traditional algorithms under specific conditions, such as varying UE application loads where the max CQI strategy falls short. Our RL algorithm, particularly PPO, adapts to mirror the max-weight algorithm yet excels in scenarios like round-robin allocations, serving one UE per TTI. This approach contrasts with max-weight's fractional resource distribution across all UEs, which, while based on a weight-control scheme, may not always yield optimal results in environments with diverse load demands.

Table 6: Throughput and Backlog Buffer Evaluation

	PPO	Max Weight	Max CQI
Realistic Channel Traces			
Scenario 1	**29.1/0.38**	26.1/0.53	14.9/0.39
Scenario 2	30.5/**0.38**	**31.9**/0.43	14.42/0.39
Scenario 3	**25.3**/1.5	22.9/1.3	18.67/**0.97**
Scenario 4	**25.9**/1.5	23.9/1.21	20.3/**1.05**
Scenario 5	**28.5/0.96**	26.3/1.46	23.3/1.01
Over the Air Experiments			
Scenario 6	**14.6/0.19**	6.4/0.45	5.7/0.44
Scenario 7	**19.33/0.05**	10.71/0.34	9.06/0.35

Table 7: RL specifications: Video Streaming

State ($s[t]$)	$B_i[t], CQI_i[t], MB_i[t]\ \forall i$
Action ($a[t]$)	$w_i[t]\ \forall i$
Reward ($\sum_i r_i[t]$)	$r_i[t] = \begin{cases} -20, & \text{if } MB_i[t] < 2\ sec \\ +2, & \text{otherwise} \end{cases} \quad \forall i$

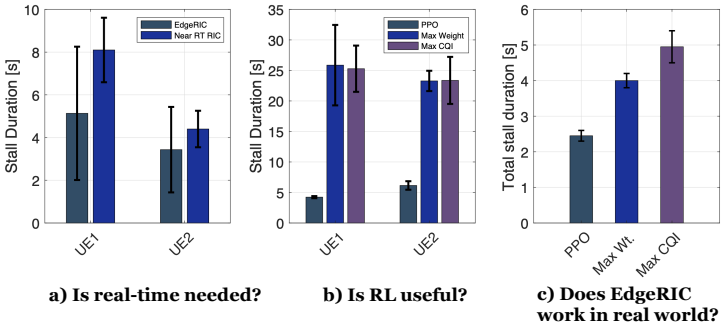


Figure 13: Snapshot of EdgeRIC performance for video streaming application

Figure 12(c) illustrates the transferability of RL models, showing that a model trained in one scenario can adapt to others with similar user numbers, highlighting its generalizability. Specifically, a model from a random walk scenario was applied to both a 2 Drone UEs setup (Scenario 1) and a combination of 2 Car UEs and 2 Turntable UEs (Scenario 2). The crucial question of RL's real-world efficacy in realtime wireless control is affirmatively addressed in Figure 12(d). Real-world tests with EdgeRIC revealed that traditional algorithms struggled with channel variability, impacting performance. In contrast, the RL policy maintained system stability and achieved consistent throughput, demonstrating RL's potential for performance gains in realtime scenarios.

7.4 Cross-Layer Optimization: Case Study

In this case study on enhancing video streaming performance through cross-layer optimization, we set up a heterogeneous environment with four users: two streaming high-quality video from an HTTP server and two generating background traffic with iPerf. We use the GPAC [20] library for DASH-style video segments and define a stall event when the media buffer drops below 2 seconds. The media buffer size, updated every 40 milliseconds to match a 24fps video, is communicated to EdgeRIC, along with network states at each TTI. To simulate realistic conditions, we introduce a 20ms delay representing uplink latency on srsRAN. An RL agent, trained with EdgeRIC, aims to optimize video playout and resource allocation, responding to both network and application states to minimize video stalls.

The RL framework specifications for this setup are presented in Table 7. Here, apart from downlink backlog $B_i[t]$, and channel quality $CQI_i[t]$ of UE i, we augment the state with the length of the media buffer (in seconds) of the video streaming application of UE i, denoted $MB_i[t]$. The reward now depends on the stall performance of the applications, with smooth playout receiving a positive reward, and a stall receiving a large negative reward. We next conduct real-world experiments on video streaming. Comparing the performance of UE1 and UE2 controlled by the RL policy and the standard RAN scheduling algorithms in real-time, the RL policy outperformed the standard algorithms by incorporating "application awareness", demonstrating its potential to provide a quality of experience (QoE)-optimized solution with proper training. Figure 13(b) summarizes this statement showing the metrics observed by playing a video for 120s. Figure 13(c) presents the results of our experiment, which evaluated the stall performance of the video streaming application over the air in a static environment with a one-streamer and one-loader scenario for 30 s videos. We see that the RL trained policy only has about a third of the stalls experienced by the other approaches. Further, to answer the question if a cloud based RIC can support a video streaming application under a highly fluctuating channel condition, we implemented a variant of the max weight scheduling algorithm (infused with the application state, $w_i[t] = CQI_i[t]B_i[t]/MB_i[t]$) and benchmarked the stall performance for a 30s video. The result is depicted in Figure 13(a) which clearly indicates that an application aware real-time control outperforms a near-RT control approach.

8 Limitations and Future Work

We presented a platform EdgeRIC and its full stack emulator. When combined together, we can train and deploy cross-layer AI-optimization algorithms that can provide decision and control at a TTI-timescale. Future directions of this work are:
Scale to 100s of users: Our demonstrations are with four users, primarily due to the instability with our setup of srsRAN for a larger number of UEs. However, we have shown that EdgeRIC platform can send the messages for 100 UEs, with a RAN to EdgeRIC latency of about 100μs (Figure 6). That said, we have not tested over-the-air with support for many more simultaneous UEs, which is key future work.
Mutliple base-stations: Our work has demonstrated performance improvements for a single base-station. We anticipate with multiple base-stations, we would have an EdgeRIC attached to each base-station/DU, with Near-RT RIC coordinating all the EdgeRIC instantiations. This would involve problems in federated learning that we will explore.
Ethical concerns: Does not raise any ethical issues.
Acknowledgement: This work was funded in part by NSF Grants CNS 2312978, CNS 1955696, ECCS 2030245, CNS 2312979 and ARO grant W911NF- 19-1-0367. All opinions and findings are of the authors.

References

[1] Luca Baldesi, Francesco Restuccia, and Tommaso Melodia. ChARM: NextG spectrum sharing through data-driven real-time O-RAN dynamic control. *arXiv:2201.06326*, 2022.

[2] Dimitri P Bertsekas. *Dynamic programming and optimal control*, volume 1,2. Athena scientific Belmont, MA, 2017.

[3] Rajarshi Bhattacharyya, Archana Bura, Desik Rengarajan, Mason Rumuly, Bainan Xia, Srinivas Shakkottai, Dileep Kalathil, Ricky KP Mok, and Amogh Dhamdhere. QFlow: A learning approach to high qoe video streaming at the wireless edge. *IEEE/ACM Transactions on Networking*, 30(1):32–46, 2021.

[4] Leonardo Bonati, Salvatore D'Oro, Stefano Basagni, and Tommaso Melodia. SCOPE: An open and softwarized prototyping platform for NextG systems. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 415–426, 2021.

[5] Leonardo Bonati, Salvatore D'Oro, Michele Polese, Stefano Basagni, and Tommaso Melodia. Intelligence and learning in O-RAN for data-driven NextG cellular networks. *IEEE Communications Magazine*, 59(10):21–27, 2021.

[6] CurveZMQ. http://curvezmq.org/, 2023.

[7] Salvatore D'Oro, Michele Polese, Leonardo Bonati, Hai Cheng, and Tommaso Melodia. dapps: Distributed applications for real-time inference and control in o-ran. *IEEE Communications Magazine*, 60(11):52–58, 2022.

[8] Harish Kumar Dureppagari, Ujwal Dinesha, Raini Wu, Santosh Ganji, Woo-Hyun Ko, Srinivas Shakkottai, and Dinesh Bharadia. Realtime intelligent control for NextG cellular radio access networks. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 567–568, 2022.

[9] Xenofon Foukas, Navid Nikaein, Mohamed M. Kassem, Mahesh K. Marina, and Kimon Kontovasilis. FlexRAN: A flexible and programmable platform for software-defined radio access networks. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 427–441, New York, NY, USA, 2016. Association for Computing Machinery.

[10] Xenofon Foukas, Bozidar Radunovic, Matthew Balkwill, and Zhihua Lai. Taking 5g ran analytics and control to a new level. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2023.

[11] Yu-Pin Hsu, Navid Abedini, Natarajan Gautam, Alex Sprintson, and Srinivas Shakkottai. Opportunities for network coding: To wait or not to wait. *IEEE/ACM Transactions on Networking*, 23(6):1876–1889, 2014.

[12] Tianchi Huang, Rui-Xiao Zhang, Chao Zhou, and Lifeng Sun. QARC: video quality aware rate control for real-time video streaming based on deep reinforcement learning. In *Proceedings of the 26th ACM international conference on Multimedia*, pages 1208–1216, 2018.

[13] Ish Kumar Jain, Raghav Subbaraman, and Dinesh Bharadia. Two beams are better than one: Towards reliable and high throughput mmwave links. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 488–502, New York, NY, USA, 2021. Association for Computing Machinery.

[14] Ish Kumar Jain, Rohith Reddy Vennam, Raghav Subbaraman, and Dinesh Bharadia. mmFlexible: flexible directional frequency multiplexing for multi-user mmWave networks. *arXiv preprint arXiv:2301.10950*, 2023.

[15] Petteri Kela, Thomas Höhne, Teemu Veijalainen, and Hussein Abdulrahman. Reinforcement learning for delay sensitive uplink outer-loop link adaptation. In *2022 Joint European Conference on Networks and Communications 6G Summit (EuCNC/6G Summit)*, pages 59–64, 2022.

[16] Woo-Hyun Ko, Ushasi Ghosh, Ujwal Dinesha, Raini Wu, Srinivas Shakkottai, and Dinesh Bharadia. EdgeRIC: Delivering realtime RAN intelligence. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 1162–1164, 2023.

[17] Manikanta Kotaru, Kiran Joshi, Dinesh Bharadia, and Sachin Katti. SpotFi: Decimeter level localization using WiFi. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 269–282, New York, NY, USA, 2015. Association for Computing Machinery.

[18] Merima Kulin, Tarik Kazaz, Ingrid Moerman, and Eli De Poorter. End-to-end learning from spectrum data: A deep learning approach for wireless signal identification in spectrum monitoring applications. *IEEE Access*, 6:18484–18501, 2018.

[19] Panqanamala Ramana Kumar and Pravin Varaiya. *Stochastic systems: Estimation, identification, and adaptive control*, volume 75. SIAM, 2015.

[20] Jean Le Feuvre, Cyril Concolato, and Jean-Claude Moissinac. GPAC: open source multimedia framework. In *Proceedings of the 15th ACM International Conference on Multimedia*, MM '07, page 1009–1012, New York, NY, USA, 2007. Association for Computing Machinery.

[21] Changqing Luo, Jinlong Ji, Qianlong Wang, Xuhui Chen, and Pan Li. Channel state information prediction for 5g wireless communications: A deep learning approach. *IEEE Transactions on Network Science and Engineering*, 7(1):227–236, 2020.

[22] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210. ACM, 2017.

[23] Mehrtash Mehrabi, Mostafa Mohammadkarimi, Masoud Ardakani, and Yindi Jing. Decision directed channel estimation based on deep neural network k -step predictor for mimo communications in 5G. *IEEE Journal on Selected Areas in Communications*, 37(11):2443–2456, 2019.

[24] Tommaso Melodia, Stefano Basagni, Kaushik R. Chowdhury, Abhimanyu Gosain, Michele Polese, Pedram Johari, and Leonardo Bonati. Colosseum, the world's largest wireless network emulator. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, MobiCom '21, page 860–861, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Arupa Mohapatra, Natarajan Gautam, Srinivas Shakkottai, and Alex Sprintson. Network coding decisions for wireless transmissions with delay consideration. *IEEE Transactions on Communications*, 62(8):2965–2976, 2014.

[26] Mustafa Mohsin, Jordi Mongay Batalla, Evangelos Pallis, George Mastorakis, Evangelos K. Markakis, and Constandinos X. Mavromoustakis. On analyzing beamforming implementation in O-RAN 5G. *Electronics*, 2021.

[27] Kanthi Nagaraj, Dinesh Bharadia, Hongzi Mao, Sandeep Chinchali, Mohammad Alizadeh, and Sachin Katti. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 188–201, New York, NY, USA, 2016. Association for Computing Machinery.

[28] Khaled Nakhleh, Santosh Ganji, Ping-Chun Hsieh, I-Hong Hou, and Srinivas Shakkottai. NeurWIN: Neural Whittle index network for restless bandits via deep RL. In *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021.

[29] Ali ParandehGheibi, Muriel Médard, Asuman Ozdaglar, and Srinivas Shakkottai. Access-network association policies for media streaming in heterogeneous environments. In *49th IEEE Conference on Decision and Control (CDC)*, pages 960–965. IEEE, 2010.

[30] Jounsup Park, Philip A. Chou, and Jenq-Neng Hwang. Rate-utility optimized streaming of volumetric media for augmented reality. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(1):149–162, 2019.

[31] Ali Parsa, Neda Moghim, and Pouyan Salavati. Joint power allocation and mcs selection for energy-efficient link adaptation: A deep reinforcement learning approach. *Comput. Netw.*, 218(C), dec 2022.

[32] Michele Polese, Leonardo Bonati, Salvatore D'Oro, Stefano Basagni, and Tommaso Melodia. ColO-RAN: Developing Machine Learning-based xApps for Open RAN Closed-loop Control on Programmable Experimental Platforms. *IEEE Transactions on Mobile Computing*, pages 1–14, July 2022.

[33] Vivek Raghunathan, Vivek Borkar, Min Cao, and P Roshan Kumar. Index policies for real-time multicast scheduling for wireless broadcast systems. In *IEEE INFOCOM 2008-The 27th Conference on Computer Communications*, pages 1570–1578. IEEE, 2008.

[34] Javad Razavilar, KJ Ray Liu, and Steven I Marcus. Jointly optimized bit-rate/delay control policy for wireless packet networks with fading channels. *IEEE Transactions on Communications*, 50(3):484–494, 2002.

[35] Robert Schmidt, Mikel Irazabal, and Navid Nikaein. Flexric: An SDK for next-generation sd-rans. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '21, page 411–425, New York, NY, USA, 2021. Association for Computing Machinery.

[36] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[37] Stefania Sesia, Issam Toufik, and Matthew Baker. *LTE-the UMTS long term evolution: from theory to practice*. John Wiley & Sons, 2011.

[38] S. Shakkottai and R. Srikant. Network optimization and control. *Foundations and Trends in Networking*, 2(3):271–379, 2007.

[39] srsRAN, Inc. https://www.srslte.com/, 2023.

[40] Raghav Subbaraman, Yeswanth Guntupalli, Shruti Jain, Rohit Kumar, Krishna Chintalapudi, and Dinesh Bharadia. BSMA: Scalable LoRa networks using full duplex gateways. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, MobiCom '22, page 676–689, New York, NY, USA, 2022. Association for Computing Machinery.

[41] Leandros Tassiulas and Anthony Ephremides. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. In *29th IEEE Conference on Decision and Control*, pages 2130–2132. IEEE, 1990.

[42] Pratheek S. Upadhyaya, Aly S. Abdalla, Vuk Marojevic, Jeffrey H. Reed, and Vijay K. Shah. Prototyping next-generation O-RAN research testbeds with SDRs, 2022.

[43] Gongwei Xiao, Muhong Wu, Qian Shi, Zhi Zhou, and Xu Chen. DeepVR: deep reinforcement learning for predictive panoramic video streaming. *IEEE Transactions on Cognitive Communications and Networking*, 5(4):1167–1177, 2019.

[44] Lin Zhang, Junjie Tan, Ying-Chang Liang, Gang Feng, and Dusit Niyato. Deep reinforcement learning-based modulation and coding scheme selection in cognitive heterogeneous networks. *IEEE Transactions on Wireless Communications*, 18(6):3281–3294, 2019.

[45] Yuanxing Zhang, Pengyu Zhao, Kaigui Bian, Yunxin Liu, Lingyang Song, and Xiaoming Li. DRL360: 360-degree video streaming with deep reinforcement learning. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1252–1260. IEEE, 2019.

A Appendix

In the appendix, we provide additional results for different scenarios, which are not critical to the performance but provide visibility into the system.

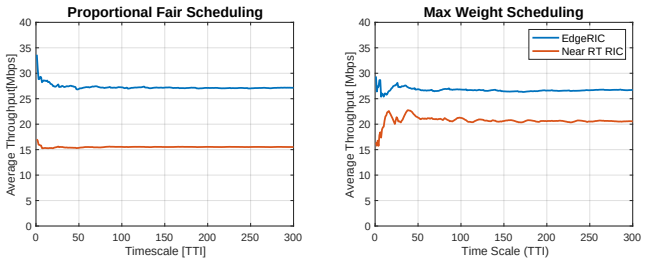


Figure 14: EdgeRIC Performance in a 4UE synthetic channel scenario.

Figure 14 shows that EdgeRIC outperforms Near Real-time RIC for implementation of microbenchmarks. This is the scenario with 4UEs on a synthetic channel trace. We implemented Proportional Fair Scheduling and Max Weight Scheduling as a μApp in EdgeRIC which exchanges state information and actions with RAN Stack over ZeroMQ-based communication. In order for performance comparison, we implemented Near RT RIC by imposing delays in message deliveries between RIC and RAN. We used synthetic CQI traces for 4 UEs and evaluated each algorithm's performance by comparing their average throughput. The graph on the left side shows average throughput of Proportional Fair Scheduling and the graph on the right side shows one of Max Weight Scheduling. A blue line in the graphs are for EdgeRIC while a red line for Near RT RIC. As Near RT RIC showed low average throughput due to the delay, EdgeRIC successfully supported those microbenchmarks to achieve their best throughput.

Table 7: Load: 30Mbps, Channel: 4UE synthetic channel

		EdgeRIC	15ms	30ms
Max CQI	Avg. Thrpt.	**26.3**	16.4	15.9
	BL[MB]	**1.22**	1.25	1.27
Prop. Fair.	Avg. Thrpt.	**24.27**	22.72	20.15
	BL[MB]	1.37	**1.08**	1.27
Max Weight	Avg. Thrpt.	**25.4**	19.8	19.1
	BL[MB]	1.33	**1.05**	0.99

Table 8: Throughput and Backlog Buffer evaluation of synthetic traces

	PPO	Max Weight	Max CQI
Synthetic Channel Traces			
Scenario 1	**27.8/0.38**	25.6/0.38	19.0/0.38
Scenario 2	**27.5/0.33**	24.7/0.51	18.9/0.38
Scenario 3	**26.8/0.38**	25.2/0.51	25.2/0.76

Figure 15 shows the real-time CQI traces we collected to characterize various CQI mobility in different environments. An x-axis is time in TTI unit and a y-axis is CQI showing real-time channel quality. The report period of CQI values was 2 TTI period which is approximately 2 mili-seconds. While a mobile robot has some drops in its CQI traces due to the radio block by the lid of the laptop mounted on the mobile robot when rotating, drone's swift motions caused radical ups and downs in its CQI traces. In car's CQI traces, its variation has a characteristic of a long period and smooth curves because of its slow acceleration and deceleration. For a turntable, we was able to generate fast angular acceleration which caused drastic and kind of periodic changes in its CQI traces by fast rotation.

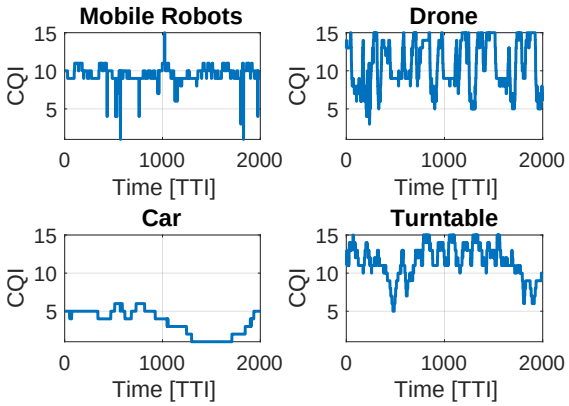


Figure 15: CQI Trace for different UEs

Figure 16 shows the training curves of RL PPO policy model and its throughput evaluations on DigitalTwin. The graphs in the first column illustrate the training curve for each scenario. We trained the policy network of RL for Scenario 1, 3 and 5 which are 2 Drone UEs, 2 Car UEs and 2 Drone UEs, and 2 Random Walk UEs and 2 Turntable UEs cases. We used specific CQI trace data to emulate channel conditions for individual scenarios. To train an RL agent using PPO, we designed a reward function to maximize throughput for general scenarios. Each iteration, we sampled 5000 data samples including previous state, previous action and current state at each iteration and updated policy network until the reward saturates. Most transient periods for training were less than 40 iterations and the training curves converged for all scenarios. The graphs in the second column describe the throughput evaluation of the trained policy model. In each graph, we compared the throughput of PPO to the ones of Max Weight and max CQI. In each algorithm, a red bar means the mean of its throughputs and a blue box their range. PPO obviously outperformed Max CQI, and even achieved almost the same throughput performance as Max Weight, which is usually considered as an optimum policy, with somehow higher ranges than Max Weight.

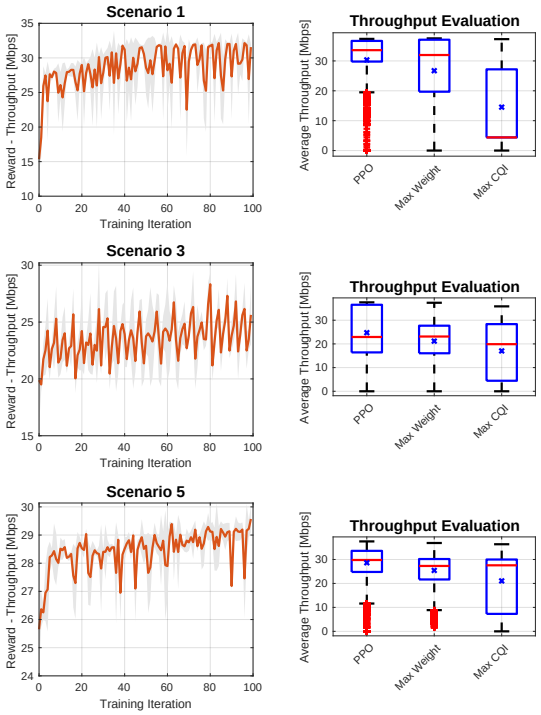


Figure 16: Can RL train and evaluate on EdgeRIC emulator?

ADR-X: ANN-Assisted Wireless Link Rate Adaptation for Compute-Constrained Embedded Gaming Devices

Hao Yin* *University of Washington*
Murali Ramanujam* *Princeton University*
Joe Schaefer *Microsoft*
Stan Adermann *Microsoft*
Srihari Narlanka *Microsoft*
Perry Lea *Microsoft*
Ravi Netravali *Princeton University*
Krishna Chintalapudi *Microsoft Research*

Abstract

The wireless channel between gaming console and accessories e.g. controllers and headsets, experiences extremely rapid variations due to abrupt head and hand movements amidst an exciting game. In the absence of prior studies on wireless packet losses for console gaming, through extensive evaluations and user studies, we find that state-of-the-art rate adaptation schemes, unable to keep up with these rapid changes, experience packet loss rates of 2-10% while loss rates that are 10× lower (0.1-0.5%) are required to ensure a high quality gaming experience. We present ADR-X, an ANN-based contextual multi-armed bandit rate adaptation technique that continuously predicts and tracks the channel and picks appropriate data rates. A key challenge for ADR-X is that it must run on power and compute constrained embedded devices under realtime constraints. ADR-X addresses this challenge by meticulously crafting an ANN that leverages existing communication theory results to incorporate domain knowledge. This allows ADR-X to achieve 10× lower packet losses than existing schemes while also running 100× faster than state-of-the-art reinforcement learning schemes, making it suitable for deployment on embedded gaming devices.

1 Introduction

Gaming is a $200 billion industry experiencing a rapid upward trajectory in growth. A typical gaming setup comprises a gaming console/PC located within 10 feet of the gamer. During gameplay, various gaming accessories such as controllers with joysticks and buttons [42, 65], headphones or specialized headsets for audio, VR, or AR [43, 66] connect to the console either via cables or wireless (Figure 1). The console continuously receives player inputs, such as button interactions and joystick maneuvers, and adjusts the game state in real-time. Subsequently, it generates and dispatches audio, video, and tactile feedback (like haptic responses) to the respective connected accessories. The accessories are typically power and compute constrained embedded devices e.g. Xbox controllers use 250-600MHz ARM based processors [1].

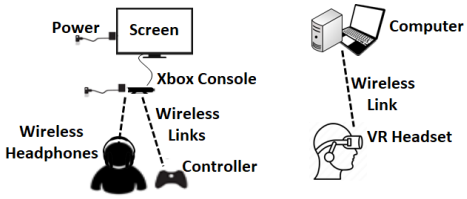


Figure 1: A typical gaming setup.

Despite wired headsets/headphones impeding free movement, ardent gamers today, prefer wired over wireless connectivity as Wi-Fi[1] packet losses deteriorate game audio/video quality. Prior user studies have determined that packet loss rates $< 0.5\%$ are necessary for a high quality gaming audio/video experience [58, 70]. However, these studies were conducted in the context of internet game streaming assuming independent (Bernoulli) packet losses over wired connections. To the best of our knowledge, no prior study examines the nature and effect of packet losses in a console gaming session over wireless links between the accessories and the console.

Amidst an exciting game (e.g. car racing or first person shooter games), the constant jerky hand (controller) and head (headset) movements of the gamer, rapidly affect the wireless channel resulting in up to 15*dB* signal strength variations within 50-100ms (Sec 3.2) – $5-10\times$ faster than usual Wi-Fi usage scenarios. We find that these rapid changes result in increased packet losses as the wireless Adaptive Data Rate (ADR) schemes are too slow to react. We also find that packet losses occur in succession, pairs and triplets, $\approx 50\%$ of the time (rather than independently as previous user studies assumed). *We conduct the first user study to quantify the effect of successive packet losses on gaming audio/video experience and find that (Sec. 3.3) consecutive losses are in fact more perceptible than random isolated losses; e.g. with 2-3 consecutive losses, gamers negatively perceive loss rates as low as*

* These authors contributed equally to this work.

[1] Although some existing gaming platforms adopt Bluetooth Low Energy (BLE) for connectivity, low throughput and high congestion in 2.4 GHz results in large latency, poor scalability.

0.1% for VR, a harsher requirement than that suggested by prior studies. **We have obtained an IRB for this study.**

While there exists a vast literature of wireless ADR schemes (Sec. 2.2), they have neither been tested nor designed for these severe gaming conditions. It is therefore no surprise that in our extensive evaluations (Table 1) of the state-of-the-art ADR techniques, as well as from traces collected from off-the-shelf gaming devices, we find $1-10\%$ packet loss during an active gaming session, falling well short of the desirable packet loss target.

Our key contribution is a novel wireless ADR technique, ADR-X, that achieves $10\times$ *lower packet losses (0.1-0.3%) than state-of-the-art during gaming conditions and an order of magnitude lesser consecutive packet losses by "closely" following and predicting the wireless channel dynamics.* ADR-X takes a hybrid approach – it leverages the well established analytical results from communication theory alongside an Artificial Neural Network (ANN) based contextual multi-armed bandit. It employs online learning to implicitly model, predict and adapt continuously to host radio hardware, background interference conditions and gamers' physical movement patterns that are hard/impossible to model analytically. As we demonstrate in our evaluations (Sec. 5.2), reinforcement learning approaches that do not leverage domain knowledge (e.g. PPO [60]) run $100\times$ slower than ADR-X and are unable to meet the real-time and low compute constraints imposed by the embedded gaming accessories.

Designing an ADR scheme that can predictively determine the ideal data rate under the fast changing gaming channel conditions poses two key challenges:

Wireless gaming channels are hard to predict analytically. Communication theory provides analytical functions that can accurately predict packet error rates over a wired medium [50]. Wireless channels however are far less amenable to modelling and prediction due to frequency selective fading that changes rapidly as the gamer changes hand and head orientations during gameplay and unpredictable background interference. As we discuss in Section 2.2 prior attempts [18] to analytically predict packet loss are extremely sensitive to small calibration and measurement errors, background interference, wireless channel asymmetry due to differences between transmitter and receiver radios, variations due to radio circuit components such as Automatic Gain Control (AGC), etc. ADR-X leverages the capability of ANNs to model and adapt to the intractable effects of the wireless channel and gamer's movements in a data-driven manner.

Computation, power and real-time Constraints. (Sec. 4.2) Gaming accessories e.g. controllers, headphones and headsets, are low power embedded devices with limited computation capabilities. For example, Xbox controllers uses 250MHz ARM processors [42] and employ sleep duty-cycling roughly 1 out of 8ms to save power. A naive strawman approach might be to employ ANN based online reinforcement learning to predict the appropriate data rate. However, as our evaluations show (Sec. 5.2), this approach proves to be extremely computationally demanding and unsuitable for embedded gaming devices. The key insight in the design of ADR-X is to rely on an ANN to model only the intractable effects while leveraging the predictive power of analytical models used in wired communication channels. This relieves the ANN from the burden of re-learning well-known communication theoretical results, making it more accurate and computationally economical.

ADR-X overview. ADR-X employs a contextual multi-armed bandit architecture (Sec. 4.1) that comprises three stages. The first stage is an ANN that transforms the wireless channel measurements (e.g. Signal to Noise Ratio (SNR), Channel State Information (CSI)) into an equivalent wired channel conducted SNR – CSNR, by implicitly and continuously learning to model the latent effects such as background interference, calibration errors, wireless channel asymmetry, hardware/firmware specifics and gamer movements. The second stage then leverages communication theory results to predict packet success rates for each of the possible data rates. In practice, to facilitate efficient gradient descent-based learning, we approximate the complex analytical communication theory models in the second stage using sigmoid functions (used commonly in training ANNs) with fixed pre-computed parameters. Finally, the third stage uses an $\varepsilon-$greedy sampling approach commonly used in multi-armed bandits to pick a suitable data rate. Further, as described in Section 4.2, ADR-X employs Wi-Fi domain specific feature engineering for the ANN to allow for a small ANN architecture.

Dealing with survivorship bias. When packets are lost, so are the associated channel measurements used to train the ANN. This leads to survivorship bias [63] i.e. the ANN learns only from successful examples leading to poor performance. To solve these problems ADR-X uses a specially crafted re-transmission strategy (Sec. 4.4) with stepped reductions in data rate for each subsequent re-transmission that serves as "training wheels" by providing second (or third) chances. When an ACK is received after one (or more) retries, ADR-X obtains negative examples corresponding to prior losses and channel state. As ADR-X learns, its reliance on these "training wheels" diminishes. Further, ADR-X uses imputation techniques [14] such as interpolation when packets are lost despite the re-transmissions.

Quick start using pretrained/federated models. Starting from a random initialization, when the player plays for the first time, ADR-X takes about 50s of gameplay to learn and converge (Sec. 5.3). We find that techniques such as (i) pre-training the ANN on IEEE channel simulations, and (ii) using federated model weights from other gamers or prior sessions reduce this to about 20s. Further, this time reduces to about 10s after a few gaming sessions. The stepped re-transmission strategy (described earlier) helps ADR-X in these initial few seconds to overcome high packet losses.

Summary of contributions.

- We provide the first user study and insights on the nature of packet losses for console-accessory wireless links and their effect on the gaming audio/video experience (we have obtained an IRB for the study). We show that existing wireless ADR schemes are unable to adapt "fast enough" in response to the high dynamism of gaming wireless channels resulting in successive losses rather than isolated incidents. To the best of our knowledge, our user study is the first to highlight that the traditionally accepted target loss rate of 0.5% for individual losses should be supplemented by 0.1% for successive losses.
- We propose ADR-X, an ANN assisted contextual multi-armed bandit based novel adaptive data rate scheme that is suited for power and compute constrained embedded gaming accessories. ADR-X predictively adjusts data rates based on the history of channel wireless measurements and packet losses.
- We evaluate ADR-X extensively across 20 diverse games*, including multiple genres such as First Person Shooter (FPS), racing, and action. In real-world experiments with practical hardware, under extensive comparisons against a bevy of state-of-the-art ADR baselines including several ML based approaches, ADR-X is the only scheme to achieve $0.1-0.3\%$ packet loss on hardware matching Xbox accessory clock speeds of 250-600MHz.

2 Background and Related Work

Signal to Noise Ratio (SNR) η, determines the data rate at which bits can be successfully sent over a communication channel. A data rate too high causes bits to be lost, and a data rate too low is wasteful in terms of time and energy. Since wireless channel conditions vary significantly over time; Adaptive Data Rate (ADR) techniques aim to adapt data rate to these changes by choosing the most appropriate data rate. The Modulation Coding Scheme (MCS) index, a number between 0 to 9, in Wi-Fi determines the data rate of transmission – the higher the MCS index, the higher the data rate. For each MCS index m, communication theory allows the bit error rate to be calculated analytically using a function $ber_m(\eta)$ [16].

2.1 Channel Measurements

In OFDM [49] modulation used by Wi-Fi, a 20MHz channel is split into 52 sub-carriers (sub-channels) in 802.11n and 242 sub-carriers in 802.11ax. Bits in a packet are spread out and transmitted over these sub-carriers.

CSI. A radio wave transmitted over the ith sub-carrier undergoes changes in amplitude and phase represented by a complex number c_i. Channel State Information (CSI) collects all these values into a vector $\mathbf{c} =< c_1, \cdots, c_C >$. While all Wi-Fi radios have to necessarily measure CSI for each received packet to decode, not all radio hardware/firmware provides access to CSI information through an API.

RSS. Almost all radios provide Received Signal Strength (RSS) measured in `dBm`, for each received packet – this measures the power level (strength) of a received signal.

SNR. Some radio APIs provide SNR, η (in `dB`) for each packet. SNR and RSS in `dB` are related to each other as [18]

$$\mathbf{SNR} = \mathbf{RSS} - \mathbf{NF} - \mathbf{AGC} \tag{1}$$

In Eqn. (1) NF is the noise floor of the radio and AGC is Automatic Gain Control of the radio which is a dynamic gain introduced by the radio circuit. While some radio APIs provide Noise Floor (in `dBm`) of the radio [22, 52], they do not provide AGC. In practice computing SNR using RSS and NF without accounting for AGC can lead to several `dB` of error.

SNR per subcarrier. As prior work [18] demonstrates, SNR is an extremely poor predictor for packet loss due to frequency selective fading, as each sub-carrier experiences a different SNR and hence experiences a different bit-error rate. The success or failure of a packet depends on the aggregate success of all the bits transmitted across all the sub-carriers. The SNR of the kth sub-carrier, η_k can be computed by scaling η by the kth component of unit vector of c,

$$\begin{aligned} \zeta_k &= \frac{C}{\sum_i \|c_i\|^2} \|c_k\| \\ \eta_k &= \eta - 10\log_{10}\zeta_k \end{aligned} \tag{2}$$

Effective SNR (ESNR). In channels undergoing frequency selective fading, to tackle the ineffectiveness of SNR in predicting packet rate, [18] introduces ESNR (η_{esnr}). It summarizes the per-subcarrier values $< \eta_1, \cdots, \eta_C >$ into a single value to represent an equivalent SNR value corresponding to a flat-fading channel (a channel without frequency selective fading) by treating the wireless channel to be composed of several narrow wired channels, one for each sub-carrier. Thus, for each MCS index m, [18] suggests computing η_{esnr} analytically by calculating the average bit rate across all the sub-carriers,

$$\eta_{esnr} = ber_d^{-1}\left(\sum_{i=1}^{i=C} ber_d(\eta_i)\right) \tag{3}$$

ESNR Computation is sensitive and error prone. Calibration errors and measurement errors of a few `dB` in CSI, RSS, noise floor, or SNR are common in all radios. Further, ber_m functions are exponential (based on the Q function) with very sharp transitions from 0 to 1 within 2-3 `dB`. This makes ESNR computation extremely sensitive to even small errors – 2-3 `dB` error can result over 10`dB` error in effective SNR. Consequently, subsequent work [13] uses an exponentially weighted mean of η_i, which is estimated during an initial calibration phase, rather than relying on Eqn. (3).

* Full list of games with media samples exhibiting the impact of packet losses is at https://muralisr.github.io/ADRX/.

2.2 Existing ADR Techniques

In this paper, we broadly classify ADR techniques into *reactive* and *proactive* ADR techniques. The former relies on recent packet loss and re-transmission statistics (e.g. using running average estimates). The latter makes use of channel measurements such as RSS, SNR, CSI etc. to make a timely choice based on the current state of the channel. Further, several recent ADR techniques, both reactive and proactive, employ ML to cope with vagaries of the wireless channel.

Reactive techniques. ARF [27], AARF [35], and CARA [31] gradually increase or decrease the rate based on the success or failure of consecutive transmission results. Several rate control algorithms are designed to optimize specific metrics instead of losses only *e.g.* average transmission time [5, 71], frame loss ratio [36, 48, 73, 74], bit error rate [64, 69] and throughput [9, 17, 47, 72, 75].

Reactive techniques Using ML. NeuRA [30] and MLRA [39] attempt to reduce sampling overhead by using a neural network model to predict the throughput of unsampled data rate. Thompson Sampling [34] uses a multi-armed bandit approach to improve the sampling of different data rates. As discussed in Section 3.2, reactive techniques in general are too slow and are unable to keep pace with the channel dynamism during gaming, leading to a poor gaming experience.

Proactive or channel measurement-based techniques. [33] uses RSS while [12, 21, 26, 38, 54, 56, 61, 68] use SNR measurements from received packets to dynamically adjust transmission rates. As discussed in Section 2.1 the performance of these schemes is limited by the fact that RSS and SNR do not take into account the effects of frequency selective fading. While ESNR-based techniques [18] account for frequency selective fading, as discussed in Section 2.1 their performance is limited due to the sensitivity of ESNR computation to hardware calibration errors and the requirement of offline calibration for each device.

Proactive schemes using ML. Recently, researchers have demonstrated the potential use of ML in proactive schemes through Simulation studies [28, 29, 39, 53]. To the best of our knowledge, EDRA [11] is the only existing implemented proactive ML-based ADR scheme. EDRA uses reinforcement learning, aiming to maximize throughput through joint rate and bandwidth adaptation by using Deep Q-Learning [19] with SNR, loss rates, and service times as inputs. However, EDRA imposes severe computational requirements – even a single inference takes 1.3-3.7ms on high-performance CPUs like i7-8700 and i5-6200U.

3 Packet Losses in Gaming

In this section, we analyze wireless packet losses during active gaming sessions. We show how off-the-shelf ADR schemes are unable to keep up with the highly dynamic gaming wireless channel and cause multiple consecutive packet losses. We then describe our user studies that show users are more sensitive to multiple consecutive losses than isolated ones.

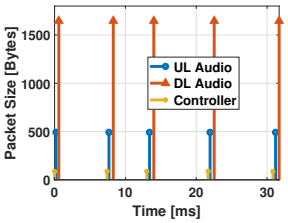


Figure 2: Xbox traffic.

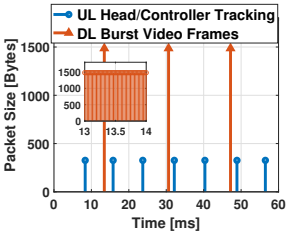


Figure 3: VR traffic.

3.1 Gaming Traffic Patterns

To understand packet flow in gaming traffic, we collect packet traces for Xbox [41] and Oculus [40] Quest 2 using a sniffer during multiple gaming sessions – we highlight the results from two games (out of the 20 games in our corpus) CrossFireX (Xbox) and Robo-Recall (Oculus Quest 2), but we note that trends persist across games. As seen in Figure 2, the Xbox console transmits one PCM game audio packet (1646 bytes), receives one chat audio packet (492 bytes) and one game input packet (88 bytes) every 8ms. In VR traffic (Figure 3), a burst of roughly 50 packets (total of about 500Kb) comprising game video and audio are transmitted once every 16.6ms (corresponding to 60Hz video frame refresh rate). A headset tracking packet (326 bytes) is transmitted every 8ms.

3.2 Packet Losses During Gaming

In this sub-section, our goal is to gain insights into how wireless packet losses occur during an active gaming session.

Figure 4: The setup for console gaming experiments.

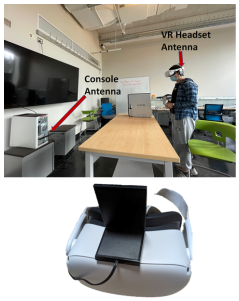


Figure 5: The setup for VR gaming experiments.

Measurement methodology. In order to emulate Xbox and Oculus Quest 2 radio firmware for controlled experiments, we collect and replay game traces between two PCs. We affix antennas on the console/desktop and the controller (since players connect to the headset via an audio jack from the controller) and VR headset (Figure 4, 5). We ask the gamers to play a game using the controller/headset with the affixed antenna. This allows the antenna to experience the same head/hand motions as the headset/controller. When actual gaming traffic is exchanged between the controller/headset and the console/desktop during a gaming session, we transmit gaming traffic traces in an interference-free DFS channel [45] between

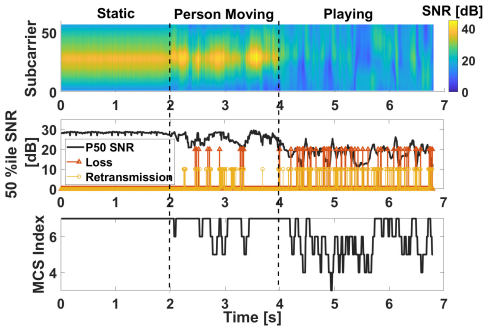


Figure 6: Channel, 50%ile SNR, re-transmissions, Losses and MCS for Xbox Traffic.

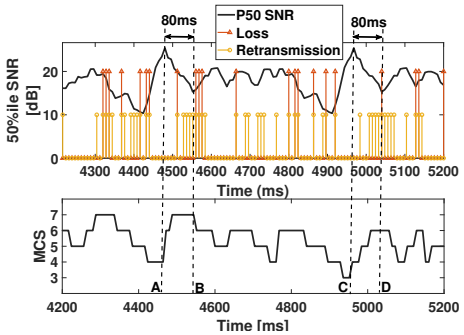


Figure 7: Rate adaptation is unable to keep up with rapid channel changes causing a cluster of losses.

the two PCs. We use PicoScenes toolbox [24, 25] to capture the transmitted packets and measure their CSI and SNR. We use these measurements to compute individual SNRs for each of the 52 Wi-Fi sub-carriers as described in Section 4.2. We consider three scenarios.

- *static* – the controller/headset is static, on a table about 5 feet from the transmitter in an empty room
- *people movement* – the controller/headset is static on a table 5 feet from the transmitter but with a person walking around in the room.
- *game play* – a gamer actively plays using the controller/ headset 5 feet from the transmitter (Figure 4, 5).

Observations. Figure 6 depicts the SNR heatmap for each of the 52 Wi-Fi sub-carriers spanning a 20MHz Wi-Fi channel as a function of time for each of the three scenarios. Figure 6 also depicts the instantaneous 50%ile SNR across all the sub-carriers. This is based on the intuition that a packet loss in Wi-Fi will occur when a "significant" fraction of sub-carriers experience fading (low SNR) so that error correction is unable to recover the correct bits. We also plot packet re-transmission events and loss events (when the re-transmission fails) and the MCS index (data rate) chosen by the native ADR scheme in an off-the-shelf device.

In the static scenario, the wireless channel is excellent with all the sub-carriers having an SNR of 30dB or higher with no losses or re-transmissions. In the people movement scenario, the channel sees variations with time due to changes in multi-path reflections, and occasional packet losses. When the gamer holds the controller or wears the headset and starts playing the game, the channel changes rapidly, leading to a large number of re-transmissions and packet losses. The ADR scheme shows large frequent variations in MCS Index.

Data rate adaptation is unable to keep up. Figure 7 shows a zoomed 1s section of the wireless channel for the playing scenario between 4.2 to 5.2 seconds. The 50%ile SNR across subcarriers shows a variation of over 15dB with a rapid decline of up to 10dB within 80ms in sections AB and CD. *The interesting observation in Figure 7 is that while the channel is degenerating, as seen by the decreasing trend in the 50%ile SNR in AB and CD, the ADR scheme actually increases data rates (MCS Index) and then starts reducing it when the channel SNR is improving – the exact opposite of the desired behavior.* This is because of the native ADR scheme, which is *reactive*, i.e. relies on recent packet re-transmission and loss statistics (e.g. running average) to adapt data rate. Before A, the channel 50%ile SNR improves by almost 15dB within a matter of 70ms. Based on the running average, after A, as the channel deteriorates, the data rate is increased causing a cluster of packet re-transmissions and losses. After B, even though the channel has an improving trend, data is decreased due to the history of failures.

Consecutive packet losses. As seen from Figure 7, due to ADR's reaction lag, packet losses occur in clusters and consecutive packets are lost when channel conditions change rapidly. Figure 8 shows the distribution of the number of consecutive losses for different distances between the controller and console. As seen from the figure, about 55-65% (100%-fraction of single packet losses) of all the packet losses occur with two or more packets lost consecutively. Our experiments with the VR headset (elided for space constraints) also show a similar cluster of losses.

3.3 Effect of Consecutive Packet Losses

Prior studies [7, 10, 57, 58, 70] in the context of internet game streaming have shown that Bernoulli (independently occurring) packet loss rates beyond 0.5% deteriorates audio/video quality for online gaming. However, to the best of our knowledge, there has not been a study quantifying the effect of **consecutive** packet losses on audio/video quality in a wireless console gaming setup. Thus, we conduct a study to measure the effect of consecutive packet losses on gaming audio and video.

Test data. For our study, we chose 30 Xbox game audio clips each 15s long, and 60 VR game video clips each 5s long as our original data set. These clips were drawn from the games in our corpus comprising 15 popular Xbox games and 5 popular VR games including racing, First Person Shooter (FPS), and other action games(full list of games with media samples exhibiting the impact of packet losses is at https://muralisr.github.io/ADRX/). Audio data is streamed as PCM audio packets (similar to Xbox) with

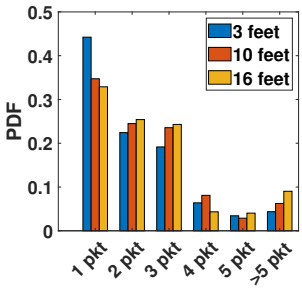


Figure 8: Consecutive packet losses due to slow rate adaptation.

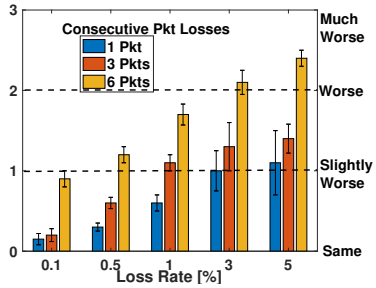


Figure 9: Effect of packet losses on game audio.

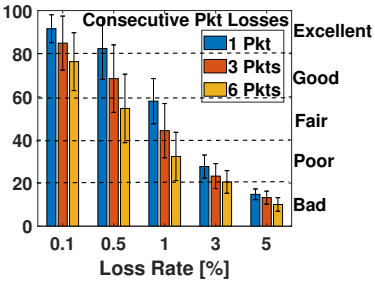


Figure 10: Effect of packet losses on game video.

8ms worth of data in each packet. Video data stream is encoded using H.264 video compression. Losses are injected for each stream randomly, parameterized by different loss rates $\rho = \{0.1\%, 0.5\%, 1\%, 5\%, 10\%\}$ and consecutive loss length $b = \{1, 3, 6\}$ packets. These parameter choices are informed by the loss patterns observed during our experimental evaluations (Sec. 5) of existing ADR schemes. This gives us 15 different combinations of $< \rho, b >$. Once a packet is randomly dropped with probability ρ, $b-1$ following packets are also dropped to capture consecutive losses during rate adaptation.

Measuring effect on game audio We conduct a user study adhering to the Comparative Mean Opinion Score (CMOS) test methodology as dictated by the ITU-T P.808 standard [44]. CMOS, as a subjective measure based on human perception, was chosen for its ability to accurately assess user-experienced audio quality degradation due to packet losses, particularly relevant for gamers. In contrast, PESQ [55], an objective algorithmic measure, lacks this direct user experience perspective, making CMOS more suitable for evaluating relative audio quality in our context. IRB approval from the author's organization was obtained prior to this study. Ten unique participants were recruited to listen and compare two audio clips – a lossy and an original clip, without knowing which clip was the original. On a scale from -3 (much worse) to +3 (much better), participants scored audio quality. During the study, care is taken to sanitize the results for faulty data points by eliminating incoherent outputs, and a calibration step is used to ensure proper audio setup on the participants' computers before the experiment (as per ITU-T P.808). Each combination of $< \rho, b >$ received 300 votes in all (30 clips $\times$10 people). As seen from Figure 9, our study reveals that listeners are more sensitive to consecutive packet losses – deterioration becomes perceptible at even 0.5% loss if $b \geq 6$.

Measuring effect on game video To quantify the effect of packet losses on VR video, we use the Video Multimethod Assessment Fusion (VMAF) [46]. VMAF is a popular full-reference objective video quality assessment model developed by Netflix that uses human-vision modeling. VMAF predicts a quality score that ranges from 0 to 100 for each video. Figure 10 shows that $\rho \leq 0.5$ and $b \leq 3$ are required to ensure the highest perception quality.

3.4 Conclusions

We summarize the observations in this section as,

- During gaming, the wireless channel varies by 10-15dB within a matter of 50-100ms.
- Rate adaptation is unable to keep up with these rapid changes resulting in consecutive packet losses.
- User studies indicate more than 2 consecutive packet losses causes "significant" visual artifacts even at the widely accepted standard of 0.5% packet loss rate. During gaming, consecutive packet losses become imperceptible only at the stricter threshold of 0.1%.

4 ADR-X

ADR-X continuously tracks the wireless channel and adapts data rates predictively using the recent history of channel measurement time series to avoid incurring losses. The fact that gaming traffic is periodic 8ms-16ms (Sec. 3.1) facilitates this approach since wireless channel measurements (e.g. CSI and SNR) can be measured each time a packet is received. ADR-X combines the strengths of ANNs to model the unmeasurable or hard-to-measure effects while leveraging the analytical simplicity of predicting packet loss over wired channels. This approach allows ADR-X to be both accurate and computationally efficient.

4.1 Overview of ADR-X

ADR-X takes a contextual multi-armed bandit approach – where a feature vector derived from the recent history of channel measurements serves as context (Sec. 4.2). ADR-X comprises three logical parts – i) *CSNR Mapper*, an ANN with learnable parameters that uses online gradient descent based learning [6] to transform the context into Conducted SNR (CSNR), ii) *PSR Calculator*, a layer pre-trained using standard communication theory results to predict the packet success rate given CSNR, and finally iii) *MCS Sampler*, an explore-exploit module using multi-armed bandit approaches to select the appropriate transmission rate to facilitate online training.

CSNR Mapper. CSNR in spirit is the same as ESNR (Sec. 3) – it maps the wireless channel measurements to a wired equivalent SNR. However, while ESNR is computed analytically,

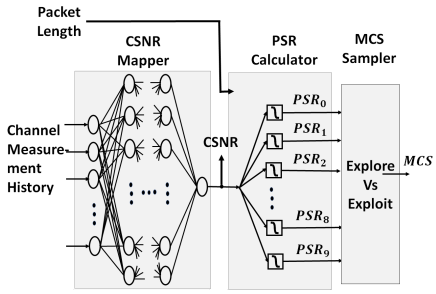


Figure 11: Architecture of ADR-X

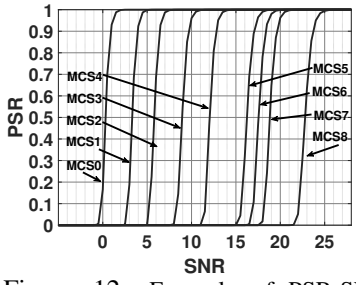


Figure 12: Example of PSR-SNR Curves for L=1536 over 802.11n.

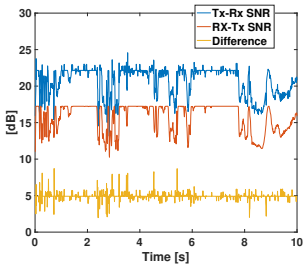


Figure 13: Channel asymmetry

treating each sub-carrier as an individual wire (Eqn. (3)), CSNR uses an ANN to learn this mapping on the fly in a data-driven manner. In this way, CSNR hopes to account for several practical hard to measure or un-measurable latent effects inherent in wireless communication – such as calibration and measurement errors, unknown AGC (Automatic Gain Control) or noise floor, background interference levels, wireless channel asymmetry, etc. as well as the nature of player's movements.

The CSNR mapper takes an input feature vector $\mathbf{f} =< f_1, f_2, .., f_n >$ extracted from the time series of channel measurements e.g. CSI, SNR, RSSI, etc. (as described in Section 4.2) and predicts the CSNR η_{csnr} in dB as,

$$\eta_{csnr} = F(\mathbf{f}, (\Theta)) \tag{4}$$

In Eqn. (4), Θ are learnable parameters – weights, biases etc.

PSR Calculator. Since CSNR represents the SNR for a flat fading channel. Communication theory provides analytical formulae $ber_m(\eta)$ to predict bit-error rate corresponding to MCS index m (data rate) and SNR η (Sec. 2). Packet Success Rate (PSR) may then be computed using $ber_m(\eta)$ and packet length L (Figure 12). These PSR functions however, tend to be complex and hard to compute analytically and typically fractional powers of $erfc(x)$ [2]. For enabling gradient descent based learning in ADR-X then, we would need to compute analytical expressions for the derivatives of these PSR functions, which are expensive to compute. Instead, ADR-X observes that the PSR-SNR curves qualitatively resemble a typical switching function [4] e.g. a Sigmoid function [8]. Thus, in ADR-X, the PSR Calculator approximates the PSR-SNR curves using the Sigmoid functions, a function commonly used in Neural Networks. The packet success rate is computed as,

$$PSR(m,L) = \frac{1}{1+e^{-(\alpha_{m,L}+\beta_{m,L}\eta_{csnr})}} \tag{5}$$

In Eqn. (5), $PSR(m,L)$ is the estimated PSR for the data rate corresponding to MCS index m, and the number of data bytes L. We first use Matlab IEEE flat fading model simulations to compute PSR for various values of SNR for different values of $< m, L >$. Then we estimate $\alpha_{m,L}$ and $\beta_{m,L}$ as a Least Mean Square (LMS) estimate to fit these values. Since there are too many possible values of L, in our implementation we compute three sets of PSR-SNR curves for broad L ranges – (0-64 bytes], (64-512 bytes] and (512-1536 bytes] by generating SNR, PSR values randomly sampling over the range of L. $\alpha_{m,L}$ and $\beta_{m,L}$ remain fixed during online training in ADR-X.

MCS Sampler. Once the PSR for each MCS Index is computed, we must choose the appropriate data rate. In ADR-X we define the optimal MCS value $MCS*$ as one that offers PSR higher than a certain threshold θ_{PSR},

$$m* = argmax_m PSR(m,L) > \theta_{PSR} \tag{6}$$

Since we aim to achieve $< 0.5\%$ PSR, we choose $\theta = 0.999$. Since the wireless channel changes with time, as described in Section 3.2, ADR-X must continuously explore to adapt and train correctly. Always sampling too conservatively will lead to ADR-X choosing lower MCS values resulting in increased power consumption and wastefully long transmission times. Thus, ADR-X must occasionally take risks and explore higher MCS rates. This problem is well known as the explore versus exploit problem in reinforcement learning literature and there exist several techniques [3, 20] to handle this tradeoff. In our implementation, we choose an ε-greedy approach as it works well in practice. $MCS*$ is chosen with the probability of ε (0.9 in our implementation) and a rate one step higher with a probability of $1-\varepsilon$.

4.2 Feature Engineering

Since ADR-X uses an ANN, it can potentially be generalized to use different kinds of channel measurements. Almost all radios provide RSS while some others provide CSI in addition. Thus, we generate two different flavors of features depending on the available channel measurements – ADR-X(CSI, RSS) and ADR-X(RSS). ADR-X(RSS) simply uses a time-series history of recently measured RSS value samples that are 8ms/16ms apart. ADR-X(CSI, RSS) uses the cumulative distribution function (CDF) of per-subcarrier SNR (η_i) described in Section 3. For computing SNR from RSS, we simply use NF as a constant -101db and let the ANN implicitly model the errors in the SNR.

For ADR-X(CSI, RSS), the time series history of per-subcarrier SNR values (Section 3) can lead to a large input to the ANN which can be either 52 (802.11n) or 242 (802.11ax) per time step. ADR-X however, must run on power and computation-constrained embedded devices such as 250-600MHz ARM processors (used in Xbox controllers) [1, 42].

To achieve this, we engineer the features to reduce the burden on the ANN itself in three steps.

1. Reducing frequency resolution. Instead of using all 52 or 242 values of per-subcarrier SNR, we compute the average SNR over spectrum bins each 2MHz wide (computing an average over the constituent subcarriers in each bin) to generate a vector $\mathbf{v} =< v_1, \cdots, v_9 >$ with nine values. This is based on the observation that magnitudes of CSI of sub-carriers that are "close" to each other in frequency are typically similar i.e. correlated.

2. Sorting SNR values. We sort the nine values of $\mathbf{v}$ in a decreasing order before presenting to the ANN - to create a CDF representation of the values - $\mathbf{v}^{sorted}$.

3. Including historical time series. Since the per-subcarrier SNRs show clear trends in 50-150ms time scales, we wish the ANN to implicitly extrapolate the current channel quality and compute PSR values. Thus, we provide the ANN with a time series history of $\mathbf{v}^{sorted}$ to allow it to make use of channel trends. In our implementation, we found that using channel information from 3 previous provides the most benefit. The feature vector at time t, $\mathbf{f}^t$ is computed as,

$$\mathbf{f}^t = \left[\mathbf{v}^{sorted}_{t-(H-1)\Delta}, \mathbf{v}^{sorted}_{t-(H-2)\Delta}, \cdots, \mathbf{v}^{sorted}_{t}\right] \tag{7}$$

Here, H is the length of the time series and the feature vector $\mathbf{f}$ comprises $9H$ numbers. Through experimentation, we chose $H = 3$ for our implementation. When ADR-X loses packets, it does not receive channel measurements. In this case, we use linear interpolation on the historical CSI values to fill in the gaps as imputation.

4.3 Online Training of ADR-X

ADR-X uses online training to allow it to adapt to changes in wireless channel conditions.

Training Data. Each time a packet is transmitted, the receipt of ACK or lack thereof indicates the success or failure of the transmission. Thus, after each transmission, we obtain the data $< L, m, r >$. Here L is the length of the packet, m is the MCS index used and r is a binary variable equal to 1 if the ACK was received or 0 if the packet was not received.

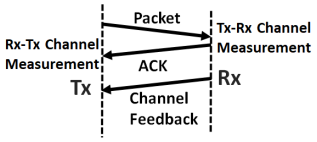


Figure 14: Channel measurements and feedback. ADR-X does not use feedbackbut relies on the reverse channel measurements.

Obtaining RSS, CSI and wireless channel asymmetry. CSI and SNR are obtained at the receiver (Rx) when a packet is received, however, ADR-X runs on the transmitter (Tx) (Figure 14). In principle, the receiver can explicitly provide a channel feedback message with CSI and SNR information (as employed in other proactive schemes). This approach constitutes an additional communication overhead and complexity of transmitting feedback. ADR-X, however, uses the channel measurements from the received packets (or ACKs) on the reverse path i.e. Rx-Tx instead of Tx-Rx. While in theory they are supposed to be the same, in practice, these measurements will be different (Figure 13), largely due to power differences but also due to AGC and hardware/firmware differences. As seen from Figure 13, there is a mean 5dB offset between the Tx-Rx and Rx-Tx channel SNRs of two devices along with a variation of up to $\pm$ 5dB. ADR-X implicitly learns to model the channel asymmetry in-situ by using online training methods to learn the PSR from real transmissions. This allows it to adapt to deployment scenarios with varying characteristics and generalize to different devices.

Loss function and weight updates in ADR-X. Suppose that the kth packet was transmitted using an MCS index m. Let $\delta_{i,k}$ depict a binary indicator variable, capturing whether or not the packet was successfully transmitted and MCS index i was used. Thus, $\delta_{i,k} = 1$ if $i = m$ and the packet's acknowledgment was received and 0 otherwise. Further, suppose that the PSR prediction of the ANN for this packet at MCS index m was P_{mk}. The ADR-X loss function is given by,

$$J_{loss} = \frac{1}{M}\sum_{k}\sum_{i}\delta_{ik}\left(P_{ik} - \delta_{ik}\right)^2 \tag{8}$$

In Eqn. (9) M is the total number of packets. Note the loss function is for the PSR is based on the Eqn. (5) with parameters from the channel model simulations, which will regulate the PSR from being too close to 1 such that the loss is minimized trivially. ADR-X trains the weights of the ANN using gradient decent-based back-propagation to minimize J_{loss} using online training. After each packet, it runs one back-propagation step using the MCS index, success/failure of the latest packet, and the current feature vector $\mathbf{f}$. We use a learning rate of 0.001 in our implementation.

Outline of proof for the validity of ADR-X's loss function. Let p_i be the true PSR for the ith MCS index and q_i be that estimated by ADR-X. As $M \to \infty$, i.e. for a large number of received packets, we have,

$$J_{loss} = p_i\left(1-q_i\right)^2 + \left(1-p_i\right)q_i^2 \tag{9}$$

$$\frac{\partial J_{loss}}{\partial q_i} = 2\left(q_i - p_i\right) \tag{10}$$

Thus, at extremum $q_i = p_i$ i.e. q_i converges to p_i. In other words ADR-X constantly updates the weights to predict the true PSRs.

4.4 Packet Re-transmission Strategy

Re-transmission is used in general to improve packet success rates by giving more chances for the packet to succeed. Packet losses in ADR-X cause two challenges – first, *survivorship bias* and second, *high pre-convergence losses*.

Survivorship bias. When a packet is lost so is the associated CSI and RSS information. This causes survivorship bias i.e.

ADR-X will be able to train only on positive examples using channel measurements and MCS rates from successful transmissions. This can undermine the learning process.

Pre-Convergence losses. In the initial stages of learning or when there are sudden changes in the wireless environment e.g. due to channel change, ADR-X might experience higher losses initially leading to poor audio/video experience.

Packet re-transmission strategy. To solve both these problems, ADR-X relies on a re-transmission strategy of using packet re-transmissions at stepped lower data rate. Upon packet loss, ADR-X re-transmits the packet immediately at three indices below the MCS index suggested by ADR-X for the original transmission. This choice provides insurance by a margin of 10dB prediction error. Optionally, we also use a second re-transmission at four MCS indices below the original transmission in case the first re-transmission is also lost providing a cover of 15dB error. This approach provides negative examples for transmissions that fail as well as positive examples when they succeed in re-transmissions. Further, these re-transmissions also serve as "training wheels" by providing second (or third) chances. As ADR-X learns, its reliance on these "training wheels" diminishes and it succeeds without re-transmissions as we demonstrate in our evaluations.

4.5 Federated Learning for Initialization

When the device is turned on the first time, ADR-X has to start learning from no prior experience. In practice, this means initializing the weights of the neural networks randomly. Starting from no experience, ADR-X may take about 20s - 2 minutes to converge depending on the interference levels in the channel (Section 5). It is however possible to reduce this initial convergence time using two strategies – first, pre-training ADR-X on simulations and second, by leveraging the experiences learned from other consoles using Federated Learning (FL) [32]. ADR-X employs both these strategies in order to reduce initial convergence time.

As we show in our evaluations (Sec. 5.3) using IEEE TGax Channel Model B [67] simulations ADR-X's convergence time reduces to 20-40s with this optimization. The second approach is based on federated learning (FL) algorithms that conduct a *weighted model aggregation* over all the users through weighted averaging of ANN network weights. If $\mathbf{W_m}$ are the weights of the mth device. To fuse the individual models, FL performs a weighted averaging operation:

$$\widetilde{\mathbf{W}} = \sum_{m=1}^{M} \Omega_m \mathbf{W}_m, \tag{11}$$

where Ω_m is the weight matrix of user m for each neuron in the neural networks. For a newly joined user (i.e., a user without any prior experience), we generate and apply a privacy-preserving model [32] with average experience from all users. Since games played on different consoles can be in different environments, the federated model represents a generic model for initialization. However, we note that safe mechanisms to collect data in a privacy-preserving manner may be required to deploy Federated Learning successfully in practice.

4.6 Implementation of ADR-X

We implement ADR-X at the application layer as a PicoScenes Plugin [23], on a desktop or portable laptop equipped with commercial off-the-shelf Qualcomm Atheros 9300 NIC Cards (802.11a/b/g/n) that provides CSI, RSS, NoiseFloor and SNR information for each received packet. We use the PicoScenes toolbox [24] APIs (on Ubuntu 20.04.3.) to collect the CSI measurements for each packet from the NIC. To customize the packet sizes and feedback, we use the packet injection mode provided by the PicoScenes Plugins [23]. PicoScenes APIs allow us to modify packet size, traffic patterns, and MCS directly from the toolbox in real-time. We implemented the ADR-X with about 1.5 K lines of C++ code for the whole ADR-X algorithm, including the gradient descent updates of the neural network.

5 Evaluation of ADR-X

In this section, we evaluate the performance of ADR-X and compare with seven representative state-of-the-art ADR schemes with respect to packet loss rates, consecutive packet loss rates, and average packet transmit times as well as their run times on embedded platforms. We also examine the convergence of ADR-X's learning and the benefit of pre-training through simulations and federated models and the impact of sudden changes in the wireless environment such as channel changes. In the end, we summarize the impact of feature engineering and architecture search for the ANN in ADR-X.

5.1 Experimental Setup

We use the identical setup described in Section 2.1 and implementation in Section 4.6 for conducting our experiments.

Multiple locations and gamers. We tested ADR-X's performance across 10 different gamers at different locations including two different apartments in different apartment complexes and university labs and conference rooms.

Diverse games and traces. We evaluate ADR-X across 20 diverse games, spanning multiple genres such as FPS, racing, and action (list of games with media samples is at https://muralisr.github.io/ADRX/). We use 15-minute (allowing each ADR algorithm sufficient time to converge to a steady state) gaming sessions on each game while recording network packet traces. Each resulting trace contains $> 10^5$ packets.

Embedded devices. Since Xbox devices use 250MHz-600MHz ARM processors, we used Raspberry PI 1 A+ with 250MHz BCM2835 chipset and Raspberry PI 2 600MHz [15] with BCM2836 chipset for evaluating run-times. These embedded devices are representative of the environment requiring link rate adaptation. Specifically, in Xbox, microphone audio transmission occurs from the controller to the console, which is a critical aspect of user experience. Similarly, in the VR scenario, not only is there a comparable audio uplink, but video streaming also occurs from the headset to the server.

Table 1: Summary of the Main Results.

Methods	Channel Info Used	Avg. Total Runtime [μs]	Channel Condition	Xbox					VR				
				Loss Rate [%]			Consective	Average	Loss Rate [%]			Consective	Average
				No Retry	1 Retry	Overall	Loss (>=3)[%]	Tx Time μs	No Retry	1 Retry	Overall	Loss (>=3)[%]	Tx Time [μs]
1. ARF [27] (reactive)	Pkt Loss	0.23	DFS	20.7	4.4	3.7	1.82	376	25.5	5	4.3	2.11	380
			5GHz	25.2	4.7	3.9	1.9	449	27.9	6.2	4.7	2.25	464
			2.4GHz	26.7	5.6	4.9	2.36	476	29.3	6.7	5.3	2.5	482
2. Minstrel [72] (reactive)	Pkt Loss	4.83	DFS	18.8	2.5	1.9	0.63	295	21.9	3.7	3.1	1.2	337
			5GHz	19.6	2.9	2.6	0.84	351	24.9	4	3.5	1.38	360
			2.4GHz	21.8	3.9	3.2	1.02	367	26.2	4.5	3.8	1.57	359
3. RAM [12] (proactive)	SNR	5.62	DFS	8.9	2.3	1.8	0.42	311	11.1	2.2	1.7	0.35	285
			5GHz	10.4	2.8	2.3	0.56	373	13	2.5	1.9	0.44	312
			2.4GHz	11.4	3.4	3	0.68	415	14	2.7	2.2	0.52	337
4. ESNR [18] (proactive)	CSI,SNR	31	DFS	6.5	2.5	2.2	0.4	295	10.5	3	2.4	0.5	280
			5GHz	8.4	3	2.6	0.46	344	12.4	3.2	2.7	0.53	291
			2.4GHz	10.1	3.3	2.7	0.54	395	13.6	3.6	3.1	0.66	312
5. TS [34] (reactive-ML)	Pkt Loss	67	DFS	6.1	2.4	1.9	0.45	281	7.3	2.4	2	0.58	259
			5GHz	7.8	2.7	2.3	0.52	306	9.7	2.6	2.2	0.65	284
			2.4GHz	8.9	3.5	2.9	0.66	325	10.3	3.3	2.6	0.78	303
6. EDRA [11] (proactive-ML)	RSS Pkt Loss	16645	DFS	4.5	2.4	1.6	0.39	264	4.85	2.5	1.4	0.53	238
			5GHz	5.9	2.7	1.8	0.48	279	6.3	2.7	1.5	0.65	270
			2.4GHz	8.1	3.2	2.1	0.62	295	8.4	2.9	2.2	0.73	289
7a. PPO(RSS) (proactive-ML)	CSI,RSS, Pkt Loss	17568	DFS	1.8	0.57	0.13	0.001	248	3.1	0.42	0.15	0.002	225
			5GHz	2.7	0.64	0.21	0.018	257	4.5	0.81	0.43	0.008	239
			2.4GHz	3.9	1.14	0.32	0.021	265	5.3	1.08	0.51	0.015	243
7b. PPO(CSI,RSS) (proactive-ML)	CSI,RSS, Pkt Loss	31081	DFS	0.82	0.23	0.06	0.001	233	0.91	0.15	0.04	0.004	218
			5GHz	1.1	0.31	0.13	0.004	247	1.3	0.26	0.11	0.012	224
			2.4GHz	1.9	0.57	0.21	0.008	266	1.8	0.41	0.17	0.021	239
ADR-X(RSS)	RSS, Pkt Loss	135	DFS	2.4	0.68	0.24	0.015	256	4.8	0.92	0.43	0.024	234
			5GHz	3.5	0.72	0.25	0.023	269	5.4	1.2	0.56	0.038	247
			2.4GHz	5.1	1.71	0.49	0.08	277	6.3	1.9	0.67	0.042	255
ADR-X(CSI, RSS)	CSI,RSS, Pkt Loss	381	DFS	1.2	0.33	0.13	0.003	245	1.5	0.27	0.14	0.007	223
			5GHz	1.4	0.39	0.15	0.01	253	1.7	0.34	0.15	0.014	231
			2.4GHz	2.3	0.79	0.25	0.024	274	2.6	0.56	0.26	0.036	252

This is particularly relevant for scenarios such as VR content sharing and live streaming, underscoring the importance of link adaptation in the embedded device side.

External interference scenarios. In a practical environment, background interference can cause packet losses. Packet losses due to interference occur independent of channel measurements making learning harder for ADR-X. We consider three different interference conditions by operating in three distinct bands – 2.4 GHz band (High Interference), 5 GHz non-DFS bands (Medium Interference), and the DFS bands (No Interference). To provide intuition, Figure 15 depicts the power level and channel width for each of the interfering Wi-Fi devices in the three different Wi-Fi bands within an apartment complex between 10:00-10:15 A.M. at one of the locations where the experiments were conducted. There were 161 active interfering Wi-Fi devices in the 2.4 GHz band, 85 in the 5 GHz band, and none in the DFS band.

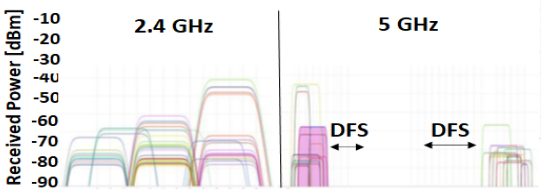


Figure 15: The Wi-Fi environment in 2.4 GHz (High Interference), 5 GHz (Medium Interference) and DFS channels (No Interference).

5.2 Performance of ADR-X

In this section, we show that ADR-X's offers 10× lower loss rate than state-of-the-art under gaming conditions while being computationally efficient. ADR-X is able to achieve this efficiency as it eschews a pure black-box ML approach in favor of synergizing an ANN with communication theoretical results as described in Sec 4.1. This allows ADR-X to sidestep computing analytical expressions for the PSR functions in favor of cheaper switching functions.

Given the large body of prior work in ADR, we choose six representative state-of-the-art ADR schemes from each of the four categories (Sec. 2.2) *reactive*, *proactive*, *reactive using ML* (reactive-ML) and *proactive using ML* (proactive-ML). Further, since Proximal Policy Optimization (PPO) is considered state-of-the-art in reinforcement learning, we implemented our own version of ADR based on PPO as described in Appendix B as an additional comparison point.

Reactive ADR schemes - 1. ARF [27], 2. Minstrel [72]. ARF, found in most off-the-shelf Wi-Fi devices due to its ease of implementation, maintains the packet loss rate as a moving average. The data rate is decreased by one MCS index after two consecutive packet re-transmissions/losses and increased after 9/10 successful transmissions. Minstrel is considered state-of-the-art among reactive ADR schemes and is typically implemented in the Linux driver as the default ADR algorithm. It dedicates 10% of its traffic to probing different rates to search for the maximum achievable throughput.

Proactive ADR schemes - 3. RAM [12], 4. ESNR [18]. Rate

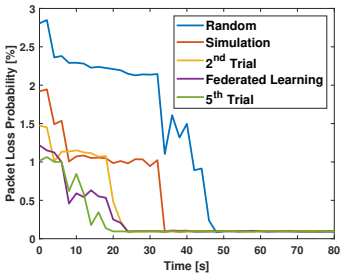


Figure 16: Example of ADR-X(CSI, RSS) convergence in 5GHz for Xbox traffic.

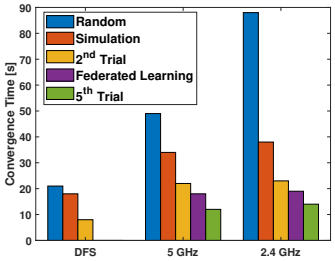


Figure 17: Loss Convergence of of ADR-X(CSI, RSS) for Xbox.

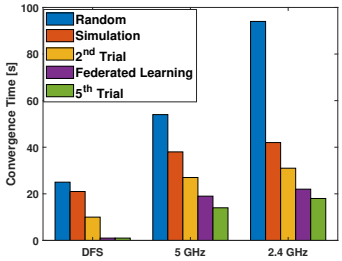


Figure 18: Loss Convergence of of ADR-X(CSI, RSS) for VR.

Adaptation in Mobile environments (RAM) represents the state-of-the-art in proactive schemes that use RSS for channel measurements. RAM maintains a throughput-vs-(rate, SNR) table. It uses measured SNR and the table to select the rate that can maximize throughput. ESNR, described in Section 2.1, represents the state-of-the-art among proactive schemes that use both CSI and RSS. It computes ESNR (Eqn. (3)) using the received packets and computes PSRs to pick the highest data rate that is predicted to have a <10% packet loss rate.

Reactive ADR with ML – 5. Thompson Sampling (TS) [34]. Thompson sampling based ADR is theoretically optimal [51] and considered the state-of-the-art in this category. It takes a multi-armed bandit [37] based on an efficient explore vs exploit approach to selecting the right data rate.

Proactive ADR with ML – 6. EDRA [11] EDRA uses Q-learning based Reinforcement Learning (RL) to select rates aiming to maximize throughput and is the state-of-the-art.

7. PPO based ADR Since, arguably Proximal Policy Optimization (PPO) is the state-of-the-art in RL, we created our own version of ADR that leverages PPO [60] (PPO-Clip) for two cases – i) when only RSS is available as a measurement and ii) both RSS and CSI are available. Due to space constraints, we describe the scheme in detail in the Appendix B. Table 1 summarizes our results averaged across all 10 gamers.

Packet loss rates. For Xbox and VR, and all three interference conditions, only ADR-X and PPO, achieve overall packet loss rates (after two MAC re-transmissions) of 0.1-0.25% with almost negligible consecutive losses – 10× lower than that for all other ADR schemes except ranges around 2-3% (Overall Loss Rate in Table 1).

Loss rates without re-transmissions. To answer the question, "how well does ADR-X pick the data rate compared to other schemes?" we examine the loss rates prior to packet re-transmissions. ADR-X and PPO rely on MAC re-transmissions only 1.2-2.3% of the time (No Retry Loss Rate in Table 1), ≈ 4× improvement over all other schemes that offer between between 6-20%. In fact even with 1 retry, ADR-X and PPO achieve < 0.5% loss rate while others offer 10× loss rates (1 Retry Loss Rate in Table 1).

Average transmission times. Choosing a rate too low will ensure packet transmission success but increase transmission time. Choosing a rate too high will result in packet losses and incur re-transmissions, once again increasing the overall packet transmission time. Thus, the ability of an ADR scheme to pick just the right rates can be measured by its ability to minimize overall packet transmission time. Packet transmission time is an important metric as it is commonly used as a proxy for energy consumption [62]. ADR schemes that minimize the transmission time also reduce the energy consumption by reducing the duration of time the radio has to be awake for transmission. In Table 1 we compute the average time to transmit a packet. The time to transmit a given packet is the sum of transmission times of all its re-transmissions and the original transmission. As seen from Table 1, ADR-X and PPO offer about ≈40% reduction compared to ARF and ≈25-30% compared to Minstrel and other schemes. An interesting observation is that even ADR-X(RSS) with only RSS, has a $20-25\%$ lower transmission times than all existing schemes.

Average run-time. Table 1 highlights the average run-time per data rate decision by each of the algorithms on computationally constrained embedded chip-sets. The run-time measurements for 250MHz ARM processor are provided in Table 1, and those for the 600MHz processor are provided in Table 2. ADR-X runs ≈ 100× faster than PPO. *It is the only scheme that satisfies the 8-16ms time budget (Sec. 3.1) with significant time left over for duty-cycling to save power.*

Effective Quality of Experience (QoE) hinges on balancing packet losses, transmission time, and power consumption. While a lower MCS reduces packet loss, it may increase transmission time and power usage. High packet loss, especially in Wi-Fi, necessitates retransmissions, further raising power consumption. The primary goal is to finely tune these parameters to minimize packet losses and reduce transmission time, thereby enhancing user experience.

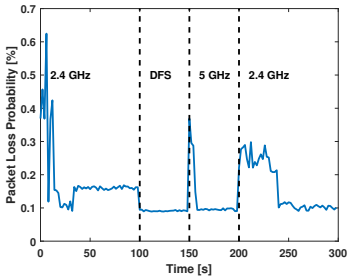


Figure 19: Effect of sudden changes in interference levels.

5.3 Convergence of ADR-X

In this section we ask the question, *"How quickly does ADR-X learn during a gaming session?"*. We define convergence

Table 2: Network Architecture Exploration.

Feature Engineering	Architecture (In-Hidden-Out)	Historical datapoints	Total parameters	Loss (%)	Convergence below 0.5%	Runtime (250 MHz) [μs]			Runtime (600 MHz) [μs]		
						Training	Inference	Total	Training	Inference	Total
Raw (Unsorted)	242-20-1	1	4881	1.31	N/A	4343	3519	7862	1752	1421	3173
	726-20-1	3	14561	0.92	N/A	12511	11515	24026	5348	4928	10276
	726-5-20-1	3	3776	0.13	44s	3496	2708	6204	1403	1107	2510
Sorted	242-20-1	1	4881	1.14	N/A	4250	3408	7658	1761	1426	3187
	726-20-1	3	14561	0.87	N/A	12958	12116	25074	5358	4935	10293
	726-5-20-1	3	3776	0.13	42s	3527	2704	6230	1411	1114	2525
Freq. Binning (Unorted)	27-20-1	3	581	1.34	N/A	478	325	803	194	132	326
	27-5-20-1	3	281	0.76	N/A	206	156	362	84	63	147
Freq. Binning (Sorted)	9-5-1	1	56	0.73	N/A	75	47	122	28	16	44
	9-20-1	1	221	0.65	N/A	182	135	317	73	53	126
	27-5-1	3	146	0.39	89s	134	104	238	55	41	96
	27-20-1	3	581	0.31	96s	461	334	794	196	138	334
	27-5-15-1	3	246	0.21	57s	183	164	347	72	63	135
	27-5-20-1	**3**	**281**	**0.15**	**49s**	**221**	**159**	**381**	**86**	**65**	**151**
	45-5-20-1	5	371	0.17	71s	303	239	542	125	94	219

as the time taken for ADR-X to reach below 0.5% overall loss rate and 0.1% for consecutive losses. We consider four different ANN weight initializations, i) *Random*- randomly initialized ANN weights; ii) Simulation based pre-training - ANN weights obtained by training on simulated CSI, RSS and packet losses using the IEEE TGax Channel Model B [67]; iii) *Federated model* - a federated model (using [32]) based on learned weights obtained from 9 different users across different locations; iv) *Successive trials* - the gamer plays multiple games, each 15 minutes long with a break of 10 minutes in between. After each successive gaming session, weights are passed on to the next session.

To provide an intuition into ADR-X's convergence during a gaming session, Figure 16 depicts the packet loss rates of ADR-X(CSI,RSS) in the 5GHz channel for Xbox traffic. As seen from Figure 16, while random initialization takes about 50s to achieve convergence, simulation-based pre-trained model and federated models take about 35s and 20s respectively. Finally, ADR-X convergence improves to 10s over 5 consecutive gaming sessions.

Figures 17, 18 capture the average convergence times for Xbox and VR over all user experiments under various interference conditions. Interference-free DFS channels are the fastest to converge. The higher the external interference, the longer ADR-X takes to converge. Federated models take about 20s to converge in highly congested channels.

5.4 Sudden Changes in Interference Levels

In this section, we ask the question "How does ADR-X adapt upon experiencing sudden changes in the ambient interference?". This can occur when the radio changes its operating channel into say a highly congested channel. We conduct an experiment starting in a 5GHz channel on a federated model (Sec. 5.3) and then change the operating channel between DFS, 5GHz, and 2.4 GHz as shown in Figure 19. Figures 19 that depicts packet loss as a function of time – each time the channel changes to higher congestion levels ADR-X adapts, however loss rates are below 0.5% at all times.

5.5 Benefits of Feature Engineering

In this section, we answer the question, "How crucial are the various steps of feature engineering to ADR-X?" The feature engineering 4.2 comprises threes aspects – i) use of historical time series of CSI, and RSS values ii) reduction the resolution of the subcarriers through frequency binning and iii) sorting the SNR per subcarriers. Table 2, summarizes performance results for ADR-X with and without different kinds of feature engineering for different ANN architectures. In all cases the including 3 historical channel measurement values brings a significant improvement in ADR-X performance. Frequency binning brings about $10-20\times$ reduction in runtimes over both 250MHz and 600MHz ARM processors. With frequency binning, sorting the SNRs makes a significant improvement in performance – a reduction in loss rates from 0.7% to 0.15%.

5.6 Architecture exploration

We use a Multilayer Perceptron architecture for our ANN. Table 2 presents a few representative examples from our extensive architecture sweep for the ANN architecture – our choice indicated in gray has a loss rate of 0.15% and compute time of 151μs on a 600MHz ARM processor.

6 Conclusions

We perform the first study into wireless losses and their effects on console gaming audio/video experience. Due to gamer hand/head motion during gaming, the wireless channel between a console and its accessories experiences rapid changes. Existing ADR schemes' inability to cope to fast channel changes results in 2-10% losses whereas our study indicates that 0.1-0.5% losses are required for a high quality experience. This paper introduces ADR-X, a contextual ANN assisted multi-armed bandit that closely tracks the channel and adapts quickly. Its novel design exploits communication theory domain knowledge to make it computationally efficient. Overall, ADR-X achieves $10\times$ lower packet loss than the ADR schemes widely used today while running $100\times$ faster than the state-of-the-art ML approaches.

References

[1] Xbox controller teardown. https://www.techinsights.com/blog/xbox-one-teardown.

[2] Milton Abramowitz and Irene A Stegun. *Handbook of mathematical functions with formulas, graphs, and mathematical tables*, volume 55. US Government printing office, 1968.

[3] Deepak Agarwal and Bee-Chung Chen. *Statistical Methods for Recommender Systems*. Cambridge University Press, 2016.

[4] Issa Batarseh and S. B. Dewan. *The Switching Function: analysis of power electronic circuits*. The Institution of Engineering and Technology, 2006.

[5] John Charles Bicket. *Bit-rate selection in wireless networks*. PhD thesis, Massachusetts Institute of Technology, 2005.

[6] Michael Biehl and Holm Schwarze. Learning by on-line gradient descent. *Journal of Physics A: Mathematical and general*, 28(3):643, 1995.

[7] Gulnaziye Bingol, Luigi Serreli, Simone Porcu, Alessandro Floris, and Luigi Atzori. The impact of network impairments on the qoe of webrtc applications: A subjective study. In *2022 14th International Conference on Quality of Multimedia Experience (QoMEX)*, pages 1–6. IEEE, 2022.

[8] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[9] Seongho Byeon, Kangjin Yoon, Changmok Yang, and Sunghyun Choi. Strale: Mobility-aware phy rate and frame aggregation length adaptation in wlans. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017.

[10] Kuan-Ta Chen, Chi-Jui Chang, Chen-Chi Wu, Yu-Chun Chang, and Chin-Laung Lei. Quadrant of euphoria: a crowdsourcing platform for qoe assessment. *IEEE Network*, 24(2):28–35, 2010.

[11] Syuan-Cheng Chen, Chi-Yu Li, and Chui-Hao Chiu. An experience driven design for ieee 802.11ac rate adaptation based on reinforcement learning. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, 2021.

[12] Xi Chen, Prateek Gangwal, and Daji Qiao. Ram: Rate adaptation in mobile environments. *IEEE Transactions on Mobile Computing*, 11(3):464–477, 2012.

[13] Riccardo Crepaldi, Jeongkeun Lee, Raul Etkin, Sung-Ju Lee, and Robin Kravets. Csi-sf: Estimating wireless channel state using csi sampling & fusion. In *2012 Proceedings IEEE INFOCOM*, pages 154–162, 2012.

[14] A Rogier T Donders, Geert JMG Van Der Heijden, Theo Stijnen, and Karel GM Moons. A gentle introduction to imputation of missing values. *Journal of clinical epidemiology*, 59(10):1087–1091, 2006.

[15] Raspberry Pi Foundation. Raspberry pi documentation - computers, 2023. Accessed: [insert date you accessed the site].

[16] Andrea Goldsmith. *Wireless Communications*. Cambridge University Press, 2005.

[17] Rémy Grünblatt, Isabelle Guérin-Lassous, and Olivier Simonin. Simulation and performance evaluation of the intel rate adaptation algorithm. In *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, MSWIM '19, page 27–34, New York, NY, USA, 2019. Association for Computing Machinery.

[18] Daniel Halperin, Wenjun Hu, Anmol Sheth, and David Wetherall. Predictable 802.11 packet delivery from wireless channel measurements. SIGCOMM '10, page 159–170, New York, NY, USA, 2010. Association for Computing Machinery.

[19] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, page 2094–2100. AAAI Press, 2016.

[20] Mikael Henaff. Explicit explore-exploit algorithms in continuous state spaces. *Advances in Neural Information Processing Systems*, 32, 2019.

[21] Gavin Holland, Nitin Vaidya, and Paramvir Bahl. A rate-adaptive mac protocol for multi-hop wireless networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, MobiCom '01, page 236–251, New York, NY, USA, 2001. Association for Computing Machinery.

[22] Intel. Intel ultimate n wifi link 5300, 2009.

[23] Zhiping Jiang and contributors. Developing Your PicoScenes Plugins. https://ps.zpj.io/plugin.html, 2021.

[24] Zhiping Jiang and contributors. PicoScenes. https://ps.zpj.io/, 2021.

[25] Zhiping Jiang, Tom H. Luan, Xincheng Ren, Dongtao Lv, Han Hao, Jing Wang, Kun Zhao, Wei Xi, Yueshen Xu, and Rui Li. Eliminating the barriers: Demystifying wi-fi baseband design and introducing the picoscenes wi-fi sensing platform. *IEEE Internet of Things Journal*, 9(6):4476–4496, 2022.

[26] Glenn Judd, Xiaohui Wang, and Peter Steenkiste. Efficient channel-aware rate adaptation in dynamic environments. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, page 118–131, New York, NY, USA, 2008. Association for Computing Machinery.

[27] Ad Kamerman and Leo Monteban. Wavelan®-ii: A high-performance wireless lan for the unlicensed band. *Bell Labs Technical Journal*, 2(3):118–133, 1997.

[28] Raja Karmakar, Samiran Chattopadhyay, and Sandip Chakraborty. Smartla: Reinforcement learning-based link adaptation for high throughput wireless access networks. *Computer Communications*, 110:1–25, 2017.

[29] Raja Karmakar, Samiran Chattopadhyay, and Sandip Chakraborty. An online learning approach for auto link-configuration in ieee 802.11ac wireless networks. *Computer Networks*, 181:107426, 2020.

[30] Shervin Khastoo, Tim Brecht, and Ali Abedi. Neura: Using neural networks to improve wifi rate adaptation. In *Proceedings of the 23rd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, MSWiM '20, page 161–170, New York, NY, USA, 2020. Association for Computing Machinery.

[31] J. Kim, S. Kim, S. Choi, and D. Qiao. Cara: Collision-aware rate adaptation for ieee 802.11 wlans. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–11, 2006.

[32] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[33] Lito Kriara and Mahesh K. Marina. Samplelite: A hybrid approach to 802.11n link adaptation. *SIGCOMM Comput. Commun. Rev.*, 45(2):4–13, apr 2015.

[34] Alexander Krotov, Anton Kiryanov, and Evgeny Khorov. Rate control with spatial reuse for wi-fi 6 dense deployments. *IEEE Access*, 8:168898–168909, 2020.

[35] Mathieu Lacage, Mohammad Hossein Manshaei, and Thierry Turletti. Ieee 802.11 rate adaptation: A practical approach. In *Proceedings of the 7th ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, MSWiM '04, page 126–134, New York, NY, USA, 2004. Association for Computing Machinery.

[36] Mathieu Lacage, Mohammad Hossein Manshaei, and Thierry Turletti. Ieee 802.11 rate adaptation: A practical approach. In *Proceedings of the 7th ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, MSWiM '04, page 126–134, New York, NY, USA, 2004. Association for Computing Machinery.

[37] T.L Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6(1):4–22, 1985.

[38] Okhwan Lee, Jihoon Kim, Jongtae Lim, and Sunghyun Choi. Sira: Snr-aware intra-frame rate adaptation. *IEEE Communications Letters*, 19(1):90–93, 2015.

[39] Chi-Yu Li, Syuan-Cheng Chen, Chien-Ting Kuo, and Chui-Hao Chiu. Practical machine learning-based rate adaptation solution for wi-fi nics: Ieee 802.11ac as a case study. *IEEE Transactions on Vehicular Technology*, 69(9):10264–10277, 2020.

[40] Meta. Quest 2. https://www.oculus.com/quest-2/, 2020.

[41] Microsoft. Xbox Series X. https://www.xbox.com/en-US/consoles/xbox-series-x, 2020.

[42] Microsoft. Xbox Wireless Controller. https://www.xbox.com/en-US/accessories/controllers/xbox-wireless-controller, 2020.

[43] Microsoft. Xbox Wireless Headset. https://www.xbox.com/en-US/accessories/headsets/xbox-wireless-headset, 2020.

[44] Babak Naderi and Ross Cutler. An open source implementation of itu-t recommendation p.808 with validation. In *Proc. Interspeech 2020*, pages 1166–1170, 2020.

[45] National Telecommunications and Information Administration. Agreement reached regarding u.s. position, 2003.

[46] Netflix. Vmaf - video multi-method assessment fusion. https://github.com/Netflix/vmaf, 2021.

[47] Ioannis Pefkianakis, Yun Hu, Starsky H.Y. Wong, Hao Yang, and Songwu Lu. Mimo rate adaptation in 802.11n

wireless networks. In *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking*, MobiCom '10, page 257–268, New York, NY, USA, 2010. Association for Computing Machinery.

[48] Ioannis Pefkianakis, Starsky H.Y. Wong, Hao Yang, Suk-Bok Lee, and Songwu Lu. Toward history-aware robust 802.11 rate adaptation. *IEEE Transactions on Mobile Computing*, 12(3):502–515, 2013.

[49] Ramjee Prasad. *OFDM for Wireless Communications Systems*. Artech House, 2004.

[50] John G Proakis. *Digital communications*. McGraw-Hill, Higher Education, 2008.

[51] Hang Qi, Zhiqun Hu, Xiangming Wen, and Zhaoming Lu. Rate adaptation with thompson sampling in 802.11ac wlan. *IEEE Communications Letters*, 23(10):1888–1892, 2019.

[52] Qualcomm. Qualcomm csrc9300 bluetooth & wi-fi combo chipset, 2014.

[53] Ruben Queiros, Eduardo Nuno Almeida, Helder Fontes, Jose Ruela, and Rui Campos. Wi-fi rate adaptation using a simple deep reinforcement learning approach, 2022.

[54] Hariharan Rahul, Farinaz Edalat, Dina Katabi, and Charles G. Sodini. Frequency-aware rate adaptation and mac protocols. In *Proceedings of the 15th Annual International Conference on Mobile Computing and Networking*, MobiCom '09, page 193–204, New York, NY, USA, 2009. Association for Computing Machinery.

[55] Antony W Rix, John G Beerends, Michael P Hollier, and Andries P Hekstra. Perceptual evaluation of speech quality (pesq)-a new method for speech quality assessment of telephone networks and codecs. In *2001 IEEE international conference on acoustics, speech, and signal processing. Proceedings (Cat. No. 01CH37221)*, volume 2, pages 749–752. IEEE, 2001.

[56] B. Sadeghi, V. Kanodia, A. Sabharwal, and E. Knightly. Opportunistic media access for multirate ad hoc networks. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, MobiCom '02, page 24–35, New York, NY, USA, 2002. Association for Computing Machinery.

[57] Steven Schmidt, Babak Naderi, Saeed Shafiee Sabet, Saman Zadtootaghaj, and Sebastian Möller. Assessing interactive gaming quality of experience using a crowdsourcing approach. In *2020 Twelfth International Conference on Quality of Multimedia Experience (QoMEX)*, pages 1–6. IEEE, 2020.

[58] Steven Schmidt, Saman Zadtootaghaj, Shijie Wang, and Sebastian Möller. Towards the influence of audio quality on gaming quality of experience. In *2021 13th International Conference on Quality of Multimedia Experience (QoMEX)*, pages 169–174. IEEE, 2021.

[59] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[60] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[61] Sayandeep Sen, Neel Kamal Madabhushi, and Suman Banerjee. Scalable WiFi media delivery through adaptive broadcasts. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*, San Jose, CA, April 2010. USENIX Association.

[62] Pablo Serrano, Andres Garcia-Saavedra, Giuseppe Bianchi, Albert Banchs, and Arturo Azcorra. Per-frame energy consumption in 802.11 devices and its implication on modeling and design. *IEEE/ACM Transactions on Networking*, 23(4):1243–1256, 2015.

[63] Gary Smith. *Standard deviations: Flawed assumptions, tortured data, and other ways to lie with statistics*. Abrams, 2014.

[64] Lixing Song and Shaoen Wu. Aarc: Cross-layer wireless rate control driven by fine-grained channel assessment. In *2015 IEEE International Conference on Communications (ICC)*, pages 3311–3316, 2015.

[65] Sony. DualSense Wireless Controller. https://www.playstation.com/en-us/accessories/dualsense-wireless-controller/, 2020.

[66] Sony. PULSE 3D Wireless Headset. https://www.playstation.com/en-us/accessories/pulse-3d-wireless-headset/, 2020.

[67] TGax. Tgax channel model document. Technical report, IEEE 802.11, 2014.

[68] Xiaozheng Tie, Anand Seetharam, Arun Venkataramani, Deepak Ganesan, and Dennis L. Goeckel. Anticipatory wireless bitrate control for blocks. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, New York, NY, USA, 2011. Association for Computing Machinery.

[69] Mythili Vutukuru, Hari Balakrishnan, and Kyle Jamieson. Cross-layer wireless bit rate adaptation. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, page 3–14,

New York, NY, USA, 2009. Association for Computing Machinery.

[70] Abdul Wahab, Nafi Ahmad, and John Schormans. Variation in qoe of passive gaming video streaming for different packet loss ratios. In *2020 Twelfth International Conference on Quality of Multimedia Experience (QoMEX)*, pages 1–4. IEEE, 2020.

[71] Shao-Cheng Wang and Ahmed Helmy. Beware: Background traffic-aware rate adaptation for ieee 802.11. In *2008 International Symposium on a World of Wireless, Mobile and Multimedia Networks*, pages 1–12, 2008.

[72] Linux Wireless. Minstrel rate control algorithm. https://wireless.wiki.kernel.org/en/developers/documentation/mac80211/ratecontrol/minstrel, 2015.

[73] Linux Wireless. PID. http://linuxwireless.sipsolutions.net/en/developers/Documentation/mac80211/RateControl/PID/, 2015.

[74] Starsky H. Y. Wong, Hao Yang, Songwu Lu, and Vaduvur Bharghavan. Robust rate adaptation for 802.11 wireless networks. In *Proceedings of the 12th Annual International Conference on Mobile Computing and Networking*, MobiCom '06, page 146–157, New York, NY, USA, 2006. Association for Computing Machinery.

[75] Wei Yin, Peizhao Hu, and Jadwiga Indulska. Rate control in the mac80211 framework: Overview, evaluation and improvements. *Computer Networks*, 81:289–307, 2015.

Appendices

A Deep Reinforcement Learning Design

A General Overview of DRL. Deep reinforcement learning (DRL) is an advanced machine learning approach that combines deep learning and reinforcement learning (RL) techniques. It leverages neural networks to interpret high-dimensional inputs, making it adept at handling complex problems. In the DRL framework, at each time instance t, the RL algorithm necessitates determining the optimal transmission rate for a packet. This decision is based on the input state vector $S(t)$, which incorporates the experience vector harvested from the experience memory. The RL algorithm then prescribes an action state $a(t)$, encompassing the appropriate data rate (MCS) to be employed. Within the RL engine, there exists a mapping from the state to action, denoted as $S(t) \rightarrow a(t)$, which is regarded as the state-action policy map, represented by π_θ. Here, θ symbolizes the parameters of the policy map function, which are updated iteratively during the learning process. Following each transmission, a reward value $r(t)$ is determined to assess the efficacy of selected action $a(t)$ at the current state $S(t)$. These rewards are pivotal in training the RL engine, guiding the optimization of the state-action policy function through the adjustment of θ to enhance the anticipation of future rewards.

State, Action, and Rewards in DRL. The efficacy of a DRL algorithm is fundamentally governed by the appropriate selection and definition of states, actions, and rewards, which are essential in training the DRL model. Initially, we delineate the general design underpinning DRL algorithms, followed by the proposition of two distinct DRL policies sharing identical state, action, and reward designs.

State $s(t)$***:*** The state must ideally comprise all information that is relevant for the actor network and enable it to predict an appropriate action (data rate). In our implementation these comprise channel quality CQ and packet length of the current packet to be transmitted l_t. Since, the most recent channel measurement is usually taken in the past (typically Tms in the past), the channel may have changed within these Tms. The actor must be able to implicitly predict the current state of the channel from history. To enable this, we provide as state, a vector comprising a history of n channel states and lengths. The state is thus computed as $S(t) =< \mathbf{CQ}(t-T), \mathbf{CQ}(t-2T), \cdots, \mathbf{CQ}(t-kT), l_t >$. We use $k = 3$ in our implementation since higher values did not provide any significant benefit. The contents of **CQ** are computed by the Input Generator and depend on the type of channel measurement available in the device. We have designed DRL for three different kinds of channel measurements, each of which uses a different **CQ**.

Action $a(t)$***:*** The action space is all the possible MCSs determined by the IEEE standards for each packet. The policy network outputs the probability estimated by the neural network for actions to achieve the highest reward. Since the action space is discrete, we choose the action with the largest possibility with 95% of the time and 5% of the time we choose the action randomly to explore other rates.

Reward $r(t)$***:*** We design the reward in DRL to specifically target packet loss and power consumption as they are the primary determining factors for user experience in a gaming scenario. In our design we prioritize packet loss more than power consumption since packet loss results in immediate loss of user experience. Further, consecutive packet losses are discouraged to greater degree as they have more significant impact on user experience. The reward function as follows:

$$r(t) = -\tau(MCS) - Q * \rho(t), \tag{12}$$

where $\tau(MCS)$ is the total transmission time by choosing the current MCS, including the retransmission time if the first packet is lost and $\rho(t)$. The term $-\tau(MCS)$ is a penalty on long packet transmission times and hence discourages DRL from choosing low data rates. The term $Q * \rho(t)$ penalizes packet losses with the weight Q determining the trade-off between the power consumption and the packet losses. To prioritize avoiding packet loss $\rho(t)$ is computed as a running average of the packet loss given by :

$$\begin{aligned} \rho(t) &= \rho(t-1) * \alpha + b(t), \\ b(t) &= \begin{cases} 0 & \text{if no packet loss} \\ 1 & \text{if packet loss} \end{cases} \end{aligned} \tag{13}$$

α denotes the reliability requirements, for instance, $\alpha = 0.99$ for the audio packets. In this manner, each packet loss has a long lasting negative impact on the reward, and consecutive packet losses have a greater impact on the reward calculation. This choice discourages DRL to quickly eliminate losses.

B Proximal Policy Optimization (PPO)

B.1 PPO Design

Overview of PPO Algorithm. PPO algorithms belong to class of RL algorithms known an Actor-Critic algorithms comprising two separate neural network models – an Actor and a Critic (Fig. 20). An actor model is a state-action policy map, π_θ to learn what action to take under a particular observed state; θ represent the weights of the neural network. The critic model V_ϕ evaluates the effectiveness of π_θ by predicting the expected future reward based on past history of action-reward pairs;its weights are represented by ϕ. The critic learns by trying to minimize the discrepancy between its past estimates of future rewards and those that it actually observes. The actor learns by attempting to maximize the expected future rewards as predicted by the critic network. Both actor and critic networks learn in conjunction taking turns based on the evolution of state, actions and rewards by employing gradient decent optimization. As time progresses, the critic learns to

Table 3: Network Architecture Exploration for PPO.

Actor		Critic		Loss [%]	Convergence below 0.5%	Runtime (250 MHz) [μs]			Runtime (600 MHz) [μs]		
Architecture (In-Hidden-Out)	Total Parameters	Architecture (In-Hidden-Out)	Total Parameters			Training	Inference	Total	Training	Inference	Total
27-120-9	4320	27-240-1	6720	2.4	NA	13017	4831	17848	5300	2085	7385
		27-40-80-1	4360	2.7	NA	10093	4825	14918	4188	2080	6268
		27-80-40-1	5400	2.8	NA	11531	5003	16534	4657	2087	6744
		27-80-80-1	8640	2.1	NA	14114	5008	19122	6133	2081	8214
27-240-9	8640	27-240-1	6720	0.69	NA	17327	10169	27496	7371	4148	11519
		27-40-80-1	4360	0.98	NA	14900	9660	24560	6239	4115	10354
		27-80-40-1	5400	0.76	NA	16294	10153	26447	6750	4154	10904
		27-80-80-1	8640	0.47	218s	18949	9970	28919	8191	4135	12326
27-40-80-9	5000	27-240-1	6720	0.49	196s	13758	5537	19296	5571	2385	7956
		27-40-80-1	4360	0.65	NA	10346	5773	16119	4454	2406	6860
		27-80-40-1	5400	0.61	NA	12169	5662	17831	4927	2417	7344
		27-80-80-1	8640	0.42	211s	15693	5638	21331	6465	2422	8887
27-80-40-9	5720	27-240-1	6720	0.36	165s	14444	6708	21152	5941	2760	8701
		27-40-80-1	4360	0.43	159s	11708	6659	18367	4844	2766	7610
		27-80-40-1	5400	0.41	162s	12835	6515	19350	5305	2749	8054
		27-80-80-1	8640	0.28	171s	16438	6557	22994	6834	2759	9593
27-80-80-9	9280	27-240-1	6720	0.16	174s	18156	10485	28641	7598	4413	12011
		27-40-80-1	4360	0.24	163s	16210	10586	26796	6561	4417	10978
		27-80-40-1	5400	0.19	169s	16731	11011	27742	7039	4452	11491
		27-80-80-1	8640	0.13	185s	20250	10825	31075	8534	4416	12950

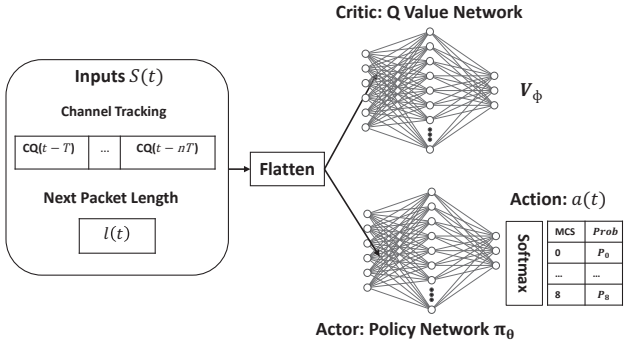


Figure 20: The PPO Architecture for Reinforcement Learning.

predict future rewards more accurately and the actor learns to take more optimal actions for each state. While updating the policy-map, PPO algorithms constrain the amount of change allowed in the policy to limit sudden/drastic changes and hence are stable to sudden changes. In our implementation, we chose a Multilayer Perceptron (MLP) with two hidden layers to represent both actor and critic models

Training Methodology. The objective of the PPO algorithm is to maximize the expected accumulative reward from current time t: $R^\theta(t) = \mathbf{E}_{(S(t),a(t))\sim\pi_\theta}\left[\sum_{j=t}^{\infty}\gamma^{(j-t)}r(j)\right]$, where $\gamma \in [0,1]$ is the discount factor (usually 0.99) used to avoid the accumulated reward to be infinity, and $r(t)$ is the reward by taking action $a(t)$ at state $S(t)$.

The Actor-Critic structure first obtains a finite mini-batch of sequential samples from the trajectory memory. The PPO algorithm randomly chooses a start point within each batch and uses the sub-sequential data to train the network. A new objective function is proposed in PPO to achieve mini-batch updates and update the policy smoothly. PPO introduces importance sampling to obtain the expectation of samples gathered from an old policy π_{old} under the new policy π_{new} we want to refine with the probability ratio $R^\theta(t) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$. They maximize the following surrogate objective function: $L(\theta) = \widehat{E}\left[\min\left(R^\theta(t), \text{clip}\left(R_t(\theta), 1-\varepsilon, 1+\varepsilon\right)\right)\widehat{A}_t\right]$, where ε is the clipping parameter. $\widehat{A}_t$ is an estimator of the advantage function at time step t. We use the generalized advantage estimator (GAE) [59] to calculate $\widehat{A}_t$. By introducing the clipped objective function, the PPO algorithm won't stick to the favoring actions with positive advantage, and make quicker update to avoid actions with a negative advantage function from a mini-batch of samples. $\theta = \theta - \eta_\theta \nabla L^\theta$, where η_θ is the learning rate for the actor model optimization.

Avoiding Losses During Convergence. When PPO is training for the first time or when the environment changes suddenly, PPO may experience higher packet losses during the time it takes to learn, adapt and converge to a steady state. In order to avoid the losses during these vulnerable times, PPO chooses a relies on a conservative retransmission strategy. While RL in PPO provides only the data rate of the initial transmission, the retransmission provides greater reliability by reducing the data rate to an MCS three steps below that suggested by PPO. Note that the corresponding longer transmission time also acts as a discouraging penalty to the reward function

B.2 PPO Network Architecture Exploration.

In this study, we explored various architectural configurations for the actor and critic networks in the PPO algorithm to evaluate their performance on different metrics, including loss

percentage, convergence time, and runtime in two different CPU frequency settings (250 MHz and 600 MHz) for the CSI based input. The architectures were delineated based on the input-hidden-output layers, and each configuration's total number of parameters was reported. A discernible trend is the general improvement in loss percentage and convergence time with more complex network architectures, characterized by a higher number of parameters. The most notable performances were observed in configurations employing the 27-80-80-9 architecture for the actor network, which consistently achieved the lowest loss percentages and reasonable convergence times below 0.5%. Specifically, the combination with the 27-80-80-1 critic architecture exhibited the most promising results, recording the lowest loss of 0.13% and a swift convergence time of 185 seconds.

While the intricate architectures, such as the 27-80-80-9 actor and 27-80-80-1 critic configuration, exhibit good performance in reducing loss and enhancing convergence times, they necessitate significantly extended runtime durations, especially at a frequency setting of 250 MHz. This increase in runtime, which encompasses both the training and inference phases, exhibits a positive correlation with the complexity of the network architectures. In summary, while the PPO network is capable of learning from scratch to find a policy that optimizes the packet loss rate, achieving this necessitates a relatively large network. This complexity poses a challenge for implementation on embedded devices, where computational resources are typically limited. This underscores the need to strike a balance between network complexity and computational efficiency to achieve optimal performance without over-burdening the system resources.

RFID+: Spatially Controllable Identification of UHF RFIDs via Controlled Magnetic Fields

Donghui Dai[1], Zhenlin An[1,2], Zheng Gong[1], Qingrui Pan[1,3], Lei Yang[1,4]
[1]*The Hong Kong Polytechnic University,* [2]*Princeton University,* [3]*The University of Edinburgh*
[4]*Shenzhen Research Institute, The Hong Kong Polytechnic University*

Abstract

In the fast-paced landscape of UHF RFID technology, achieving precise spatial-selective identification is of critical importance in the logistics and retail domain. This work introduces RFID+, a magnetically-driven UHF RFID system that leverages the matching loops of commercial-off-the-shelf UHF RFID tags for efficient energy harvesting from tailored magnetic fields. The RFID+ delivers a level of spatial precision comparable to that of HF NFC systems, effectively mitigating issues of miss-reading and cross-reading. Our primary contributions reside in the development of a specialized multi-turn, capacitor-segmented coil antenna and an innovative fast inventory algorithm. The RFID+ seamlessly integrates traditional radiative coupling with the innovative magnetic coupling in UHF RFID systems, bolstering their overall performance and efficiency. Real-world pilot studies in warehouses and logistics settings reveal that RFID+ significantly diminishes the miss-reading rate from 22.9% down to a remarkable 1.06%, while entirely eliminating cross-reading challenges. Moreover, our RFID+ variant demonstrates better resilience against materials traditionally challenging for UHF RFID, such as water bottles and containers. These advancements make RFID+ exceedingly relevant for practical applications in logistical networks.

1 Introduction

Ultra-high frequency (UHF) Radio-Frequency IDentification (RFID) has arisen as a transformative non-contact identification paradigm in the logistics and retail domains [1–3]. By harnessing wireless radio frequency (RF) communication, RFID infrastructures can effortlessly access unique Electronic Product Codes (EPC) from passive RFID tags affixed to various items. A key criterion is the pinpoint identification of tagged objects within a predetermined Region of Interest (ROI). For instance, in automated shopping arenas, RFID systems discern customer selections within a specified vicinity near the departure point, commonly termed the checkout zone; in environments such as airports or manufacturing units, RFID systems adeptly oversee an array of items transitioning on conveyors; within warehouse contexts, RFID systems diligently oversee item dynamics at ingress and egress junctures. The efficacy of the RFID paradigm hinges on its prowess to execute these operations expeditiously and with unparalleled accuracy, even amidst challenging operational conditions.

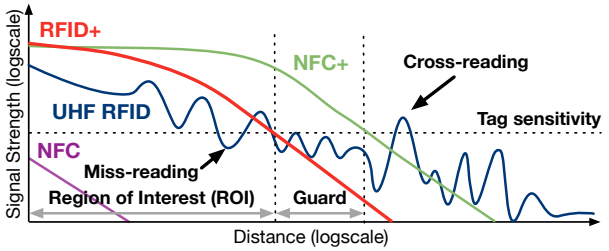


Fig. 1: The unpredictable propagation behavior of RF signals has traditionally rendered UHF RFID susceptible to anomalies of miss-reading and cross-reading. RFID+ defies this norm by introducing a controlled magnetic field, effectively addressing both miss-reading and cross-reading challenges similar to the HF NFC+.

Nevertheless, without fully realizing the extensive capabilities of RFID, present-day RFID systems remain under the shadow of two significant challenges: *miss-reading anomaly* and *cross-reading anomaly*. Conventionally, UHF RFID tags operate in far-field domains, where the RF signal propagates as planar electromagnetic (EM) waves, decaying at a pace of $O(1/r^2)$ in relation to distance r. As depicted in Fig. 1, governing the far-field RF signals is intricate due to their multifaceted interplay with RF-inimical materials such as metals or liquids. These signals are prone to reflection or absorption. Particularly, within the ROI, there is the potential for multipath signals to nullify each other, culminating in "blind zones" where the RF intensity is not sufficient to trigger relevant RFID tags (i.e., the miss-reading anomaly). In contrast, outside the designated ROI, multipath signals may combine constructively, inadvertently leading to the recognition of undesired tags (i.e., the cross-reading anomaly). For instance, as illustrated in Fig. 2(a-b), the confined and crowded nature of the checkout lane in an automated store inherently leads to unavoidable tag miss-reading and cross-reading anomalies.

Modern RFID systems face challenges in addressing cross-reading and miss-reading anomalies [4, 5] simultaneously. While increasing transmission power and sensitivity can improve detection, it often leads to unintended cross-readings. On the other hand, mitigating cross-reading can cause missed detections. Although recent debates favor a localization-based approach over the traditional binary reading, its effectiveness is questioned due to challenges like the "garbage in, garbage out" [6,7]. While RFGo [8] leverages machine learning for enhanced detection, it falls short in dynamic scenarios. NFC+ [7] previously explored the potential of long-range, magnetical-coupling HF NFC at 13.56MHz as an alternative to UHF RFID for improved ROI management. Magnetic signals inherently minimize cross-reading owing to their rapid decay,

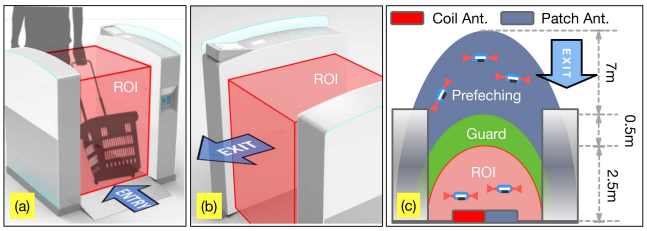


Fig. 2: Checkout lane. (a) and (b) show the entrance and the exit of the lane. These two figures are reproduced from [8]. (c) shows our prefetching augmented ROI coverage.

and diminish the likelihood of miss-reading because of their pronounced penetration through challenging materials such as liquids and metals. However, NFC+ still faces three challenges: (1) HF tags are substantially more expensive than UHF tags, posing deployment barriers; (2) existing infrastructures, heavily reliant on UHF RFIDs [9], face challenges in adopting NFC due to compatibility issues; (3) NFC's reading rate of 50 tags/s is dwarfed by RFID's 200 tags/s, making NFC less suitable for high-speed inventory tasks such as those involving moving tags on conveyors or checkout zones.

Contrary to HF NFC tags that use *coil antennas* for inductive or magnetic coupling, UHF RFID tags commonly utilize *dipole antennas* to capture and emit electromagnetic waves. The absence of coils may give the *misleading impression that UHF RFID tags are ill-suited for energy harvesting from magnetic fields.* However, as illustrated in Fig. 24 (Appendix A), the architecture of a standard UHF RFID tag includes a small loop (highlighted in blue) situated near the chip. Commonly referred to as the "matching" or "tuning" loop, this component is critical in optimizing energy harvesting and offering protection against over-voltage issues. Our findings indicate that this inherent loop structure can, in fact, enable UHF RFID tags to harvest energy from magnetic fields, thereby mimicking the inductive coupling characteristics as seen in HF NFC.

In this work, we introduce RFID+, a magnetically-driven UHF RFID system that innovatively repurposes the matching loop of a UHF RFID tag as an RF frontend for both energy harvesting and communication. The system's objective is the spatially selective identification of UHF RFIDs through precision-controlled magnetic fields. As depicted in Fig. 1, RFID+ not only emulates the propagation features of NFC+ but also overcomes its limitations. Specifically, RFID+ effectively interfaces with widely-used, economical UHF RFIDs while maintaining a high reading rate.

However, translating RFID+ into practice poses three significant engineering hurdles:

- Initially, the feasibility of harnessing energy from magnetic fields through the matching loops of commercial UHF RFIDs has not been thoroughly investigated. To fill this research void, we conduct an exhaustive feasibility study that shines light on the underlying principles of magnetically-driven UHF RFID systems. Through empirical evaluations conducted in real-world scenarios, we not only establish the viability of this technique but also substantiate its effectiveness and broad applicability.
- Secondly, the necessity arises for the integration of a coil antenna into a UHF RFID reader, enabling the creation of a controllable magnetic field. Traditional wavelength-matched loop antennas (single-coil) commonly seen in HF NFC (i.e., 13.54 MHz) become problematic when shifted to the UHF band (i.e., 860 – 960 MHz). This difficulty stems from the notably shorter wavelength at UHF relative to the loop's circumference, potentially leading to certain regions experiencing mutual magnetic field nullification. To address this concern, we put forward a tailored multi-turn, capacitor-segmented coil antenna complemented by a high-impedance reflector, striving for an even and directional magnetic field distribution.
- Thirdly, RFID systems driven by magnetic fields have a range that caps at about 3m, substantially less than the potential 10m span of those powered by radiative methods, largely attributed to the swift dissipation of magnetic fields. This confined ROI intensifies the need for prompt tag reading. To mitigate potential delays, especially with fast-moving tags, we initially engage a conventional far-field, radiatively-coupled reader to prefetch potential tags moving toward the ROI. Subsequently, we incorporate Bloom filters alongside the near-field, magnetically-coupled reader to streamline the inventory procedure.

Contributions. This study re-examines the intricate ROI management in UHF RFID through the lens of inductive coupling. We present three major contributions: 1) We validate the potential of magnetically-driven UHF RFID; 2) We introduce innovative coil antenna designs and a tailored inventory algorithm for RFID+; 3) Through rigorous tests and real-world pilots, we demonstrate RFID+'s efficacy. We hope that this research revitalizes academic discourse surrounding magnetically-driven UHF RFID systems in typical scenarios.

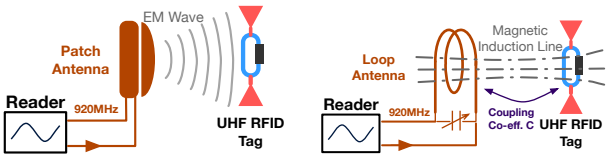
2 Magnetically-Driven UHF RFID

In this section, we start by exploring the basics of UHF RFID, introduce inductive coupling via tags' matching loops, and finally assess the feasibility across COTS UHF tags.

2.1 Background

The coupling mechanism describes the means by which systems engage in the interchange of energy or information. The mechanisms adopted by HF NFC and UHF RFID vary significantly, largely owing to the different frequency ranges in which these systems operate.

• **Inductively-Coupled HF NFC**: Operating at 13.56MHz, HF NFC employs inductive coupling facilitated by coil antennas. In this arrangement, an oscillating electrical current flows through the reader's coil, generating a variable magnetic field in the surrounding area. When an NFC tag comes within this field's sphere of influence, the magnetic flux induces a current in the tag's coil antenna, thereby activating the NFC chip and initiating communication with the reader. Given the reactive



(a) Radiatively-Coupled RFID (b) Inductively-Cdoupled RFID

Fig. 3: Rationale behind Magnetically-driven UHF RFID

nature of these fields, the effective communication range is generally limited to a short span of several centimeters.

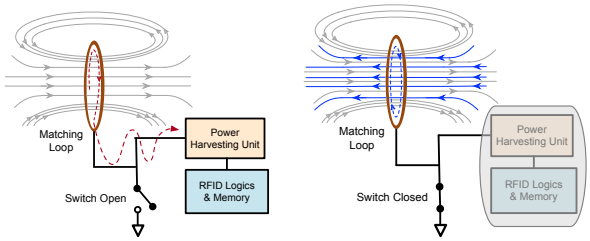
• **Radiatively-Coupled UHF RFID**: Typically operating within the 860-960 MHz frequency range, UHF RFID utilizes radiative coupling, commonly executed through dipole antennas. In this scenario, the reader emits a UHF radio wave that, upon reaching an RFID tag, induces an electric current in the tag's antenna. Utilizing backscatter technology, the tag then modifies and reflects the wave back to the reader. This method allows for a substantially extended operational range, often reaching up to several meters.

In essence, while HF NFC primarily employs inductive coupling for close-proximity interactions, UHF RFID leverages radiative coupling to enable longer-distance engagements.

2.2 Inductive Coupling via Matching Loops

Inductive coupling usually requires coil-configured antennas that the UHF RFID tags are short of. Thus, it seems that UHF RFID tags are not reactive to magnetic fields at first glance. Actually, as illustrated in Fig. 25 (Appendix A), every RFID tag inherently incorporates a single-turn coil, referred to as the matching loop. This essential component bridges the gap between dipole-style antennas and the tag's integrated chips (ICs), serving three primary functions: Firstly, it facilitates impedance matching between the antenna and the chip, ensuring optimal power conveyance. Secondly, this loop aids in adjusting the voltage levels to align with the chip's requisites. Lastly, by safeguarding the correct impedance and voltage calibrations, it inadvertently fortifies the RFID chip against potential over-voltage detriments.

These loops can essentially act as standalone coil antennas. When exposed to a magnetic field, these loops capture an induced electrical current that then flows to energize the chip, successfully accomplishing inductive coupling. Therefore, it becomes feasible to power and interface with UHF RFID tags through magnetic fields, akin to the operation seen in NFC systems. To elucidate further, let us juxtapose the operational mechanics of conventional radiatively-coupled RFID systems with those of our innovative magnetically-coupled ones. Fig.3(a) portrays the conventional system where the reader is equipped with a patch antenna – a specific variation of a dipole antenna. A tag derives its power from the transmitted electromagnetic waves fully using its own dipole antenna. On the other hand, Fig.3(b) illustrates our approach: the reader comes outfitted with a coil antenna, purposed for the generation of concentrated magnetic fields. This magnetic flux subsequently traverses the tag's matching loop, thereby



(a) Non-reflective (b) Reflective

Fig. 4: Magnetic backscatter communication. (a) With the switch is open, the induced current is directed towards the harvesting unit; (b) With the switch is closed, the coil experiences a short-circuit. The current generated within the coil fosters an opposing magnetic field, which subsequently resonates with the reader's coil, acting as a reflection.

inducing an electric current. The distinction between the two approaches lies solely in their signal propagation mechanisms – via either a patch or a coil antenna. All other system parameters, including the operating frequency and transmission power, remain unaltered, making this a financially judicious solution.

2.3 Communication Immutability

A concern may arise regarding the necessity of altering communication protocols when passive tags obtain power through magnetic fields. This concern can be addressed by examining the two communication links of UHF RFID: the downlink and the uplink.

• **Downlink (Reader ⇒Tag)**: In the downlink, where the reader communicates to the tag, the goal is to query tags or send specific commands. The reader employs an Amplitude Shift Keying (ASK) method, encoding different bit values with varying amplitude levels. This ASK causes magnetic strength changes, inducing a current with amplitude variation in the tag's matching loop. Thus, even with magnetic power derivation, the tag's chip can decode ASK commands.

• **Uplink (Tag ⇒ Reader)**: In the uplink, where tags communicate to the reader, the primary goal is to send the stored EPC. When energized by the reader's signal, passive tags use "backscatter modulation" to transmit data. This process involves a switch that connects the tag's peripherals to its IC, as depicted in Fig. 4. When the switch is on, the tag directs the induced current from the loop to the chip, making it absorbent and non-reflective. However, when off, the tag grounds the chip, and the loop-initiated current, following Lenz's law, creates an opposing magnetic field. This renders the RFID tag reflective, sending the loop-induced power back to the reader. The tag thus communicates bits by toggling between its reflective and non-reflective states or backscattering.

Thus, the modulation techniques employed for both downlink and uplink communication remain unmodified.

2.4 Experimental Verification

We verify the feasibility of the magnetically-driven approach using standard COTS RFID hardware. Fig. 5 provides a visual representation of the experimental configuration, showcasing the tags evaluated. We utilized a widely-used commercial RFID reader (Impinj R420 [10]) set to a frequency of

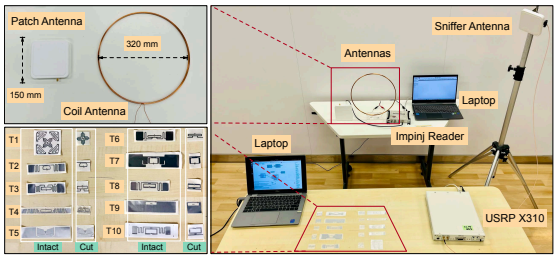


Fig. 5: Experiment Validation. A commercial Impinj reader is equipped with a patch antenna and a coil antenna, respectively. Ten types of commercial tags are tested.

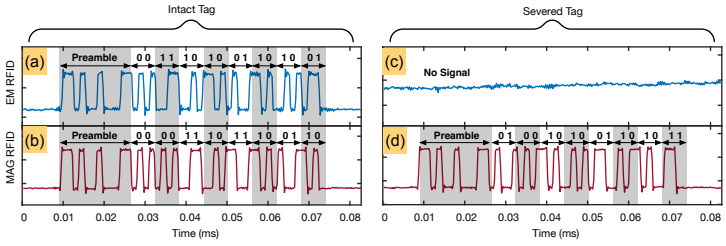


Fig. 6: Captured RN16 signals from a tag across four distinct conditions. (a) and (b) display signals procured from the unaltered tag, powered by EM waves and magnetic fields, correspondingly. (c) and (d) illustrates signals from the modified tag whose dipole antenna is cut off. As a result, it is unresponsive to EM-wave query anymore but still functions effectively when exposed to magnetic fields.

920MHz. Alongside this, we deployed the USRP X310 [11] device as a dedicated sniffer to capture the communication between the reader and the tag. Our tests incorporated two distinct reader antennas: a 15×15 cm^2 patch antenna and a coil antenna with a 32cm diameter. The reader is alternately connected to each of these antennas, with the transmission power maintained at 30dBm. The sniffer was strategically placed close to the reader antenna, roughly 30cm away. For each trial, we arranged each tag in an approximately parallel orientation to the loop to maximize the capture of magnetic flux.

Comparative Tests on Loop Role. Fig. 26 has validated that COTS RFID tags can be activated and queried using magnetic fields while the communication protocol remains consistent[1]. To test if tags can continue to harness energy from magnetic fields using only matching loops, we conducted two comparative inventory tests with an M4 tag (i.e., T1) positioned 50cm away. We alternated between using a patch and coil antenna for the reader. The first test assessed the `RN16` responses of the unaltered tag. The signals from both antennas were largely consistent, as seen in Fig. 6(a) and (b). In the second test, after cutting off the tag's dipole antenna and leaving only the matching loop (see Fig. 5), the tag was unresponsive to EM-waves but still functions effectively when exposed to magnetic fields, as shown in Fig. 6(c) and (d). The comparative results emphasize the crucial role of the matching loop in energy harvesting. The comparative analyses were also conducted for the other nine tag types (T2-T10, as depicted in Fig. 5). The outcomes were consistent across all tests as summarized in Table 2.

3 Overview

At a high level, RFID+ is elegantly simple: We strategically replace the conventional electric patch antennas of UHF RFID readers with tailored coil antennas, in the pursuit of precisely controlled magnetic field emission aimed at UHF RFID tags within a specified ROI. Benefiting from rapid attenuation and strong penetration capabilities, magnetically-driven RFID+ minimizes miss-reading and cross-reading anomalies. To this end, we introduce a unique multi-turn, capacitor-segmented coil design for our UHF RFID system in §4 and devise a prefetching-based algorithm to speed up readings in §5. The subsequent sections elaborate on the technical details.

Scope. The matching loops inherent in existing UHF RFID tags are clearly not optimized for energy extraction from magnetic fields, resulting in low radiation efficiency and a relatively limited range. Our approach balances compatibility with controllability, designed to integrate seamlessly with existing UHF RFID tags while offering precise ROI management in near-field settings like conveyor belts or checkout zones. In such environments, the desired operational range is approximately 100-200cm, consistent with prior findings [7, 8, 12]. Thus, our intent is not to supplant existing radiatively-coupled UHF RFID systems or to extend the long-range reading capability, but rather to complement them by offering improved spatial controllability within confined ROIs.

4 Spreading Magnetic Fields

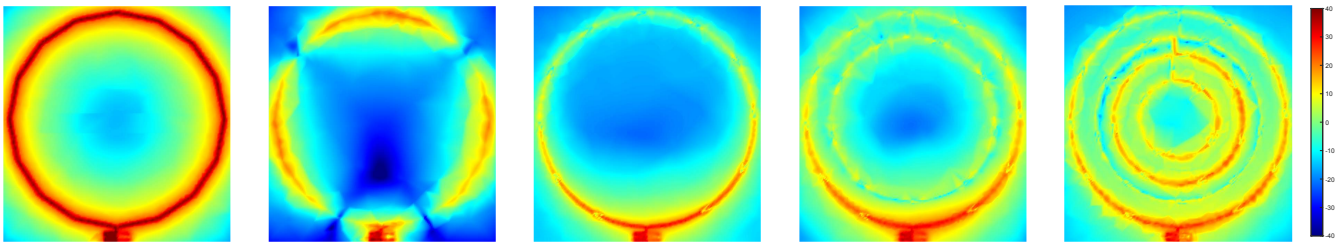
In this section, we delve into the novel coil antenna design tailored for our magnetically-driven UHF RFID system.

4.1 Necessity of a Novel Coil Antenna

Why are traditional coil antennas inadequate for the specific requirements of magnetically-driven UHF RFID systems? To answer this question, we use ANSYS HFSS [13] software to simulate the distributions of magnetic fields generated by a 7.5cm radius single-turn coil at both 13.56MHz (HF NFC norm) and 920MHz (UHF RFID standard). The findings, illustrated in Fig. 7(a), reveal a uniform magnetic field distribution at the 13.56MHz frequency. However, at 920MHz, the field distribution shown in Fig. 7(b) exhibits irregularities, including four intensified regions surrounding the loop and five weaker areas, or "blind zones," both centrally and at the corners. Such uneven distribution highlights the limitations of conventional coil antennas for UHF RFID applications, specifically in terms of potential activation failures in these blind zones.

The observed unevenness can be explained as follows: the coil is considered as a composition of countless tiny elements, each generating its unique magnetic field with an initial phase. The overall field is the sum of these individual fields. Phase alignment results in constructive field addition, while misalignment can cause mutual nullification. To better visualize

[1] More details can be found in Appendix A.



(a) Loop@13.56 MHz (b) Loop@920MHz (c) One coil@920MHz (d) Two coils@920MHz (e) Four coils@920MHz

Fig. 7: Simulated magnetic intensity in the vicinity of five distinct antenna structures. (a) denotes an unsegmented loop actuated by a 13.56 MHz HF excitation signal; (b) illustrates an unsegmented loop energized by a 920 MHz UHF signal; (c) offers a depiction of a segmented loop equipped with lumped capacitors, resonating to a 920 MHz UHF signal; (d) presents a two-turn segmented loop integrated with fork capacitors, subjected to a 920 MHz UHF excitation; and (e) displays a multi-turn segmented spiral loop with fork capacitors, stimulated by a 920 MHz UHF signal.

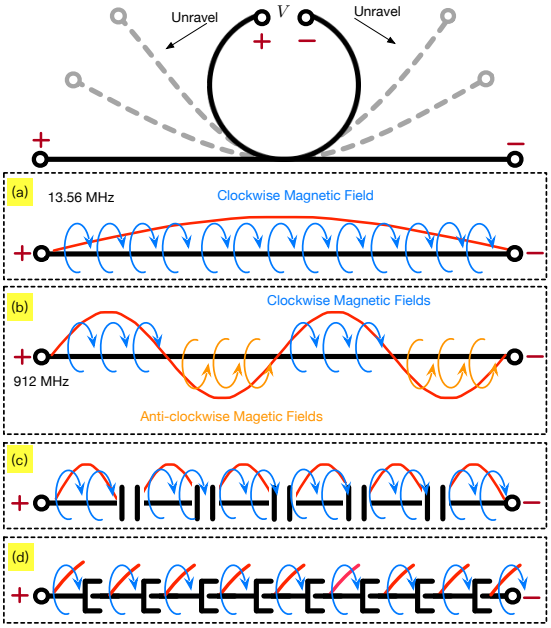


Fig. 8: Analysis on a loop antenna. (a) Magnetic fields remain in consistent directions at 13.5MHz; (b) The magnetic fields change their directions alternatively along the coil. (c) The capacitor-segmented coil eliminates the disparities in the initial phase. (d) The microstrip lumped caption-segmented coil.

the phase variations as the signal traverses the coil, we represent the loop linearly, depicted in Fig.8. With HF signals, characterized by a 22-meter wavelength, phase discrepancies across coil elements are negligible due to the significant disparity between the wavelength and the coil's 50 cm circumference. A maximum phase difference is just 0.14 radians, ensuring a predominantly in-phase superposition, as displayed in Fig.8(a). Conversely, the UHF signal has a wavelength of 32 cm, comparable to the coil's circumference. As the signal traverses the coil, it completes about two cycles, as depicted in Fig. 8(b). This causes each element's initial phase to fluctuate between 0 and 2π repeatedly. The clockwise magnetic fields stem from positive currents, while negative currents yield anti-clockwise ones. When refashioned into a loop, these opposing magnetic fields combine out-of-phase, resulting in the observed uneven distribution. Thus, there arises a pressing demand to pioneer a novel coil antenna design tailored for the proposed magnetically-driven UHF RFID systems.

4.2 Capacitor-Segmented Coil Antenna

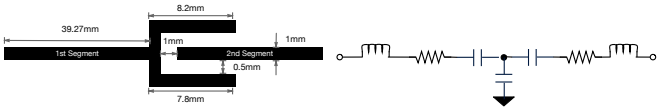
The loop antenna's limitation stems from the incoherent amalgamation of different elements because of phase discrepancies. A straightforward remedy might be to shrink the loop's size to 1/4 wavelength, ensuring better phase alignment. Yet, antenna theory fundamentals suggest that a loop antenna resonates (evident as a purely real impedance) only when its circumference is roughly equal to a wavelength [14]. More accurate dimensions need to be ascertained through specialized 3D electromagnetic simulation.

Capacitor-Segmented Loop. To counteract the loop antenna's limitations, we propose segmenting the loop physically and inserting capacitors between adjacent segments, as shown in Fig. 8(c). Each segment can then be modeled as an equivalent RLC circuit. Let R, L and C represent the intrinsic resistance and inductance of a segment, and the capacitance respectively. Capacitors are known to resist sudden voltage changes. Under AC conditions, the current behind a capacitor obtains a phase shift ϕ, which is given by:

$$\phi = \arctan\left(\frac{2\pi f L - \frac{1}{2\pi f C}}{R}\right) \quad (1)$$

where f symbolizes the frequency of the signal. This equation indicates that by fine-tuning the capacitor's value, one can methodically counterbalance a desired phase shift. As shown in Fig. 8(c), segmentation at intervals of half a wavelength along the line results in a 180° phase change each time. By strategically selecting the capacitance values, a corrective phase shift of -180° is introduced by each capacitor. This ensures the RF signal flowing between consecutive capacitors retains a uniform initial phase shift. Such alignment gives rise to uniform clockwise magnetic fields across segments. When the linear arrangement is formed into a loop, this coherence is maintained, yielding a consistent magnetic field distribution. This technique can uphold the loop's size while guaranteeing a balanced field distribution.

Fork-Shaped Lumped Capacitor. Utilizing a single discrete capacitor is straightforward, but integrating several capacitors onto a PCB might result in unexpected power consumption and enlarged dimensions. Contemporary antenna systems lean towards microstrip lumped antennas that



(a) Microstrip Lumped Capacitor (b) Equivalent circuit

Fig. 9: Fork-shaped Lumped Capacitor. (a) shows the structure and the size of the lumped capacitor. (b) shows the equivalent circuit.

smartly embed lumped components within the antenna's design. Inspired by prior designs in microstrip antennas [15–17], we adopt the fork-shaped lumped capacitors. As shown in Fig. 9(a), the capacitor is implemented by the microstrip line. In the figure, two segments are positioned 1mm apart. The former segment ends with a three-sided forked shape that interfaces with the beginning of the subsequent segment. Specifically, the dimensions of the top, left, and bottom sides are 8.2mm, 4mm, and 7.8mm, respectively. The bottom side is slightly shortened compared to the top to fit the arc design. Fig.9(b) presents the equivalent circuit where the three sides are modeled as individual capacitors.

The capacitance of the lumped capacitor is approximated by $8\varepsilon\sqrt{A/\pi}$, where A represents the area of the forked configuration and ε is the permittivity coefficient. This equation suggests that a considerable area is needed to achieve a substantial capacitance and the corresponding large phase shift. Thus, to avoid bulky capacitors, we trim the segment length, thereby adjusting the required phase. Each segment, featuring the aforementioned fork-shaped capacitor, was iteratively determined in HFSS by tuning the phase shifts according to segment length. Specifically, our design sets the segment length at 3.927cm, roughly 11.9% of the 32.872cm wavelength, deviating from the standard half-wavelength model. It is crucial to note the compromise: this design necessitates four times as many lumped capacitors than its half-wavelength counterpart. Fig. 8(d) showcases this coil design. As the signal traverses a 3.927cm segment, its phase shifts by 42.84°. The lumped capacitor then counters this shift by −42.84°, ensuring consistent signal alignment across segments.

4.3 Spiral Coil Antenna

As illustrated in Fig.7(c), the capacitor-segmented loop offers a markedly even distribution of magnetic field intensity compared to the conventional loop design. The updated design spreads this energy more uniformly. However, a region of diminished intensity remains near the center. This drawback can be remedied by using multi-turn coils. Accordingly, we introduce a spiral coil antenna, whose design and implementations are shown in Fig.10. This design features four distinct coil turns, artfully arranged in a spiral layout, with both the start and end points of the coil connected using via-holes. A greater coil density might seem beneficial but poses a challenge: the potential coupling between neighboring coils. To alleviate this potential interference, we reserve an 18mm spacing between them (i.e., half of the segment length) after iterative optimization. The simulated magnetic intensity distribution of two-turn and four-turn coil antenna are shown in Fig. 7(d) and

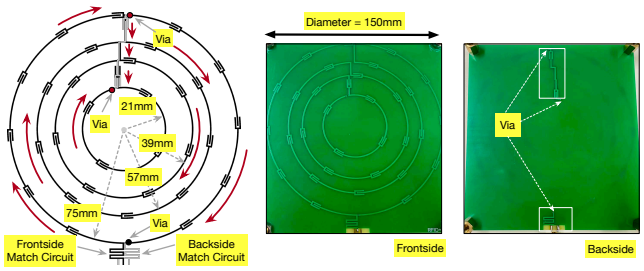


Fig. 10: Spiral Coil Antenna. Multi-turns of coils are arranged in a spiral fashion.

(e), respectively. The simulation results reinforce our hypothesis that a spiral configuration not only enhances the effective electrical length of the loop antenna but also accentuates its intensity with the addition of more turns.

4.4 Directional Coil Antenna

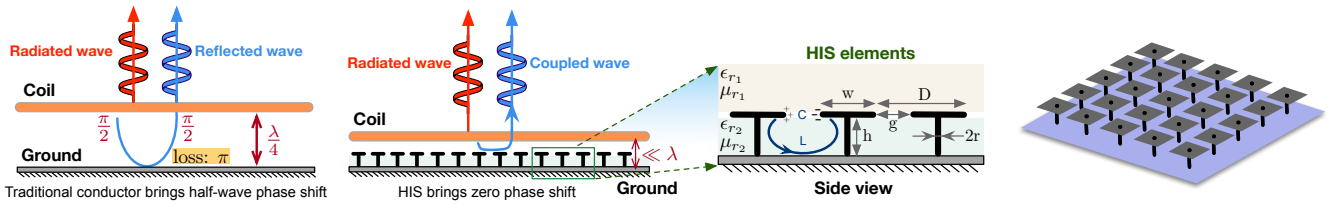
Many applications, like inventory management, require directional coverage to meet user expectations. It is also notable that today's EM-driven UHF RFID systems use directional patch antennas. Thus, aligning with industry norms, our next steps will focus on crafting a directional coil antenna. A conventional solution is to place a metal reflector behind the coil antenna to direct magnetic fields forward. However, this approach faces the issue of half-wave loss. When RF signals transition from low to high impedance boundaries, the reflected wave undergoes a 180° phase shift. As depicted in Fig. 11(a), if the coil-reflector gap is $< \lambda/4$, these shifted magnetic fields destructively interfere with those from the opposite side. The constructive superpositions occur only when the gap is set to $\lambda/4$. For our 32cm wavelength, a separation of about 8cm is required, increasing antenna thickness.

Mushroom-like HIS. Inspired by the artificial magnetic conductors [18], we introduce a high-impedance surface (HIS) approach to intrinsically mitigate the half-wave loss. The core principle behind HIS is illustrated in Fig.11(b) and (c). Resting atop a metallic base, a series of compact square patches are arranged in a grid pattern. These mushroom-like patches (called HIS elements) connect to the base via central via-holes, with a deliberate spacing between them. As a result, adjacent elements essentially function as capacitors, connected through the via-hole to the base metal below. Given the inherent parasitic resistance, two proximate HIS elements together resemble a standard parallel resonant RL circuit. The impedance related to an HIS can be expressed as:

$$Z_{\text{HIS}} = \frac{j\omega L}{1-\omega^2 LC} = \frac{j\omega L}{1-(\omega/\widehat{\omega})^2} \tag{2}$$

where ω signifies the angular frequency of the prevailing magnetic field, while $\widehat{\omega} = 1/\sqrt{LC}$ represents the resonant angular frequency. The L and C refer to the parasitic resistance and lumped capacitance, respectively. Their values are calculated as outlined in [19]:

$$L = \frac{\eta_s}{\omega}\tan(\beta h) \text{ and } C = \frac{1}{\pi} w\varepsilon_0(\varepsilon_{r_1}+\varepsilon_{r_2})\cosh^{-1}\left(\frac{D}{g}\right) \tag{3}$$



(a) Traditional Metal Reflector (b) Sideview of HIS (c) Bird's-eye View of HIS

Fig. 11: Reflection-Induced Phase Shift. (a) When magnetic signals encounter a metallic ground, they experience a 180° phase shift due to half-wave loss. To counterbalance this phase shift, the separation between the coil and the ground must be set to $\lambda/4$. If the gap is less than $\lambda/4$, reflected signals will destructively interfere with the upward-propagating magnetic waves. (b) Illustrates a side view of the mushroom-structured HIS, designed to minimize the phase shift from reflections, effectively bringing it close to zero. (c) Depicts a top-down or bird's-eye view of the HIS.

Here, $\eta_s = \sqrt{\mu_0\mu_{r_2}/\varepsilon_0\varepsilon_{r_2}}$ and $\beta = \omega\sqrt{\mu_0\mu_{r_2}\varepsilon_0\varepsilon_{r_2}}$ stand for the intrinsic wave impedance and propagation constant, respectively. The constants ε_0 and μ_0 are the permittivity and permeability of a vacuum, with relevant parameters defined in Fig. 11(b). By varying the size and spacing of these elements, one can fine-tune the resonant frequency and achieve the desired impedance characteristics for specific applications.

Zero-Phase Shift. When the HIS's resonant frequency $\widehat{\omega}$ aligns with the frequency ω of the impinging magnetic field, the HIS manifests as an "infinite" impedance surface, as evident from Eqn. 2. This phenomenon arises because the equation's denominator approaches zero. In this scenario, the magnetic field's reflected phase by the HIS is articulated by

$$\theta = \mathrm{Im}\left(\ln\left(\frac{Z_{\mathrm{HIS}}-\eta_0}{Z_{\mathrm{HIS}}+\eta_0}\right)\right) \approx 0 \tag{4}$$

because $Z_{\mathrm{HIS}} \gg \eta_0$. This implies that an HIS can proficiently eliminate the 180° phase shift induced by the half-wave loss, facilitating constructive coupling of the reflected waves via the HIS. By leveraging the properties of the HIS, it becomes possible to significantly reduce the required separation between the HIS and the coil antenna to well below $\lambda/4$. Simultaneously, nearly all of the magnetic energy is constructively redirected to the opposing side of the antenna. The simulated magnetic field distributions with and without an HIS reflector are compared in Appendix B.

4.5 Coil Antenna Array

In near-field communications, interactions are primarily driven by magnetic fields, which are divided into reactive and radiative near-field domains. The reactive near-field resides close to the antenna, typically within $0.62\sqrt{D^3/\lambda}$, while the radiative near-field or Fresnel Region extends to about $2D^2/\lambda$, with D representing the antenna's maximum linear dimension(i.e., aperture). For optimal performance, our single coil's outermost circumference is set to λ, making its diameter $D = \lambda/\pi$. Theoretically, the near-field range of a single-turn coil antenna becomes approximately 12cm, which falls short for many practical applications. Despite incremental advancements from components such as capacitor-segmented, multi-turn coils, and the zero-phase reflection of HISs, the range expansion is still insufficient, extending to merely 50cm. To

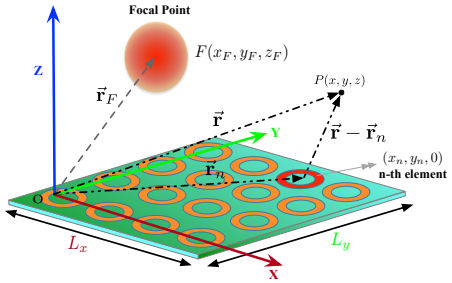


Fig. 12: Magnetic Antenna Array forms a near-field focal point at inventory region.

ensure comprehensive coverage, we utilize an array of coil antennas to shape the detection zone as needed. As previously discussed, the near-field range R is influenced by the square of the antenna aperture D, following the relationship $2D^2/\lambda$. Consequently, augmenting the aperture by a factor of three through an array setup could lead to a ninefold enhancement in the near-field range.

For an antenna array with N coils, as shown in Fig. 12, adjacent coils are spaced by $\lambda/2$. The nth coil's position is represented by $\vec{r}_n$ or coordinates $(x_n, y_n, 0)$ with n spanning from 1 to N. Each coil follows a uniform radiation pattern, denoted as $\widehat{B}(\vec{r})$, indicating the magnetic field vector. This pattern is adjusted based on the decay factor $\frac{1}{R}e^{-\mathbf{J}2\pi R/\lambda}$, where R denotes the maximum operational range. The total magnetic field generated by the array at an observation point $P(x,y,z)$ or $\vec{r}$ is given by:

$$B(\vec{r}) = \sum_{n=1}^{N} C_n B_n(\vec{r}) = \sum_{n=1}^{N} A_n e^{\mathbf{J}\varphi_n} \cdot \widehat{B}(\vec{r}-\vec{r}_n) \cdot \frac{e^{-\mathbf{J}2\pi||\vec{r}-\vec{r}_n||/\lambda}}{||\vec{r}-\vec{r}_n||} \tag{5}$$

where $C_n = A_n e^{\mathbf{J}\varphi_n}$ is the nth coil's complex excitation coefficient. Each coil is activated with an amplitude A_n and a modifiable phase φ_n.

To direct the magnetic field towards the ROI, the phase shifts of the N coils' excitation coefficients in the array must be tuned. For a desired concentration of the magnetic field at point $\vec{r}_F$, the distance to the origin is $R_F = ||\vec{r}_F||$ and the unit vector pointing to this spot is $\vec{\hat{r}}_F = \vec{r}/||\vec{r}||$. The conjugate phase method, as referenced in [20, 21], suggests setting each antenna's phase φ_n as:

$$\varphi_n = \frac{2\pi}{\lambda}\|\vec{r}_F - \vec{r}_n\| = \frac{2\pi}{\lambda}\sqrt{R_F^2 + \|\vec{r}_n\|^2 - 2R_F\vec{\hat{r}}_F\cdot\vec{r}_n} \tag{6}$$

When R_F is much larger than coil size L, the phase adjustments needed to focus on point F are linear and quadratic. The Fresnel approximation, cited in [22], captures this:

$$\varphi_n \approx -\frac{2\pi}{\lambda}\left(\vec{r}_F \cdot \vec{r}_n\right) + \frac{2\pi}{\lambda}\frac{\|\vec{r}_n\|^2}{2R_F} \tag{7}$$

A constant phase term, $-2\pi R_F/\lambda$, is omitted as it's relatively insignificant. This approximation is valid with an error under $\pi/8$ if $R_F > \sqrt[3]{L^4/8\lambda}$. However, for closer focal points, necessitating $F \gg L$ to be invalid, Eqn. 6 should be used for accurate phase fine-tuning. To achieve complete coverage of a ROI, the array simply needs to adjust its focal point for meticulous traversal of the area. The small size of the area renders the scanning process effortlessly manageable in practical applications. Furthermore, inventory advancements [9, 23, 24] that prioritize tag localization before communication can be swiftly integrated into RFID+, given its primary focus on enhancing the RF frontend, which permits extensive customization for the signal processing backend.

5 Fast Inventory

In this section, we incorporate the far-field UHF RFID system with the proposed near-field UHF RFID system to expedite inventory processing for tags within the ROI.

5.1 Dual-Coupling Systems

Magnetically-driven RFID systems, despite recent advancements, can only reach a maximum range of about 2.5m, significantly less than the 12m of radiatively-driven RFIDs due to magnetic properties. This constrains their use to smaller ROIs like gates or checkout lanes. Within these narrow confines, a high reading rate, the number of tags recognized per second, becomes essential. Slow readings might miss rapidly moving tags. Fig. 2 shows a self-service checkout scenario, where the system must quickly detect all tagged items in a brief timeframe to ensure a smooth customer experience. Currently, prevalent RFID systems use the Q-adaptive anti-collision protocol, a time-division-based ALOHA derivative. As described in [25], the peak efficiency of such protocols is approximately 36.8%. This means nearly 74% of the time is lost to channel contention, posing a significant efficiency challenge.

To mitigate the low efficiency in the channel competition, we present a prefetching mechanism that harmoniously combines both radiatively-driven RFID and magnetically-driven RFID systems to enhance the reading speed within the ROI. In this configuration, one reader interfaces with a patch antenna, while another is connected to our innovatively designed coil antenna. For clarity, the two readers are called *far-field reader* and *near-field reader*, respectively. Both antennas are strategically positioned toward the direction from which the tags approach. As depicted in Fig. 2(c), a conceivable setup would have the two antennas suspended above the checkout lane, angled antero-inferiorly. This arrangement ensures that the coil antenna encompasses the entirety of the near-field ROI (approximately the 3m-long lane), while the patch antenna extends its coverage to a broader 10m-long far-field region, inclusive of the ROI and its surrounding area. Leveraging the extended reach of the far-field RFID reader, it becomes feasible to preemptively identify a set of candidate tags expected to traverse through the ROI. While this set might occasionally register cross-readings or omit certain tags, it still offers a substantial advantage by expediting the operations of the near-field reading. In summary, the far-field antenna initially pre-fetch a set of potential tags, enabling the near-field reader to swiftly verify their presence in the ROI based on the far-field's prior knowledge. This time-divided dual-stage approach guarantees that operations in the far-field and near-field do not interfere with each other.

5.2 Acceleration via Perfected Bloom Filter

A Bloom filter (BF) is a space-efficient probabilistic data structure to represent a set of elements. It can be used to fast test whether an element is a member of the set. More introduction about the BF refers to Appendix. C. Initially, the far-field reader collects a set of candidate tags, denoted by T, utilizing the time-consuming Q-adaptive algorithm. The `EPC`s of these collected tags are then used to construct a Candidate Bloom Filter (BF). Notably, the construction does not necessitate any back-and-forth communication between the reader and the tags, but it is generated by the algorithm. Both the `EPC`s and the BF are passed to the near-field reader via Ethernet cables.

In the next phase, the near-field reader uses the previously obtained BF to check for the presence of tags within the ROI quickly. Rather than transmitting their full 96-bit `EPC`s, the tags merely send 16-bit `RN16` packets to indicate their presence, thus speeding up the inventory process significantly. We adopt the method outlined in previous work [26,27] to test the BF on-site directly, which has theoretically demonstrated that the acquisition overhead is reduced by approximately 60%. The algorithm subsequently performs a comparative analysis to identify which candidate tags are genuinely in the ROI, i.e., checking the slots during which a desired tag responds based on the hashing results. This can be seen as a streamlined polling algorithm that verifies tags against a predefined list of names without the need for channel competition. Finally, for missing tags that are not indicated by the BF, the near-field reader uses the `Select` command to explore them.

6 Implementation

In this section, we introduce the system implementation of RFID+ and conduct the microbenchmark on coil arrays.

6.1 Fabrication

Given that RFID+ utilizes dual-coupling to blend far-field and near-field characteristics, it necessitates two distinct sets of reader hardware for implementation in actual deployments. Specifically, a commercial Impinj R420 reader [10], accompanied by its patch antenna, serves as the far-field reader in

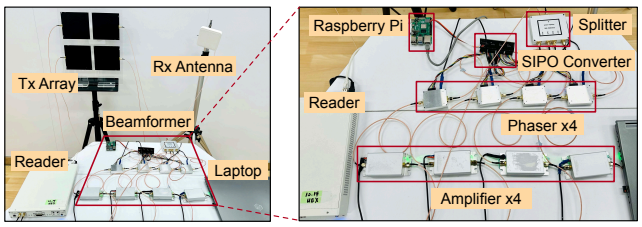


Fig. 13: Experiment setup.

our prototype. In contrast, the near-field reader is custom-constructed around the foundation of the USRP X310 [11].

• **Near-Field Reader**. Our near-field reader prototype is anchored in the capabilities of the USRP X310, which boasts two independent RF frontend configurations. One interface is dedicated to an array of coil antennas for transmission (TX). The secondary interface liaises with a compact patch antenna designed for reception (RX). It should be noted that integrating the TX and RX into a monostatic mode is technically feasible and would further reduce the system's size, benefiting dynamic scenarios. The architectural blueprint of the reader is shown in Appendix D. Upon upconversion in the USRP, the resultant signal is split into four distinct pathways. Each trajectory is orchestrated by an analog phaser (i.e., PHSA-152+ from Mini-Circuits [28]). These phasers enable meticulous phase tuning with an eight-bit granularity. The signals undergo amplification to achieve a cumulative transmission power of 30 dBm before they are broadcasted by the quartet of coil antennas. A Raspberry Pi 4 Model B [29] orchestrates the coordination among the phaser units.

• **Coil Array**. At the heart of the system lies a 2×2 coil antenna array, encompassing four distinct coil antennas, each of which has a 75mm radius and about 1.4λ circumference. (see Fig. 10). The exact size and shape of the antenna are determined through iterative optimization to enhance the near-field magnetic intensity. As depicted in Fig. 13, these antennas are integrated onto a two-layer 1.6mm FR-4 printed circuit board (PCB), with HIS reflectors secured to the rear via plastic standoffs. They operate primarily around the 920MHz. Performance enhancement is achieved with an impedance-matching circuit, aligning the antenna to the 50Ω RFID reader standards. Each antenna unit, measuring 17cm by 16cm, incurs an approximate manufacturing expense of $65.53, as detailed in Appendix E.

6.2 Microbenchmark

Before embarking on a detailed performance evaluation, we initially provide a quantitative analysis of the performance improvements attributable to each antenna design component.

• **Component Gain**. We used a magnetic field probe to measure the H-field strength induced by each component of the designed antenna, assessing their respective gains. Different coils, such as conventional single-turn, segmented-line single-turn, multi-turn, along with HIS reflectors, were constructed for evaluation. The data in Tab. 1 show that the gains from the capacitor-segmented, multi-turn, and directional components were 3.01 dB, 1.96 dB, and 1.68 dB, respectively, contributing to a total increase of 6.65 dB in the H-field's peak strength.

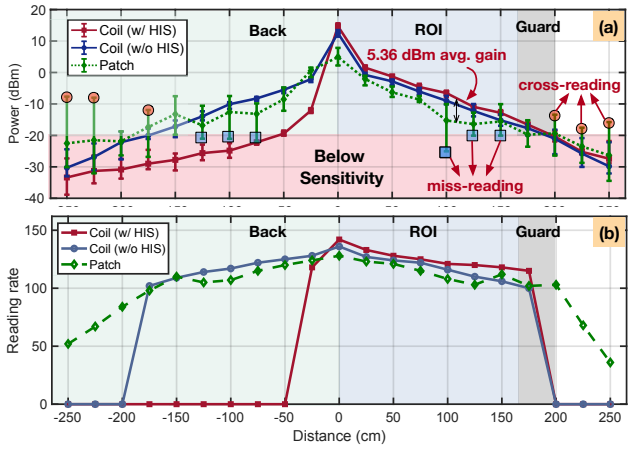


Fig. 14: Controllability Analysis across Different Antennas.(a) and (b) show the signal strength and reading rate as a function of distance, respectively.

Next, we characterize the coil array by analyzing the engineered coil antenna's attributes. We examined three distinct configurations: the novel coil array with and without an HIS, and a traditional electrical patch antenna devoid of a reflector. For experimental purposes, the antenna was positioned on the XOY plane. Here, positive (or negative) distances signify locations either ahead of (or behind) the antenna along Z-axis. Each configuration was subjected to 20 trials. The reader's transmitting power was consistently maintained at 30dBm.

• **Strength Distribution**. We analyzed the magnetic field distribution in radiative regions using a spectrum analyzer [30] coupled with magnetic field probes, as depicted in Fig. 14(a). (1) Frontside: In front of our coil antenna, magnetic strength gradually decreased with little variation. Conversely, the EM field from the patch antenna fluctuated significantly, with strengths ranging from -25 to 11dBm@100cm. Considering a sensitivity threshold of -20dBm [31], the patch antenna could cause miss-readings @100-200cm and potential cross-readings @200-250 cm due to EM field unpredictability. (2) Backside: Behind the coil, the strength quickly fell below the sensitivity level, attributed to null-phase-shift reflections from the HIS. These reflections added an average 1.68dB gain to the frontside magnetic field, equivalent to a 1.5 times power gain. Our findings emphasized the heightened issues of miss-reading and cross-reading in traditional RFID antennas. In contrast, magnetic antennas exhibited enhanced controllability due to reduced multipath propagation effects. This study underscores the superiority of the proposed coil antenna and HIS in spatial control.

• **Effective Coverage**. Reading rates, defined as the number of readings recognized per second (r/s), were gathered across a range of -250 to 250 cm. The results are portrayed in Fig. 14(b). Frontside: Within 25-175 cm, the coil antenna con-

Table 1: Each Components' Impact on H-Field Boost.

Elements	Segmented	Multi-turn	HIS	**Total**
Gain (dB)	3.01	1.96	1.68	**6.65**

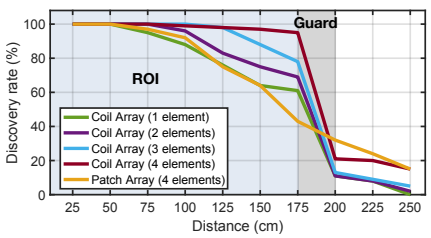


Fig. 15: Inventory Accuracy

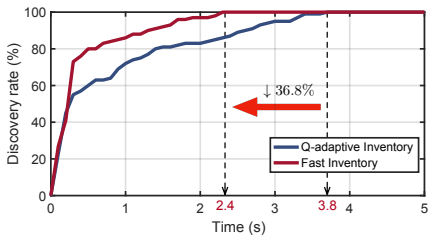


Fig. 16: Inventory Efficiency

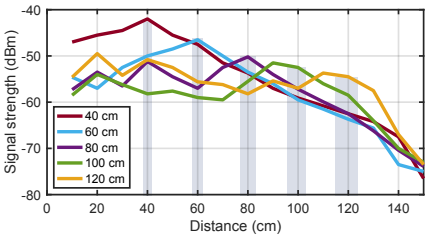


Fig. 17: Spatial Manipulation

sistently showed high reading rates of 115-133 r/s. Beyond 175 cm, this rate drops to zero, mirroring the observed decline in magnetic strength and indicating precise ROI control. Conversely, the patch antenna's reading capability persists beyond 200 cm, capturing nearly 36 r/s at 250 cm. Backside: Beyond -50 cm, the coil antenna w/ HIS reading rate swiftly drops to zero. However, both the coil w/o HIS and the patch antenna display similar reading rates on the backside as seen on the frontside.

Summary. The outcomes of these two experiments emphatically demonstrate the heightened controllability presented by the proposed coil array in both the physical and application dimensions relative to traditional electronic patch antennas. Further, the data suggests a maximum effective range of 175cm plus a 25cm guard zone.

7 Evaluation

In this section, we conduct a group of experiments within a sizable office measuring 50 m^2 full of multipath reflectors to evaluate RFID+ comprehensively.

7.1 Inventory Accuracy

We conducted a comparative analysis of RFID+ versus conventional RFID systems, centered on inventory accuracy. Both systems employed a 2×2 antenna array, with the former utilizing the proposed coil and the latter using a conventional patch antenna. In our test, a dense collection of 100 tags were fixed to a flat surface. The accuracy was measured based on the discovery rate, i.e., the percentage of unique tags identified from the total of 100 tags.

The results are shown in Fig. 15. From the figure, we have two main findings: First, RFID+ boasts near-perfect detection within the ROI, identifying almost all tags for distances up to 175cm. Beyond this zone, the detection drops drastically. Traditional RFIDs maintain a good detection rate only up to 100cm, and their performance diminishes past this point, mostly due to environmental reflections. Second, the number of coils in the array influences detection reach. Distances of flawless detection were reduced to 125cm, 100cm, and 75cm for arrays with 3, 2, and 1 coils, respectively. Overall, RFID+ offers better ROI management than standard RFIDs.

7.2 Inventory Efficiency

We introduce a Bloom Filter-enhanced fast inventory algorithm to expedite the near-field reader's inventory process as discussed in Sec. 5. Leveraging prefetched tags, the far-field reader constructs Bloom Filters to ascertain tags within the ROI swiftly. We compared this approach against conventional inventory methods. Initially, an electronic antenna gathers `EPC`s from 100 tags, out of which only 30 prefetched and 20 miss-read tags arrived at the near-field ROI. The efficiency is gauged by the discovery rate over time. As illustrated in Fig. 16, our fast inventory approach completes the discovery in 2.4 seconds, contrasted with the 3.8 seconds by the Q-adaptive algorithm. This marks a 36.8% efficiency boost, primarily due to the omission of anti-collision procedure.

7.3 Spatial Controllability

Next, we examine RFID+'s capability in finely tuning the beamforming focal point using the coil array. The focal point represents the peak of energy concentration derived from the coordinated quartet of coils. We directed the system's focus to five positions along the Z-axis: 40, 60, 80, 100, and 120cm. For each position, a tag was moved from 10cm to 120cm, and the backscattered signal strength was measured.

The findings are illustrated in Fig. 17. As anticipated, signal strength peaked exactly at the 40cm, 60cm, and 80cm marks when directed there. Beamforming focal point increases the average received signal strength (RSS) by approximately 7.73 dB at these locations. However, at 100cm and 120cm, the peaks lagged by 10cm, likely due to power dissipation effects, namely, the combined power from the four coils is unable to compensate for the losses experienced over extended distances. This observation is further mirrored in the peak values decreased with distances, i.e., from -42.3dBm to -46.5dBm, -50.2dBm, -51.5dBm, and -53.7dBm. Such precision in focal adjustments is unparalleled, making it invaluable for specialized scenarios like detecting tags in containers or conveyors.

7.4 Penetrability

We assess the penetration capabilities of RFID+, NFC+ [7], and traditional RFID when tags are positioned on the front (LoS) and backside (NLoS) of various liquid products. Our experiment evaluates six distinct products: M1 (64mm-thick bottled water), M2 (48mm-thick canned Coke), M3 (40mm-thick bottled Coke), M4 (85mm-thick boxed milk), M5 (45mm-thick boxed milk), and M6 (64mm-thick Bottled beer). Such liquid bottles are placed 50cm ahead of the antenna array, which is linked to an Impinj R420 reader used to measure the signal strength.

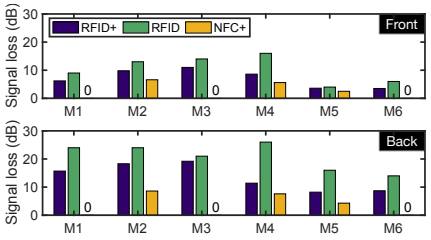


Fig. 18: Penetrability

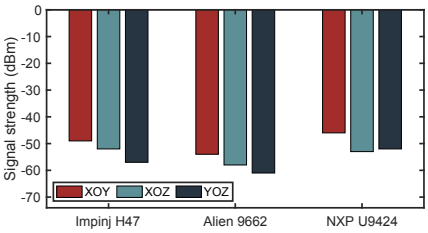


Fig. 19: Impact of tag orientation

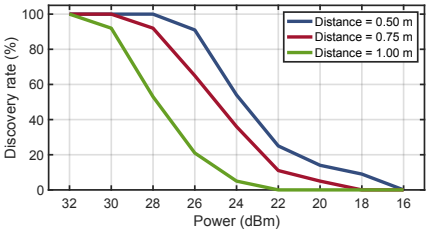


Fig. 20: Impact of transmission power

Fig. 18 illustrates the difference of signal strengths (i.e., loss) acquired when a tag is either affixed to a product or not, respectively. Notably, NFC+, operating at 13.56MHz, exhibits superior performance in most scenarios due to its reduced vulnerability to water interference. Conversely, UHF frequency signals experience higher absorption by water molecules, leading to increased signal loss. Yet, RFID+ surpasses standard RFID systems, registering average losses of 7.1dB and 13.5dB on the product's front. These losses rise to 13.5dB and 20.8dB for RFID+ and RFID, respectively, on the back. In summary, RFID+ demonstrates potent penetration for liquid products with thicknesses less than 60mm, particularly near the ROI.

7.5 Impact Analysis

Finally, we consider the two potential factors that affect the performance of RFID+.

• **Impact of Tag Orientation.** We investigate the role of tag orientation in determining RSS. Three distinct tags, namely Impinj H47, Alien 9662, and NXP U9424 are positioned 50cm from the coil array. These tags are aligned parallel to the XOY, XOZ, and YOZ planes, with the coil array set on the XOY plane. Fig. 19 presents the RSS from the backscatter signals for each tag across the three configurations. Clearly, optimal RSS is achieved when the tag orientation is parallel to the coil array (i.e., XOY), as this allows maximum magnetic flux to traverse the tag's matching loops. Conversely, the least favorable setup is the YOZ orientation, where minimal flux interacts with the coils. This orientation sensitivity poses a recurring challenge in RFID systems, attributed to the use of planar antennas in tags. A practical workaround involves deploying multiple coils in varied orientations to mitigate such orientation-based discrepancies.

• **Impact of Transmission Power**. We investigated the influence of transmission power on the discovery rate, situating tags at intervals between 0.5 m and 1 m from the reader. As depicted in Fig. 20, the discovery rate correlates inversely with decreasing power levels from 32 dBm to 16 dBm. Notably, when transmission power diminishes below 26 dBm, the discovery rate plunges to under 20% for a 1 m distance setting. Moreover, an increased separation between the reader and tags further depresses the rate. For example, at a robust transmission power of 28 dBm, the discovery rate remains optimal at 100% for a 0.5 m distance. Yet, when the distances extend to 0.75 m and 1 m, the rates taper off to 70% and 48%, respectively. This underscores that RFID+, akin to traditional RFID systems, is power-sensitive, primarily because energizing the passive tags consumes a significant portion of the transmitted energy.

8 Pilot Study: Logistic Network Evaluation

8.1 Warehouse Management

We tested RFID+ in a textile factory warehouse with an annual revenue of 100 million USD for contact manufacturing of branded apparel. The factory uses both RFID and barcodes for identification, shown in Fig. 21 (a-c). We set up two 2×2 RFID+ antenna arrays near storage shelves. Over 200 tags were attached to clothing items like T-shirts and jeans, packed in garment boxes, and moved on a Manual Hand Pallet Jack. The number of tags is subject to the Manual Hand Pallet Jack's maximum capacity and aligns with values reported in earlier studies [8, 12]. As items passed through the scanning zone, RFID+ logged the detected products. Fig. 22 depicts the performance of RFID+, as determined by the mean outcome of ten replicated trials that collectively involved more than 2,000 tags. It detected 98.94% of tags at the gateway, outperforming the commercial radiatively-coupled RFID system's 77.14%. This is because approximately 10% of the regions are blind spots for reading with conventional radiatively-coupled RFID electrical antennas [4, 32, 33]. When five volunteers, including two students and three workers, manually counted with barcode scanners, the traditional optical barcode identification system noted about 97%. Hence, RFID+ demonstrated superior accuracy against both commercial RFID and manual counts. For cross-reading accuracy, we established a 2m×2m ROI around the coils/antennas and placed tags randomly at its edges. The results in Fig. 22 show RFID+ had a negligible 0.09% cross-reading rate, significantly less than the UHF RFID system's 42%. The manual method registered 1.4%, a number potentially increasing with working overtime. In summary, RFID+ excels in minimizing cross-reading and miss-reading when contrasted with both traditional RFID systems and manual inventory methods. It is even capable of approaching the performance of the state-of-the-art (SOTA) system, NFC+ [7], which records a mere 0.03% rate of miss-readings and a zero cross-reading rate for randomly oriented objects. The comparison in Appendix E also highlights RFID+'s outperformance over another SOTA system RFGo [8] in terms of generality, affordability, and ease of deployment.

Fig. 21: Deploying RFID+ in real-world logistics networks. (a-c) show the application in warehouse management, whereas (d-e) illustrate the application in supply chain planning.

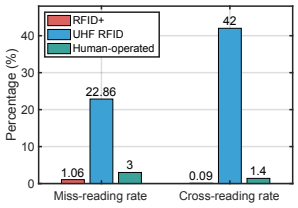


Fig. 22: Performance in Warehouse

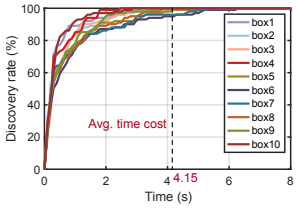


Fig. 23: Performance in Logistics

8.2 Supply Chain Planing

We further explored RFID+'s efficacy in a supply chain setting, wherein boxed products traverse through a conveyor scanning gateway. This conveyor bridges the warehouse and the truck's cargo hold. We demarcated an inventory zone (i.e., ROI) on the conveyor, dimensions being $1.5 \times 0.5 \times 1.6\text{m}^2$, as visualized in Fig. 21(d-e). Each box, housing around 150-200 garments based on the apparel type, is strategically positioned on the conveyor at 1.5m gaps. Whenever a box enters the designated zone, the conveyor intuitively reduces its speed for inventory purposes. Given the tags' movement on the conveyor and the minimal interference risk from neighboring tags due to the deliberate spacing, our system consistently showcased flawless performance without any miss-reading or cross-reading instances. Fig. 23 portrays the discovery rate vis-a-vis the time expenditure for ten sequential boxes. An average time consumption of roughly 4.15 seconds is observed for each box. This testing underscores RFID+ 's prowess to seamlessly integrate into real-world industrial settings, where UHF RFID consistently sidesteps both miss-reading and cross-reading, all within a reasonable timeframe.

9 Related Work

The related work can be categorized into the following areas:

(1) UHF RFID Communication. Over time, UHF RFID systems have been thoroughly tested and understood [5, 7, 12, 34–37]. Their vulnerabilities, including missed readings, are heightened in RF-unfriendly environments [38] or due to multipath issues [5]. Historical attempts at remedying these inaccuracies have explored beamforming [39], nonlinear backscattering [40], wideband signal [12] and wave optimization [41,42]. Besides missed readings, cross-reading errors are a challenge [43]. Localizing UHF RFID tags has been a primary method to sieve out unwanted tags from the ROI [12,34]. Still, most localization solutions are antenna-intensive, with RFGo [8] using eleven antennas, proving costly and intricate for industrial settings. The NFC+ solution [7] proposes enhanced NFC technology as a UHF RFID substitute. Although NFC is highly secure [44], its low throughput is its Achilles' heel. The industry has explored merging HF NFC and UHF RFID technologies by developing dual-frequency tags [45]. However, this approach compromises the low-cost benefit of standard tags due to the need for custom-designed alternatives. In contrast, RFID+ innovatively combines the best features of both technologies, enabling accurate UHF RFID tag detection with spatially controllable magnetic fields while also ensuring full compatibility with legacy RFID tags.

(2) Magnetic Antenna Design: Recent studies have focused on developing magnetic antennas for near-field UHF RFID applications [46, 47], highlighting this technology's potential to substitute LF/HF RFID in tagging individual items [48]. Traditional solid-line loop antennas primarily be utilized as RFID reader antennas in low frequency [49] and high frequency [50, 51] RFID systems. However, these antennas struggle to maintain a uniform magnetic distribution within the ROI at the UHF band, mainly due to their inability to be electrically small [14]. Addressing this, Dobkin et al. introduced the use of lumped series capacitors in loop antennas to counteract current nulls and enhance magnetic strength in the UHF range [52]. Later studies [15, 17] examined the properties of segmented line capacitors, eliminating the need for discrete circuit components. This led to investigations into various distributed capacitor designs [53–56]. While our work is inspired by existing research, RFID+ is the first to harmoniously integrate disparate elements (e.g., capacitor-segmented loops, multi-turn UHF coils, HISs, coil arrays, etc) into a unified practical system.

10 Conclusion

This work introduces RFID+, a highly accurate and reliable system for RFID tag inventory. RFID+ utilizes the tailored magnetic field to achieve a 99% discovery rate within the ROI, simultaneously preventing cross-reading of tags outside this area. Our warehouse analysis indicates that RFID+ has the potential to revolutionize the logistics industry.

Acknowledgements

We thank all anonymous reviewers and our shepherd Dr. Zerina Kapetanovic for their help and insightful feedback. This work is supported by the NSFC Key Program (No. 61932017), UGC/GRF (No. 15204820, 15215421), Innovation and Technology Fund (ITS/099/21), and Shenzhen Fundamental Research Program (No. JCYJ20230807140410022). Zhenlin An and Lei Yang are co-corresponding authors.

References

[1] E. C. Jones and C. A. Chung, *RFID in logistics: a practical introduction*. CRC press, 2007.

[2] Z. Zhang, G. Xu, and E. C. Kan, "Outlooks for uhf rfid-based autonomous retails and factories," *IEEE Journal of Radio Frequency Identification*, 2022.

[3] S. Pradhan, E. Chai, K. Sundaresan, S. Rangarajan, and L. Qiu, "Konark: A rfid based system for enhancing in-store shopping experience," in *Proceedings of the 4th International on Workshop on Physical Analytics*, 2017, pp. 19–24.

[4] C. Loo, A. Elsherbeni, F. Yang, and D. Kajfez, "Experimental and simulation investigation of rfid blind spots," *Journal of Electromagnetic Waves and Applications*, vol. 23, no. 5-6, pp. 747–760, 2009.

[5] M. Taguchi and H. Mizuno, "Analysis of dead zone of rfid system," in *2006 IEEE Antennas and Propagation Society International Symposium*. IEEE, 2006, pp. 4759–4762.

[6] J. Wang, L. Chang, O. Abari, and S. Keshav, "Are rfid sensing systems ready for the real world?" in *Proc. of ACM MobiSys*, 2019, pp. 366–377.

[7] R. Zhao, P. Wang, Y. Ma, P. Zhang, H. H. Liu, X. Lin, X. Zhang, C. Xu, and M. Zhang, "Nfc+ breaking nfc networking limits through resonance engineering," in *Proc. of ACM SIGCOMM*, 2020, pp. 694–707.

[8] C. Bocanegra, M. A. Khojastepour, M. Y. Arslan, E. Chai, S. Rangarajan, and K. R. Chowdhury, "Rfgo: a seamless self-checkout system for apparel stores using rfid," in *Proc. of ACM MobiCom*, 2020.

[9] Y. Ma, N. Selby, and F. Adib, "Minding the billions: Ultra-wideband localization for deployed rfid tags," in *Proc. of ACM MobiCom*, 2017, pp. 248–260.

[10] Impinj, "Rain rfid readers, connectivity devices for iot solutions," https://www.impinj.com/products/readers, accessed: 2024-02-27.

[11] Ettus Research, "Usrp x310 high performance software defined radio," https://www.ettus.com/all-products/x310-kit/, accessed: 2024-02-27.

[12] B. Liang, P. Wang, R. Zhao, H. Guo, P. Zhang, J. Guo, S. Zhu, H. H. Liu, X. Zhang, and C. Xu, "Rf-chord: Towards deployable rfid localization system for logistic networks," in *Proc. of USENIX NSDI*, 2023.

[13] Ansys, "Ansys hfss: 3d high frequency simulation software," https://www.ansys.com/products/electronics/ansys-hfss, 2024, accessed: 2024-02-27.

[14] W. L. Stutzman and G. A. Thiele, *Antenna theory and design*. John Wiley & Sons, 2012.

[15] Y. Zeng, Z. N. Chen, X. Qing, and J.-M. Jin, "Modeling and characterization of zero-phase-shift lines and optimization of electrically large zpsl loop antennas for near-field systems," *IEEE Transactions on Antennas and Propagation*, vol. 64, no. 11, pp. 4587–4594, 2016.

[16] A. H. Muqaibel, A. Safaai-Jazi, and S. M. Riad, "Fork-coupled resonators for high-frequency characterization of dielectric substrate materials," *IEEE transactions on instrumentation and measurement*, vol. 55, no. 6, pp. 2216–2220, 2006.

[17] Z. N. Chen, X. Qing, J. Shi, and Y. Zeng, "Review of zero-phase-shift-line loop antennas for uhf near-field rfid readers," *IEEE journal of radio frequency identification*, vol. 1, no. 4, pp. 245–252, 2017.

[18] R. Dewan, M. Rahim, M. Hamid, M. Yusoff, N. A. Samsuri, N. Murad, and K. Kamardin, "Artificial magnetic conductor for various antenna applications: An overview," *International Journal of RF and Microwave Computer-Aided Engineering*, vol. 27, no. 6, p. e21105, 2017.

[19] T. H. Loh, "High impedance surface electromagnetic band gap metamaterials: Design approach and applications for antenna engineering," *Invited Paper*, 2011.

[20] P. Nepa and A. Buffi, "Near-field-focused microwave antennas: Near-field shaping and implementation." *IEEE Antennas and Propagation Magazine*, vol. 59, no. 3, pp. 42–53, 2017.

[21] A. Buffi, P. Nepa, and G. Manara, "Design criteria for near-field-focused planar arrays," *IEEE Antennas and Propagation Magazine*, vol. 54, no. 1, pp. 40–50, 2012.

[22] Wikipedia, "Fresnel diffraction," https://en.wikipedia.org/wiki/Fresnel_diffraction, 2024, accessed: 2024-02-27.

[23] J. Wang and D. Katabi, "Dude, where's my card? rfid positioning that works with multipath and non-line of sight," in *Proc. of ACM SIGCOMM*, 2013, pp. 51–62.

[24] L. Yang, Y. Chen, X.-Y. Li, C. Xiao, M. Li, and Y. Liu, "Tagoram: Real-time tracking of mobile rfid tags to high precision using cots devices," in *Proc. of ACM MobiCom*, 2014, pp. 237–248.

[25] D. Tse and P. Viswanath, *Fundamentals of wireless communication*. Cambridge university press, 2005.

[26] L. Yang, Q. Lin, C. Duan, and Z. An, "Analog on-tag hashing: Towards selective reading as hash primitives in gen2 rfid systems," in *Proc. of ACM MobiCom*, 2017, pp. 301–314.

[27] Z. An, Q. Lin, L. Yang, W. Lou, and L. Xie, "Acquiring bloom filters across commercial rfids in physical layer," *IEEE/ACM Transactions on Networking*, vol. 28, no. 4, pp. 1804–1817, 2020.

[28] Mini-Circuits, "Sphsa-152+ datasheet," https://www.minicircuits.com/pdfs/SPHSA-152+.pdf, 2024, accessed: 2024-02-27.

[29] Raspberry Pi Foundation, "Raspberry pi 4 model b," https://www.raspberrypi.com/products/raspberry-pi-4-model-b/, accessed: 2024-02-27.

[30] Rohde & Schwarz, "Hameg hms3010 3 ghz spectrum analyzer," https://scdn.rohde-schwarz.com/ur/pws/dl_downloads/dl_common_library/dl_brochures_and_datasheets/pdf_1/HAMEG_DB_EN_HMS3000_3010.pdf, accessed: 2024-02-27.

[31] D. Dobkin, *The rf in RFID: uhf RFID in practice*. Newnes, 2012.

[32] L. Ukkonen, D. Engels, A. Sydanheimo, and M. Kivikoski, "Reliability of passive rfid of multiple objects using folded microstrip patch-type tag antenna," in *2005 IEEE Antennas and Propagation Society International Symposium*, vol. 2. IEEE, 2005, pp. 341–344.

[33] R. H. Clarke, D. Twede, J. R. Tazelaar, and K. K. Boyer, "Radio frequency identification (rfid) performance: the effect of tag orientation and package contents," *Packaging Technology and Science: An International Journal*, vol. 19, no. 1, pp. 45–54, 2006.

[34] Z. Wang, M. Xu, N. Ye, R. Wang, and H. Huang, "Rf-focus: Computer vision-assisted region-of-interest rfid tag recognition and localization in multipath-prevalent environments," in *Proc. of ACM IMWUT*, 2019.

[35] L. Dodds, N. Naeem, A. Eid, and F. Adib, "Software-controlled polarization for longer-range rfid reading and localization," in *2023 IEEE International Conference on RFID (RFID)*. IEEE, 2023, pp. 90–95.

[36] L. Dodds, I. Perper, A. Eid, and F. Adib, "A handheld fine-grained rfid localization system with complex-controlled polarization," in *Proc. of ACM MobiCom*, 2023.

[37] J. Wang, J. Zhang, R. Saha, H. Jin, and S. Kumar, "Pushing the range limits of commercial passive rfids," in *Proc. of USENIX NSDI*, 2019, pp. 301–316.

[38] Y. Ma, Z. Luo, C. Steiger, G. Traverso, and F. Adib, "Enabling deep-tissue networking for miniature medical devices," in *Proc. of ACM SIGCOMM*, 2018, pp. 417–431.

[39] S. Chen, S. Zhong, S. Yang, and X. Wang, "A multiantenna rfid reader with blind adaptive beamforming," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 986–996, 2016.

[40] D. Vasisht, G. Zhang, O. Abari, H.-M. Lu, J. Flanz, and D. Katabi, "In-body backscatter communication and localization," in *Proc. of ACM SIGCOMM*, 2018, pp. 132–146.

[41] Z. An, Q. Lin, Q. Pan, and L. Yang, "Turbocharging deep backscatter through constructive power surges with a single rf source," in *Proc. of IEEE INFOCOM*, 2021.

[42] S. Sabesan, M. J. Crisp, R. V. Penty, and I. H. White, "Wide area passive uhf rfid system using antenna diversity combined with phase and frequency hopping," *IEEE Transactions on Antennas and Propagation*, vol. 62, no. 2, pp. 878–888, 2013.

[43] M. Bolic, M. Rostamian, and P. M. Djuric, "Proximity detection with rfid: A step toward the internet of things," *IEEE Pervasive Computing*, vol. 14, no. 2, pp. 70–76, 2015.

[44] D. Dai, Z. An, Q. Pan, and L. Yang, "Magcode: Nfc-enabled barcodes for nfc-disabled smartphones," in *Proc. of ACM MobiCom*, 2023, pp. 1–14.

[45] HUAYUAN RFID NFC Manufacturer, "Dual frequency rfid cards," https://www.huayuansh.com/products/rfid-smart-cards/dual-frequency-rfid-cards/, 2024, accessed: 2024-02-27.

[46] B. Shrestha, A. Elsherbeni, and L. Ukkonen, "Uhf rfid reader antenna for near-field and far-field operations," *IEEE Antennas and Wireless Propagation Letters*, vol. 10, pp. 1274–1277, 2011.

[47] X. Qing, C. Goh, and Z. Chen, "Segmented loop antenna for uhf near-field rfid applications," *Electronics letters*, vol. 45, no. 17, pp. 872–873, 2009.

[48] P. V. Nikitin, K. Rao, and S. Lazar, "An overview of near field uhf rfid," in *2007 IEEE international conference on RFID*. IEEE, 2007, pp. 167–174.

[49] A. Diet, Y. Le Bihan, C. Conessa, F. Alves, M. Grzeskowiak, M. Benamara, G. Lissorgues, M. Biancheri-Astier, and A. Pozzebon, "Lf rfid chequered loop antenna for pebbles on the beach detection," in *2016 46th European Microwave Conference (EuMC)*. IEEE, 2016, pp. 41–44.

[50] M. Benamara, M. Grzeskowiak, A. Diet, G. Lissorgues, Y. Le Bihan, S. Protat, and C. Conessa, "A twisted loop antenna to enhance hf rfid detection for different tag positioning," in *2016 10th European Conference on Antennas and Propagation (EuCAP)*. IEEE, 2016, pp. 1–5.

[51] X. Qing, Z. N. Chen, and A. Cai, "Multi-loop antenna for high frequency rfid smart shelf application," in *2007 IEEE Antennas and Propagation Society International Symposium*. IEEE, 2007, pp. 5467–5470.

[52] D. M. Dobkin, S. M. Weigand, and N. Iyer, "Segmented magnetic antennas for near-field uhf rfid," *Microwave Journal*, vol. 50, no. 6, p. 96, 2007.

[53] Y. S. Ong, X. Qing, C. K. Goh, and Z. N. Chen, "A segmented loop antenna for uhf near-field rfid," in *2010 IEEE Antennas and Propagation Society International Symposium*. IEEE, 2010, pp. 1–4.

[54] A. L. Borja, A. Belenguer, J. Cascon, and J. R. Kelly, "A reconfigurable passive uhf reader loop antenna for near-field and far-field rfid applications," *IEEE Antennas and Wireless Propagation Letters*, vol. 11, pp. 580–583, 2012.

[55] Y. Zeng, X. Qing, Z. N. Chen, and J.-M. Jin, "Directional uhf near-field rfid reader antenna with an improved magnetic field distribution," in *2016 IEEE Region 10 Conference (TENCON)*. IEEE, 2016, pp. 135–137.

[56] Z. D. Wang, Y. Z. Yin, X. Yang, and J. J. Wu, "Design of a wideband horizontally polarized omnidirectional antenna with mutual coupling method," *IEEE Transactions on Antennas and Propagation*, vol. 63, no. 7, pp. 3311–3316, 2015.

[57] Mike Lenehan, "Monza 4 datasheet – impinj support portal," https://support.impinj.com/hc/en-us/articles/202756908-Monza-4-Datasheet, accessed: 2024-02-27.

[58] Impinj, "Rain rfid tag chips for the internet of things," https://www.impinj.com/products/tag-chips, 2024, accessed: 2024-02-27.

[59] Alien Technology, "Tags," https://www.alientechnology.com/products/tags/, accessed: 2024-02-27.

[60] NXP Semiconductors, "Ucode rain rfid uhf," https://www.nxp.com/products/rfid-nfc/ucode-rain-rfid-uhf:MC_50483, 2023, accessed: 2024-02-27.

[61] Laxcen, "RFID," http://www.laxcen.com/rfid.html, 2023, accessed: 2024-02-27.

[62] Z. An, Q. Lin, L. Yang, and W. Lou, "Embracing tag collisions: Acquiring bloom filters across rfids in physical layer," in *Proc. of IEEE INFOCOM*, 2019, pp. 1531–1539.

[63] B. Sheng, Q. Li, and W. Mao, "Efficient continuous scanning in rfid systems," in *Proc. of IEEE INFOCOM*, 2010.

[64] L. Xie, Q. Li, X. Chen, S. Lu, and D. Chen, "Continuous scanning with mobile reader in rfid systems: An experimental study," in *Proc. of ACM MobiHoc*, 2013, pp. 11–20.

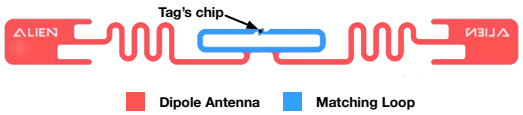


Fig. 24: Structure of a Typical UHF RFID Tag. It consists of an integrated chip, matching loop and a dipole antenna.

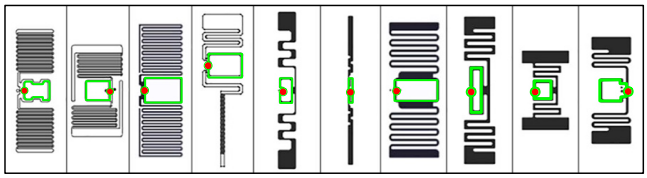


Fig. 25: The Popular UHF RFIDs. Each type of tag contains a matching loop (green) that bridges the dipole antenna with the chip.

Appdendix

A Magnetically-driven UHF RFID Systems

Without loss of generality, Fig. 24 displays a standard UHF RFID tag design, showcasing a small loop (highlighted in blue) adjacent to its chip. Consistently, as Fig. 25 further demonstrates, it is a fundamental trait for nearly every RFID tag to embody a single-turn coil, known as the matching loop. This loop is instrumental in the magnetic energy harvesting process within our system. To inventory such tags, the reader must adhere to the UHF RFID Gen2 air interface protocol, which involves several steps: (1) The process begins with the reader initiating an inventory session by sending out a `Select` command to select a group of tags for participation. (2) This is followed by a `Query` command to initiate a new frame, during which each tag that has not yet been identified chooses a random time slot to respond. (3) In its allocated slot, a tag first sends out a 22-bit short response (i.e., `RN16`) to aid in detecting signal collisions. (4) If the `RN16` is successfully decoded by the reader, it indicates a collision-free transmission from a single tag (known as a singleton slot). (5) Subsequently, the reader requests a longer, 128-bit response (i.e., `EPC`) by issuing an `ACK` command. Thus, a tag is required to first transmit an `RN16` response before it can send its full EPC response.

Inventory with the Coil Antenna. To test the viability of a magnetically-driven UHF RFID system, we initiated a preliminary experiment using the configuration depicted in Fig.5. An Impinj M4 tag [57] was placed 10cm away as part of the experimental setup. *The Impinj reader is equipped with the coil antenna for tag activation and querying*. Concurrently, the sniffer discreetly recorded the leaked RF transmissions. As depicted in Fig. 26, the normalized amplitude of the intercepted signal is showcased. The figure clearly delineates the reader's command signals and the tag's backscattered responses. The inventory process kicks off with a `Select` command, promptly followed by a `Query`. Responding to the query, the tag emits a `RN16` reply that includes a fixed 6-bit preamble for signal identification, succeeded by 16 random bits to facilitate channel contention, as illustrated in Fig. 6.

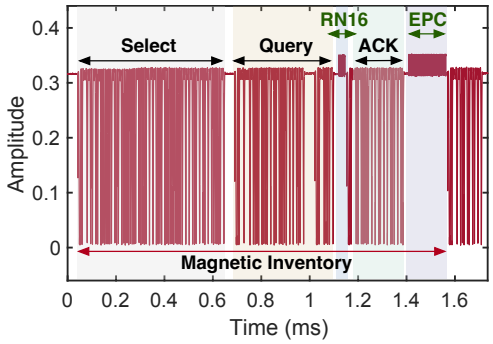


Fig. 26: RF signal acquired by the sniffer when the tag was queried by a magnetically-driven UHF RFID reader.

This is subsequently acknowledged by the `ACK` command, indicating the slot's availability. Conclusively, the tag broadcasts its `EPC`. This process closely parallels the inventory sequence observed in radiatively-coupled RFID Gen2 systems. It validates that COTS RFID tags can be activated and queried using magnetic fields while the communication protocol remains consistent.

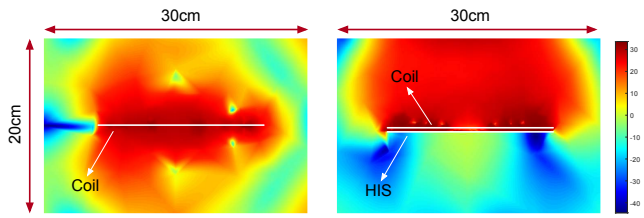
Versatility Validation. To underscore the versatility of the magnetically-driven approach, we further tested ten tags from leading manufacturers such as Impinj [58], Alien [59], NXP [60], and Laxcen [61]. Every tag is placed 50cm in front of RFID+ to assess its readability. Their performance is reflected in reading rates, outlined in Table. 2. Reading rates ranged from 105 to 129, with Impinj's H47 tag outperforming the rest and Alien's 9654 tag at the lower end. These differences are likely due to variations in the tags' internal loop structures. Broadly speaking, tags with larger loop diameters tend to register a heightened coupling coefficient, leading to better reading rates. However, every tag was effectively recognized by the magnetically driven reader, highlighting the universal efficacy of our magnetically-driven RFID solution.

B High-Impedance Surface

Fig. 27 showcases the simulated results of the HIS reflector, utilizing Ansys HFSS [13] as the simulation platform. Throughout the simulation process, the spiral antenna was strategically positioned a mere 10 mm ($\ll \lambda/4$) above the HIS substrate. In the absence of the reflector, as visualized in Fig. 27(a), the magnetic field displays a balanced distribution across both the superior and inferior facets of the an-

Table 2: The configuration and reading rate of different tags

Tag(#)	MFR.	IC	Model	Size(mm^2)	Reading rate (r/s)
T1	Impinj	Monza 4QT	H47	50×50	129
T2		Monza R6	ER62	74×18	119
T3	Alien	Higgs 3	9662	70×17	121
T4		Higgs 3	9640	94.8×8.25	126
T5		Higgs 3	9654	93×19	105
T6		Higgs 3	9962	73.5×20.2	117
T7	NXP	Ucode8	U9624	98×27	105
T8		UR108	U7015	70×15	126
T9	Laxcen	Monza 4QT	C90G	90×20	107
T10		Monza 5	C50D	50×30	106



(a) Without HIS (b) With HIS

Fig. 27: Simulated magnetic intensity without and with an HIS reflector attached to the spiral coil antenna. (a) shows the magnetic field distribution on the antenna's tangent radiation plane without HIS; (b) illustrates the magnetic field distribution with the HIS in place.

tenna. Yet, when the reflector is introduced beneath the UHF magnetic antenna, as illustrated in Fig. 27(b), there is a pronounced intensification of the field on the antenna's top surface, accompanied by a significant attenuation on its lower side. These results vividly demonstrate the prowess of the HIS-based reflector in steering the magnetic field, accentuating the radiation efficiency and gain in the desired direction while simultaneously attenuating undesired emissions. The incorporation of this reflector could lead to power conservation for the antenna due to its innate ability to enhance signal superposition constructively.

C Fundamentals of Bloom Filter

Utilizing a set of prefetched tags (i.e., candidate tags), there is no longer a need for exhaustive inventory processing in the near-field. Instead, we employ Bloom filters (BF) to swiftly ascertain the presence or absence of these candidate tags in the region of interest [26,62–64]. BF is a time-efficient probabilistic data structure that accurately represents the existing set of tags. As depicted in Fig. 28, a Bloom filter succinctly characterizes a set $T = \{t_1, t_2, \ldots, t_n\}$ comprising n tags through an array of M bits, which are initialized to 0. By leveraging K distinct hash functions, denoted as $\{h_1, h_2, \ldots, h_K\}$, each tag is mapped to an integer within the span of $\{1, \ldots, M\}$. For every tag t in T, the bits corresponding to $H = \{h_1(t), \ldots, h_K(t)\}$ are assigned the value 1. Even though a bit might encounter collisions, its value remains 1. To determine if a tag t resides in T, it suffices to verify whether all bits associated with $\{h_1(t), \ldots, h_K(t)\}$ are indeed 1. If even one isn't, then t is not a member of the set. As a case in point, tag t_2 is not part of T as, in the test BF, its seventh bit does not hold the value of 1. If all related bits are 1, we postulate that t is within T, albeit with a caveat: there is a minuscule chance (e.g., < 0.001) of misclassification, leading to a false positive. Moreover, if a particular bit is 0 in the candidate BF but transitions to 1 in the test BF, this alteration indicates the presence of a previously unaccounted-for tag in the ROI. For example, the last bit '1' in the test BF is caused by the uncollected tag t_{n+1} (i.e., $t_{n+1} \notin T$).

Time Efficiency. In less complex situations (e.g., $n < 100$), using a dictionary search method suffices to meet the goals, specifically enabling the system to expedite the inventory

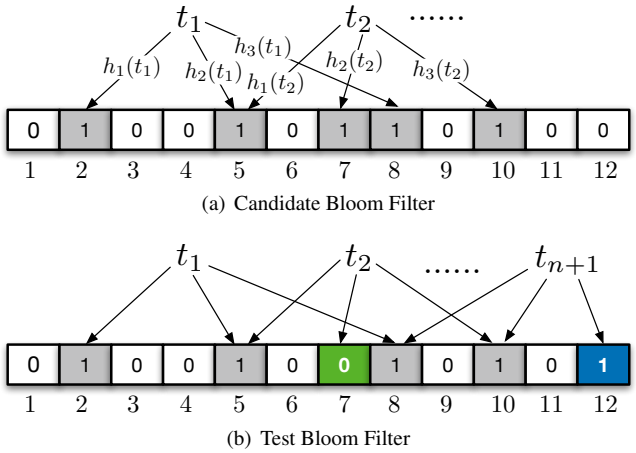


Fig. 28: Fast Inventory with Bloom Filters. (a) shows a candidate Bloom filter, which is a concise bitmap representing the collected candidate tags, acquired by the far-field reader. (b) shows the test Bloom filter acquired from the near-field reader on site. By comparing these two bitmaps, we can swiftly discern the tags residing within the ROI. For example, the tag t_2 is a cross-reading, i.e., absent from the ROI, while tag t_{n+1} represents a tag undetected by the far-field reader.

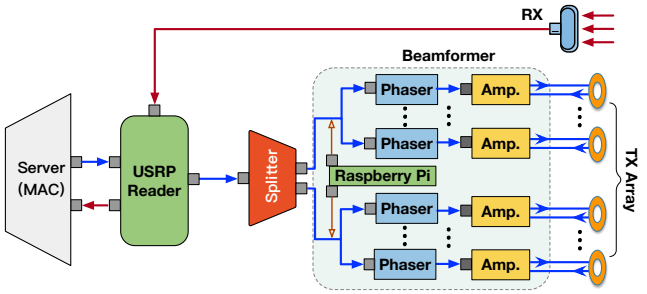


Fig. 29: Illustration of Near-Field Reader

process through straightforward dictionary queries. However, with the increase in the number of tags, the search time complexity of a dictionary scales linearly as $O(n)$. On the other hand, BFs offer efficient lookups for tag presence, maintaining fast query times regardless of the dataset's size n, with a complexity of $O(\log_2(n))$. Therefore, BF is ideally suited for rapid inventory applications in warehouses.

D Architecture of Near-Field Reader

The detailed architecture of the near-field reader is illustrated in Fig. 29. After undergoing upconversion in the USRP, the emergent signal is bifurcated, creating four separate paths. These distinct routes are meticulously controlled by analog phasers, specifically the PHSA-152+ model from Mini-Circuits [28]. These state-of-the-art phasers grant a high level of precision in phase adjustments, offering an impressive eight-bit granularity. Following this, the signals are subjected to an amplification process, ensuring they achieve a robust transmission power of 30 dBm. Subsequently, these fortified signals are disseminated by an ensemble of four coil antennas. To streamline and synchronize the operations of the phaser units, a Raspberry Pi 4 Model B [29] is strategically deployed, acting as the central coordination hub with the aid of a cus-

tomized Serial-In to Parallel-Out (SIPO) Converter.

E Compared with Related Systems

In a parallel comparison, RFID+ outperforms the state-of-the-art RFID inventory solution RFGo [8] with respect to generality, cost, and deployment ease: First, RFID+ enables seamless plug-and-play functionality in dynamic environments without the extensive data collection and training RFGo requires. Moreover, RFID+ proves to be more budget-friendly, employing a limited number of lower-cost hardware components like magnetic coils, phase shifters, and HIS reflectors (see cost breakdown in Tab. 3), in contrast to RFGo's intricate setup involving 11 USRP X310 units and numerous antennas. Lastly, setting up RFID+ is straightforward, requiring only two perpendicular surfaces, unlike RFGo's complex three-dimensional antenna configuration.

Table 3: Pricing Estimation for BOM List of One Antenna Unit

Item (#)	Component	Description	Quantity	Price ($)
1	Coil Antenna	Two-layer PCB	1	5.42
2	HIS Reflector	Two-layer PCB	1	3.06
3	RF Amplifier	SKY65111-348LF	1	2.64
4	Phase Shifter	SPHSA-152+	1	50.41
5	Micro Controller	Raspberry Pi Pico	1	4.00
Total Cost				**65.53**

SMUFF: Towards Line Rate Wi-Fi Direct Transport with Orchestrated On-device Buffer Management

Chengke Wang[ℙ], Hao Wang[𝕊], Yuhan Zhou[ℙ], Yunzhe Ni[ℙ], Feng Qian[ℂ], Chenren Xu[ℙℤ𝕂*]

[ℙ]*School of Computer Science, Peking University* [𝕊]*Shenzhen Kaihong Digital Industry Development Co., Ltd.*

[ℂ]*University of Southern California* [ℤ]*Zhongguancun Laboratory*

[𝕂]*Key Laboratory of High Confidence Software Technologies, Ministry of Education (PKU)*

Abstract – Wi-Fi direct transport provides versatile connectivity that enables convenient data sharing and improves the productivity of mobile end users. However, as today's smartphones are capable of near-Gbps wireless data rates, current solutions do not efficiently utilize the available bandwidth in this single-hop environment. We show that existing transport schemes suffer from resource-intensive reliable delivery mechanisms, inadequate congestion control, and inefficient flow control for achieving line-rate transmission in peer-to-peer Wi-Fi direct links. In this paper, we present SMUFF, a reliable file transfer service that achieves nearly the practical line rate of the underlying wireless bandwidth. We note a unique feature of direct transport – the sender can monitor each buffer along the data path and determine an optimal sending rate accordingly. Therefore, SMUFF can maximize throughput by strategically backlogging the appropriate amount of data in the bottleneck buffer. We have deployed SMUFF on four different phone models, and our evaluations with other transport schemes show that SMUFF achieves up to 94.7% of the practical line rate and 22.6% throughput improvement with a 37% reduction in CPU usage and a 15% reduction in power consumption, compared to state-of-the-art solutions.

1 Introduction

Peer-to-peer wireless connectivity is compelling for scenarios where quick and easy data sharing is critical. It not only improves data transport efficiency by reducing protocol overhead and backhaul traffic but also prevents potential data privacy leakage from the cloud. The industry has taken note of these advancements. An increasing number of smartphone manufacturers are introducing their own file transfer solutions, such as Apple AirDrop [1, 2], Huawei Share [3], Xiaomi Mobile Direct Fast Exchange [4], and Samsung Quick Share [5]. Looking ahead, these innovations will play a crucial role in supporting emerging mobile device applications. For instance, a VR headset may seamlessly load a 2 GB game asset from a PC [6], autonomous vehicles may exchange road traffic information via peer-to-peer wireless links [7], and devices involved in distributed ML training within robotic IoT networks or mobile federated learning may need to exchange model gradients over the wireless channel [8–10]. With such a wide range of applications, there is a need to unleash the full potential of peer-to-peer wireless data transfer.

With the ubiquitous deployment of Wi-Fi networks, Wi-Fi Direct [11] is an attractive local wireless data transmission technology to provide direct connectivity between devices. The standard claims that 802.11ac (Wi-Fi 5) already provides theoretical physical layer bandwidth up to 867 Mbps, and the practical wireless line rate can reach nearly 700 Mbps (§2.2). However, TCP (CUBIC), the most widely used transport scheme, averages only 442 Mbps on Wi-Fi Direct links, and other commercial solutions perform even worse (§2.3). Our further investigation reveals three key reasons that prevent the current transport schemes from achieving wireless line rate performance: *i)* The current reliable delivery mechanism uses a per-packet ACKing policy, which exacerbates channel contention on Wi-Fi links, and the compute-intensive network stack imposes more packet processing overhead. Both of these drawbacks significantly reduce throughput. *ii)* Today's widely deployed TCP (or QUIC) relies primarily on congestion control algorithms (CCAs) to probe and estimate the network bandwidth and adjust the sending rate, which is unsuitable for the one-hop single-flow scenario as it exhibits unnecessary startup phase and overreaction to the lossy wireless link. *iii)* The flow control mechanisms in existing transport schemes are inefficient in achieving line rate transmission in Wi-Fi Direct links as they are unaware of some on-path buffers and suffer from delayed ACK feedback and fixed parameter configurations. All of the above factors contribute to the reduced link bandwidth utilization.

In the presence of the above problems, we notice a distinctive characteristic of peer-to-peer direct data transmission that can be exploited to achieve high link utilization. Specifically, we are able to monitor the state of each individual buffer in the whole packet lifecycle because the two communicating devices are the only entities involved. This is in sharp contrast to a data center or Internet connection that relies on intermediate nodes (routers and switches) to forward the traffic – in this case, the sender typically needs to use a sophisticated CCA to *indirectly* infer the buffer states that are not available to the end host [12, 13] or obtain buffer states with the help of in-network devices [14–16]. In our scenario, the *direct* access to buffer states offers a unique opportunity to achieve a high data rate by backlogging data in the buffers.

*✉: chenren@pku.edu.cn

However, designing an efficient mechanism allowing a receiver to report the buffer state to the sender is not straightforward. First, existing flow control shipped with today's TCP and QUIC uses a per-packet ACK to allow the receiver to report its available space to the sender. However, a dense ACK stream can significantly reduce the throughput of the high-speed wireless direct connection [17,18]. Besides, there exists a non-negligible control latency between the buffer state report and sender decision-making even in our one-hop scenario (§3.3.1). Therefore, the sender can only receive infrequent feedback from the receiver and has to strategically determine a sending rate according to out-of-date buffer states. Furthermore, although our scenario only involves two entities, there are multiple buffers along the data path (§2.1). The sender thus needs to coordinate these buffers and ensure its sending rate does not cause underflow or overflow in any buffer.

In this paper, we present SMUFF, a file transfer service that improves Wi-Fi Direct transport throughput to line rate by orchestrating the on-device buffers. We model the transport data path as a series of linearly connected buffers, and consider the traffic as a fluid that sequentially traverses each buffer. Our core idea is to identify and address the bottleneck component within this data path. To maximize throughput, we maintain an appropriate backlog of data in the bottleneck buffer. The amount of backlogged data must satisfy two constraints: *i)* Enough data must be backed up to prevent a buffer underflow, *i.e.,* an empty router queue results in throughput less than the link rate; and *ii)* sufficient buffer space must be reserved to avoid buffer overflows, which can cause packet drops and complicate the subsequent packet recovery process. This leads us to define a "safe range" for the amount of buffered data. This range ensures that even in situations where state information is infrequently updated or out of date due to limited feedback, the buffer remains protected from both underflow and overflow. To accomplish this, SMUFF systematically collects state information from on-path buffers and calculates an optimal sending rate. This rate is carefully tuned to keep the bottleneck buffer size within the safe range.

SMUFF requires no proprietary hardware or modification to the device kernel, making it portable across different device models and easy to deploy. We have deployed the SMUFF service on four different Android mobile phone models with release dates spanning five years. Our evaluation shows that SMUFF achieves an average link utilization of 88.7% and 91.8% for 802.11ac and 802.11ax, respectively. It improves the link utilization by up to 22.6% while reducing CPU utilization by 37% and energy consumption by 15%, compared to the state-of-the-art solution.

Our contributions are as follows:

- We reveal the fundamental limitations of existing end-to-end transport schemes for Wi-Fi Direct. These schemes designed for multi-hop networks suffer from link under-utilization in peer-to-peer wireless data transfer.
- We design SMUFF, a file transfer service that is dedicated to Wi-Fi Direct towards practical line rate. SMUFF maximizes throughput by deriving the optimal sending rate according to the buffer states.
- We implement SMUFF to be compatible with Android and demonstrate its ability to reach up to 94.7% utilization of the wireless line rate.

This work does not raise any ethical issues.

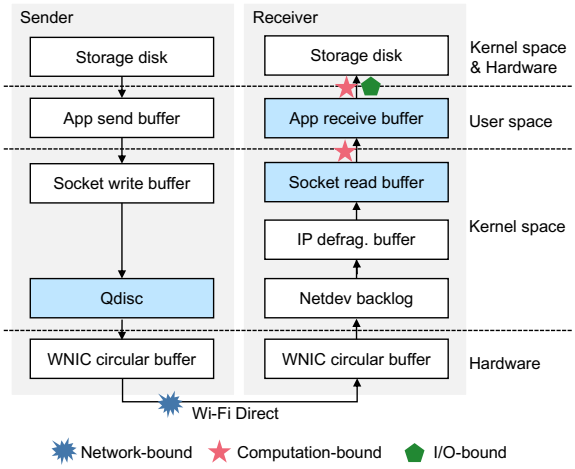


Figure 1: The end-to-end file transfer path. The Qdisc buffer, the socket read buffer, and the application receive buffer (highlighted in blue) are vulnerable to overflows.

2 Background and Motivation

In this section, we introduce the background and data packet lifecycle of Wi-Fi Direct (§2.1). We then examine the achievable wireless line rate (§2.2) and the limitations of existing transmission schemes that prevent them from achieving the line rate of peer-to-peer direct links (§2.3).

2.1 Wi-Fi Direct and Its Packet Life-Cycle

Wi-Fi Direct is a link layer (L2) technology that allows mobile devices to establish a direct Wi-Fi connection with each other directly, without the need for a central access point or any intermediate nodes. This technology is primarily used for improving daily life productivity such as file transfer between devices [20]. Note that it only provides L2 connectivity and does not specify any transport protocols or default applications. Device vendors need to choose or implement upper-layer protocols to support reliable data transfer.

The end-to-end pipeline of a file transfer task on a typical Android device (using Wi-Fi Direct) is shown in Fig. 1. To simplify the illustration and bypass the heavy TCP network stack, we use the UDP socket as an example.

Sender. The sender reads data from the disk and encapsulates it into a series of UDP datagrams. The datagrams are then sent from userspace and enqueued into the queueing discipline (Qdisc), which is a software queue that allows traffic shaping and prioritization. Note that for UDP, the socket write buffer is a virtual buffer that only counts memory allocation but has

Buffer Type	Overflow Risk Analysis
Application Send Buffer	No, the application can stop sending data when the buffer reaches its capacity.
Socket Write Buffer	No. For TCP, the kernel sets the socket as non-writable when the buffer is full. For UDP, the socket write buffer is a virtual buffer that only tracks packet memory allocation.
Qdisc	Yes, this buffer is shared among all traffic and may overflow if the Wireless Network Interface Card (WNIC) cannot drain it quickly enough.
WNIC Circular Buffer	No. At the sender, the NIC driver notifies the kernel when the firmware buffer is full. At the receiver, the kernel prioritizes the interrupt handler responsible for receiving data.
Netdev Backlog	Almost never, as the high-priority interrupt handler can process data.
IP Defragmentation Buffer	No overflow in single-hop scenarios. Overflow typically results from Denial-of-Service (DoS) attacks [19].
Socket Read Buffer	Yes, if the receiving process lacks sufficient CPU time to copy data from it.
Application Receive Buffer	Yes, if the receiving system is I/O-bound.

Table 1: Analysis of buffer overflow risk in the Wi-Fi Direct transport data path.

no real queue to buffer data. The kernel blocks the application from sending more data when the memory counter reaches a memory allocation threshold. Finally, the WNIC driver fetches packets from the Qdisc and sends them to the peer.

Receiver. The received packets are stored in the network device queue (*i.e.,* netdev backlog) by the interrupt handler and then processed by the kernel. After processing the received data, the kernel stores the UDP datagrams in the read buffer and waits for the userspace program to read them. The receiving application reads the data from the socket read buffer, performs packet reordering and loss recovery, and asks the kernel to write to disk.

Buffer Overflow Analysis. Buffer overflow is the primary cause of packet drops when we send UDP packets at a high rate. Table 1 gives an analysis of overflow risk. We need to consider these buffers when designing our system. We identify three buffers that are susceptible to overflow because of bounded device capability, as highlighted in blue in Fig. 1: *i) Qdisc*. Packet drops happen at Qdisc when the sending rate is faster than the available wireless bandwidth and the queue is full. The bounded network capacity cannot drain the queue fast enough; *ii) Socket read buffer.* When the system is under heavy load, or the processor clock frequency is constrained by the mandatory thermal throttling mechanism to avoid overheating, the receiving application process may not have sufficient CPU time slices to copy the data from the kernel. The packets could drop if the socket read buffer is full; *iii) Application receive buffer.* Like the socket read buffer, the application buffer is prone to overflow when there is no available CPU time. This buffer also overflows when the disk I/O rate cannot keep up with the receiving rate.

In a nutshell, from a buffer management perspective, we need to carefully pace the sending rate to avoid overflow in any of these buffers, preferably at the upper layer (*e.g.,* userspace transport) instead of inside the kernel or firmware in WNIC for the sake of programmability and deployability.

2.2 Practical Line Rate Transmission of Wi-Fi Direct

Theoretical transmission rate cannot be achieved in the real world. While Wi-Fi standards such as 802.11ac (Wi-Fi 5) and 802.11ax (Wi-Fi 6) claim impressive physical bandwidth limits[1] of up to 867 Mbps and 1.2 Gbps, respectively [21, 22], real-world peak throughput falls far short of these advertised rates. The gap between the line rate and the physical rate is inevitable for two reasons: *i) Protocol overhead.* Network protocols from the MAC layer to the transport layer require the process of header information in each data packet. This additional overhead reduces the effective throughput that can be achieved; *ii) Channel contention*. In areas with a high density of wireless devices or networks, Wi-Fi devices adhere to Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) mechanism [22] when accessing the wireless channel, resulting in decreased throughput [23]. To illustrate the discrepancy between the achievable transmission rate and the theoretical rate, we conduct an experiment using two 802.11ac supporting devices with a maximum physical bandwidth limit of 867 Mbps. We use the `iperf` tool [24] with fine-tuned parameters and the `iw` tool [25] to monitor the physical layer data rate reported by the wireless NIC (WNIC). As shown in Fig. 2, for the first 30 seconds we place the two devices head-to-head, allowing them to reach the maximum achievable throughput. However, although the physical layer rate consistently reaches its theoretical maximum, the actual throughput fluctuates and remains below that rate. From 30 seconds to 60 seconds, we move the devices 5 meters apart while keeping them stationary. This change caused instability in the data rate due to increased collisions and retransmissions caused by other devices sharing the same wireless channel. Between 60 seconds and 90 seconds, we deliberately introduce channel contention by running TCP traffic on additional devices operating on the same Wi-Fi channel. The actual throughput is well below the physical layer rate.

Defining practical line rate transmission. While the physical layer rate cannot serve as the metric of practical transmission rate due to protocol overhead and channel contention, previous research has explored techniques for estimating this

[1] From a wireless communication perspective, it is the instantaneous highest data rate based on the modulation and coding scheme determined by the channel condition (*e.g.,* Signal to Interference plus Noise Ratio, or SINR) and channel bandwidth according to the Wi-Fi specification.

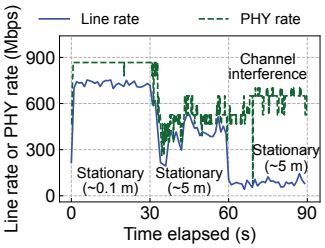


Figure 2: Traces of the physical rate and the actual UDP throughput.

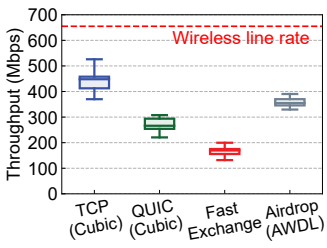


Figure 3: The Wi-Fi Direct link throughput of two 802.11ac devices 3 meters apart.

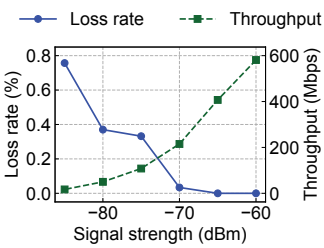


Figure 4: Wi-Fi Direct cannot guarantee packet loss recovery (at lower layer).

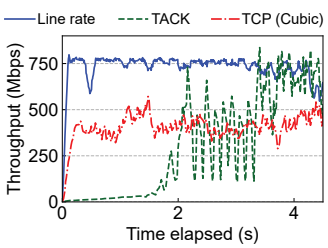


Figure 5: Congestion control fails to (quickly) achieve high utilization of a wireless link.

rate using specific hardware [15, 26]. However, these estimates are inherently approximate and hardware dependent. In the rest of this paper, we take the empirical approach in our experiment and define *practical line rate* as the maximum instantaneous throughput achievable by `iperf` UDP over a one-second window. Such a definition provides a practical and real-time measure of network performance and is a valuable metric for our analysis.

2.3 Limitations of Existing Transport Schemes

Existing peer-to-peer transmission schemes fail to achieve line rate. While the achievable line rate is below the claimed physical line rate, existing peer-to-peer transmission schemes still cannot achieve such a line rate over Wi-Fi Direct link. As shown in Fig. 3, we initiate data transmission tasks over a Wi-Fi Direct link with two transport layer schemes (TCP CUBIC and QUIC CUBIC) and two commercial services (Xiaomi Fast Exchange and Apple AWDL). While the wireless line rate is 655 Mbps in this setting, TCP (CUBIC), as the most widely used transport solution, only achieves 442 Mbps on average. QUIC and other commercial solutions perform even worse. This pilot study indicates a great opportunity to design a dedicated transport scheme for Wi-Fi Direct towards its wireless line rate. In the rest of the section, we investigate the drawbacks of basic transport layer elements in Wi-Fi Direct links and their implications on a dedicated transport solution.

Resource-Intensive Reliable Delivery Mechanism. The Medium Access Control (MAC) layer of Wi-Fi performs "best effort" retransmission upon packet loss[2]. Therefore, Wi-Fi Direct is only partially reliable, and packet loss over a wireless channel is unavoidable. We conduct a Wi-Fi Direct data transfer experiment and confirm that packet loss is not negligible when the signal is "weak" (*i.e.,* below -70 dBm), as shown in Fig. 4, which means that the reliable delivery (including loss recovery and packet reordering) is still a key requirement for Wi-Fi Direct and should be handled by the transport layer. TCP is widely used for reliable delivery, but it has the following two drawbacks when applied to Wi-Fi Direct:

• *Per-packet ACKing intensifies channel contention.* A recent study [18] shows that over a Wi-Fi link, TCP's default ACKing policy (*i.e.,* at least one ACK every other packet [28]) would hurt the throughput by 33% compared to ACKing every 16 packets. This is because uplink and downlink data in a shared wireless channel will cause extra contention and MAC protocol overhead, suggesting a reduced ACK frequency for a delicate transport layer design.

• *Computation-intensive network stack stresses computation resource.* The TCP stack is known to be computationally intensive and causes more packet processing overhead [29]. More recent transport protocols such as QUIC opt for high efficiency based on UDP, which bypasses the TCP stack and gives more flexibility to the user space. Therefore, we need a lightweight transport protocol based on UDP datagrams for its efficient in-kernel implementation.

Unsuitable Congestion Control. TCP or QUIC relies heavily on congestion control algorithms (CCAs) to adjust the sending rate. Basically, the goal of CCA is to evenly distribute bandwidth among the flows traversing a bottleneck link in a multi-hop path. However, CCAs are unsuitable in the one-hop, single-flow setting in Wi-Fi Direct links and face the following two problems:

• *Unnecessary Startup Phase.* CCAs need to probe the available bandwidth and send packets conservatively to avoid congestion, adding unnecessary start time to quickly reach the peak sending rate. Our experiment in Fig. 5 shows that Cubic spends 0.15 seconds on the slow start phase, while TACK [18], a TCP variant that reduces ACK frequency to alleviate channel contention, spends 3.3 seconds on the congestion avoidance phase. In the peer-to-peer scenario, the slow start should be avoided because of the absence of congestion.

• *Overreact to Lossy Link.* CCAs could reduce the congestion windows to lower the sending rate due to packet loss or transient delay spikes, which are common on wireless links. This overreaction leads to link under-utilization. As shown in Fig. 5, both CCAs cannot reach the available link bandwidth and the throughput oscillates due to wireless link fluctuation. A wireless transport layer should be insensitive to a lossy link.

Inefficient Flow Control. Flow control is designed to prevent the sender from overwhelming the receiver. To achieve line

[2]It retries for failed transmission until the attempt reaches `dot11ShortRetryLimit` (7 by default) or the retransmission time exceeds `dot11MaxTransmitMSDULifetime` (512 ms by default) [27].

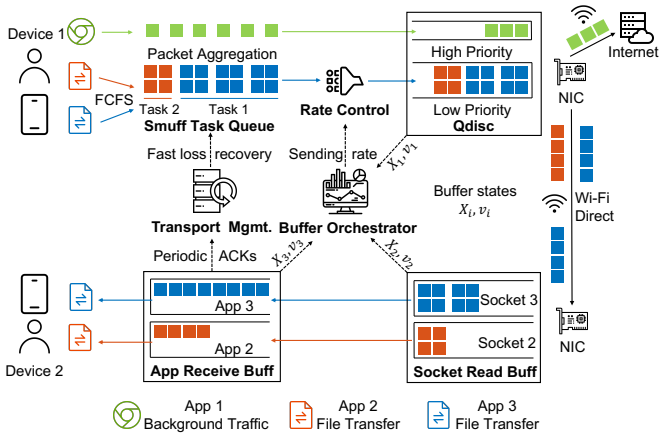


Figure 6: SMUFF Overview.

rate transmission in Wi-Fi Direct, more delicate flow control should be used because *i)* the sender needs to decide a proper packet sending rate to achieve line rate while avoiding packet drops. An insufficient sending rate will not fully utilize the WNIC potential, causing link under-utilization, while an excessive sending rate may cause packet drops at the sender Qdisc due to the limited queue length. *ii)* With Gigabit wireless hardware equipped, the receiver could fall short of handling such a high speed due to CPU and I/O bottlenecks and incur buffer overflow. The existing flow control in TCP and QUIC is inefficient in the following three aspects:

• *Unaware of other On-path Buffers.* Conventional TCP flow control only considers transport-layer buffers and is agnostic to other buffers in the packet life-cycle – underflow or overflow on any of them can still cause link under-utilization – a new flow control design should take all buffers into account.

• *Delayed ACK Feedback.* The flow control window in TCP and QUIC assumes ACK-driven updates, which may be delayed due to a longer packet queue or ACK frequency reduction [18]. The window may be slow to adjust to the fluctuating wireless link due to delayed feedback. The effectiveness of flow control should not depend on timely ACKs.

• *Fixed Parameter Configuration.* The maximum flow control window size is a fixed value during transmission [30, 31] that varies between implementations and device configurations. Such a value will not be ideal for other buffers of different sizes in the packet lifecycle and different link capacities. Flow control should better handle all of this heterogeneity, dynamics, and complexity.

3 SMUFF Design

In this section, we propose SMUFF, a Wi-Fi Direct file transfer service to achieve line rate by orchestrating on-device buffers. We first present the design goals of SMUFF.

3.1 Design Goals

The main idea of SMUFF is to orchestrate multiple buffers by actively monitoring on-device buffer states and continuously obtaining an optimal sending rate. We design SMUFF to achieve the following three goals.

G1: Link Bandwidth Utilization Maximization. As presented in §2.3, existing transport solutions often fall short of achieving line rate transmission. SMUFF should be designed to make the full utilization of available Wi-Fi direct link bandwidth by dynamically adapting its sending rate at the transport layer to saturate the network interface.

G2: On-path Buffer Overflow Avoidance. A straightforward idea is to continuously send packets to saturate the link. However, this may cause unnecessary packet loss due to buffer overflow and complicate the loss recovery process. SMUFF should effectively manage the critical packet buffers along the data path by monitoring the buffer states and making timely adjustments to the sending rate to avoid packet loss.

G3: Practical Deployment on Mobile Devices. SMUFF is designed for easy portability to different phone models. To make SMUFF practical for deployment, it should not rely on vendor-specific hardware (*e.g.,* proprietary NICs). In addition, it must have a low CPU usage footprint and minimize its impact on concurrent flows on the same devices.

3.2 System Overview

SMUFF is designed as a system service, and an overview of its workflow and main system components are shown in Fig. 6. This service provides a platform for applications on the sender side to submit file transfer tasks, which are then managed and executed by SMUFF on a first-come, first-served (FCFS) basis. SMUFF needs to efficiently transfer the files to the corresponding application on the receiving device.

In the rest of this section, we first present our theoretical analysis to achieve line rate data delivery (§3.3). We then put our analysis into practice with two key system components: *Buffer orchestrator* (§3.4) collects buffer states and calculates an optimal sending rate to maximize system throughput; *Transport manager* (§3.5) ensures reliable data delivery and facilitates packet loss recovery.

3.3 Buffer Management Analysis

As shown in §2.1, data is repeatedly processed and transferred to the next buffer until it reaches its final destination. This property allows us to take a buffer-by-buffer approach for flow management. For ease of exposition, we will start with the management of a single buffer (§3.3.1) and then extend the methodology to multiple buffers (§3.3.2).

3.3.1 Single Buffer Management

To derive the optimal sending strategy, we begin with the simple case where there is only one buffer as shown in Fig. 7. The packet buffer stores the incoming data until the data can be processed by the next system component. We suppose the maximum size of the buffer is $X_{\max}$. We denote the packet ingress and egress rate as v_{in} and v_{out} respectively and denote the current amount of data in the buffer as X. The main purpose of the buffer is to serve as a cushion to absorb the

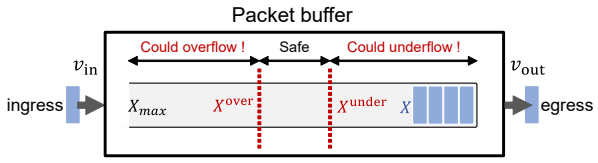


Figure 7: Single buffer management.

variation in the ingress and egress rate. In order to make full use of the buffer, we translate the design goals G1 and G2 into the following two constraints.

C1: Backlog enough data to avoid buffer underflow. To keep the system working at its full rate, the buffer needs to be backlogged (*i.e.,* the buffer never runs out of data). If the buffer goes empty, the egress rate will decrease and fail to achieve the line rate.

C2: Reserve enough buffer room to avoid buffer overflow. To prevent data loss, it is also necessary to allocate sufficient buffer space to handle incoming data. A buffer overflow occurs when the data pushed into the buffer exceeds its capacity, resulting in potential loss of information and disruption of system functionality.

Challenge Brought by Control Latency. Both C1 and C2 can be easily satisfied if we are able to control the ingress rate v_i just in time. We can increase v_i when the buffer is about to underflow and decrease it when the buffer is about to overflow. However, the conventional assumption of low-latency data delivery in single-hop wireless networks does not hold for the line rate transmission. There is a non-negligible *control latency* in buffer management, which we define as the time interval between buffer state reporting and the rate adjustment taking effect. For example, the buffer states at the receiver must be relayed to the sender, and the control latency is dominated by the RTT of the data flow. To demonstrate such latency, we initiate TCP connections with an increasing sending rate over an 802.11ac Wi-Fi direct link and obtain RTT from the kernel data structure. As shown in Fig. 8, the RTT increases up to 25 ms with the increase in the system throughput even in this single-hop setting. This intriguing phenomenon finds its roots in the well-documented bufferbloat issue [32]. Notably, this issue becomes increasingly pronounced as throughput scales to larger proportions, contrasting the experiment result in previous work [15,33,34] where the latency is no more than a few mill-seconds due to a low throughput of much less than 20 Mbps. Therefore, in the presence of the control latency, we need to strategically select the ingress rate v_i to maintain the right amount of data in the buffer.

Buffer Occupancy Estimation. The change in buffer occupancy X depends on the delta of the ingress rate and the egress rate. This can be formulated as:

$$\dot{X} = v_{\text{in}} - v_{\text{out}} \tag{1}$$

where $\dot{X}$ is the derivative over the time t (*i.e.,* $\dot{X} = \frac{\mathrm{d}}{\mathrm{d}t}X$). Suppose at time t_0, the buffer reports its current states, including $X(t_0)$, $v_{\text{in}}(t_0)$, and $v_{\text{out}}(t_0)$. The rate adjustment takes effect at a later time $t_1 = t_0 + d$ with a control delay d. Therefore, we need to estimate the buffer occupancy during t_0 to t_1 based on the reported values to adjust the sending rate at t_1. We make estimations by exploiting the fact that there exists a lower and an upper bound for the processing rate of any system components. For example, the wireless transport rate must be greater than zero and less than the physical limit. Therefore, we can assume a value range for $v_{\text{in}}(t)$ and $v_{\text{out}}(t)$ for the duration from t to $t+d$:

$$v_{\text{in}}^{\text{lo}}(t) \le v_{\text{in}}(t) \le v_{\text{in}}^{\text{up}}(t) \tag{2}$$

$$v_{\text{out}}^{\text{lo}}(t) \le v_{\text{out}}(t) \le v_{\text{out}}^{\text{up}}(t) \tag{3}$$

As a result, we can infer the value range of $\dot{X}$ by Eqn. 1, Eqn. 2, and Eqn. 3:

$$v_{\text{in}}^{\text{lo}}(t) - v_{\text{out}}^{\text{up}}(t) \le \dot{X} \le v_{\text{in}}^{\text{up}}(t) - v_{\text{out}}^{\text{lo}}(t) \tag{4}$$

In order to satisfy C1, we must have a sufficient amount of data in the buffer. From Eqn. 4 we know that from time t_0 to t_1, the maximum amount of data that can be consumed from the buffer is

$$X^{\text{under}} = -d \times (v_{\text{in}}^{\text{lo}}(t_0) - v_{\text{out}}^{\text{up}}(t_0)) \tag{5}$$

As a result, if the data in the buffer is less than X^{under} at t_0, the buffer may run out of data, resulting in underutilization.

In order to satisfy C2, we need to reserve enough room in the buffer. Similarly, the maximum amount of data that can be backlogged from t_0 to t_1 is $d \times (v_{\text{in}}^{\text{up}}(t) - v_{\text{out}}^{\text{lo}}(t))$. Therefore, to avoid overflow and packet loss, the data in the buffer at t_0 should be kept below a threshold

$$X^{\text{over}} = X_{\text{max}} - d \times (v_{\text{in}}^{\text{up}}(t_0) - v_{\text{out}}^{\text{lo}}(t_0)) \tag{6}$$

To satisfy both constraints, the buffer occupancy X has a "safe" range between two thresholds X^{under} and X^{over} at t_0, as shown in Fig. 7. If X falls within this safe range, we can ensure that the next system component will consistently operate at full speed and that no packet loss will occur for the next time period of control latency d. The existence of this safe region depends on $X^{\text{under}} < X^{\text{over}}$, meaning that the buffer size X_{max} must exceed a certain threshold, specifically $d \times (v_{\text{out}}^{\text{up}}(t_0) - v_{\text{in}}^{\text{lo}}(t_0) + v_{\text{in}}^{\text{up}}(t_0) - v_{\text{out}}^{\text{lo}}(t_0))$.

3.3.2 *Multiple Buffer Management*

There are typically multiple buffers (or queues) connecting different system components in a transport system. The data starts from the sender, gets processed and traverses each component and buffer, and arrives at the receiver. In our case as shown in Fig. 9, the Qdisc at the sender-side stores the packets sent by the application and waits for the network interface driver to fetch data to send. Also, the application buffer stores the data that is read from the transport layer and waits for the I/O requests to be ready.

While there are three buffers in our Wi-Fi direct use case, this idea can be generalized to more buffers. Without loss of generality, suppose there are n buffers. We let X_i denote the buffer occupancy (*i.e.,* buffer size or queue length) of the i-th

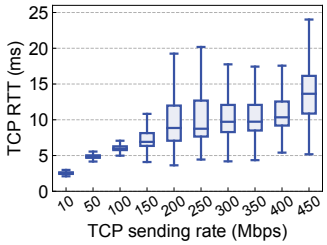


Figure 8: TCP control latency (RTT) increases as system throughput increases.

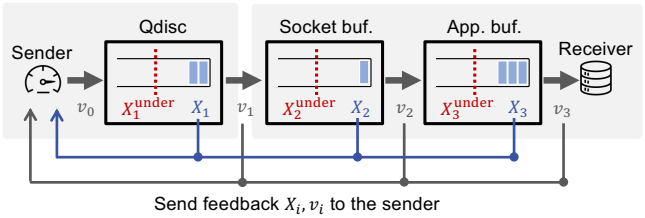


Figure 9: Multiple buffer management for the transport over the Wi-Fi direct connection.

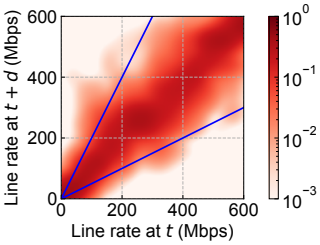


Figure 10: Throughput transition probability within a control latency ($d = 40$ ms).

buffer. We let v_i denote the instantaneous data rate from the i-th buffer to the $(i+1)$-th buffer. Note that v_0 and v_n are the sending and receiving rates, respectively.

Our goal is to adjust the sending rate to maximize the overall throughput of the system. While the overall throughput is the processing rate of the bottleneck system component, our idea is to identify the bottleneck component and maintain a moderate backlog of data in the buffer connected to that particular component. Our algorithm to determine the sending rate v_0 works in two steps: *i)* calculate the target amount of data to backlog for each buffer; *ii)* adjust the sending rate to maintain enough data at the bottleneck buffer.

Calculating the target backlog size. We need to ensure that the backlog size remains within the safe range at all times. To minimize the buffer usage, we set the *target backlog size* X_i^{target} for the i-th buffer to the lower bound of the safe range (X_i^{under}). On each latest feedback signal (v_i, X_i) from the i-th buffer, we update X_i^{target} according to Eqn. 5 as follows:

$$X_i^{\text{target}} = -d_i \times \begin{cases} (v_{i-1}^{\text{lo}} - v_i^{\text{up}}) & \text{if } i > 1 \\ -v_i^{\text{up}} & \text{if } i = 1 \end{cases}, \tag{7}$$

where d_i is the control latency for the i-th buffer, $v_i^{\text{lo}}, v_i^{\text{up}}$ are the lower bound and upper bound of v_i. We leave the settings of $v_i^{\text{lo}}, v_i^{\text{up}}$, and d_i in practice to §3.4. Note that we can also set the target backlog size X_i^{target} to any other value within the safe range (for example, the midpoint of the safe range $(X_i^{\text{under}} + X_i^{\text{over}})/2$).

Calculating the sending rate. For each feedback signal, the sender determines the sending rate based on the following:

$$v_0 \leftarrow \max\left(\min_{1 \le i \le n}\left\{v_i + \frac{X_i^{\text{target}} - X_i}{d_i}\right\}, 0\right), \tag{8}$$

The interpretation of this rule is as follows. For the i-th buffer, when the current buffer occupancy is less than the target buffer backlog size ($X_i < X_i^{\text{target}}$), this buffer could underflow, resulting in link underutilization. Therefore, we need to increase the rate v_i to increase the buffer occupancy in a control delay time d_i. Similarly, when the buffer occupancy exceeds the target backlog size ($X_i > X_i^{\text{target}}$), we decrease the sending rate. The outer min operator is used to identify the bottleneck of the whole system.

3.4 Buffer Orchestrator

The buffer orchestrator bridges the gap between theoretical buffer management analysis (§3.3) and practical implementation. Effective buffer management depends on the ability to quickly and accurately estimate system conditions. Therefore, the buffer orchestrator needs to perform two tasks: continuously monitoring buffer states and dynamically adjusting the sending rate.

Monitoring the Buffer States. There are four variables that characterize a buffer state.

- Buffer Occupancy (X): This variable indicates the current amount of data in the buffer and can be obtained directly by reading from the buffer.
- Data Consumption Rate (v): This variable represents the rate at which data is being consumed from the buffer. It is estimated over the duration of the control delay and further smoothed using an Exponentially Weighted Moving Average (EWMA) filter.
- Lower and Upper Bounds of the Data Rate (v^{lo} and v^{up}): These two variables specify the range of values for the data rate. Different buffers may have distinct properties, which we discuss as follows.

• *Bounds of the network data rate.* We conduct an empirical study to observe throughput variations. Our experiment involves two mobile devices running 802.11ac; the link is capable of a maximum PHY rate of 867 Mbps. The devices are held by two users moving randomly around a room. To saturate the link, we run an iperf UDP test while fine-tuning the iperf parameters to ensure that the maximum throughput is achievable. The result, shown in Fig. 10, is presented in a heatmap format and plot the transition probabilities from data rate at time t to rate at time $t+d$, where d is the control latency. Two reference lines in blue, $y = \frac{1}{2}x$ and $y = 2x$, are included to denote throughput reductions of $0.5\times$ and increases of $2\times$, respectively. Our results show that within a control latency duration, the network throughput is highly unlikely to experience a $> 2\times$ increase or a $< 0.5\times$ decrease, with a probability greater than 99%. Therefore, for the Qdisc, we set the lower bound $v_1^{\text{lo}} = 0.5v_1$ and upper bound $v_1^{\text{up}} = 2v_1$. Note that this empirical measurement is specific to our setup.

The throughput variation is affected by the complex wireless channel environment. To completely prevent buffer overflows due to rapidly changing channel conditions, SMUFF can use strict bounds on the data rate, setting the lower bound to zero and the upper bound to the maximum theoretical data rate.

• *Bounds for CPU processing and I/O rate.* To determine the bounds of the processing rate of the socket buffer and the application buffer at the receiver, we take advantage of the insight that these rates are largely unaffected by the network, but are CPU- and I/O-bound (Fig. 1). Therefore, they tend to exhibit stability within a device that is largely dependent on the underlying hardware configuration. As a result, for the socket buffer and application buffer, we set the lower and upper bounds to the values of 0.9 and 1.1 times the estimated data rate, respectively, to allow for error tolerance.

Controlling the Sending Rate. In response to signals indicating buffer state updates, the buffer orchestrator calculates the new sending rate using the formula in Eqn. 8 and adjusts v_0. While the sending rate is determined at the sender, the control latency d_i varies depending on the location of the buffers: *i)* When dealing with the Qdisc buffer at the sender, the control latency can be set to a relatively small value (4 ms in our implementation), because polling the buffer state information from the sender buffers can be done simply by a system call; *ii)* When dealing with the two buffers at the receiver, the control latency is set to the network latency.

3.5 Transport Manager

While buffer orchestrator carefully monitors buffer states and adjusts the sending rate accordingly, packet loss could still occur due to lossy wireless link. The loss detector guarantees data delivery reliability based on the sliding window mechanism and ACKing with two improvements.

Reduce ACK Frequency. Frequent ACK messages exaggerate channel contention and have a negative impact on throughput in Wi-Fi direct transport (§2.3). To address this issue, we use periodic ACKs and send four ACKs per RTT to reduce the ACK frequency. This setting proves to be robust in practice [18]. In addition, an ACK message also carries the states of the buffers at the receiver side.

Speed up Loss Recovery. We propose three optimizations to improve the efficiency of the loss recovery procedure. *i)* The use of periodic ACKs prevents the receiver from notifying the sender of lost events until the next ACK is sent. To speed up recovery, SMUFF immediately sends an ACK containing a retransmission request when a loss event is detected. *ii)* To reduce overhead, we use Negative Acknowledgments (NACKs), similar to the SACK option [35] in TCP and the ACK range in QUIC [31], to inform the sender of missing data. *iii)* We use PING messages to periodically probe the RTT and use an EWMA filter to obtain a smoothed RTT estimate. This design mitigates RTT measurement bias caused by reduced ACK frequency and does not require additional computation or maintenance of a complex per-packet data structure.

3.6 Other Design Considerations

In practice, network-tuning is also important to achieve high throughput [29, 36–38]. Here we highlight two important considerations in the SMUFF design.

3.6.1 Packet Aggregation

Packet aggregation can reduce processing overhead in software [29]. SMUFF uses packet aggregation to improve throughput and reduce mobile computation and power costs. Specifically, SMUFF generates full-size UDP packets, approaching 64 KB in size, instead of the usual 1500-byte packets. The IP layer then fragments or reassembles these packets to meet the Maximum Transmission Unit (MTU) limit, conforming to the IP fragmentation feature [39]. Since a substantial amount of energy is drained by the processing of packet units (*i.e.,* independent of their size, air time, or modulation and coding scheme) [40], SMUFF is more computation- and energy-efficient with packet aggregation. The effectiveness of packet aggregation is evaluated in §5.4.

However, packet aggregation can lead to increased susceptibility to packet loss. In cases where a single packet within a fragmented small packet is lost, the entire IP packet is considered lost. To illustrate this concept mathematically, if the network path loss rate is denoted as L, then the packet loss rate for a full-sized IP packet can be expressed as $1-(1-L)^n$, where n is the number of fragments into which the IP packet is broken. For example, if the path loss rate is 1% and a full-size UDP packet is segmented into 43 fragments, the total packet loss rate rises to an unacceptable 35.1%, significantly reducing the available throughput. To address this issue, we consider the infrequency of packet loss events in most usage scenarios, a phenomenon that contrasts with Internet connections where loss is often attributed to network congestion. Therefore, we empirically set a conservative threshold of 0.1%. Packet aggregation is only enabled if the current network loss rate is below this threshold. We provide a evaluation of its performance and the rationale for choosing this threshold in the evaluation (§5.3).

3.6.2 Flow Prioritization

Data transfer is often a background job. Other foreground traffic flows are potentially at a disadvantage when sharing the same outbound device with SMUFF because the buffer orchestrator will try to keep the Qdisc blocklogged (if the network is the bottleneck). This can affect interactive traffic such as web browsing and video streaming. To address this issue, we strictly prioritize other foreground traffic over the SMUFF flow. Specifically, there are multiple queues with different priorities at the Qdisc layer, and the packet in the queue with the higher priority is sent first. Therefore, by setting a lower priority for the SMUFF packets, we can reduce the impact on other traffic. The effectiveness of flow prioritization is evaluated in §5.3.

4 Implementation

We have developed SMUFF for commodity Android devices as a dedicated system daemon running entirely in userspace, with a primary focus on minimizing development and integration complexity. Our implementation consists of about 8000 lines of C++ code and seamlessly integrates with the NDK toolchain as well as the native dependencies inherent to the Android platform. SMUFF extends its functionality by providing a file transfer service over a Linux socket. Applications wishing to use this service can effectively communicate their file transfer requirements by interacting with this socket. We use the `SO_PRIORITY` option to define the priority for all packets to be sent by SMUFF. In addition, we set the maximum buffer size $X_{\max}$ to twice the bandwidth-delay product (BDP) to ensure that the safe area exists.

In SMUFF, we use standard system APIs to retrieve buffer state information. Specifically, to retrieve Qdisc information, we use the `RTM_GETQDISC` option of the Linux routing socket. The Linux routing socket is part of Netlink, which allows userspace programs to communicate with the kernel via a socket interface. To get the length of the Qdisc queue, we first create a socket with the domain `AF_NETLINK` and the protocol `NETLINK_ROUTE`. After sending the request to the socket, the kernel takes over the processing of the request and places the result in the socket receive buffer. We then read from the socket and get the length of the Qdisc queue. To get the state of the receive buffer, we use the socket option `SO_MEMINFO` to get memory-related information. It's worth noting that all APIs used are available after Linux 4.12 and have been supported since Android 9. This compatibility ensures a high degree of compatibility; we verify that SMUFF works on at least four different phone models without any code changes.

5 Evaluation

Testbed. We use two Pixel 4 devices for most of the experiment because Android is based on Linux and more readily customizable. The devices support up to 2 MIMO spatial streams, 80 MHz bandwidth, short guard interval, and default rate adaptation, enabling speeds of up to 867 Mbps. To ensure the robustness of our results, each test is repeated for 30 times, guaranteeing statistical reliability and comprehensive data collection. The experiment is conducted in a public office with over 10 Wi-Fi APs. This experiment environment closely mirrors common real-world scenarios involving direct device-to-device connections.

Baselines. We compare SMUFF with the following transport schemes as baselines:

- TACK [18, 41] is a variant of TCP. It aims to improve wireless transport performance by minimizing the ACK frequency to reduce wireless channel contention.
- CUBIC [13] uses the standard TCP socket to send the data. To make sure it gets the maximum throughput, we set the socket buffer size to 20 MB and enable the TCP window size scaling [42, 43].
- BBR [12] uses the standard TCP with BBR congestion control algorithm. As BBR currently does not ship off-the-shelf with the phones, we recompile the Android kernel to include BBR and flash the new kernel.
- QUIC [31, 44] is designed to improve transport performance for HTTPS traffic. It is based on UDP and is implemented entirely in userspace as SMUFF.
- UDP with the default system configuration is also evaluated. Note that it does not provide reliable delivery.

Signal Strength	-20 dBm	-40 dBm	-60 dBm	-80 dBm
Distance	$<$ 0.1 m	2~3 m	8~10 m	N/A
Scenario	head-to-head	face-to-face	meeting	behind walls

Table 2: Wireless signal strength for typical scenarios.

5.1 Transport Performance

Throughput. We begin by evaluating SMUFF for different levels of signal strength. The typical conditions and common usage scenarios are shown in Table 2 for reference. The average throughput of different transport schemes transferring a 1 GB file is illustrated in Fig. 11. *For good Wi-Fi signal* (≥ -60 dmB), we maintain control over signal strength by manipulating the distance between the mobile phones, thus simulating different channel environments relevant to file transfer scenarios. In all cases, SMUFF outperforms other transport schemes. Across the signal strength settings, it achieves throughput improvement from 17.8% to 18.2% compared to TACK, the state-of-the-art variant designed for wireless networks, and a remarkable 22.6% to 44.5% improvement compared to CUBIC. In terms of throughput variance, SMUFF exhibits minimal variation across different signal strengths, ensuring a consistent and reliable transmission service. This stability is due to SMUFF's ability to effectively manage data within buffers. BBR also exhibits low variance, but it achieves lower throughput due to its goal to achieve the minimum `RTprop` time. As saturating the link can result in increased latency, BBR is conservative in increasing its sending rate. In contrast, other transport schemes such as TACK, CUBIC, and QUIC, exhibit significant throughput variance as they struggle to accurately measure wireless bandwidth. Under conditions of *bad Wi-Fi signal*, where the signal strength is at -80 dBm, all solutions perform similarly, with SMUFF still approaching the practical line rate. Overall, our results consistently demonstrate the superior network throughput performance of SMUFF across different signal strength scenarios, confirming its effectiveness as a reliable data transfer solution.

Flow Completion Time. We compare flow completion times (FCT) when transferring files of different sizes, as shown in Fig. 12. For smaller files (*i.e.,* 10 MB), the FCT is mainly influenced by the time taken to reach the maximum link rate. SMUFF's ability to bypass the slow start process leads to remarkable reductions in transfer times, ranging from 55.7% to 91.4% compared to alternative solutions. In particular, TCP

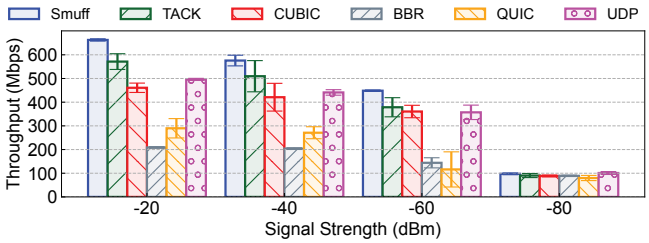


Figure 11: Throughput of different transport schemes at different signal strengths.

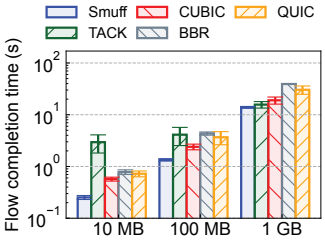


Figure 12: FCT of different transport schemes.

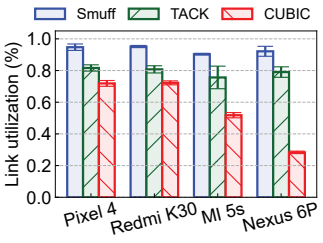


Figure 13: Link utilization on different device models.

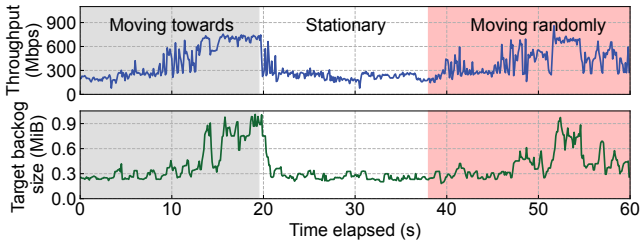


Figure 14: Trace of the real-time throughput and the target backlog size of the Qdisc.

variants and QUIC relying on the conventional slow start mechanism exhibit longer transfer times. TACK, on the other hand, shows significant variance and a much longer FCT compared to all other schemes, mainly due to its extended slow start phase caused by ACK shortages. This result is consistent with the analysis presented in §2.3. For medium-sized files (*i.e.,* 100 MB), SMUFF remains the best scheme with a 44.2% to 67.4% reduction in FCT over the other schemes. For larger files (*i.e.,* 1 GB), FCT is mainly determined by the link utilization. Consequently, TACK shows better performance compared to transferring smaller files. SMUFF still outperforms TACK by reducing the FCT by 12.1%. Furthermore, when averaged across CUBIC, BBR, and QUIC, SMUFF reduces transfer time by an impressive 26.3% to 64.4%. This experiment shows that SMUFF can significantly reduce the FCT by fully utilizing the link capacity.

Robustness to Mobility. As discussed in §3.3.2, the buffer orchestrator continuously computes the target backlog size based on newly received buffer states. To evaluate how effectively SMUFF adapts to mobility, we perform mobility tests focusing on the target backlog size of the Qdisc buffer. Our experiment involves two users with devices moving around a room, simulating realistic mobility scenarios. The real-time throughput collected at the receiver and the target backlog size of the Qdisc are recorded, as shown in Fig. 14. The experiment includes three different movement patterns: For the first 19 seconds (the gray region), both users move toward each other. Between 19 seconds and 38 seconds (the white region), both users move away from each other and then remain stationary. After 38 seconds (the red region), the users roam randomly in the room. The results show that the target backlog size closely matches the throughput. Such synchronization demonstrates SMUFF's ability to make robust adjustments to its sending rate to prevent overflow or underflow under mobility. This adaptability to mobility ensures stable and reliable performance in dynamic network environments.

	Redmi K30 Pro	Google Pixel 4	MI 5s	Nexus 6P
SoC	Snapdragon 865	Snapdragon 855	Snapdragon 821	Snapdragon 810
RAM	6 GB	4 GB	3 GB	3 GB
Battery	4700 mAh	2800 mAh	3200 mAh	3450 mAh
Wi-Fi	802.11a/b/g/n/ac/ax	802.11a/b/g/n/ac	802.11a/b/g/n/ac	802.11a/b/g/n/ac
Released	2020	2019	2016	2015

Table 3: Device specifications.

5.2 Compatibility

Different Device Models. To demonstrate the deployability and adaptability of SMUFF, we conduct deploy SMUFF and performance tests on four different smartphone models listed in Table 3. We compare SMUFF with two sub-optimal schemes (TACK and CUBIC) and initiate our tests by designating each device as the sender, with the Pixel 4 serving as the receiver. This configuration was chosen because the sender actively manages the sending rate based on buffer states, providing a robust metric. The results, shown in Fig. 13, show that SMUFF outperforms TACK and CUBIC on all devices, achieving a remarkable link utilization rate (achieved throughput divided by line rate) of up to 94.7%. It's worth noting that CUBIC's performance shows some variability across devices, likely due to differences in processing capabilities. Overall, our experiments confirm that SMUFF works seamlessly across different device models, delivering consistently high link utilization.

Wi-Fi Specifications. To evaluate link utilization with different Wi-Fi specifications, we conduct experiments using a stable signal with a strength of -40 dBm. This evaluation is performed on both 802.11ac (Wi-Fi 5) and 802.11ax (Wi-Fi 6) networks. We evaluate the throughput of SMUFF against all other transport schemes. The results, shown in Fig. 15, highlight the impressive performance of SMUFF, which is consistently close to the line rate. It achieves an average link utilization of 88.7% on 802.11ac (equivalent to 576 Mbps) and an even more remarkable 91.8% on 802.11ax (equivalent to 645 Mbps). These results demonstrate SMUFF's ability to efficiently utilize available network resources across different Wi-Fi specifications.

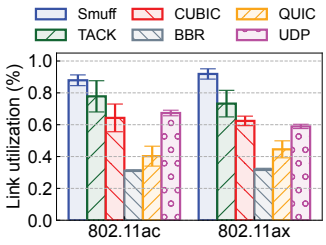


Figure 15: Link utilization of different transport schemes and Wi-Fi specifications.

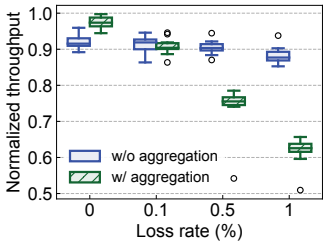


Figure 16: Impact of loss rate on the packet aggregation.

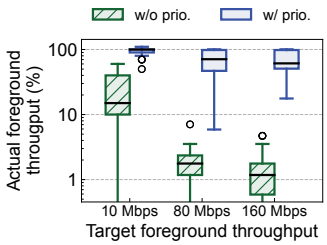


Figure 17: Flow prioritization improves the foreground throughput.

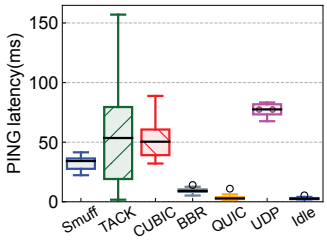


Figure 18: PING latency of different transport schemes.

5.3 Microbenchmark

We perform some micro-benchmarks to investigate the impacts of design choices made in the development of SMUFF.

Packet Aggregation Robustness. As discussed in §3.6.1, packet aggregation can make a network more susceptible to packet loss. To investigate how SMUFF behaves under various conditions, we perform tests at -40 dBm signal strength, both with and without packet aggregation. To simulate packet loss, we drop packets randomly at the receiver. As shown in Fig. 16, in scenarios where no packet loss occurs, packet aggregation effectively reduces processing overhead and improves performance, resulting in an average improvement of 5.4%. However, as the loss rate escalates from 0.1% to 1%, throughput drops dramatically by up to 62.4%. Such a severe reduction is primarily due to the vulnerability introduced by packet aggregation, where the loss of a single IP fragment packet results in the loss of the entire packet. In contrast, when packet aggregation is disabled, SMUFF shows efficiency in recovering lost packets. Therefore, SMUFF disables aggregation when the path loss rate exceeds a predefined threshold, ensuring reliability and preserving performance in the presence of a high packet loss rate.

Flow Prioritization. As discussed in §3.6.2, SMUFF can potentially cause foreground traffic to starve. To evaluate the effectiveness of flow prioritization in mitigating this problem, we perform tests by running foreground TCP flows alongside SMUFF. The TCP flows are configured with different target throughputs (10, 80, and 160 Mbps). We measure the actual throughput achieved by the TCP flows with flow prioritization enabled and disabled. As shown in Fig. 17, without flow prioritization, the foreground TCP flows achieve an average throughput of 2.44 Mbps, 2.74 Mbps, and 2.54 Mbps for different target throughputs, indicating a disadvantaged state. With flow prioritization enabled, the achieved throughput improves significantly, reaching averages of 2.44 Mbps, 59.1 Mbps, and 118.2 Mbps for the respective target throughputs. While SMUFF cannot completely eliminate the impact on other flows (*e.g.,* only 73.9% of the target throughput is achieved for the 160 Mbps flow) due to the lack of prioritization in the device firmware buffers, flow prioritization significantly alleviates the problem of foreground traffic starvation.

	SMUFF	TACK	CUBIC	BBR	QUIC	UDP	Idle
Energy (J)	50.7	62.6	59.6	86.6	72.8	55.2	40.6
Overhead (%)	24.8	54.2	48.2	113.3	74.4	36.0	N/A

Table 4: Average energy consumption for transferring a 1 GB file via peer-to-peer wireless data transfer. "Overhead" is the additional energy consumption compared with the idle state.

5.4 System Overhead

Computation Overhead. To compare the computational overhead of SMUFF, we transfer a 1 GB file using SMUFF and other transport schemes and measure their CPU usage. As shown in Fig. 19, SMUFF achieves the lowest CPU usage. There are two reasons for this: First, SMUFF achieves nearly the wireless line rate and completes the transmission process much faster; second, SMUFF is based on UDP and thus avoids the overhead of a compute-intensive TCP stack. We also evaluate the effectiveness of packet aggregation. As shown in Fig. 20, aggregation can reduce the CPU usage of SMUFF by 18.0% and 21.9% on the sender and receiver side, respectively. This result shows that SMUFF can reduce system load on resource-constrained mobile devices.

Energy Overhead. We also examine power consumption, since mobile devices are typically powered by limited batteries. We use the Monsoon power monitor [45] to measure the average power consumption while transferring a 1 GB file using SMUFF and other transport schemes. The monitor works by precisely measuring the input power of the smartphone with a sampling rate of 5000 Hz and is widely used in the research community for measuring power consumption [46, 47]. As shown in Table 4, SMUFF consumes less energy than all other schemes (24% overhead over baseline) because it bypasses the computationally intensive TCP stack and completes the transmission earlier. The result shows that SMUFF is more energy efficient than other solutions.

Transmission Latency. As buffer saturation may increase transmission latency for all data flows between peer devices, we also evaluate the impact of the transport schemes on transmission latency. To measure the latency overhead of SMUFF, We conduct experiments by sending the ICMP PING messages [48] every second in the background during the file transmission. As shown in Fig. 18, compared with TACK

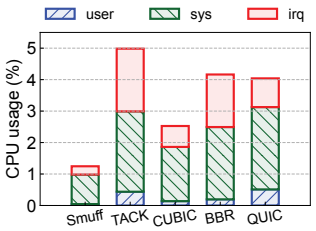


Figure 19: CPU usage of different transport schemes.

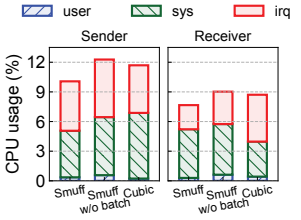


Figure 20: CPU usage reduction of packet aggregation.

and TCP CUBIC, SMUFF reduces average latency by 36.9% and 39.2% because it uses an adaptive adjustment on the backlogged data at Qdisc buffer. BBR and QUIC maintain a low latency but fail to fully utilize the available link. SMUFF keeps the latency below 40 ms with a small variance – such an increase in latency is acceptable in file transfer scenarios.

6 Discussion

Operating in Non-Wi-Fi One-Hop Networks. SMUFF does not require the underlying NIC to be a Wi-Fi device. Therefore, SMUFF could be seamlessly deployed over any NICs that consume packets from Qdisc. The effectiveness of SMUFF is based on two premises: *i)* Sending at line rate would generate the best performance. *ii)* NIC itself has some mechanism like BQL [49] to minimize the underlying buffering. Such conditions hold for most cases. Therefore, SMUFF theoretically also works in other one-hop networks such as Apple Wireless Direct Link (AWDL) [2], Neighbor Awareness Networking (NAN) [50], and wired links.

Benefiting Other Applications. SMUFF can also be used for other data transfer applications, such as 360-degree video streaming. SMUFF can introduce a minimal initial delay due to its extended queue length, but allows for higher video resolution due to its increased throughput once the video begins. To minimize latency, SMUFF can also be configured to maintain a near-zero queue length by setting the target queue size to zero. This strategy effectively reduces queue latency within the monitored buffers. However, it is important to note that latency cannot be similarly reduced for buffers that are not directly monitored, such as the NIC firmware buffer.

7 Related Work

Queue-based Transport Improvement. The packet queue provides lower-layer information to the transport control algorithm. This information enables the transport layer to perform better sending rate adjustment and accomplish different optimization goals. XCP [51], RCP [52], and ABC [15] leverage the dequeuing rate and the current queue length to let the network middleboxes compute the right sending rate and signal this rate to the sender. The signal lets the sender "jump" to the correct sending rate to achieve high throughput. Active Queue Management (AQM) schemes such as RED [53], CoDel [54], and PIE [55] drop packets when queue length or queue delay exceeds a threshold to signal congestion. Swift [56] and QCut [32] monitor the end-host queuing delay and adjust sending rate accordingly. PowerTCP [16] and HPCC [14] leverage in-network measurements at programmable switches to accurately obtain the bottleneck link state. SMUFF also works on the queue length, but it calculates its variation and decides to send how many packets so as to backlog the queue.

Wireless Transport Layer Protocol Design. Wireless links have different characteristics due to unstable channel characteristics. This motivates designs to optimize different aspects of the wireless transport. I-TCP [57] splits a TCP connection into two parts (the wireless part and the wired part) and handles frequent interruptions on the lossy wireless link. Snoop [58] performs local retransmissions over the wireless link to hide the packet loss event. TACK [18] reduces channel contention between data packets and ACKs to improve throughput. ELN [59] allows senders to distinguish between packet corruption and congestion losses and respond accordingly. SMUFF is designed to improve the network performance of the one-hop wireless link. It addresses the challenge of how to saturate a rapidly changing wireless link.

End-to-end Congestion Control Designed for Wireless Link. Wireless link speeds vary greatly over time. There are several ways to predict link speed and improve throughput. Sprout [33] observes packet arrival times to predict how many bytes can be sent from the sender. Verus [34] learns the relationship between packet delay and transmit window size and uses it to adjust the size of the transmit window. CQIC [60] and PBE-CC [61] use the physical layer bandwidth information exchanged between base stations and handsets to determine link capacity. SMUFF also needs to capture link capacity variations, but it does not need to handle the congestion on the one-hop network and can accurately track the line rate.

8 Conclusion

The rising demand for high-speed peer-to-peer mobile data transfer calls for an extremely efficient single-hop transport solution. In this work, we have addressed the challenge of achieving near wireless line rate transport based on Wi-Fi Direct by effectively orchestrating all the on-path buffers to maximize throughput while avoiding packet loss. We believe that our work identifies an overlooked yet important problem and our design can benefit other one-hop wireless networks (*e.g.,* 5G and vehicular) with different PHY/MAC protocols.

Acknowledgments

We are grateful to the anonymous NSDI reviewers for their constructive critique and valuable comments, all of which have greatly helped us improve this paper. This work is supported in part by the National Key Research and Development Plan, China (Grant No. 2023YFB2903902) and the National Natural Science Foundation of China (Grant No. 62022005, 62272010 and 62061146001). Chenren Xu is the corresponding author.

References

[1] Use AirDrop on iPhone to send items to nearby devices - Apple Support. https://support.apple.com/guide/iphone/use-airdrop-to-send-items-iphcd8b9f0af/ios.

[2] Milan Stute, David Kreitschmann, and Matthias Hollick. One Billion Apples' Secret Sauce: Recipe for the Apple Wireless Direct Link Ad hoc Protocol. In *ACM MobiCom*, 2018.

[3] Huawei share. https://consumer.huawei.com/en/support/huaweishare/.

[4] Xiaomi, OPPO and Vivo partner to bring new wireless file transfer system to global users – Mi Blog. https://c.mi.com/thread-2776586-1-0.html.

[5] What is Quick Share on Galaxy? - The Official Samsung Galaxy Site. https://www.samsung.com/global/galaxy/what-is/quick-share/.

[6] VRChat. https://hello.vrchat.com/.

[7] Abderrahmane Lakas and Moumena Shaqfa. Geocache: Sharing and Exchanging Road Traffic Information Using Peer-to-Peer Vehicular Communication. In *IEEE Vehicular Technology Conference*, 2011.

[8] Xiuxian Guan, Zekai Sun, Shengliang Deng, Xusheng Chen, Shixiong Zhao, Zongyuan Zhang, Tianyang Duan, Yuexuan Wang, Chenshu Wu, Yong Cui, et al. ROG: A High Performance and Robust Distributed Training System for Robotic IoT. In *IEEE/ACM MICRO*, 2022.

[9] Ang Li, Jingwei Sun, Pengcheng Li, Yu Pu, Hai Li, and Yiran Chen. Hermes: An efficient federated learning framework for heterogeneous mobile clients. In *ACM MobiCom*, 2021.

[10] Jinliang Yuan, Mengwei Xu, Xiao Ma, Ao Zhou, Xuanzhe Liu, and Shangguang Wang. Hierarchical Federated Learning through LAN-WAN Orchestration. arXiv 2010.11612, 2020.

[11] Daniel Camps-Mur, Andres Garcia-Saavedra, and Pablo Serrano. Device-to-device communications with Wi-Fi Direct: overview and experimentation. *IEEE wireless communications*, 2013.

[12] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control. *ACM Queue*, 2016.

[13] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review*, 2008.

[14] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High precision congestion control. In *ACM SIGCOMM*, 2019.

[15] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. ABC: A Simple Explicit Congestion Controller for Wireless Networks. In *USENIX NSDI*, 2020.

[16] Vamsi Addanki, Oliver Michel, and Stefan Schmid. POWERTCP: Pushing the Performance Limits of Datacenter Networks. In *USENIX NSDI*, 2022.

[17] Lynne Salameh, Astrit Zhushi, Mark Handley, Kyle Jamieson, and Brad Karp. HACK: Hierarchical ACKs for Efficient Wireless Medium Utilization. In *USENIX ATC*, 2014.

[18] Tong Li, Kai Zheng, Ke Xu, Rahul Arvind Jadhav, Tao Xiong, Keith Winstein, and Kun Tan. TACK: Improving Wireless Transport Performance by Taming Acknowledgments. In *ACM SIGCOMM*, 2020.

[19] Yossi Gilad and Amir Herzberg. Fragmentation considered vulnerable. *ACM Transactions on Information and System Security (TISSEC)*, 15(4):1–31, 2013.

[20] Wi-Fi Direct Phone Product List. https://www.wi-fi.org/product-finder-results?sort_by=certified&sort_order=desc&categories=3.

[21] IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements–Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications–Amendment 4: Enhancements for Very High Throughput for Operation in Bands below 6 GHz. 2013.

[22] IEEE Standard for Information Technology–Telecommunications and Information Exchange between Systems Local and Metropolitan Area Networks–Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 1: Enhancements for High-Efficiency WLAN. 2021.

[23] Shangqing Zhao, Zhe Qu, Zhengping Luo, Zhuo Lu, and Yao Liu. Comb Decoding towards Collision-Free WiFi. In *USENIX NSDI*, 2020.

[24] Iperf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/, 2023.

[25] iw - linux man page. https://linux.die.net/man/8/iw, 2023.

[26] Ranveer Chandra, Ratul Mahajan, Thomas Moscibroda, Ramya Raghavendra, and Paramvir Bahl. A Case for Adapting Channel Width in Wireless Networks. In *ACM SIGCOMM*, 2008.

[27] IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area networks–Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. 2016.

[28] Robert Braden. RFC1122: Requirements for Internet hosts-communication layers, 1989.

[29] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *ACM SIGCOMM*, 2021.

[30] RFC 793: Transmission Control Protocol, 1981.

[31] Jana Iyengar and Martin Thomson. RFC 9000: QUIC: A UDP-Based Multiplexed and Secure Transport. 2021.

[32] Yihua Guo, Feng Qian, Qi Alfred Chen, Zhuoqing Morley Mao, and Subhabrata Sen. Understanding On-device Bufferbloat for Cellular Upload. In *ACM IMC*, 2016.

[33] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *USENIX NSDI*, 2013.

[34] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive Congestion Control for Unpredictable Cellular Networks. In *ACM SIGCOMM CCR*, 2015.

[35] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. RFC 2018: TCP Selective Acknowledgment Options, 1996.

[36] Sanjib Sur, Ioannis Pefkianakis, Xinyu Zhang, and Kyu-Han Kim. WiFi-Assisted 60 GHz Wireless Networks. In *ACM MobiCom*, 2017.

[37] Shivang Aggarwal, Swetank Kumar Saha, Pranab Dash, Jiayi Meng, Arvind Thirumurugan, Dimitrios Koutsonikolas, and Y. Charlie Hu. Poster: Can Mobile Hardware Keep Up with Today's Gigabit Wireless Technologies? In *ACM MobiCom Poster*, 2019.

[38] UDP Tuning Technique from ESnet. https://fasterdata.es.net/network-tuning/udp-tuning.

[39] RFC 791: Internet Protocol, 1981.

[40] Andres Garcia-Saavedra, Pablo Serrano, Albert Banchs, and Giuseppe Bianchi. Energy consumption anatomy of 802.11 devices and its implication on modeling and design. In *ACM CoNEXT*, 2012.

[41] fillthepipe/fill-the-pipe. https://github.com/fillthepipe/fill-the-pipe.

[42] TCP optimization for network performance | Compute Engine Documentation | Google Cloud. https://cloud.google.com/compute/docs/networking/tcp-optimization-for-network-performance-in-gcp-and-hybrid.

[43] David Borman, Robert T. Braden, Van Jacobson, and Richard Scheffenegger. RFC 7323: TCP extensions for high performance, 2014.

[44] litespeedtech/lsquic: LiteSpeed QUIC and HTTP/3 Library. https://github.com/litespeedtech/lsquic.

[45] Monsoon High Voltage Power Monitor. https://www.msoon.com/online-store/High-Voltage-Power-Monitor-p90002590.

[46] Arvind Narayanan, Xumiao Zhang, Ruiyang Zhu, Ahmad Hassan, Shuowei Jin, Xiao Zhu, Xiaoxuan Zhang, Denis Rybkin, Zhengxuan Yang, Zhuoqing Morley Mao, Feng Qian, and Zhi-Li Zhang. A variegated look at 5G in the wild: Performance, power, and QoE implications. In *ACM SIGCOMM*, 2021.

[47] Junxian Huang, Feng Qian, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4G LTE networks. In *ACM MobiSys*, 2012.

[48] Jon Postel. RFC 792: Internet control message protocol. Technical report, 1981.

[49] Tom Herbert. BQL: Byte Queue Limits. https://lwn.net/Articles/454378/.

[50] Wi-Fi Aware | Wi-Fi Alliance. https://www.wi-fi.org/discover-wi-fi/wi-fi-aware.

[51] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *ACM SIGCOMM*, 2002.

[52] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. In *ACM SIGCOMM CCR*, 2006.

[53] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM ToN*, 1993.

[54] Kathleen Nichols and Van Jacobson. Controlling queue delay. *Communications of the ACM*, 2012.

[55] Rong Pan, Preethi Natarajan, Chiara Piglione, Mythili Suryanarayana Prabhu, Vijay Subramanian, Fred Baker, and Bill VerSteeg. PIE: A lightweight control scheme to address the bufferbloat problem. In *IEEE HPSR*, 2013.

[56] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *ACM SIGCOMM*, 2020.

[57] A. Bakre and B.R. Badrinath. I-TCP: Indirect TCP for mobile hosts. In *IEEE ICDCS*, 1995.

[58] Hari Balakrishnan, Srinivasan Seshan, and Randy H. Katz. Improving reliable transport and handoff performance in cellular wireless networks. *Wireless Networks*, 1995.

[59] Hari Balakrishnan and Randy H Katz. Explicit Loss Notification and Wireless Web Performance. In *IEEE Globecom*, 1998.

[60] Feng Lu, Hao Du, Ankur Jain, Geoffrey M. Voelker, Alex C. Snoeren, and Andreas Terzis. CQIC: Revisiting Cross-Layer Congestion Control for Cellular Networks. In *ACM HotMobile*, 2015.

[61] Yaxiong Xie, Fan Yi, and Kyle Jamieson. PBE-CC: Congestion Control via Endpoint-Centric, Physical-Layer Bandwidth Measurements. In *ACM SIGCOMM*, 2020.

Vulcan: Automatic Query Planning for Live ML Analytics

Yiwen Zhang[†], Xumiao Zhang[†], Ganesh Ananthanarayanan[‡], Anand Iyer[§],
Yuanchao Shu[††], Victor Bahl[‡], Z. Morley Mao[†¶], Mosharaf Chowdhury[†]
[†]University of Michigan, [‡]Microsoft, [§]Georgia Institute of Technology, [††]Zhejiang University, [¶]Google

Abstract

Live ML analytics have gained increasing popularity with large-scale deployments due to recent evolution of ML technologies. To serve live ML queries, experts nowadays still need to perform manual query planning, which involves pipeline construction, query configuration, and pipeline placement across multiple edge tiers in a heterogeneous infrastructure. Finding the best query plan for a live ML query requires navigating a huge search space, calling for an efficient and systematic solution.

In this paper, we propose Vulcan, a system that automatically generates query plans for live ML queries to optimize their accuracy, latency, and resource consumption. Based on the user query and performance requirements, Vulcan determines the best pipeline, placement, and query configuration for the query with low profiling cost; it also performs fast online adaptation after query deployment. Vulcan outperforms state-of-the-art ML analytics systems by 4.1×-30.1× in terms of search cost while delivering up to 3.3× better query latency.

1 INTRODUCTION

Recent years have witnessed a growing demand for machine learning (ML) analytics. *Live ML analytics* – with applications in edge-assisted autonomous driving [62, 71, 72], live traffic analysis [1, 43], and real-time speech recognition [18, 66] – stands out due to its large-scale deployments [1,2,4]. Live ML analytics involves ML pipelines at its core, where each pipeline consists of a series of operators to perform specific ML tasks. For example, an autonomous driving perception query that detects surrounding objects of an autonomous vehicle may contain filtering operators for road surface removal [23] and 3D data compression [22,53], along with a 3D object detector to perform object detection [37,68].

Live ML analytics differentiates itself from ML on stored data in two key characteristics. First, live ML pipelines are deployed across *heterogeneous infrastructure*, spanning multiple tiers such as device edges, on-premise edges, public MEC, and cloud datacenters [1–3,6]. Second, live ML queries have *latency requirements* besides accuracy targets as the analytics is based on real-time data. Some analytics, such as object detection in autonomous vehicles, may have more stringent requirements than others depending on the priority of the task. Therefore, before deploying live ML analytics, each query describing the ML task must go through careful *query planning*. Specifically, this involves (i) constructing the pipeline by selecting a series of operators and ordering them; (ii) determining the physical placement of pipeline operators across infrastructure tiers; and (iii) selecting configurations of the pipeline operators, to optimize for query performance. A joint optimization of the three aspects is required for optimal performance and resource consumption.

Recent research on these topics have been piecemeal, focused on compute alone, and hence largely sub-optimal for live ML analytics. Although declarative query languages (e.g., SQL) have been proposed for ML queries, there exists no systematic approach to automatically construct pipelines based on query's end-to-end latency requirement. This includes selecting and ordering the filtering modules during pipeline construction, which have a great impact on the performance characteristics of ML pipelines. Furthermore, when choosing physical placement of pipeline components, one cannot afford to exhaustively search for the optimal placement through real-world deployments. As a result, deployments often rely on past experience with simple heuristics [3,67]. While recent solutions have focused largely on selecting query configurations [12,27,28,63,70], they assume the ML analytics component to be a monolithic module instead of a pipeline. As such, they are profiled for compute alone, assuming they are running in a homogeneous datacenter. In the process, networking resources are ignored, and the complexities of multi-resource planning of compute and network are overlooked altogether. Finally, these solutions rely heavily on domain-specific insights of video content, which are not applicable to general ML scenarios beyond video analytics [28,45,70].

After deploying the query in the wild, one must also adapt the query plan based on runtime dynamics such as data content and/or resource changes [12,28,52,54]. Prior solutions in providing online adaptation [28,70] focus on data content changes alone but do not adapt to compute and network resource changes, which are common in edge environments [52]. An ideal solution should change query configuration *and* placement during online adaption.

In this work, we consider how to perform automatic query planning – i.e., constructing, placing, and configuring ML pipelines, along with adapting to runtime dynamics – for live ML queries based on user-provided performance requirements. The goal is to find query plans that optimize latency and accuracy, while minimizing the network and compute resource consumption. Generating such query plans, however, is challenging as jointly optimizing pipeline construction,

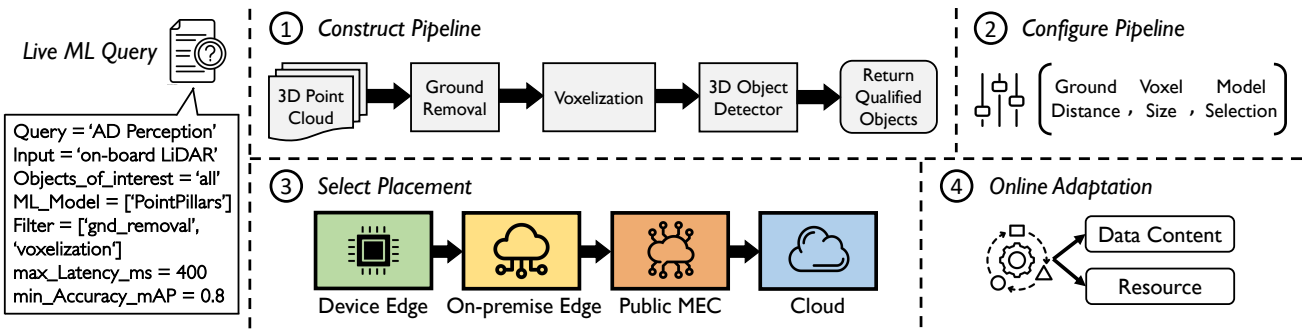


Figure 1: The existing workflow of query planning for a live ML query.

placement, and query configuration leads to a much larger search space. Moreover, as query performance is resource and data dependent, considering all possible resource and data dynamics that may or may not happen in the future can explode the search space. Clearly, a more efficient online adaptation technique is needed for faster convergence and lower cost.

We present Vulcan, an ML analytics system that performs automatic query planning for live ML queries. Its design includes the following key ideas to overcome the aforementioned challenges:

(1) Vulcan defines a novel metric to quantify each filtering operator in the pipeline by combining query precision, recall, resource usage, and latency (§4.2). This converts the complexity of filter ordering from exponential to linear.

(2) Vulcan carefully identifies components of ML pipelines that are *independent* of placement. This allows Vulcan to dramatically prune the search space of placement options.

(3) Vulcan efficiently explores the best combination of configuration knobs using Bayesian Optimization (BO) [14, 51]. It designs BO's priori assumptions and acquisition function to jointly optimize placement and configuration selection.

(4) Vulcan adapts quickly to dynamic changes in data and resources by (i) designing programming interfaces that allow for dynamic updates to live pipelines modules without disrupting them, and (ii) leveraging prior knowledge to make faster decisions on modifying configurations and placement.

We evaluate Vulcan using real-world datasets on a wide range of applications including traffic monitoring, autonomous driving perception, and automatic speech recognition. Experiments are conducted under an edge hierarchy that represents real compute and network resource setting of our production infrastructure. Vulcan generates query plans with better profiling cost by 4.1×-30.1× over state-of-the-art ML analytics systems while delivering 3.3× better query latency performance. Vulcan outperforms existing solutions for ML query configuration and placement selection with up to 2.8× better query latency and 174× lower network resource consumption. Vulcan also achieves consistently better 99th-p latency by up to 2.5× by adapting to both data and resource changes during online adaptation (§6).

In summary, we make the following research contributions:

- We provide an end-to-end system design for live ML queries for a wide range of real-world ML applications.
- We propose novel solutions on search space reduction for constructing, placing, and configuring ML pipelines.
- We implement interfaces and control loop for online adaption for fast re-profiling and dynamic re-configuration.

2 BACKGROUND AND MOTIVATION

We start with an overview of the workflow of live ML query processing, followed by motivating examples to highlight the key aspects when choosing query plans.

2.1 Processing Live ML Queries

We explain the query planning workflow by walking through an example query which detects surrounding objects in edge-assisted autonomous driving, as shown in Figure 1. The input query specifies input data, object of interests, pipeline operators, and performance requirements on accuracy and latency.

① **Constructing the pipeline.** The first step is constructing the ML pipeline by choosing a series of operators to perform the task. Specifically, it involves choosing *filtering operators*, such as voxelization and ground removal, to be deployed for substantial resource efficiency prior to the ML models, such as the 3D object detector [37, 68] in the case of Figure 1. The ordering of the filtering operators has a significant impact on performance. The best ordering is dependent on the data, resource availabilities, and performance requirements.

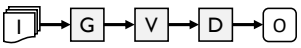
② **Configuring the pipeline.** After the operators are chosen in the pipeline, the next step is selecting *configuration knobs* to achieve the best tradeoff between query performance and resource consumption [12, 26, 28, 45, 70]. In our example, configuration knobs include ground distance, the voxel size, and the choice of a 3D object detection model.

③ **Selecting physical placement.** The next step is placing pipeline operators across the heterogeneous edge infrastructure, starting from the device edge and to the cloud (Figure 1). This heterogeneity introduces complexity in placement of the operators and influences the *end-to-end latency performance* of the ML pipeline.

④ **Performing online adaptation.** After a query is deployed, its performance is affected by *runtime dynamics* due to resource changes and data variations. Resource changes are

A_{req}	L_{req} (ms)
0.8	400

(a) Performance Requirement

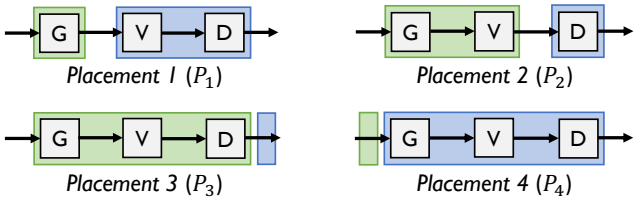


(b) Pipeline

Config	A	L (ms)	Size (MB)
C_1	0.85	260	6|4.4|3.2
C_2	0.8	270	6|3.6|2.2

(c) Query Configuration Results

Figure 2: (a) Performance requirement on accuracy (A_{req}) and latency (L_{req}) of the example query. (b) Query's pipeline ($Input \rightarrow GroundRemoval \rightarrow Voxelization \rightarrow ObjectDetector \rightarrow Output$). (c) Offline profiling results. The last column records the size of the data at different stages of the pipeline: data at the source | after 'G' | after 'V'. Data after 'D' is not shown due to negligible size.



(a) Placement Choices

Config	Placement	A	L (ms)
C_1	P_1	0.85	480
C_1	P_2	0.85	420
C_1	P_3	0.85	520
C_1	P_4	0.85	560

Config	Placement	A	L (ms)
C_2	P_1	0.8	450
★ C_2	P_2	0.8	380
C_2	P_3	0.8	540
C_2	P_4	0.8	570

(b) End-to-end Performance Results

Figure 3: Placement choices and corresponding end-to-end performance. (a) All four feasible placement choices in a two-tier setting (Device Edge→Datacenter). (b) The baseline approach which first acquires the optimal combinations of configuration knobs during offline profiling (C_1) and then selects placement fails to meet the latency target. An ideal solution should select C_2 & P_2 by jointly optimizing configuration and placement. Note accuracy does not depend on placement and remains unchanged.

common because of other workloads on the edge infrastructure (e.g., 5G RAN containers) or network outages [39, 52]. The content of the data, such as lighting or object densities [12,28], can also change during the lifetime of a query. An ideal solution should adapt to runtime dynamics by adjusting the pipeline's filters, its configurations, and placement.

2.2 Motivating Examples

We now highlight some of the key challenges toward performing query planning for live ML queries using toy examples.

Jointly optimizing configuration and placement. Current practice considers pipeline placement and query configuration separately [12, 28, 33, 49, 67]. Designers first perform offline profiling to determine the optimal query configuration, and then select placement based on heuristics (prioritize network or compute, etc.) or greedy algorithms [67]. Such an approach fails to consider additional query latency introduced by pipeline placement, leading to sub-optimal query plans.

A_{req}	L_{req} (ms)
0.6	300

(a) Performance Requirement

(b) Pipeline

Config	A	L (ms)	Size (MB)
C_1	0.7	180	6|4|2.7
C_2	0.6	190	6|2.8|2

(c) Query Configuration Results

Config	Placement	A	L (ms)
C_2	P_2	0.6	290

(d) Optimal Query Plan

Figure 4: Given an updated performance requirement, a new pipeline is required to meet the latency target. The new pipeline swaps the order of the two filters ('V' & 'G'), delivering a different performance characteristic than the old one in Figure 2b.

To illustrate this, take an example autonomous driving query, whose performance requirements and pipeline are shown in Figure 2a and 2b. Figure 2c records the offline profiling results assuming only 2 sets of configuration knobs (C_1 and C_2) exist. In this case, the baseline approach identifies C_1 as the optimal configuration since it performs better both in terms of accuracy and latency than C_2. It then pairs C_1 with one of the placement choices in our simplified two-tier infrastructure shown in Figure 3a.

However, as shown in Figure 3b, none of placement choices would satisfy the query's end-to-end latency requirement if C_1 is selected. The end-to-end latency is composed of the compute latency (time spent in the object detector) and additional network latency occurred over the edge. For example, the additional network latency in P_1 is computed as the time it takes to transmit the output data of the ground removal module using the link bandwidth, which is set to 20MBps in this example.[1] An ideal solution should select C_2 by jointly considering both placement and configuration.

Constructing pipelines based on performance requirements. Different orderings of filtering operators lead to different performance characteristics of ML pipelines. Therefore, an ideal solution must construct the pipeline based on query-specific performance requirements. For instance, if we change the performance requirements of the original example query to a lower latency target with more tolerance on accuracy (shown in Figure 4a), then none of the query plans in Figure 3b can satisfy the new latency target as long as they use the pipeline from Figure 2b. Instead, we need to use a new pipeline which swaps the order of the two filters (Figure 4b). As filters are not independent to each other, placing 'V' before 'G' reduces more data, leading to better latency but lower accuracy (Figure 4c). Figure 4d shows the end-to-end results with the optimal query plan using the new pipeline. We omit for brevity the same process of finding the optimal placement and configuration as we did earlier.

Building an ideal solution that handles all the aforementioned aspects of live ML query planning is non-trivial, as selecting the right pipeline, placement, and configuration jointly

[1] In this example, the compute latency is assumed to be doubled when placing the detector on the device edge (i.e., in P_3).

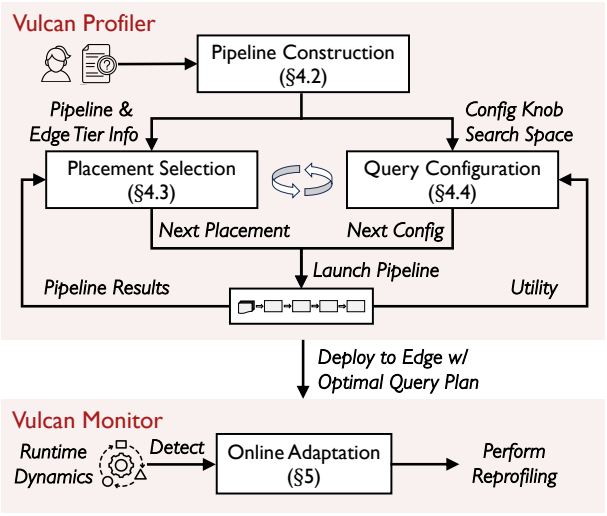


Figure 5: High-level workflow of Vulcan.

leads to a huge search space both during offline profiling and online adaptation. We next describe how Vulcan overcomes these challenges in a high-level system overview.

3 SYSTEM OVERVIEW

Vulcan is an ML analytics system that provides automatic query planning for live ML queries. It takes charge of the entire lifecycle of a ML query by constructing, configuring, and placing its ML pipeline, and performing online adaptation after the query is deployed.

Figure 5 presents a high-level system diagram of Vulcan. A user launches a live ML analytics task by submitting an input query to Vulcan, along with performance requirements. An example of Vulcan input queries can be found in Figure 1. Upon parsing the query, Vulcan Profiler generates its query plan by determining the query pipeline, placement of pipeline operators, and pipeline configuration. In Vulcan, query plans are evaluated using a utility function we define that combines the query performance and resource consumption (§4.1). To determine which pipeline to use for a query, Vulcan first constructs an initial pipeline by mapping user query specification to a general template optimized for performance and resource efficiency, and then determines the best ordering of filtering operators based on a new metric we define to capture the impact of filters on query latency and accuracy (§4.2).

Given a constructed pipeline, Vulcan jointly searches for the best placement and query configuration which, when combined, gives the highest utility. To explore placement choices with low cost, Vulcan reuses intermediate results from pipeline runs, such that a pipeline with the same configuration only needs to be offline profiled once. In the meantime, it also early prunes unpromising placement choices to further reduce the profiling cost (§4.3). For each placement, Vulcan searches for the best query configuration by leveraging Bayesian Optimization to explore a large number of query configurations with a small number of trials (§4.4).

After query deployment, Vulcan continues to monitor query performance to detect runtime dynamics. During such events, Vulcan reprofiles the pipeline in a quick and low-cost fashion by leveraging prior knowledge (§5).

4 VULCAN: PROFILER DESIGN

This section introduces the utility function we define to compare query plans, followed by how Vulcan construct, place, and configure the ML pipelines for live ML queries.

4.1 Defining Utility of Query Plans

We start by defining a utility function to evaluate the value of a query plan as we explore the search space. Existing literature has proposed various utility functions [9, 29, 50, 61] to combine query accuracy and latency. We build on top of these works and extend the utility function introduced in VideoStorm [70] to make it *resource-aware*. Resource consumption for live ML queries is as important as query performance, because each ML pipeline is deployed over edge infrastructure, where limited network and compute resources must be shared between applications. A query plan with good performance but excessive resources is undesirable.

Given a pipeline q with placement p and pipeline configurations c, we define $U_{q,p,c}$, the utility function of a query plan, as the ratio of the query performance to resource consumption:

$$U_{q,p,c} = P_{q,p,c} / R_{q,p,c} \quad (1)$$

such that the higher the utility value is, the better performance and cost for the query plan. $P_{q,p,c}$ combines query accuracy (A) and end-to-end latency (L) by calculating the reward (penalty) for achieving good (bad) performance based on a minimum accuracy target (A_m) and a maximum latency target (L_m):

$$P_{q,p,c}(A,L) = \gamma \cdot \alpha_A \cdot (A - A_m) + (1-\gamma) \cdot \alpha_L \cdot (L_m - L) \quad (2)$$

, where $\gamma \in (0,1)$. γ allows users to express their preference between accuracy and latency. $R_{q,p,c}$ combines the compute and network resource consumption of the pipeline:

$$R_{q,p,c} = \alpha_{gpu} \cdot R_{gpu} + \alpha_{net} \cdot R_{net} \quad (3)$$

The consumption of the compute (R_{gpu}) is calculated as the fraction of the GPU processing time used by the query. In Vulcan, we assume compute cost is dominated by GPU cost, as the queries we tackle rely heavily on GPU-based DNN models. The network resource consumption (R_{net}) is calculated as the sum of the fraction of the network bandwidth used by the pipeline on *each* network path between the edges. The constants α_A, α_L, α_{gpu}, and α_{net} are set by the operator to balance query performance and resource usage. Note that Vulcan's solution is orthogonal to the utility function and works for any utility function defined by the operator.

4.2 Determining the Query Pipeline

The first step in profiling is to construct the query pipelines. Vulcan performs pipeline construction by first generating an initial pipeline, and then determines the best ordering of pipeline operators to carry to the later profiling stages.

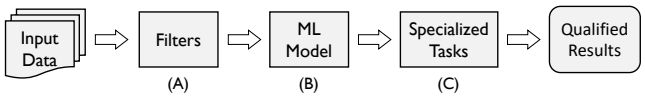


Figure 6: Template used by Vulcan to construct the initial pipeline.

4.2.1 Constructing the Initial Pipeline

Given a user query, Vulcan generates an initial pipeline using a general template with several types of building blocks, as shown in Figure 6. Starting from the data source, Vulcan constructs the initial pipeline by inserting (A) filtering modules which reduce the data size or data rate via sampling or filtering techniques, (B) the ML model to perform the actual inference task such as object detection, and (C) specialized modules for additional tasks required by the query, such as an object tracker (e.g., re-identification modules [31, 38, 40]), which can be performed only after the major ML inference task. Vulcan uses a pool of filter and ML modules that are readily available, provided by users, infrastructure providers, or third-party developers and organizations (e.g., public ML model zoos) to handle user queries. Filters and the ML model to use for the query is specified by the user in the input query (Figure 1). Based on the chosen operators, Vulcan generates a list of configuration knobs among which the profiler searches for an optimal set of configurations (§4.4).

The key insight behind arranging the building blocks in this way is to reduce the amount of data transfer across the edge *earlier* in the pipeline and leave operators with higher computation cost in *later* pipeline stages. This maximizes the savings in both network and compute resource as less data is transmitted and processed across the edge tiers. The design can improve end-to-end query latency by reducing the network latency as well as the GPU processing delay with potentially smaller data size for ML inference.

This initial pipeline leaves us with a follow-up given the impact of filter ordering on query performance (§2.2): *In what order should we place the filters?*

4.2.2 Selecting the Ordering of Filters

A naïve solution for selecting the filter ordering is to explore pipeline placement and configuration for all possible orderings; this, however, does not scale as the number of filters increases. In Vulcan, we propose a new solution that compares the impact of different filter orderings on query's accuracy, latency, and resource consumption by evaluating ***recall*** and ***precision*** of filters. We first explain how it works for a single filter before moving on to the multi-filter scenario. We define recall of a filter as the fraction of samples in the original data that contains the objects of interest (i.e., relevant data) that passes through the filter. On the other hand, precision of a filter is the fraction of data samples in its output that contains relevant data. For a given filter, we would expect its recall to be high such that it still captures most of the desired data, and query *accuracy* is preserved. A filter with low recall drops true positive samples which cannot be recovered later in the pipeline. Among filters with the same recall, we prefer the ones with higher precision because these filters provide higher data reduction rate by picking up fewer irrelevant samples, leading to better *latency and resource savings*.

We can now define the metric to evaluate how a given filter affects a query plan's utility ($U_{q,p,c}$). The metric should also handle a query's preference between latency and accuracy, based on the parameter γ defined in Eq (2) in our utility function. To this end, we leverage a variation of the F-measure in information retrieval theory [58] to encode this preference. Denote F_γ as the score for a given filter with its precision and recall measurements:

$$F_\gamma = (1+\beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}} \tag{4}$$

where $\beta = \gamma/(1-\gamma)$. In the F-measure definition, the value of β captures how many times recall is considered more important than precision. As we explained earlier, recall and precision corresponds to query accuracy and latency respectively, and thus the value of $\gamma/(1-\gamma)$ (see Eq (2)) is used to capture our accuracy-latency preference.

Measuring F_γ tells us how well a single filter fits into a query's optimal query plan. However, two challenges remain. First, directly applying F_γ to sort a sequence of filters does not work well as the recall of a filter can change based on its preceding filter. Second, filters may have their own configuration knob that leads to different precision or recall measurements. Applying one set of configuration for all filters oversimplifies the problem with inaccurate estimation, whereas evaluating too many configuration sets increases profiling cost.

We address multiple filters by treating a sequence of filters as a *bulk filter* with input being the data source and the output being the one from the last filter, and measure the overall F_γ using representative data for each permutation of the available filters. To deal with filters with various configurations, we choose a few representative configuration settings to capture the effect of configuration knobs on filters, where in each setting all the filters are configured at the x-th percentile in the range of their configuration knob. Filters with no configuration parameter remain unchanged. We empirically find x = 20-th, 50-th and 80-th good enough at picking promising pipelines, which results in $\{3 \times$ total number of filter ordering$\}$ F_γ values to collect. Vulcan then picks the ordering which achieves the highest F_γ to complete the pipeline construction.

4.3 Determining Placement Choices

After constructing the pipeline, the next step is placing each of the pipeline operators across the edge infrastructure (see Figure 5). On the one hand, rule-based solutions are good at reducing search cost but may fail to explore all promising placement choices. On the other hand, exhaustively searching through all placement choices requires high search cost during profiling as we deploy the query across the edge. Vulcan combines the benefits of the two approaches by (i) reducing search cost by reusing intermediate results from pipeline runs, and (ii) early pruning unpromising placement choices.

Algorithm 1: Placement Selection

```
1  Notation:
     q: constructed pipeline (from §4.2.1),
     P: all feasible placement choices given q,
     A_m: accuracy target, L_m: latency target, U: utility,
     res: pipeline results used to calculate utility
2  Function SelectBestPlacement(q, A_m, L_m):
3    P ← GenerateAllPlacementChoices(q)
4    foreach placement p ∈ P do
5      c ← NextConfig()        ▷ from §4.4
6      count = 0               ▷ reset early pruning counter
7      if c.hasExplored() then
8        res = LoadFromCache(q, c)
9      else
10       res ← LaunchPipeline(q, c)
11       CachePipelineResults(res)
12     U_{p,c} = CalculateUtility(q, p, res)
13     if U_{p,c} ≤ U_{p,c}(A_m, L_m) then
14       count ← count + 1
15       if count ≥ EarlyPruneCount then
16         continue
17   return arg max_{p∈P,c} U(p,c)
```

4.3.1 Reusing Pipeline Results

The idea of reusing pipeline results is based on two key observations we make in live ML pipelines. First, query accuracy does *not* depend on the placement choices of a pipeline with the same query configuration. Second, the amount of data generated after each pipeline operator is independent of placement choices. These observations allow us to deploy the pipeline ***offline*** *in the datacenter* ***only once*** *per selected query configuration* during the profiling stage to collect pipeline operator results, and *reuse* those results to evaluate a new placement choice by calculating additional *latency and resource consumption components* introduced by the placement, while reusing the same query accuracy result. Given a total of M placement choices and N combinations of pipeline configurations, our solution improves the search complexity from $O(MN)$ to $O(N)$ for a given pipeline.

Algorithm 1 describes how Vulcan evaluates placement choices. Given a pipeline, we begin with generating a collection of all feasible placement choices. Obviously infeasible choices where operators are placed in a different order than they appear in the pipeline (e.g., placing the DNN model in front of the filters) are excluded from the collection. For each placement choice, Vulcan explores promising query configurations to evaluate the utility of the query plan (details of how Vulcan picks query configurations are in Section 4.4). For every new set of pipeline configurations, Vulcan launches the pipeline inside the datacenter. In this case, Vulcan not only collects query performance and resource consumption for utility calculation but also caches intermediate results from each pipeline operator, including the operator's output size, output bandwidth, and data processing time (i.e., time spent in a filtering module or GPU inference time)[2] (Algorithm 1 lines 10-11). If a chosen pipeline configuration c has been explored by a previous placement choice, Vulcan calculates the utility for the new placement p by estimating the new query end-to-end latency $L_{p,c}$ and resource consumption $R_{p,c}$ without launching the pipeline (lines 7-8,12). $L_{p,c}$ is estimated by summing up the total processing time of each operator measured offline excluding the DNN module, the additional network latency introduced by placement, and the updated GPU inference latency as shown below:

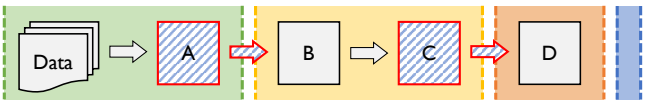


Figure 7: An example of how additional network latency is calculated for a pipeline with 4 operators placed across the edge infrastructure (i.e., device edge → on-premise edge → public MEC → cloud). Output sizes of shaded operators are used to calculate the additional latency introduced by the placement.

$$L_{p,c} = L_{offline,total} - L_{offline,gpu} + \sum L_{p,net} + L_{p,gpu} \quad (5)$$

The network component of the new latency, $\sum L_{p,net}$, is calculated by summing up the network latency going across two adjacent tiers. Figure 7 illustrates how this process works. The latency is calculated by taking the ratio of a component's output size (cached per configuration) to the assigned link bandwidth capacity the query data traverse through. Note that only the components sending data to the next tier in the infrastructure are considered. The GPU inference latency, L_{gpu}, is updated by multiplying with a coefficient based on the GPU type to reflect the performance difference, which we determined by profiling all GPUs available in our cluster. $R_{p,c}$ is estimated in a similar way by including additional network bandwidth and GPU processing time introduced by the placement. After all placement choices are explored, Vulcan selects the placement (together with the optimal configuration) that achieves the highest utility (line 17).

4.3.2 Pruning Unpromising Placement Choices

Although caching intermediate results allows Vulcan to never re-launch a pipeline with the same configuration, exhaustively exploring all feasible placement choices may still incur large search cost as the profiler strives to find a good configuration for an unpromising placement. For example, a query preferring latency performance will not favor the placement that places the filtering module too far away from the data source.

To apply early pruning, we set a utility threshold which is equal to the utility value when both minimum performance metrics is used for the given query configuration and placement, namely $U(Q_m, L_m)$, which is evaluated to zero in our utility definition (§4.1). When the profiler obtains N utility values below the threshold, we early prune the current placement

[2] Vulcan also collects the size and bandwidth of the data source to account for the case where only the data source is placed on the first tier.

choice (Algorithm 1 lines 13-16). Compared to other alternative pruning solutions, such as building a statistical model for pruning decisions, this simple scheme works well because Vulcan profiler is designed to always pick more promising configuration than its last attempt (§4.4), and consecutive bad utility values indicate a high probability of unpromising placement. This also allows us to set N to a small value ($N = 3$ in our implementation) as otherwise the profiler would have already terminated itself after a small number of attempts.

Placing split ML models. Recent research [21] has proposed the idea of splitting the layers of a large DNN model and placing them at different edge tiers. Vulcan can handle such design as long as the split layers are properly specified in the input queries as individual modules.

4.4 Determining Query Configuration

For each placement choice, Vulcan leverage Bayesian Optimization (BO) [14, 51] to efficiently explore pipeline configurations that achieve the best performance with minimized resource consumption (see Figure 5).

4.4.1 Why Do We Choose BO?

BO is a methodology for optimizing expensive objective functions and is widely applied to many computer systems [8, 13, 35, 42, 48, 64]. At a high level, it learns the shape of the objective function by picking inputs that has the highest probability of reaching the global maximum of the function. As BO accumulates more observations, it becomes more confident on the actual shape of the objective function. This is a good fit for our solution as we need to quickly find the optimal configuration that maximizes the utility function without exploring too many choices.

More importantly, we observe BO has many advantages over other popular optimization schemes in the context of live ML queries. Greedy Hill Climbing has been applied in query configuration for video analytics queries [70], but it is purely exploitative and cannot perform as efficient global exploration as BO does (§6.3 provides detailed evaluation). Multi-Armed Bandit (MAB) [25] is another popular optimization scheme for sequential decision making, but it is designed for optimizing cumulative rewards, contrasting our goal of finding the one configuration with the highest utility. In addition, BO tunes the entire set of input configurations all together for each iteration no matter how large the input vector is, whereas MAB can only adjust one knob at a time. We also prefer BO over population-based optimization scheme such as Particle Swarm Optimization (PSO), as applying PSO in query configuration requires launching many ML pipelines in parallel, leading to a much high computation cost (e.g., GPU resourse).

4.4.2 Applying BO to Query Configuration

We define the objective function of BO as $f(\vec{x})$, which models how good a given query plan is based on a given pipeline and a physical placement choice. The input $\vec{x}$ is the set of query configuration knobs, and the output of f is the utility value, $U_{q,p,c}$ for a given pipeline q with placement p and a set of configurations c. For each iteration, Vulcan launches the pipeline with the configurations suggested by BO (i.e., $\vec{x}$), and collects the measurements to compute $U_{q,p,c}$, which is then fed back to BO as the new observation.

Choice of prior and acquisition functions. Internally, BO learns an objective function by leveraging a *prior function* and an *acquisition function*. The former represents the belief about the space of possible objective functions, whereas the latter guides BO to choose the next promising input where the value of acquisition function is maximized. We choose Gaussian Process as the prior function and use *Matern 5/2* as its covariance function to describe the smoothness of the prior distribution, which is known to perform well among systems applying BO [8, 64]. Among three major choices of the acquisition function, namely probability of improvement [36] (PI), expected improvement [30] (EI), and upper confidence bound [65] (UCB), we select UCB as it works the best for our workloads. We defer a dynamic approach of selecting acquisition functions [15] to future work.

Starting and stopping BO. We start with N random sets of input query configurations as initial observations for BO to learn the rough shape of the objective function. We set $N = 3$ in all of our experiments and found it works well for various workload settings. Vulcan stops BO when the improvement of the utility value is less than a threshold for a few consecutive runs (i.e., 10% for 5 consecutive runs, which empirically works well). We include a sensitivity analysis on how the parameters we use in the starting and stopping conditions affect BO's performance in Section 6.7.

One alternative design which seems promising but we do not consider is to directly apply BO for placement selection and pipeline construction by treating available placement choices and pipelines as another two knobs in the total configuration space. In BO, input parameters are specified in a continuous range. For example, the voxel size in the voxelization module of an AD perception pipelines is chosen between 0.1 and 0.5. As BO moves within this range, the behavior of the module changes with the size of the voxel, leading to a smoother shape of the objective function with more predictable outputs. In contrast, the behavior of different placement choices or filter orderings is hard to predict, and going through the configuration space of those (i.e., placement or pipeline choices indexed by numbers) leads to very rough shape of the objective function for BO to grasp; thus it becomes much more difficult to find the global optimum.

5 ONLINE ADAPTATION

This section describes how Vulcan performs online adaptation to handle runtime dynamics after query deployment. Vulcan currently does not support concurrent execution of different ML queries, and we defer the support of multi-resource sharing among concurrent live ML pipelines to future work.

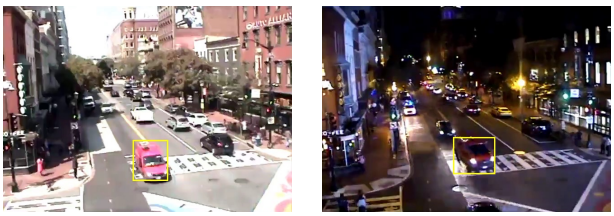


Figure 8: Scenes taken during different time of day from a video query detecting red vehicles.

5.1 Detecting and Handling Runtime Dynamics

Vulcan leverages two design ideas to quickly converge back to the best query plan during online adaption: (i) monitor utility change to detect runtime dynamics, and (ii) leverage prior knowledge during reprofiling.

Detecting runtime dynamics. Vulcan detects runtime dynamics by monitoring the change in a query's utility values, as any runtime dynamics (content, network, or compute based) leads to a change in utility. This includes all the measurements required to calculate up-to-date utility (i.e., latency, accuracy, module output size and bandwidth), and they are periodically reported to Vulcan monitor via HTTP requests. To obtain real-time ground truth on query accuracy given unlabeled live data, Vulcan launches a duplicated pipeline with the most expensive configuration inside the cloud, which only receives live data periodically to minimize network cost. A substantial change in the utility triggers reprofiling which deploys the query in the cloud similar to the case of offline profiling. We set the threshold of utility change empirically (10% in our implementation) via profiling.

Leveraging prior knowledge in reprofiling. We observe that most of the query, such as the object of interest and where the query takes place, remains the same after deployment; meaning, we can take the advantage of prior knowledge from offline profiling. Figure 8 shows two example scenes taken by the same camera during different time of the day from a query that detects red vehicles. We observe a high level of similarity between the two scenes except for environment illumination. Let us define the distance between two configurations, C_A and C_B, to be the total number of steps needed for each configuration knob in C_A to change to each knob value in C_B. In this example, applying the same configuration from daytime to nighttime scenes leads to an average 26.2% utility drop among all placement choices, but it requires only an average distance of 2.47 steps to converge back to the query plan with the highest utility (figures omitted in the interest of space).

To apply prior knowledge, Vulcan makes the following changes to the normal profiling process. Vulcan keeps track of the most recent top-K and worst-K configuration per placement choices ($K = 3$), and applies them as initial data points in BO such that BO can quickly grasp the shape of the objective function. We also tune up the exploitation factor in the acquisition function (κ in UCB [65]) to $1/10$ of its original value to focus more on exploitation than exploration. Note that Vulcan does not just stick to the best placement choice but choose to re-perform placement selection. As runtime dynamics can involve network and compute resource changes, this allows Vulcan to change the placement of the pipeline when merely adjusting query configuration makes little impact on recovering query performance. We evaluate how Vulcan performs under different types of runtime dynamics in Section 6.6.

```
1  HttpProcessor _httpProcessor = new HttpProcessor(
       Config.ENDPOINT_URL_SAMPLING);
2  Int sampling_rate = 3; // selecting 1 out of 3 frames
3  var form = new MultipartFormDataContent();
4  form.Add(new StringContent(sampling_rate.ToString()),
       "frameSamplingRate");
5  var result = await _httpProcessor.PostAsync(form);
```

(a) Sending Configuration Updates

```
1  // POST: api/Config
2  [Produces("application/json")]
3  [HttpPost]
4  public async Task<IActionResult> PostConfig([FromQuery
       ] string frameSamplingRate = null)
5  {
6    if (frameSamplingRate != null)
7    {
8      Config.FRAME_SAMPLING_RATE = Int32.Parse(
         frameSamplingRate);
9      /* sampling rate updated for subsequent frames */
10   }
11   return Ok();
12 }
```

(b) Posting Configuration Updates

Figure 9: Code snippets of Vulcan APIs on dynamically updating query configuration. (a) Vulcan Profiler sending the configuration updates to deal with runtime dynamics. (b) Vulcan Controller at the container updates the configuration to use the updated value.

5.2 Enabling Online Adaptation

After identifying the right query plan upon runtime dynamics, we must also enforce this at production scale. Unfortunately, existing frameworks for large-scale deployment, such as Kubernetes [7], do not support container modification during execution. Therefore, Vulcan adds its own implementation.

Dynamically updating query configuration. Figure 9 shows the example interfaces of Vulcan sending configuration updates and posting them inside the container upon a change in the frame sampling rate of a traffic monitoring query. In Vulcan, each pipeline module is installed and launched by a container. Query configurations are updated in real time without stopping the containers. If any configuration knobs need to be updated after reprofiling, Vulcan sends out updated configuration using HTTP requests to the containers that launch the corresponding modules (Figure 9a). The exact location (ENDPOINT_URL_SAMPLING) of the containers is acquired when Vulcan first deploys the pipeline. The containers receiving the request update the configuration right away, without stopping the current ML task (Figure 9b).

Handling placement changes. If the new query plan involves a placement change (e.g., during network or compute resource change), Vulcan migrates the container of the corresponding module to the updated tier. To perform the migration, Vulcan

Table 1: ML model variations used in evaluation.

Model	# Parameters (Million)				
YOLO [56]	**v5n6**	**v5s6**	**v5m6**	**v5l6**	**v5x6**
	3.2	12.6	35.7	76.8	140.7
PointPillars [37]	**SECFPN**		**SECFPN (FP16)**		
	4.9		4.9		
SSN [73]	**SECOND**		**RegNet**		
	6.2		7.1		
CenterPoint [68]	**DCN**		**Circular NMS**		
	6.0		6.1		
wav2vec2 [10]	**base**	**large_10m**		**large_960h**	
	94.4	315.5		315.5	
HuBERT [69]	**large**		**xlarge**		
	315.5		962.5		

first launches the module with the same query configuration on the target tier. It then updates the location of the new container to the upstream module via the same API for configuration updates described in §5.1, before removing the old container on the current tier. The new location of the container is also updated in Vulcan monitor for future adaptation.

6 EVALUATION

We evaluate the effectiveness of Vulcan in performing automatic query planing in terms of profiling time, query performance, and query resource consumption. Our key findings:

(1) Vulcan generates query plan with better profiling cost by 4.1×-30.1× over state-of-the-art ML analytics systems while delivering up to 2.0×-3.3× better latency (§6.2).

(2) Vulcan performs better query configuraiton, placement selection, and pipeline construction by outperforming existing solutions with up to 2.8× better query latency and 174× lower network resource consumption (§6.3-§6.5).

(3) Vulcan achieves consistently better 99th-p latency performance (by up to 2.5×) during online adaptation over the-state-of-the-art (§6.6).

6.1 Experiment Setup

Live ML Queries. We illustrate Vulcan's performance in performing query planning for three example live ML queries:

- *Video Monitoring*: monitors the traffic volume by examining live video frames and counting the vehicles of a specific color (white color in our examples).
- *Autonomous Driving Perception*: takes 3D point cloud as input and generates a real-time perception (represented as 3D bounding boxes) surrounding a vehicle.
- *Automatic Speech Recognition*: converts live human speech (with background noises) into written text.

Datasets. We adopt several real-world datasets for each query example to perform a comprehensive evaluation. The video monitoring queries use videos captured by traffic cameras among different metropolitan areas in Bellevue and Washington D.C. The autonomous driving queries use LiDAR sensor data from nuScenes [16]. Automatic speech recognition queries use the VOiCES dataset [57] with background noise enabled. Appendix A.1 has more details.

Pipelines. Unless otherwise specified, video monitoring queries in our evaluation experiments use the same pipeline which consists of the following components in order: a background subtractor to detect moving vehicles, a color filter, and a variations of YOLOv5 [5] object detector. The autonomous driving pipeline feeds 3D point clouds into a ground removal module, followed by a voxelization module and a variations of PointPillars [37], SSN [73], or CenterPoints [68] 3D object detector. The speech recognition queries use a pipeline that consists of an audio sampler, a noise reduction module, a variation of wav2vec2 [10] or HuBERT [69] model, and a decoder. Table 1 records all ML model variations we use in the evaluation. Appendix A.1 describes the details of the query configuration knobs for all queries.

Baselines. We consider the following baselines for Vulcan.

• *Exhaustive search.* To determine the optimal placement and configuration for a ML pipeline, we use exhaustive search to explore all possible placement choices and configurations.

• *ML analytics systems.* We implement four state-of-the-art ML analytics systems: VideoStorm [70], Chameleon [28], JellyBean [67], and LLAMA [60]. Both VideoStorm and Chameleon adopt a variant of greedy-hill climbing during query profiling, and Chameleon applies spatial and temporal correlations to reduce the search cost during online adaptation. JellyBean selects models with target accuracy and lowest cost, and applies beam search to determine query placement in a greedy fashion. LLAMA dynamically explores query configuration by computing per-invocation pipeline latency.

• *Pipeline placement strategies.* Besides JellyBean's greedy placement, we implement the following commonly-adopted placement strategies: (1) *prioritizing network (PN)* which places operators closer to the device edge, (2) *prioritizing compute (PC)* which places operators closer to the cloud, and (3) *balancing network and compute (NC)* which places filtering modules closer to the device edge and the remaining operators to the cloud.

Emulation Setup. We emulate a setup of 4 edge tiers consisting of the device edge, the on-prem edge, the public MEC, and the cloud, which is consistent with our production edge infrastructure by adding additional network latency and compute overhead. The network bandwidth and compute cost are emulated based on the real network and hardware settings in our production infrastructure.

We run all experiments 20 times with different random seeds, and collect 10th, 50th, and 90th percentiles of data to include the effect of randomness in Vulcan and other baselines. The 10th and 90th percentiles are plotted via error bars unless otherwise specified. We set L_m for video monitoring, autonomous driving, and speech recognition queries as 2000ms, 400ms, and 500ms, respectively. Q_m is set at 0.8 × the accuracy achieved by the most expensive configuration for each type of queries. We set γ (preference of query accuracy over latency) to be 0.5 for all queries unless otherwise specified.

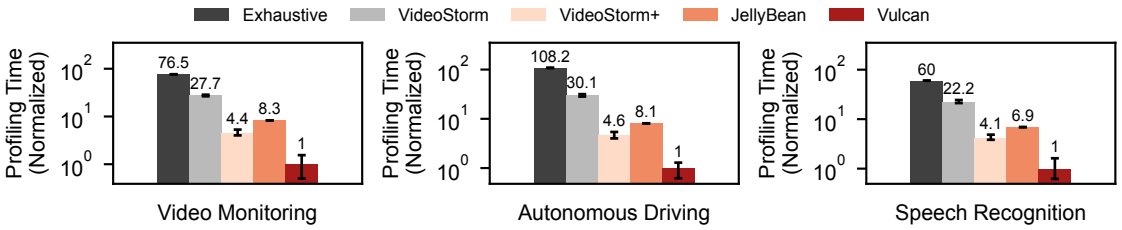


Figure 10: Comparing the profiling cost of video monitoring, autonomous driving, and speech recognition queries.

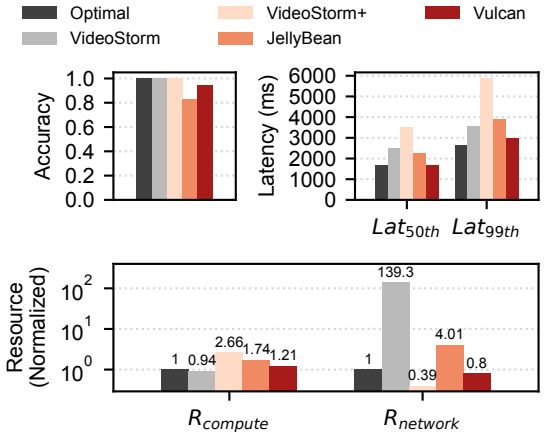


Figure 11: Comparing end-to-end performance and resource consumption for video monitoring queries.

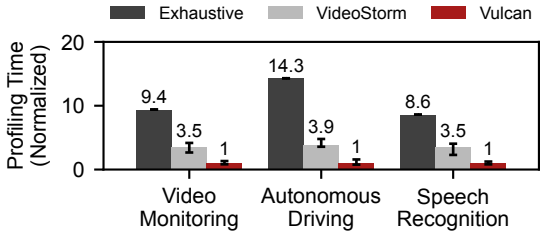


Figure 12: Profiling cost of query configuration given the same pipeline and placement.

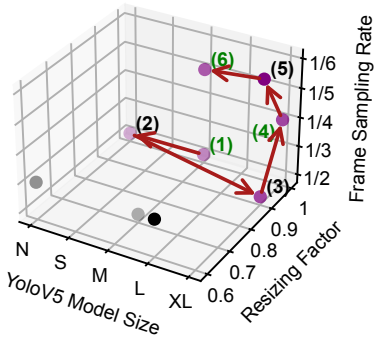


Figure 13: BO's search path in selecting configurations. The new maximum in utility is marked in green. The initial random configurations are shown in black.

6.2 End-to-End Improvement

We start with showing the end-to-end improvement of Vulcan over other baselines in generating query plans for all three types of queries. We compare exhaustive search, original VideoStorm that explores all placement choices, VideoStorm+ that explores a combination of all three baseline placement strategies (i.e., PN, PC and NC) on top of VideoStorm, and JellyBean. We compare Vulcan with Chameleon and LLAMA during online adaptation in §6.6 as their query configuration by design happens in the online phase.

We record profiling time normalized by the time spent by Vulcan, which achieves the smallest value in both types of queries. We also record performance (accuracy, median latency, and 99th-p latency) and resource consumption (network and compute) achieved by the query plan and normalize the results based on the optimal query plan generated by exhaustive search. For VideoStorm, VideoStorm+, and JellyBean, we record the query plan achieved with highest utility given the same profiling time as Vulcan.

Figure 10 records the profiling cost for all types of queries, and Figure 11 summarizes the performance and resource consumption for video monitoring queries. Results for autonomous driving and speech recognition queries are similar and moved to Appendix A.2 in the interest of space. Vulcan achieves significantly lower profiling cost than exhaustive search, VideoStorm, VideoStorm+, and JellyBean by at least 60×, 22.2×, 4.1×, and 6.9× respectively across all three types of queries, and its query performance and resource consumption are very close to the optimal one achieved through exhaustive search, achieving up to 2.0×-3.3× better tail latency than other baseline approaches. Vulcan's performance improvement comes from its joint optimization of pipeline placement and configuration with low cost. Vulcan avoids exhaustively searching for optimal placement by reusing pipeline results. VideoStorm+ improves profiling time by only exploring a few placement choices, at the cost of worse latency and network resource consumption, and it is still outperformed by Vulcan by at least 4.4×. JellyBean's placement algorithm greedily finds promising placement choices but separately optimizes query configuration and placement, leading to sub-optimal latency, accuracy, and resource usage.

6.3 Selecting Better Query Configurations

We next delve into the performance of each profiling components in Vulcan, starting with query configuration. Figure 12 shows the profiling cost among Vulcan and other baselines with the same pipeline and the best placement choice. Even without the benefits achieved in efficient placement selection, Vulcan can still outperform VideoStorm, which leverages greedy hill climbing, by 3.5× in profiling time. To illustrate how Vulcan achieves this, we plot one of Vulcan's example

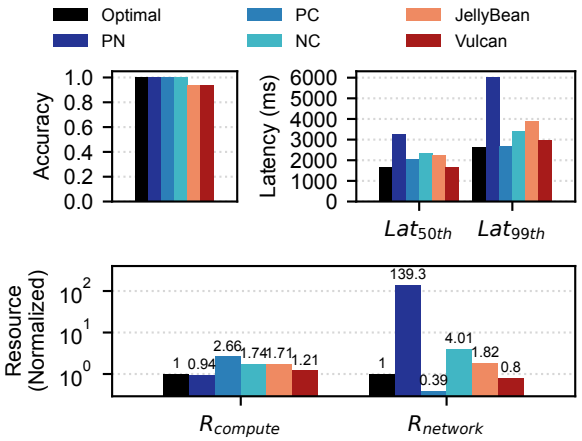


Figure 14: Comparing Vulcan with different placement strategies on serving video monitoring queries.

BO search path in the video monitoring query in Figure 13. BO focuses its exploration on larger resizing factor after verifying that smaller values lead to worse performance, and finds the optimal query configuration at the 6th step. BO takes 5 more steps to confirm we are not likely to encounter better results before stopping, which we do not plot for legibility.

6.4 Selecting Better Placement

We evaluate Vulcan's placement decisions by comparing with common placement strategies (§6.1) and JellyBean. Each baseline placement strategy uses exhaustive search to find the optimal pipeline configuration, and all comparisons use the same pipeline. We ignore the profiling time improvement of Vulcan over other baselines and only focus on comparing the achieved performance and resource consumption. Figure 14 illustrates the results of video monitoring queries. Results for autonomous driving and speech recognition queries are similar and shown in Appendix A.3. We observe that PN prioritizes network latency but fails to consider the large ML inference latency on slower compute nodes, resulting in worse end-to-end query latency. PC optimizes ML inference latency, which does not always lead to better latency. Moreover, it consumes significantly more network resources. NC tries to strike a balance between PN and PC but still achieves worse latency. JellyBean's placement algorithm is based on a fixed query configuration that is determined in a prior stage, leading to a sub-optimal placement choice. In comparison, Vulcan always achieves the same placement choices as the exhaustive search and delivers up to 2.8× better query latency and 174× network resource consumption than other baselines, thanks to its joint optimization between placement and configuration. The performance gap between Vulcan's query plan and the optimal query plan is caused by Vulcan picking a different pipeline configuration, in which case Vulcan takes much less profiling time with similar performance and resource consumption.

6.5 Selecting Better Pipelines

We now evaluate how well Vulcan selects the filter ordering during pipeline construction based on performance requirements of the queries. We compare the performance of fixed pipelines with Vulcan's choices, which is dynamically determined based on F_γ in Eq 4 (§4.2). To focus on whether Vulcan makes the correct choice of filter ordering, we remove the potential noise of BO by using the best pipeline configuration determined by exhaustive search. We sweep γ and record the utility value achieved by picking the corresponding pipeline. Figure 15 shows that Vulcan is able to select the best pipeline in 8 out of 9 cases. On the other hand, sticking to the other two fixed pipeline settings leads to only 6 out of 9 cases and 3 out of 9 cases for selecting the correct filter ordering.

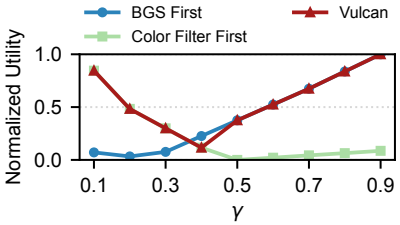


Figure 15: Compare Vulcan's selection of filter ordering with fixed pipeline settings in video monitoring queries.

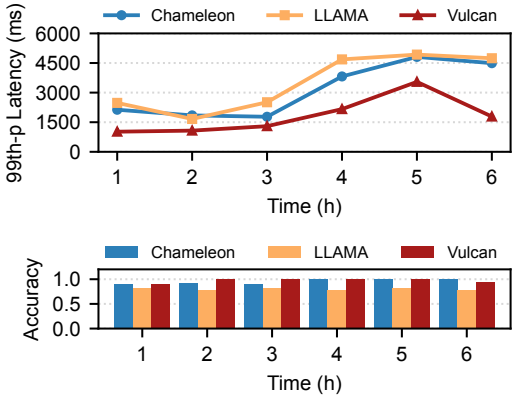
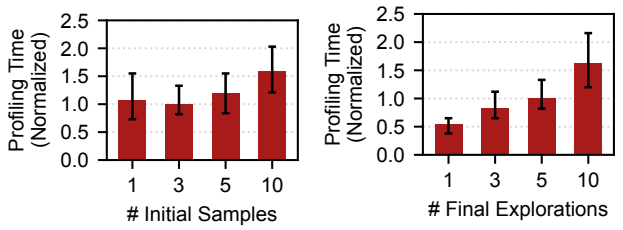


Figure 16: Comparing Vulcan with Chameleon during online adaptation for video monitoring queries.

6.6 Handling Runtime Dynamics

To evaluate Vulcan's online adaptation, we compare it with Chameleon and LLAMA, the state-of-the-art solution in adapting runtime dynamics in ML queries. We took a continuously running 6-hour long video from our Washington D.C. video dataset to monitor red vehicles. Performance (99th-p latency and accuracy) was collected and reported at the end of every hour. All three systems started with the same video monitoring pipeline and placement, where the background subtractor and the color filter are placed on the device edge, and the object detector is placed on the public MEC. At the end of the 3rd hour, we reduced the link capacity from the on-premise edge to the Public MEC by 20× to simulate a network outage. We see in Figure 16 that Vulcan achieves consistently better latency than Chameleon and LLAMA by up to 2.5×. Neither Chameleon nor LLAMA updates pipeline placement upon resource changes, causing worse latency after the outage. On the other hand, Vulcan adjusted the pipeline



(a) Starting Condition **(b)** Stopping Condition

Figure 17: Sensitivity analysis of BO's parameters.

placement by placing the object detector onto the on-premise edge after the outage, thus mitigating the latency hikes.

6.7 Sensitivity Analysis

We analyze the sensitivity of two parameters used in BO during pipeline configuration in Figure 17. We use the same experiment setting in Figure 12 with video monitoring queries. Figure 17a plots the profiling time normalized by the value picked in Vulcan implementation when sweeping the number of initial random configurations passed to BO. The number of random configurations does affect the profiling time but has a negligible impact on the quality of the query plan achieved. Figure 17b sweeps the stopping threshold of BO, which is the number of consecutive rounds without significant improvement. BO needs at least 3 rounds to obtain a good chance of finding optimal configurations, but adding more rounds won't further improve the quality of profiling.

7 RELATED WORK

Configuration management for ML analytics. Finding better configurations for ML analytics is a well studied research topic [28, 70]. VideoStorm [70] investigates query's accuracy-latency profile and applies a greedy hill-climbing approach to search for query configurations. Chameleon [28] applies spatial and temporal correlations of video frames to reduce the search cost of query configurations during online adaptation. Although VideoStorm and Chameleon works well for ML quries with fixed pipeline placement, Vulcan jointly explores placement and configurations for live ML queries deployed across a heterogeneous infrastructure.

Constructing ML pipelines with declarative queries. Researchers have proposed declarative query languages, especially for video analytics, to construct ML pipelines [11, 19, 44, 59]. MIRIS [11] provides a declarative interface to select video object tracks and selects corresponding modules based on user specification. Viva [59]'s declarative interface allows users to specify relational hints to express the relationship among different modules, which helps Viva to construct and optimize the pipeline. However, those works only consider accuracy requirement when optimizing pipelines and ignore the resource demands especially network usage, providing no guarantees to end-to-end latency performance.

Domain-specific optimization for ML analytics Applying domain-specific knowledge to perform optimization for ML analytics is an active research area spanning many use cases including video analytics [11, 19, 28], autonomous driving perception [55, 71, 72], and automatic speech recognition [20, 41]. BlazeIt [19] leverages spatiotemporal information of objects in video to optimize aggregation and limit queries. VI-Eye [72] achieves better accuracy and latency performance by exploiting domian knowledge in autonomous driving scenarios to recognize key semantic objects that can be used to align vehicle-infrastructure point cloud pairs. Vulcan's design is orthogonal to domain-specific techniques and can be applied to different use cases without additional changes.

Continuous learning for ML analytics. Another line of work that gets increasingly more attention nowadays is continuous learning in ML analytics [12, 34, 46]. Ekya [12] jointly schedules and allocate resources to ML retraining and inference to handle data drift. RECL [34] integrates model reusing with model retraining to quickly adapt to a lightweight expert DNN model for each specific video scenes. Techniques leveraging continuous learning is complementary to Vulcan and can be applied to Vulcan's online adapation phase. On the other hand, Vulcan's design can be applied to those works as well, including dynamically updating query configurations and fast detection of data changes by monitoring utility changes.

Reducing cost of ML analytics via filtering. There have been recent studies on applying different types of filtering techniques to reduce the resource consumption of ML analytics without compromising accuracy [17, 24, 26, 32, 47]. NoScope [32] searches for and trains a cascade of models that preserves the accuracy of the ML inference but with far less computation cost. Focus [26] uses cheap convolutional network classifiers (CNNs) to construct an approximate index of all possible object classes in the video frame, which reduces the use of expensive CNNs during query time. Probabilistic Predicates [47] constructs and applies binary classifiers to filter out data blob that will not pass query predicate and thus accelerate queries with expensive user-defined functions. Vulcan builds up on the idea of applying filtering and propose a novel technique to order the filters for ML pipelines.

8 CONCLUSION

Serving live ML analytics involves constructing, placing, and configuring ML pipelines as well as their online adaptation. However, existing query planning solutions for live ML queries remain elusive with piecemeal and sub-optimal. We present Vulcan, an ML analytics system that performs automatic query planning for live ML analytics. Vulcan automatically construct the pipeline and determine the best ordering of filtering operators for query performance. It efficiently explores placement choices by reusing of intermediate pipeline profiling results, and leverage Bayesian optimization with prior knowledge to handle query configuration and online adaptation. Vulcan outperforms state-of-the-art solutions on profiling time, query latency, and resource consumption when serving queries with real-world datasets.

ACKNOWLEDGEMENTS

We would like to thank Jae-Won Chung, the anonymous NSDI reviewers, and our shepherd, Danyang Zhuo, for providing valuable feedback. This work was supported in part by NSF grants CNS-1845853, CNS-2104243, and CNS-2106184.

REFERENCES

[1] Microsoft rocket for live video analytics. https://www.microsoft.com/en-us/research/project/live-video-analytics/, 2020.

[2] Build modern connected applications at the edge with 5g. https://azure.microsoft.com/en-us/blog/how-developers-can-benefit-from-the-new-5g-paradigm/, 2022.

[3] Edge video service (evs). https://azure.microsoft.com/en-us/blog/microsoft-and-att-demonstrate-5gpowered-video-analytics/, 2022.

[4] Microsoft rocket for live video analytics. https://azure.microsoft.com/en-us/blog/microsoft-and-att-are-accelerating-the-enterprise-customer-s-journey-to-the-edge-with-5g/, 2022.

[5] Yolov5. https://github.com/ultralytics/yolov5, 2022.

[6] Azure public multi-access edge compute (mec). https://azure.microsoft.com/en-us/solutions/public-multi-access-edge-compute-mec/, 2023.

[7] Kubernetes. https://github.com/kubernetes/kubernetes, 2023.

[8] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, 2017.

[9] Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, and Olivier Verscheure. Adaptive control of extreme-scale stream processing systems. In *ICDCS*, 2006.

[10] Alexei Baevski, Henry Zhou, Abdelrahman Mohamed, and Michael Auli. wav2vec 2.0: A framework for self-supervised learning of speech representations, 2020.

[11] Favyen Bastani, Songtao He, Arjun Balasingam, Karthik Gopalakrishnan, Mohammad Alizadeh, Hari Balakrishnan, Michael Cafarella, Tim Kraska, and Sam Madden. Miris: Fast object track queries in video. In *SIGMOD*, 2020.

[12] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *NSDI*, 2022.

[13] Eric Brochu, Tyson Brochu, and Nando de Freitas. A bayesian interactive optimization approach to procedural animation design. In *SCA*, 2010.

[14] Eric Brochu, Vlad M. Cora, and Nando de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning, 2010.

[15] Eric Brochu, Matthew W. Hoffman, and Nando de Freitas. Portfolio allocation for bayesian optimization. In *UAI*, 2011.

[16] Holger Caesar, Varun Bankiti, Alex H Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuscenes: A multimodal dataset for autonomous driving. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11621–11631, 2020.

[17] Christopher Canel, Thomas Kim, Giulio Zhou, Conglong Li, Hyeontaek Lim, David G Andersen, Michael Kaminsky, and Subramanya Dulloor. Scaling video analytics on constrained edge nodes. In *MLSys*, 2019.

[18] Google Cloud. Speech-to-text: Automatic speech recognition. https://cloud.google.com/speech-to-text, 2022.

[19] Matei Zaharia Daniel Kang, Peter Bailis. Blazeit: Optimizing declarative aggregation and limit queries for neural network-based video analytics. In *VLDB*, 2020.

[20] Nilaksh Das, Monica Sunkara, Dhanush Bekal, Duen Horng Chau, Sravan Bodapati, and Katrin Kirchhoff. Listen, know and spell: Knowledge-infused subword modeling for improving asr performance of oov named entities. In *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7887–7891, 2022.

[21] John Emmons, Sadjad Fouladi, Ganesh Ananthanarayanan, Shivaram Venkataraman, Silvio Savarese, and Keith Winstein. Cracking open the dnn black-box: Video analytics with dnns across the camera-cloud boundary. In *HotEdgeVideo*, 2019.

[22] Alireza Ghasemieh and Rasha Kashef. 3d object detection for autonomous driving: Methods, models, sensors, data, and challenges. *Transportation Engineering*, 8:100115, 2022.

[23] Tiago Gomes, Diogo Matias, André Campos, Luís Cunha, and Ricardo Roriz. A survey on ground segmentation methods for automotive lidar sensors. *Sensors*, 23(2), 2023.

[24] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *MobiSys*, 2016.

[25] Daniel N Hill, Houssam Nassif, Yi Liu, Anand Iyer, and S V N Vishwanathan. An efficient bandit algorithm for realtime multivariate optimization. In *KDD*, 2017.

[26] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *OSDI*, 2018.

[27] Samvit Jain, Xun Zhang, Yuhao Zhou, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Paramvir Bahl, and Joseph Gonzalez. Spatula: Efficient Cross-camera Video Analytics on Large Camera Networks. In *ACM/IEEE Symposium on Edge Computing (SEC)*, 2020.

[28] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: Scalable adaptation of video analytics. In *SIGCOMM*, 2018.

[29] Ramesh Johari and John N. Tsitsiklis. Efficiency loss in a network resource allocation game. *Mathematics of Operations Research*, pages 29(3):407–435, 2004.

[30] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, page 13(4):455–492, 1998.

[31] Kai Kai Jüngling and Michael Arens. Local feature based person reidentification in infrared image sequences. In *IEEE AVSS*, 2010.

[32] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: Optimizing neural network queries over video at scale. In *PVLDB*, 2017.

[33] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *ASPLOS*, 2017.

[34] Mehrdad Khani, Ganesh Ananthanarayanan, Kevin Hsieh, Junchen Jiang, Ravi Netravali, Yuanchao Shu, Mohammad Alizadeh, and Victor Bahl. Recl: Responsive resource-efficient continuous learning for video analytics. In *NSDI*, 2023.

[35] Yuki Koyama, Issei Sato, Daisuke Sakamoto, and Takeo Igarashi. Sequential line search for efficient visual design optimization by crowds. In *ACM Transactions on Graphics*, 2017.

[36] Harold J. Kushner. A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. *Journal of Basic Engineering*, page 86:97–106, 1964.

[37] Alex H Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12697–12705, 2019.

[38] Wei Li, Rui Zhao, Tong Xiao, and Xiaogang Wang. Deepreid: Deep filter pairing neural network for person re-identification. In *CVPR*, 2014.

[39] Zhuqi Li, Yuanchao Shu, Ganesh Ananthanarayanan, Longfei Shangguan, Kyle Jamieson, and Paramvir Bahl. Spider: A Multi-Hop Millimeter-Wave Network for Live Video Analytics. In *ACM/IEEE Symposium on Edge Computing (SEC)*, 2021.

[40] Giuseppe Lisanti, Iacopo Masi, Andrew D. Bagdanov, and Alberto Del Bimbo. Person re-identification by iterative re-weighted sparse ranking. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014.

[41] Linda Liu, Yi Gu, Aditya Gourav, Ankur Gandhe, Shashank Kalmane, Denis Filimonov, Ariya Rastrow, and Ivan Bulyko. Domain-aware neural language models for speech recognition. In *ICASSP 2021*, 2021.

[42] Daniel Lizotte, Tao Wang, Michael Bowling, and Dale Schuurmans. Automatic gait optimization with gaussian process regression. In *IJCAI*, 2007.

[43] Franz Loewenherz. Video analytics towards vision zero. https://bellevuewa.gov/sites/default/files/media/pdf_document/video-analytics-presentation-ITE-conference-021317.pdf, 2017.

[44] Chenglang Lu, Mingyong Liu, and Zongda Wu. Svql: A sql extended query language for video databases. In *IJDTA*, 2015.

[45] Yan Lu, Shiqi Jiang, Ting Cao, and Yuanchao Shu. Turbo: Opportunistic Enhancement for Edge Video Analytics. In *ACM Conference on Embedded Network Sensor Systems (SenSys)*, 2022.

[46] Yan Lu, Zhun Zhong, and Yuanchao Shu. Multi-View Domain Adaptive Object Detection in Surveillance Cameras. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2023.

[47] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. Accelerating machine learning inference with probabilistic predicates. In *SIGMOD*, 2018.

[48] Ruben Martinez-Cantin, Nando de Freitas, Eric Brochu, Jose Castellanos, and Arnaud Doucet. A bayesian exploration-exploitation approach for optimal online sensing and planning with a visually guided mobile robot. *Autonomous Robots*, pages 27(2):93–103, 2009.

[49] Massimo Merenda, Carlo Porcaro, and Demetrio Iero. Edge machine learning for ai-enabled iot devices: A review. *Sensors*, 20(9), 2020.

[50] Robert C. Merton. *Continuous-Time Finance*. Blackwell, 1990.

[51] Jonas Mockus. *Bayesian Approach to Global Optimization.* Kluwer Academic, 1989.

[52] Shadi Noghabi, Landon Cox, Sharad Agarwal, and Ganesh Ananthanarayanan. The emerging landscape of edge-computing. In *ACM SIGMOBILE GetMobile*, 2020.

[53] Pirouz Nourian, Romulo Gonçalves, Sisi Zlatanova, Ken Arroyo Ohori, and Anh Vu Vo. Voxelization algorithms for geospatial applications: Computational methods for voxelating spatial datasets of 3d city models containing 3d surface, curve and point data models. *MethodsX*, 3:69–86, 2016.

[54] Arthi Padmanabhan, Neil Agarwal, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Guoqing Harry Xu, and Ravi Netravali. GEMEL: Model Merging for Memory-Efficient, Real-Time Video Analytics at the Edge. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[55] Hang Qiu, Pohan Huang, Namo Asavisanu, Xiaochen Liu, Konstantinos Psounis, and Ramesh Govindan. Autocast: Scalable infrastructure-less cooperative perception for distributed collaborative driving. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '22, 2022.

[56] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. In *CVPR*, 2017.

[57] Colleen Richey, Maria A. Barrios, Zeb Armstrong, Chris Bartels, Horacio Franco, Martin Graciarena, Aaron Lawson, Mahesh Kumar Nandwana, Allen Stauffer, Julien van Hout, Paul Gamble, Jeff Hetherly, Cory Stephenson, and Karl Ni. Voices obscured in complex environmental settings (voices) corpus, 2018.

[58] Van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, 1979.

[59] Francisco Romero, Johann Hauswald, Aditi Partap, Daniel Kang, Matei Zaharia, and Christos Kozyrakis. Optimizing video analytics with declarative model relationships. *Proc. VLDB Endow.*, 16(3):447–460, nov 2022.

[60] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *SoCC*, 2021.

[61] sBrian T. Ratchford. Cost-benefit models for explaining consumer choice and information seeking behavior. *Management Science*, 28, 1982.

[62] Shuyao Shi, Jiahe Cui, Zhehao Jiang, Zhenyu Yan, Guoliang Xing, Jianwei Niu, and Zhenchao Ouyang. Vips: Real-time perception fusion for infrastructure-assisted autonomous driving. In *MobiCom*, 2021.

[63] Jiang Shiqi, Lin Zhiqi, Li Yuanchun, Shu Yuanchao, and Liu Yunxin. Flexible High-resolution Object Detection on Edge Devices with Tunable Latency. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2021.

[64] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *NIPS*, 2012.

[65] Niranjan Srinivas, Andreas Krause, Sham M. Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. In *ICML*, 2010.

[66] Voci. Voci: Real-time speech recognition. https://www.vocitec.com/ads/real-time-speech-to-text, 2022.

[67] Yongji Wu, Matthew Lentz, Danyang Zhuo, and Yao Lu. Serving and optimizing machine learning workflows on heterogeneous infrastructures. In *VLDB*, 2023.

[68] Tianwei Yin, Xingyi Zhou, and Philipp Krahenbuhl. Center-based 3d object detection and tracking. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11784–11793, 2021.

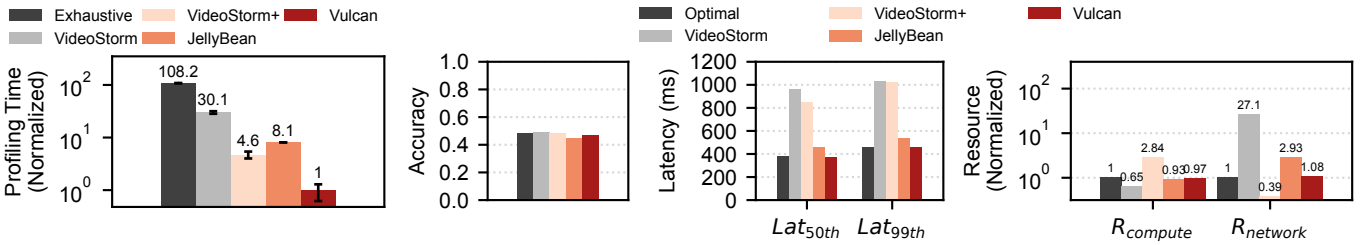
[69] Ji Won Yoon, Beom Jun Woo, and Nam Soo Kim. Hubert-ee: Early exiting hubert for efficient speech recognition, 2022.

[70] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *NSDI*, 2017.

[71] Xumiao Zhang, Anlan Zhang, Jiachen Sun, Xiao Zhu, Y. Ethan Guo, Feng Qian, and Z. Morley Mao. Emp: Edge-assisted multi-vehicle perception. In *MobiCom*, 2021.

[72] Xumiao Zhang, Anlan Zhang, Jiachen Sun, Xiao Zhu, Y. Ethan Guo, Feng Qian, and Z. Morley Mao. Emp: Edge-assisted multi-vehicle perception. In *MobiCom*, 2021.

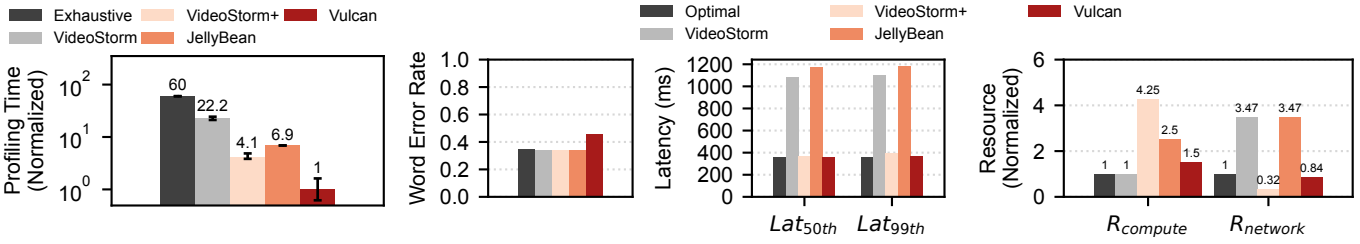
[73] Xinge Zhu, Yuexin Ma, Tai Wang, Yan Xu, Jianping Shi, and Dahua Lin. Ssn: Shape signature networks for multi-class object detection from point clouds. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXV 16*, pages 581–597. Springer, 2020.



(a) Profiling Cost

(b) Performance and Resource Consumption

Figure 18: End-to-end performance when exploring the best placement and query configuration for autonomous driving queries.



(a) Profiling Cost

(b) Performance and Resource Consumption

Figure 19: End-to-end performance when exploring the best placement and query configuration for speech recognition queries.

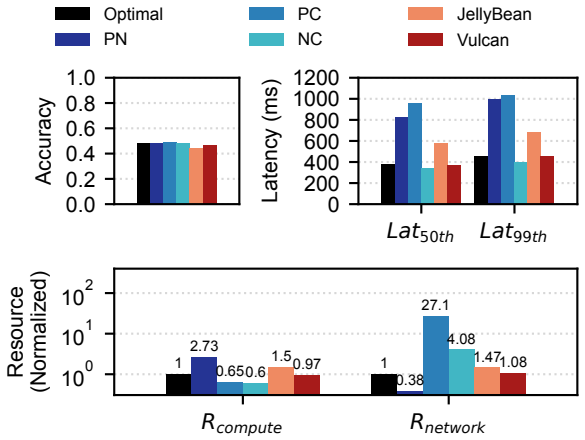


Figure 20: Comparing Vulcan with different placement strategies on serving AD perception queries.

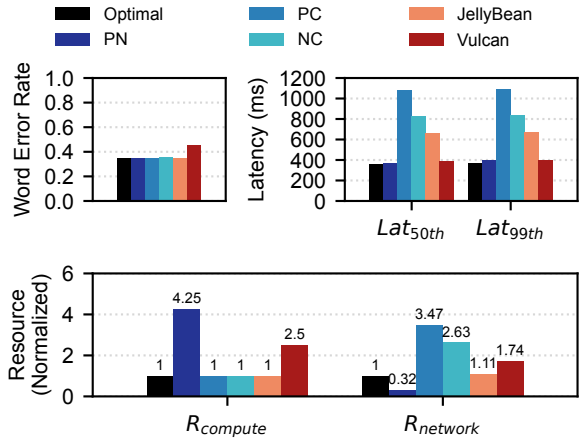


Figure 21: Comparing Vulcan with different placement strategies on serving ASR queries.

A ADDITIONAL EVALUATION RESULTS

A.1 Datasets and Query Configuration

We describe below the details of the datasets and query configuration we used in the paper.

Datasets:

The traffic monitoring queries use videos captured by traffic cameras among different metropolitan areas in Bellevue and Washington D.C. The videos are encoded in MP4 format (1280x720p, 30fps, and 120 seconds long) and randomly sampled from two-hour long videos among 24 days.

For autonomous driving perception queries, we use LiDAR sensor data from nuScenes [16], a large-scale autonomous driving dataset. The dataset features 1000 20-second driving scenes collected over months, in Boston and Singapore encompassing a diverse range of challenging driving situations.

Automatic speech recognition queries use the VOiCES dataset [57], an English speech audio dataset by male and female speakers. The dataset contains 3903 audio files (total 15 hours long) containing different room settings, simulated head movement, and various background noise patterns.

Configuration Knobs. Besides the variation of ML models described in §6.1, our queries in Evaluation configure the following additional configuration knobs.

- Video monitoring queries configure input frame sampling rate (1/2, 1/3, 1/4, 1/5, 1/6) and frame resizing factor (0.6, 0.7, 0.8, 0.9, 1) for frame resolution.
- Autonomous driving queris configure the ground removal factor[3] and voxel size, each with 5 configuration values (0.1, 0.2, 0.3, 0.4, 0.5).
- Speech recognition queires configure audio sampling rate (8k, 10k, 12k, 14k, 16k) and frequency mask width (500,

[3] corresponds to the maximum distance between a ground point and the estimated 2D ground plane.

1000, 2000, 3000, 4000) of the noise reduction module.

A.2 End-to-End Improvement

Figure 18 and Figure 19 shows the comparison between Vulcan and other baselines in the end-to-end performance of autonomous driving and speech recognition queries. The same conclusion can drawn as mentioned in §6.2.

A.3 Selecting Better Placement

Figure 20 and Figure 21 records the performance and resource consumption of autonomous driving and speech recognition queries, where PN, PC, and NC use the optimal query plan achieved by exhaustive search. Similar to §6.4, Vulcan always achieves the best placement choice, outperforming other baselines in query performance and resource consumption.

CASSINI: Network-Aware Job Scheduling in Machine Learning Clusters

Sudarsanan Rajasekaran[†] *Manya Ghobadi*[†] *Aditya Akella*[‡]

[†]*Massachusetts Institute of Technology* [‡]*UT Austin*

Abstract

We present CASSINI, a network-aware job scheduler for machine learning (ML) clusters. CASSINI introduces a novel geometric abstraction to consider the communication pattern of different jobs while placing them on network links. To do so, CASSINI uses an Affinity graph that finds a series of time-shift values to adjust the communication phases of a subset of jobs such that the communication patterns of jobs sharing the same network link are interleaved with each other. Experiments with 13 common ML models on a 24-server testbed demonstrate that compared to the state-of-the-art ML schedulers, CASSINI improves the average and tail completion time of jobs by up to 1.6× and 2.5×, respectively. Moreover, we show that CASSINI reduces the number of ECN marked packets in the cluster by up to 33×.

1 Introduction

The ever-growing increase in dataset and model sizes of deep learning has created a massive demand for efficient GPU clusters. Several studies have demonstrated that as the number of GPUs increases, the communication overhead of distributed Machine Learning (ML) training workloads quickly takes up a significant portion of training iteration time [12, 15, 28, 33, 45, 47, 55]. Yet state-of-the-art ML schedulers tend to ignore the communication pattern of ML training jobs when placing workers on GPUs.

In this paper, we develop a simple but effective approach, called CASSINI, that integrates with existing ML schedulers to allow them to efficiently place multiple ML jobs on network links while minimizing the chances of network congestion. Our approach requires no special support, such as reservations and priorities, from switches/NICs and does not require any changes to the congestion control protocol.

We demonstrate that for a specific combination of jobs, introducing a small time-shift to delay the start of one of the iterations enables CASSINI to interleave the computation and communication patterns of different jobs, thereby improving the training time. We refer to such combinations of jobs as *compatible* and develop CASSINI as a pluggable module to augment prior ML schedulers to consider a novel *compatibility metric* when determining where to place jobs and control how jobs compete on network links.

Augmenting ML schedulers to take links and servers into account is inherently challenging because jobs are likely to traverse multiple links and may compete with different jobs on different links. To address this challenge, we propose a *geometric abstraction* that leverages the periodic communication pattern of Deep Neural Network (DNN) training workloads. The key idea of our abstraction is to "roll" time around a circle whose perimeter is proportional to the training iteration time of ML jobs. To determine the compatibility score of two (or more) jobs on a link, CASSINI places each job on its corresponding circle and overlays the circles on top of each other. It then uses an optimization formulation to rotate the circles into a position that maximizes interleaving. The rotation angle of each job corresponds to a time-shift value to delay the start of the next immediate training iteration to achieve compatibility.

Looking beyond a single link and extending to jobs running across a topology, we generalize the geometric abstraction to cluster-level by introducing a bipartite Affinity graph whose vertices are a subset of jobs and links. An edge in the Affinity graph indicates a job is traversing a link. We then use a new graph traversal algorithm to find unique time-shifts for all jobs while maintaining their compatibility on all links. Using our geometric abstraction and Affinity graph, we augment Themis [40] and Pollux [50] with ≈1000 lines of code.

To evaluate CASSINI, we build a testbed with 24 servers, each with one NVIDIA A100 GPU [8] and one 50 Gbps RDMA NIC. Our experiments with 13 representative DNN models (VGG11 [26], VGG16 [18], VGG19 [32], ResNet50 [27], WideResNet101 [72], BERT [20], RoBERTa [39], XLM [17], CamemBERT [43], GPT-1 [51], GPT-2 [52], GPT-3 [11], and DLRM [6]) show that CASSINI improves the tail completion time of jobs by up to 2.2× and 2.5×, compared to Themis [40] and Pollux [50], respectively. Moreover, we show that CASSINI reduces the number of ECN marked packets in the cluster by up to 33×.

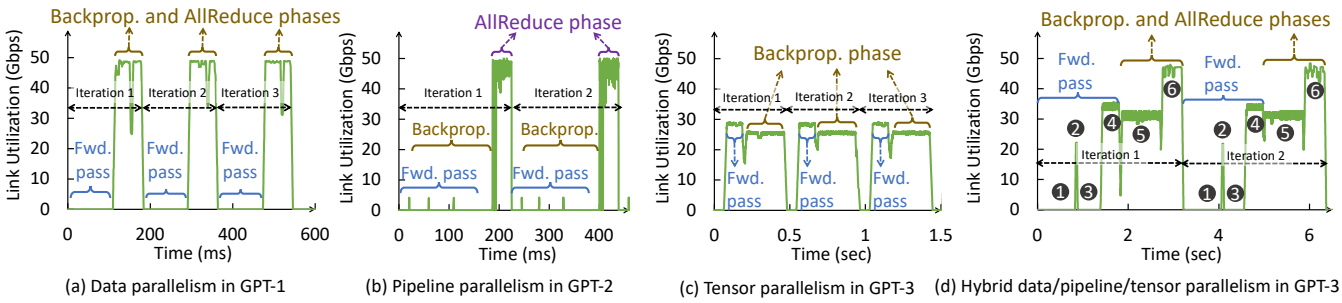


Figure 1: The traffic pattern of different parallelization strategies when training GPT-1, GPT-2, and GPT-3 models.

2 Background and Motivation

2.1 Distributed DNN Training Traffic Pattern

CASSINI is designed for large GPU clusters with hundreds of training jobs distributed with data, pipeline, and/or model parallel training paradigms. To this end, we study the impact of different parallelization strategies on network demand using a series of measurements. Each server in our testbed has one A100 GPU and one ConnectX-5 Mellanox RDMA NIC with 50 Gbps capacity. In all our experiments, we choose batch sizes such that the GPU utilization is higher than 80%, and intra-job pipelining is enabled.

Data parallelism. In data parallel training, the DNN model is copied into the memory of each GPU while the dataset is distributed across them. Figure 1(a) shows the communication pattern of a GPT-1 [51] model (12 layers, 9 GB memory) distributed across four GPU servers using data parallelism. The figure shows the traffic pattern of three back-to-back training iterations. Each iteration contains a forward pass with near-zero network demand, followed by a period of high utilization corresponding to the backpropagation and AllReduce phases.

Model/Pipeline parallelism. In model parallel training, the DNN model is partitioned across workers [29, 35], and parts of the DNN model are computed on different workers. The two common techniques for model parallelism are tensor parallelism and pipeline (or layer) parallelism [10]. In pipeline parallelism, the model is partitioned vertically at the layer boundaries [28, 47]. Figure 1(b) shows the communication pattern of a GPT-2 [52] model (24 layers, 27 GB memory) distributed across two servers using pipeline parallelism. We partition the model vertically in half (i.e., server_1 contains layer_1 to layer_{12}, and server_2 contains layer_{13} to layer_{24}) and use PipeDream's approach [47] to divide the batch size into three minibatches. The three small communication peaks during the forward pass correspond to the activation parameters of these three minibatches. The heavy communication demand following the peaks corresponds to the AllReduce operation between the embedding layers in the model.

Model/Tensor parallelism. Another variant of model parallel training is tensor parallelism [58, 59]. Tensor parallelism techniques partition the model horizontally such that different tensors are distributed across workers [31, 64]. Figure 1(c) shows the communication pattern of a GPT-3 [11] model (96 layers, 35 GB memory) distributed across two servers using tensor parallelism. We partition the model horizontally in half, where each server contains half of all the layers. The figure shows that both forward and backpropagation phases introduce roughly 25 Gbps traffic followed by a short period of near-zero network demand during data loading.

Hybrid data/pipeline/tensor parallelism. Today's DNN training systems tend to use a hybrid of data/pipeline/tensor parallelism to train large DNN models [21, 33, 46, 66]. Figure 1(d) shows the communication pattern of a GPT-3 [11] model (96 layers, 155 GB memory) distributed across eight servers using hybrid data/pipeline/tensor parallelism. We use pipeline parallelism to partition the model's layers vertically into two parts. Then, we divide the layers in each partition horizontally to obtain a total of four partitions. Next, we assign each of these four partitions to a server. Finally, we replicate the same process across another group of four servers and use data parallelism to distribute the data between these two groups of four servers. The figure shows the communication demand of the forward, backpropagation, and AllReduce phases where each phase has a different network demand.

Key takeaways. We repeat the above experiments using common DNN models, such as BERT [20], DLRM [6], WideResNet101 [72], RoBERTa [39], and VGG [62] and observe similar traffic patterns. Our key takeaways are: (*i*) the network demand repeats itself across all iterations, as long as the training hyper-parameters remain the same; (*ii*) the network demand of an iteration may consist of multiple Up and Down phases. The exact magnitude of the network demand during these Up and Down phases depends on the parallelization strategy and hyper-parameters. For instance, Figure 1(d) shows each training iteration has six Up-Down phases, labeled as ❶ to ❻. Section 3 describes CASSINI's approach to capture the duration and bandwidth of Up-Down phases.

2.2 Interleaving the Up and Down Phases

CASSINI's goal is to augment ML schedulers to consider the traffic demand of training jobs when making scheduling decisions. In particular, given the key takeaways in the previous

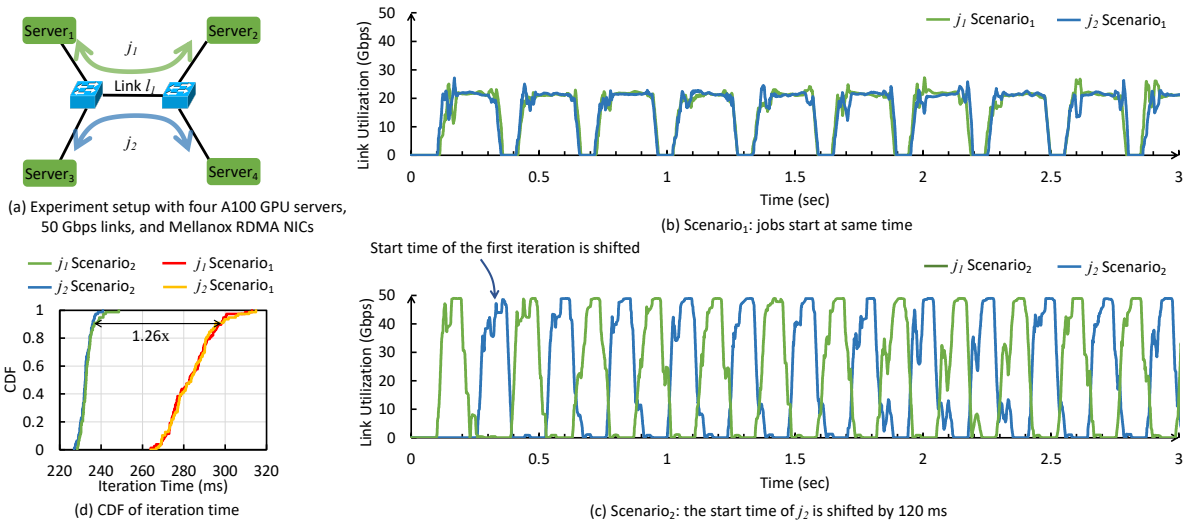


Figure 2: Impact of interleaving the Up-Down phases of two VGG19 jobs sharing link l_1.

section, we aim to interleave the bandwidth demand of Up and Down phases of different jobs to leverage the periodic network demand of distributed DNN training jobs.

To demonstrate the power of Up-Down network demand interleaving, we consider two data parallel training jobs, j_1 and j_2, as shown in Figure 2(a). Each job has one Up and one Down phase at every training iteration. We run each job for 1,000 iterations under two scenarios. In the first scenario, two VGG19 [62] jobs start simultaneously and share l_1 fairly. The communication uses the RDMA-based DCQCN congestion control algorithm [77]. Figure 2(b) shows that both jobs achieve roughly 22 Gbps bandwidth (i.e., half of l_1's capacity). In the second scenario, shown in Figure 2(c), we interleave the Down phase of j_1 with the Up phase of j_2 and vice versa, by shifting the start time of j_2 by 120 ms (Section 3 describes how we obtained this value). In this scenario, the jobs do not compete for bandwidth during their respective Up phases, giving both jobs the entire available bandwidth. Figure 2(d) plots the CDF of training iteration times for both scenarios demonstrating that scenario$_2$ accelerates the 90th percentile tail iteration time of both jobs by 1.26×.

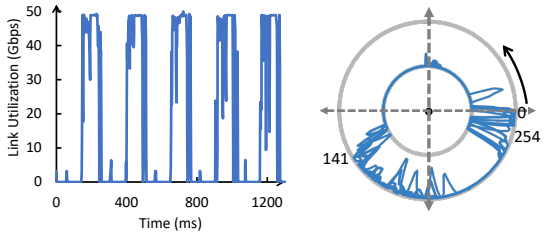
Perfectly interleaving the Up and Down phases of different jobs is not always possible. For instance, when BERT [20] and VGG19 [62] models share a link, no suitable time-shift can achieve perfect interleaving. But when WideResNet101 [72] and VGG16 [62] share a link, shifting VGG16 by 150 ms enables perfect interleaving. Instead of relying on perfectly matching Up and Down phases of jobs, we define a metric called *compatibility score* that captures the potential degree of interleaving across jobs sharing the network. Section 3 describes a novel technique to determine the compatibility score and the amount of required time-shift to achieve it.

3 Geometric Abstraction

Consider a time-series representation of the network demand for a job running in a dedicated cluster with no congestion. As shown in Section 2, different training jobs have different Up and Down patterns but the duration and bandwidth demand of the same job remain more or less the same across training iterations. The key idea of our abstraction is to *roll* time around a circle whose *perimeter* is equal to the iteration time of a job. Consequently, the Up-Down phases of all iterations will appear on approximately the same angles of the circle.

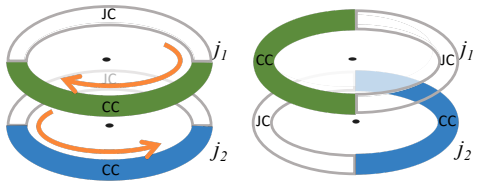
Figure 3(a) illustrates the time-series network demand of a data parallel VGG16 training job with a training iteration time of 255 ms. Figure 3(b) shows a circle with perimeter 255 units where the time-series data is plotted around it. The figure demonstrates that the Up and Down phases of different iterations cover the same angles of the circle. Our geometric abstraction captures this property, as shown in Figure 3(c). The perimeter of the circle is the iteration time, set to 255 units. The Down phase spans 141 units, represented by the uncolored arc with 200° angle, starting at 0°, on the x-axis. The Up phase represented by the colored arc occupies the remainder of the circle.

Rotate the circle to interleave Down and Up phases of different jobs. To determine the compatibility score of two (or more) jobs on a link, we place each job on its corresponding circle and overlay the circles on top of each other. Congestion occurs when the total bandwidth demand of a particular angle is higher than the capacity of the link, as shown in Figure 4(a). To find the best interleaving, we rotate the circles to a position where the summation of the bandwidth demands is less than the capacity of the link for all angles in the circle, as shown in Figure 4(b). If such a rotation is found, the jobs are fully compatible.



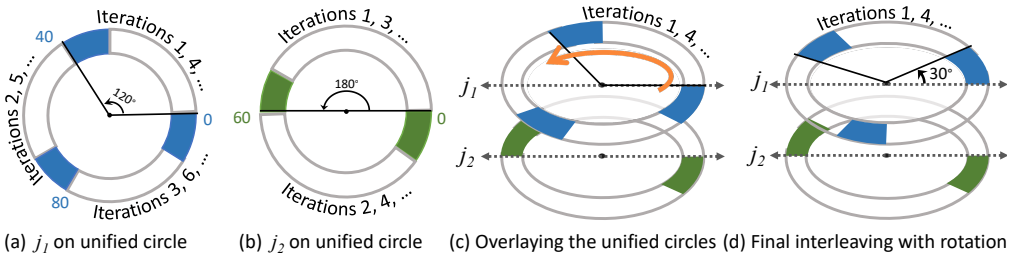
(a) Network demand (b) Network demand rolled around a circle (c) Geometric abstraction

Figure 3: CASSINI's geometric abstraction.



(a) CC phases are colliding (b) CC and JC phases are interleaved

Figure 4: Rotating the circles enables interleaving the network demand of j_1 and j_2.



(a) j_1 on unified circle (b) j_2 on unified circle (c) Overlaying the unified circles (d) Final interleaving with rotation

Figure 5: CASSINI's unified circles for jobs with different iteration times.

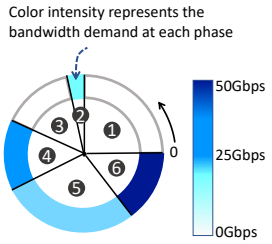


Figure 6: Geometric circle for the job in Fig. 1(d).

Capturing jobs with different iteration times. The above technique only works when the perimeters of the circles are the same. To generalize to the case where jobs have different iteration times, we place each job on a *unified circle* whose perimeter is equal to the Least Common Multiple (LCM) of the iteration time of all jobs competing on the link. For instance, consider two jobs j_1 and j_2 competing on a bottleneck link with iteration times 40 ms and 60 ms, respectively. To determine the compatibility score of the two jobs, we place them on a circle with a perimeter equal to $LCM(40,60) = 120$ units. Figure 5(a) shows j_1 on this unified circle. As the perimeter of the circle is $3\times$ j_1's iteration time, there are three periods of Up and Down phases in the figure. Similarly, Figure 5(b) shows j_2 on the unified circle. We then overlay the unified circles on top of each other (shown in Figure 5(c)) and rotate the circles to determine the compatibility score. Figure 5(d) shows that by rotating j_1 by $\Delta = 30°$ counter-clockwise, the sum of bandwidth demands on all angles of the unified circles is lower than the link capacity, giving these two jobs a compatibility score of 1 (i.e., fully compatible).

Capturing the bandwidth demand of model parallel training jobs. For clarity of presentation, the examples in this section contain data parallel training jobs with one Up and one Down phase during each iteration. However, CASSINI's geometric abstraction is generic and can capture more complex traffic patterns induced by various parallelization paradigms. Consider the communication pattern of the GPT-3 model with hybrid data/pipeline/tensor parallelism shown in Figure 1(d). Here, GPT-3's communication pattern consists of six Up-Down phases with different durations and bandwidth demands. The geometric circle of this job contains six colored arcs where the length and color intensity of each arc corresponds to the duration and bandwidth demand of each Up-Down phase of the model, as shown in Figure 6. Next, we formalize our geometric representation and show an optimization formulation that uses the geometric abstraction to find rotation angles to interleave the Up-Down phases of multiple jobs sharing a link, irrespective of the parallelization strategy.

Finding rotation angles. Once jobs are placed on their unified circles, CASSINI uses an optimization formulation, shown in Table 1, to find the best angle of rotation for jobs to maximize their compatibility.

Optimization input. The input is a set of ML jobs $J^l = \{j\}$ competing on a link l. We profile each job j to construct its unified circle, denoted by unified_circle$_j$. The perimeter of the unified circle is the LCM of the iteration times of all jobs $j \in J^l$. The data structure of unified_circle$_j$ contains a series of bandwidth demands $bw_circle_j(\alpha)$, where $\alpha \in [0, 2\pi]$ identifies an arc of the circle that corresponds to an Up or Down phase in the communication pattern. The total capacity of link l is denoted by C^l.

Optimization objective and output. The optimization goal is to overlay the unified circles of each job and rotate them such that the excess bandwidth demand across all angles is minimized. We define the compatibility score as $score = 1 - \text{average}(Excess(demand_\alpha))$, where $Excess$ is the excess bandwidth demand of all jobs at a particular angle α (Equation 1). To make the score a unitless metric, we divide the average excess bandwidth by the link capacity C^l. Note

Input	$J^l = \{j\}$	Set of ML jobs $j \in J^l$ competing on link l.
	$\{\text{unified_circle}_j\}$	Set of unified circles for $\forall j \in J$. Each circle is a data structure that contains the angles and bandwidth demand of Up or Down phases.
	$bw_circle_j(\alpha)$	Bandwidth demand at angle α on unified_circle$_j$
	r_j	Number of iterations of j in its unified_circle$_j$.
	$A = \{\alpha\}$	Set of discrete angles $\alpha \in [0,2\pi]$. $\lvert A\rvert$ denotes the number of discrete angles.
	C^l	Total link capacity of link l.
Output	$demand_\alpha$	Total bandwidth demand at angle α when considering the demand of all jobs $j \in J$.
	Δ_j^l	Rotation angle of $j \in J$ on link l, in radians.
	$score$	Compatibility score of jobs sharing link l.

Auxiliary definitions:

$$Excess(demand_\alpha) = \begin{cases} demand_\alpha - C^l & if\, demand_\alpha > C^l \\ 0 & otherwise \end{cases} \quad (1)$$

Maximize: $$score = 1 - \frac{\sum_\alpha Excess(demand_\alpha)}{|A|C} \quad (2)$$

Subject to:

$$\forall \alpha : \sum_j bw_circle_j(\alpha - \Delta_j^l) \leq demand_\alpha \quad (3)$$

$$\forall \Delta_j^l : 0 \leq \Delta_j^l \leq \frac{2\pi}{r_j} \quad (4)$$

Table 1: CASSINI's optimization formulation.

that when the excess bandwidth demand is zero, the compatibility score is 1 (i.e., 100% compatible). However, when there are many jobs with large excess bandwidth demands, it is possible for the score to become negative, indicating a highly incompatible combination. The optimization objective is to maximize this compatibility score, and the output of the optimization is a rotation angle Δ_j^l for each job.

Optimization constraints. Equation 3 computes the sum of the bandwidth demands across all the jobs sharing link l at a particular angle α on their unified circles, rotated by angle Δ_j^l. We bound this value by the output parameter $demand_\alpha$. Equation 4 bounds the rotation angle Δ_j^l between 0 and $\frac{2\pi}{r_j}$ because the unified_circle$_j$ contains r_j iterations of job j. Hence, setting an upper limit of $\frac{2\pi}{r_j}$ ensures that the rotation angle is in the first iteration and eliminates duplicate solutions.

4 Augmenting ML Schedulers with CASSINI

This section describes how CASSINI extends its link-level geometric abstraction to the entire cluster.

4.1 CASSINI Affinity Graph

Translating angular rotations to time-shifts. Consider a set of jobs $j \in J^l$ sharing link l. Using the formulation in Table 1, CASSINI computes a rotation angle Δ_j^l for $\forall j \in J^l$ such that the compatibility score is maximized. Each Δ_j^l corresponds to a time-shift t_j^l to delay the start time of j to maximize its compatibility with all other jobs in J^l. Given that the perimeter of the unified circle p^l, is the LCM of the iteration times of all jobs $j \in J^l$, CASSINI computes these time-shifts by multiplying the normalized rotation angle with p^l. Formally:

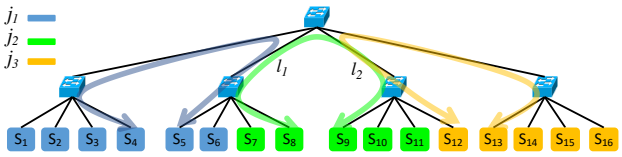


Figure 7: Example illustrating a cluster-scale compatibility challenge: CASSINI must ensure a unique time-shift for j_2.

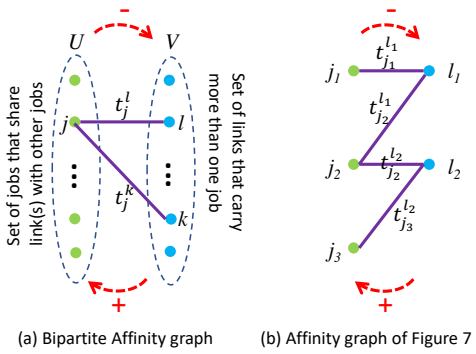


Figure 8: CASSINI's Affinity graph. Traversing left to right incurs a negative sign on the weight of edges and vice versa.

$$\forall j \in J^l, t_j^l = (\frac{\Delta_j^l}{2\pi} \times p^l) \mod iter_time_j \quad (5)$$

Challenge: ensuring a unique time-shift for each job. In a large-scale cluster, jobs are likely to traverse multiple links, and they may compete with different jobs on different links. Consider the case depicted in Figure 7 where job j_1 competes with job j_2 on link l_1, and j_2 competes with job j_3 on link l_2. Theoretically, it is possible to migrate the jobs to pack workers of the same job under the same rack to avoid sharing the links altogether, but our experiments show that today's ML scheduling systems frequently end up with fragmented placements because of the dynamic nature of their scheduling decisions and job arrival patterns. In fact, no scheduler guarantees it can maintain perfect placement throughout time without continuously migrating jobs to defragment the cluster. For the case depicted in Figure 7, computing the time-shifts of j_2 using Equation 5 would result in two time-shift values $t_{j_2}^{l_1}$ and $t_{j_2}^{l_2}$. Given the interdependence between all servers participating in a training job, CASSINI must find a *unique* time-shift value for each job across links without compromising the compatibility on any link.

Simple approach. A potential approach to address the above challenge is to simply break the tie by choosing one of the t_j^l values at random. But this approach cancels out the benefits of compatibility because it does not respect the carefully computed time-shifts for different links.

Algorithm 1 Traversing the Affinity graph

```
1:  procedure BFSAffinityGraph
    ▷ Input Graph G = (U,V,E): Cassini's Affinity graph
    ▷ Output time_shifts_G: Time-shifts of jobs in G
2:      time_shifts_G = { }
3:      for all connected subgraphs H ∈ G, H = (U_H,V_H,E_H) do
4:          time_shifts_H = { }
            ▷ BFS traversal
5:          Mark all vertices u ∈ U_H as not-visited
6:          u = randomly_select_vertex(U_H)
7:          t_u = 0 and mark u as visited
            ▷ Only enqueue vertices from U (jobs)
8:          Q.enqueue(u)
9:          while Q is not empty do
10:             j = Q.dequeue()
                ▷ Find the corresponding links and jobs
11:             for all neighbors l of j do
12:                 for all neighbors k of l do
13:                     if k is not visited then
14:                         Q.enqueue(k) and mark k as visited
                            ▷ Find the edge from U to V
15:                         e_1 = E_H(j,l)
                            ▷ Find the edges from V to U
16:                         e_2 = E_H(l,k)
                            ▷ Compute the final time-shift
17:                         t_k = (t_j − w_{e_1} + w_{e_2}) % iter_time_k
18:                         time_shifts_H[k] = t_k
19:         time_shifts_G = time_shifts_G ∪ time_shifts_H
20:     return time_shifts_G
```

Complex approach. Another potential approach is to expand the footprint of our geometric abstraction from link-level to cluster-level. This approach requires expanding the optimization formulation in Table 1 to include all jobs that share their paths with any other jobs in the cluster and to encode a unique Δ_j in the constraints. This approach is not scalable because it requires expanding the perimeter of the unified circle to become the LCM of the iteration times of a large number of jobs in the cluster. Thus, finding a unique rotation angle for each job requires adding an exponential number of constraints to the optimization formulation which increases the complexity and overhead of the formulation dramatically.

Cassini's approach. Cassini introduces a bipartite Affinity graph $G = (U,V,E)$, where U and V are two sets of vertices, and E denotes the edge set between U and V, shown in Figure 8(a). Each vertex $u \in U$ represents a job that is sharing its path with other jobs somewhere in the network. Each vertex $v \in V$ represents a link that carries more than one job. An undirected edge $e = (j,l) \in E$ exists between a job $j \in U$ and a link $l \in V$ if j traverses l. The weight of edge $e = (j,l) \in E$ is the time-shift of job j on link l; i.e., $w_e = t_j^l$.

Traversing the Affinity graph. Cassini uses a graph traversal algorithm to find unique time-shifts t_j for all jobs $j \in J$ while maintaining compatibility on all links. To consolidate t_j^l values for each job j and link l into a unique t_j value,

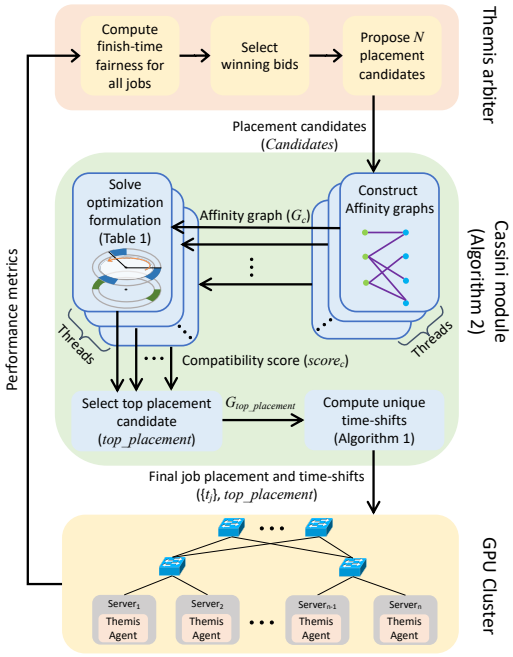


Figure 9: Using Cassini to augment Themis [40].

Cassini first randomly selects one of the jobs in the Affinity graph as the reference point with $t_j = 0$ and then traverses the graph to compute unique time-shifts for all others. Algorithm 1 describes the pseudocode of our graph traversal. In the general case, the Affinity graph is not necessarily a connected graph, hence, the algorithm traverses each connected subgraph separately (line 3). The traversal algorithm extends the Breadth First Search (BFS) algorithm in two ways. First, only vertices in U are added to the BFS queue (Q) because the time-shifts correspond to jobs, not links (lines 6- 14). Second, traversing from jobs ($j \in U$) to links ($l \in V$) incurs a negative sign on the t_j^l weight on edge $e = (j,l)$, whereas traversing the reverse direction incurs a positive sign (lines 15-18). As soon as the vertex corresponding to job j is visited, its unique time-shift is determined by the algorithm (line 18).

Theorem 1 (Correctness and Uniqueness Guarantee)**.** *Given a cluster with J jobs and a loop-free Affinity graph, $G = (U,V,E)$, Algorithm 1 guarantees both correct and unique time-shifts t_j for all jobs $j \in J$.*

Proof. The key insight behind this theorem is that our graph traversal maintains the same *relative* time-shift for all job pairs in the Affinity graph. The full proof uses induction and is provided in Appendix A, along with an example corresponding to the Affinity graph in Figure 8(b). □

4.2 Putting It All Together

This section uses Themis [40] as a running example of a scheduler augmented by Cassini.

Overview of Themis. Themis uses a fairness metric, called finish-time fairness, to achieve long-term fairness across the entire cluster by periodically updating the placement of jobs. To achieve fairness, workers in Themis lease resources and go through periodic auction epochs to help jobs that are farthest in terms of their fairness metric bid for more resources. Themis's central arbiter determines the global winning bids to maximize the aggregate improvement in the finish-time fair metrics across all bidding jobs. To capture network overheads, Themis uses a slowdown penalty based on whether the workers are under the same rack or across racks.

Augmenting Themis with CASSINI. Figure 9 shows how CASSINI augments Themis. First, CASSINI modifies Themis's arbiter to return a set of potential placement candidates instead of a single placement. Then, CASSINI selects the top placement candidate based on its compatibility metric and computes unique time-shifts for jobs that share the network. CASSINI transfers the time-shifts to Themis's agent running on servers. Finally, Themis's agent applies the time-shifts at the start of the epoch. Note that CASSINI respects the hyper-parameters, such as batch size or the number of workers, decided by Themis (or other schedulers that CASSINI is augmenting). Next, we describe each step in detail.

Step 1. Discover placement candidates. In this step, CASSINI decouples the process of finding the number of workers for each job to improve finish-time fairness from the exact worker placement in the cluster. To do so, instead of returning the precise job placements at the end of the auction phase, we configure Themis to return up to *N candidate placements*. These candidate placements all achieve the same finish-time fairness, but their worker placements are different. For instance, consider a case where jobs j_1 and j_2 each place a bid on two additional workers, and they both win, while job k_1 is losing one worker, and job k_2 is losing three. In this case, there are two ways to distribute workers: (i) k_1 and k_2 each give up one worker to j_1, and k_2 gives two workers to j_2; or (ii) k_1 and k_2 each give up one worker to j_2, and k_2 gives two workers to j_1. Both options are candidate placements. Moreover, selecting which workers in k_1 and k_2 should be reassigned creates another set of candidate placements. CASSINI collects these candidate placements and feeds them as input to the next step. This process requires changing only ≈300 lines of code in Themis.

Step 2. Find unique time-shifts. This step is listed in Algorithm 2 and includes CASSINI's key contributions. CASSINI first constructs an Affinity graph G_c for each placement candidate $c \in Candidates$ (lines 3-12). Following Theorem 1, to ensure correctness, we discard placement candidates with loop(s) in any of their Affinity subgraphs (line 15). Then, CASSINI constructs the unified circles for each job and solves the optimization formulation in Table 1 for all links in G_c to obtain the compatibility metric for each link in V_c (lines 17-22). Given that the placement candidates are independent of each other, our implementation uses multiple threads to parallelize this computation. Once the compatibility score of all candidate placements is determined, CASSINI sorts each placement candidate based on the average compatibility score of its member links to find the top placement candidate $top_placement \in Candidates$ (lines 24-25).[1] Then, it executes Algorithm 1 on $top_placement$'s Affinity graph $G_{top_placement}$ to obtain unique time-shifts $\{t_j\}, \forall j \in V_{top_placement}$ for jobs that share links with other jobs in this placement (line 26). Finally, $top_placement$ and its corresponding time-shifts are transferred to Themis's agent running on the servers (line 27).

Step 3. Apply time-shifts. When a time-shift t_j is received by the Themis agent running job j, it delays the start of the

Algorithm 2 CASSINI Module's Pluggable Algorithm

```
1:  procedure CassiniModule
      ▷ Input Jobs: Array of active training jobs in the cluster
      ▷ Input Links: Array of all links in the cluster
      ▷ Input Candidates: Array of candidate placements for jobs
      ▷ Output top_placement, {t_j}: Top placement and time-shifts
2:    for c ∈ Candidates do  ▷ (Loop is executed with threads)
        ▷ Construct CASSINI's Affinity graph corresponding
        to this placement (§4.1)
3:      G_c = (U_c, V_c, E_c)
4:      for all j ∈ Jobs, l ∈ Links do
5:        if j shares links with other jobs then
6:          U_c = U_c ∪ j
7:          if l carries more than one job then
8:            V_c = V_c ∪ l
9:            if j is traversing l then
10:             e = new Edge between {(j, l)}
11:             E = E ∪ e
12:             w_e = 0
        ▷ Discard this candidate if Affinity graph has a loop
13:     if there is a loop in G_c then
14:       Candidates.remove(c)
15:       continue
16:     score_c = { }
17:     for l ∈ V_c do  ▷ (Executed with threads)
          ▷ List of jobs traversing link l
18:       J^l = {}
19:       for all neighbors j of l do
20:         J^l = J^l ∪ j
          ▷ Solve CASSINI optimization (Table 1)
21:       score_l = CassiniOptimization(J^l)
22:       score_c = score_c ∪ score_l
        ▷ Set the compatibility score of candidate c
23:     c.score = score_c
      ▷ Sort placements based on compatibility metric
24:   SortCandidates(Candidates, "Decreasing")
25:   top_placement = Candidate[0]
      ▷ Find unique time-shifts (Algorithm 1)
26:   {t_j} = BFSAffinityGraph(G_top_placement)
27:   return {t_j}, top_placement
```

[1] Instead of averaging, tail or other metrics may also be used.

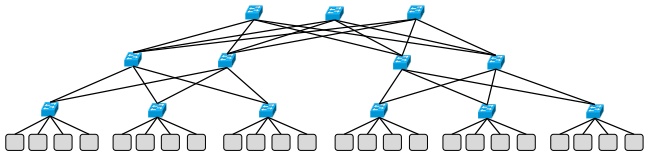


Figure 10: Logical topology of our testbed.

next immediate training iteration by t_j. However, even though the workers of the same job apply a unique time-shift, the time-shift values can drift due to noise, stragglers, and other unpredictable events. CASSINI updates the agent on each server to measure the drift and adjust the time-shifts. Our evaluations show that time-shift adjustments are rare (§5.7).

5 Evaluations

We evaluate CASSINI on a 24-server cluster and compare its performance to that of other state-of-the-art ML schedulers. First, we describe our evaluation methodology and setup (§5.1). Then, we compare CASSINI's performance gains with respect to the state-of-art ML schedulers for a mix of data and model parallel DNN training jobs (§5.2). Next, we evaluate the impact of data parallelism (§5.3), model parallelism (§5.4), partial compatibility (§5.5), and having multiple GPUs per server on CASSINI's performance (§5.6). Finally, we evaluate the frequency of time-shift adjustments and CASSINI's overhead (§5.7). CASSINI's source code is available at http://cassini.csail.mit.edu.

5.1 Methodology and Setup

Setup. We build a prototype to demonstrate the gains of CASSINI in real-world settings. Our prototype includes 24 ASUS ESC4000A-E10 servers, each with one A100 Nvidia GPU [8] (40 GB of HBM2 memory) and one 50 Gbps Mellanox ConnectX5 NIC. We use RoCEv2 for communication and enable DCB [5] and PFC on these interfaces to support a lossless fabric for RDMA. The servers run Ubuntu 18.04 LTS. We use PyTorch [36] version 1.8.0, CUDA version 11.1, and NCCL version 2.11.4 in our training framework.

Topology. We use a Tofino switch to construct the logical topology illustrated in Figure 10 with 13 logical switches. The Mellanox ConnectX5 NICs on each of the servers are connected to the Tofino switch. The Tofino switch emulates 13 logical switches and 48 bi-directional links for a 2:1 oversubscribed topology. We use flow table rules that match on <input port, destination MAC> to forward packets to the correct output port and physical loopback cables for switch-to-switch links. We use the default RDMA-based DCQCN congestion control algorithm [77]. ECN is enabled through WRED with min and max thresholds set to 1000 and 2000 cells. The PFC skid buffer threshold of each virtual switch is 4000 cells.

DNN workloads. We experiment with 13 popular DNN models: VGG11 [26], VGG16 [18], VGG19 [32], ResNet50 [27], WideResNet101 [72], BERT [20], RoBERTa [39], XLM [17], CamemBERT [43], GPT-1 [51], GPT-2 [52], GPT-3 [11], and DLRM [6]. All models have an equal probability of occurrence and the training duration time is randomly selected between 200 - 1,000 iterations. Table 3 (Appendix B) provides details about model configurations and batch sizes used in this paper.

Parallelization strategy. We use data parallelism to train the VGG, ResNet, and BERT family of models using Pytorch's DistributedDataParallel framework [38]. This framework distributes the dataset across GPUs and uses RingAllreduce to update the gradients during each training iteration. We train the DLRM and GPT family of models using a hybrid of data/model parallelism. Following prior work [66], we use Meta's opensource codebase for training DLRM [6] where the embedding tables are partitioned across GPUs, while the rest of the model is replicated on all GPUs. Finally, we use Microsoft's DeepSpeed tool [7] to partition the GPT models across GPUs using hybrid data/model parallelism.

Traces. Following prior work [40, 44, 50, 75], we use three sets of traces in our evaluations: (*i*) *Poisson trace*: we use a Poisson distribution for job arrivals where the job arrival time is determined by the load parameter defined as the average fraction of GPUs that are serving active jobs in the cluster. We vary the load between 80% and 100%; (*ii*) *dynamic trace*: where a set of DNN training jobs are present in the cluster, and a new set of jobs arrive; (*iii*) *snapshot trace*: we take several snapshots of the cluster where all jobs are present at the start of the experiment.

We implement the following schemes in our testbed.

• **Themis.** We use the default Themis [40] scheduler as one of our baselines. The bidding period (epoch) is set to 10 mins. Jobs participate in an auction where they send bid values for different GPU allocations. An assignment of GPU servers is valid until the period of lease time. When the lease time expires, the job gives up the server, and a new auction is conducted for all the released servers. When a job arrives, its initial number of requested workers is randomly selected between 1 to 12 GPUs. As the experiment progresses, the number of workers is automatically tuned based on Themis's finish-time-fairness metric.

• **Th+CASSINI.** Themis augmented with CASSINI as described in Section 4.2. In particular, this scheduler takes up to 10 placement candidates from Themis, constructs geometric circles and Affinity graphs for each placement to capture the cluster-level compatibility, solves our optimization formulation to find time-shifts for jobs that are competing on bandwidth, selects the top placement candidate based on compatibility ranks, and finally computes a unique time-shift for jobs. The unique time-shifts and final placement are given to the Themis agent running on GPUs. Unless otherwise stated, we use 5° as the angle discretization precision (Table 1) to compute the time-shifts.

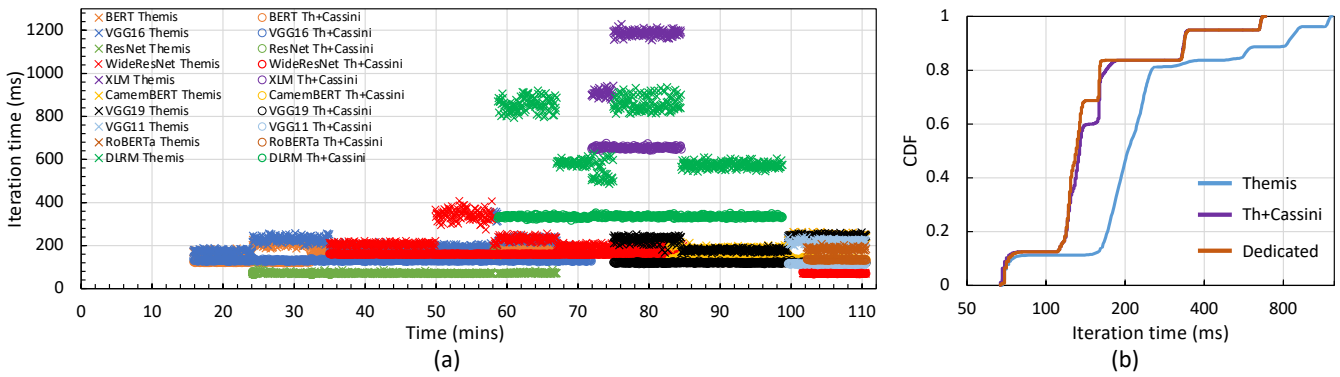


Figure 11: [Poisson trace] (a) Time series of DNN training jobs and their iteration times. (b) CDF of the iteration times.

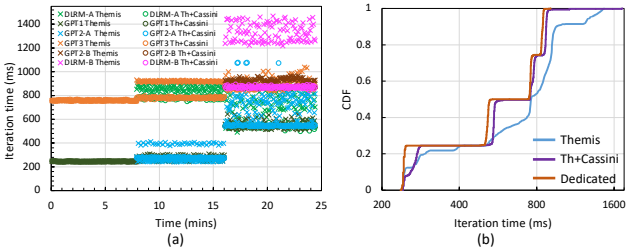


Figure 12: [Poisson trace] (a) Time series of model parallel jobs and their iteration times. (b) CDF of the iteration times.

• **Pollux.** We use Pollux as a second baseline [50]. Pollux considers the system throughput and statistical efficiency to maximize cluster-wide training performance. It periodically queries jobs and reassigns GPUs to maximize the overall goodput of the cluster. Pollux also models migration costs and avoids frequent job migrations.

• **Po+CASSINI.** We augment Pollux with CASSINI using an approach similar to that described in Section 4.2 except that Pollux uses overall goodput instead of finish-time-fairness to adjust hyper-parameters during scheduling epochs. Hence, the number of workers assigned to each job does not always agree with Themis. To make an apples-to-apples comparison, all CASSINI-related parameters in Po+CASSINI and Th+CASSINI are the same.

• **Ideal.** An ideal scheduler that runs each training job on a dedicated cluster. This scheduler incurs no congestion, as the entire cluster is dedicated to one job, and there is no need to take job compatibility into account.

• **Random.** A random placement scheduler that places workers for each job randomly. This scheduler has the highest network overhead, because it does not take locality or compatibility into account.

Profiling DNN models. Similar to Themis and Pollux, we profile each DNN using Pytorch and Infiniband port counters. Our profiling script executes a few iterations of each job to measure iteration times and collect link utilization patterns for various batch sizes and numbers of workers. Fine-grained link utilization data from the port counters enables CASSINI to build the geometric circles and the corresponding bandwidth demands for our optimization ($bw_circle_j(\alpha)$ in Table 1).

5.2 Performance Gains

We evaluate CASSINI's performance gains using job arrivals and departures from our Poisson trace. Figure 11(a) plots the time series of events in the cluster for Themis and Th+CASSINI. In this experiment, we train a combination of DNN models. We use model parallelism for the DLRM [6] model because of its large model size, and we use data parallelism for all the other DNN models. Placement changes are triggered by job arrivals, job departures, and when the lease time of any of the servers expires. Given the dynamic nature of the trace, the servers are occupied gradually, and their lease times are not synchronized. For instance, at time $t = 72$ mins, a data parallel training job for the XLM [17] model arrives at the cluster, and Themis places it such that one of the links is shared with WideResNet101 [72] without the knowledge that XLM and WideResNet101 are not compatible jobs. In contrast, Th+CASSINI improves the iteration time of XLM by placing it with compatible jobs. Figure 11(b) plots the CDF of iteration times of all the data points in Figure 11(a) and shows that compared to Themis, Th+CASSINI improves the average and 99th percentile tail iteration times by 1.6× and 1.8× respectively. We observe similar gains between Po+CASSINI and Pollux. The figure also shows that Th+CASSINI achieves similar performance as our Ideal benchmark.

To evaluate CASSINI's performance with model parallelism, we measure iteration times of various jobs trained using model parallelism, as shown in Figure 12(a). We use our Poisson trace for the job arrivals and departures. Note that this trace contains different training instances of the same DNN models where they differ in their hyper-parameters and number of workers (details in Appendix B). We use suffixes on their names to distinguish between the instances, for example, GPT2-A and GPT2-B are two different training jobs, as shown in the legend of Figure 12(a). GPT2-A has a batch

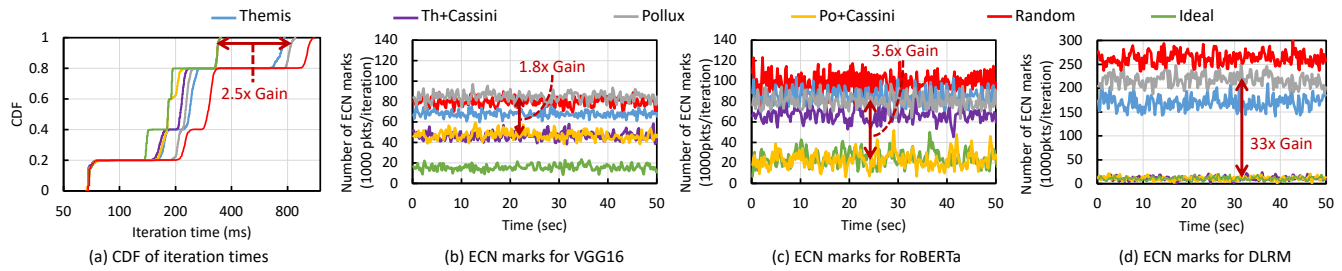


Figure 13: [Dynamic trace] CDF of training iteration times and the number of ECN marked packets per iteration.

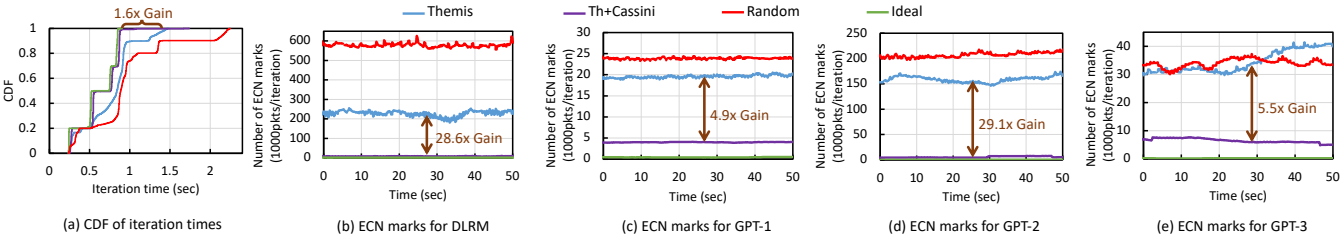


Figure 14: [Dynamic trace, model parallelism] CDF of training iteration times and the number of ECN marked packets.

size of 24 with a model hidden size of 1536 (as defined by Deepspeed's codebase [7]), while GPT2-B has a batch size of 70 with a model hidden size of 1184. For instance, at time $t = 8$ min, a model parallel GPT-2 [52] training job (labeled as GPT-2-A) arrives at the cluster and without considering the communication demands, Themis places this job such that it shares a link with another large GPT-3 [11] model in the cluster. GPT-2-A and GPT-3 models are not compatible, causing both training jobs to slow down. In contrast, Th+CASSINI improves GPT-2-A's iteration time by placing it with a compatible GPT-1 model. Figure 12(b) plots the CDF of iteration times of all the data points in Figure 12(a) and shows that compared to Themis, Th+CASSINI improves the average and 99th percentile tail iteration times by 1.2× and 1.6× respectively.

5.3 CASSINI Reduces Congestion

We next demonstrate the effectiveness of CASSINI in reducing the congestion in the network. We use our dynamic trace to trigger the arrival of DLRM and ResNet50 to the cluster while the cluster is busy running other jobs. Given the contrast between the network demand between these two models, this experiment serves as a stress test to highlight the importance of compatible job placement on network congestion. In this case, both Pollux and Themis end up placing DLRM on servers that share network links with other non-compatible jobs which hurts the iteration times. In contrast, Th+CASSINI and Po+CASSINI flip the placements of DLM and ResNet50 to achieve compatibility, thereby improving the training iteration times, as depicted in Figure 13(a). Compared to Themis, Th+CASSINI improves the average and 99th percentile tail iteration times by 1.5× and 2.2×, respectively. Similarly, compared to Pollux, Po+CASSINI improves the average and 99th percentile tail iteration times by 1.6× and 2.5×, respectively.

The gains in iteration times are a direct consequence of CASSINI's ability to reduce network congestion. Figures 13(b) to (d) show the number of ECN marked packets per iteration for different models. The figure shows that Th+CASSINI and Po+CASSINI consistently maintain a lower number of ECN marks per iteration across the models. In particular, Figure 13(d) shows that, on average, DLRM is experiencing 27× and 33× more ECN marks in Themis and Pollux, compared to their CASSINI-augmented counterparts.

5.4 Impact of Model Parallelism

To ensure CASSINI's gains are not limited to data parallel jobs, we run a series of experiments in which all jobs in the trace use model parallelism. As shown in Section 2.1, model parallel jobs have several Up and Down phases in each iteration where the duration and bandwidth demand of each phase depends on the parallelization strategy and hyper-parameters. Similar to the data parallel case, we use CASSINI's geometric abstraction to capture the duration and bandwidth demand of Up and Down phases of a series of model parallel jobs. We then use CASSINI's optimization formulation and Affinity graph to compute the time-shifts for the jobs sharing the same network links. We use our dynamic trace to trigger the arrival of multiple GPT and DLRM models while the cluster is training other model parallel jobs.

Figure 14(a) shows the CDF of training iteration times. We find that similar to the data parallel case, Themis ends up placing non-compatible jobs, such as <GPT-3 and GPT-2> or <GPT-1 and DLRM>, on the same network link, whereas Th+CASSINI places compatible jobs, such as <GPT-1 and

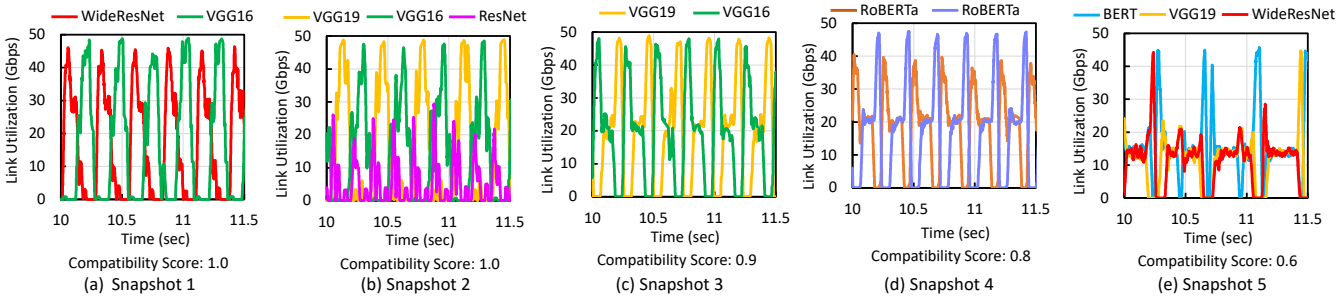


Figure 15: [Snapshot trace] Link utilization of compatible and partially compatible snapshots.

Snapshot ID	Competing jobs (batch size)	Th+CASSINI	Themis	Compatibility score	time-shift (ms)
1	WideResNet101 (800)	138 ms	205 ms	1.0	0 ms
	VGG16 (1400)	148 ms	199 ms		150 ms
2	VGG19 (1400)	168 ms	220 ms	1.0	0 ms
	VGG16 (1700)	163 ms	220 ms		158 ms
	RESNET50 (1600)	59 ms	55 ms		46 ms
3	VGG19 (1024)	166 ms	176 ms	0.9	0 ms
	VGG16 (1200)	168 ms	177 ms		100 ms
4	RoBERTa (12)	164 ms	210 ms	0.8	0 ms
	RoBERTa (12)	180 ms	208 ms		60 ms
5	BERT (8)	209 ms	213 ms	0.6	0 ms
	VGG19 (1400)	294 ms	292 ms		42 ms
	WideResNet101 (800)	265 ms	266 ms		191 ms

Table 2: [Snapshot trace] Compatibility score of DNN jobs.

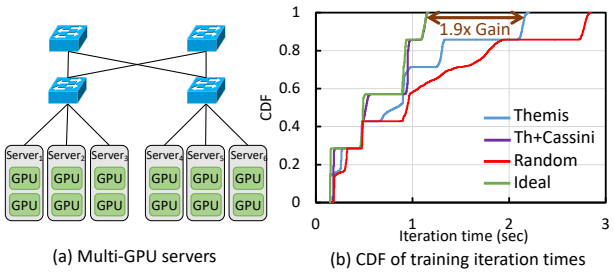


Figure 16: [Dynamic trace] Multi-GPU experiment.

GPT-2> or <GPT-3 and DLRM>, on the same network links. Consequently, Th+CASSINI improves the average and 99th percentile tail iteration times by 1.2× and 1.6×, respectively. We observe similar gains between Pollux and Po+CASSINI.

Figures 14(b) to (e) depict the number of ECN marked packets per iteration for the models in this experiment. Depending on the status of congestion, different models experience different numbers of ECN marked packets. For instance, Figure 14(d) shows that compared to Themis, Th+CASSINI reduces the average number of ECN marked packets by 29×.

5.5 Impact of Partial Compatibility

An important consideration for practical deployment of CASSINI is to evaluate the impact of placing *partially* compatible jobs on the same link(s). Intuitively, the higher the compatibility score, the better interleaving is achieved. As the compatibility score reduces, the gains also diminish. To evaluate the impact of partial compatibility, we take five snapshots of the cluster, as shown in Table 2, and compute the compatibility scores and time-shift values from our optimization formulation (§3) for each snapshot. We then measure the average communication time of each model under Themis and Th+CASSINI. The table shows that when the compatibility score is 0.6, CASSINI's gain compared to Themis starts to diminish. Note that CASSINI avoids placing jobs with low compatibility score (e.g., snapshot 5) on the same link.

We demonstrate the reason behind diminishing returns by plotting the link utilization of each snapshot in Figure 15. When the compatibility score is high, the opportunity for interleaving is large, and jobs end up interleaving their network usage most of the time, as shown in Figures 15(a)–(d). However, as the compatibility score is reduced, jobs are forced to share the link most of the time, as shown in Figure 15(e). Additionally, Figure 15(b) demonstrates a desirable feature of our optimization formulation where compatibility does not require *strict* interleaving. In this snapshot, only VGG19 and VGG16 are interleaved, and ResNet's communications overlap with both jobs because its network demand is not significant and can co-exist with the other jobs.

5.6 Impact of Multiple GPUs per Server

Although having multiple GPUs per server enables allocating more GPUs within the same server to a job, today's large-scale training jobs require hundreds of workers [45, 66], making it impossible to avoid network congestion entirely by relying on multi-GPU servers. In such cases, CASSINI's gains are more pronounced for models that are distributed outside the boundary of a server.

We evaluate CASSINI's gains with multi-GPU servers by removing GPUs from some of our single-GPU servers and adding them to other servers to compose servers with two GPUs. We create a topology with six servers, each with two GPUs, as shown in Figure 16(a). We then use a mix of data parallel and model parallel jobs and generate a series of job arrivals using our dynamic trace.

Figure 16(b) demonstrates that compared to Themis, Th+CASSINI improves the average and 99th percentile tail iteration times by 1.4× and 1.9×, respectively. These gains are

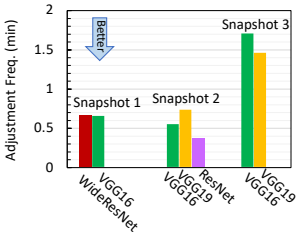


Figure 17: [Snapshot trace] The frequency of adjusting time-shifts for snapshots 1–3.

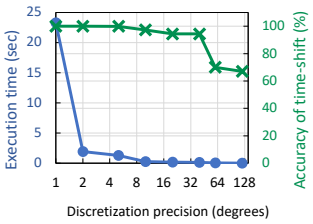


Figure 18: Impact of angle discretization on execution time and time-shift accuracy.

achieved because some jobs require more than two GPUs to train. For instance, at a particular instant in our dynamic trace, the XLM and ResNet50 models each require three GPUs to train. With the arrival of a network-intensive model DLRM requesting three more GPUs, Themis decides to place DLRM such that it shares a server with a non-compatible model (XLM), making both jobs experience congestion. In contrast, Th+CASSINI selects a placement where DLRM shares a link with a compatible model (ResNet50), thereby improving the training iteration times of both models.

5.7 Adjusting Time-Shifts and Overhead

To maintain CASSINI's interleaving, workers must respect the time-shift values given to them through the scheduler. Given that our servers are not running perfectly in sync, we evaluate the frequency of automatic time-shift adjustments by the Themis (or Pollux) agents running on the servers. Note that respecting the time-shift is only required for compatible jobs. All other jobs in the cluster can send packets at any time. A worker triggers an adjustment when the start of the communication phase deviates by more than five percent of the ideal iteration time. Figure 17 shows the average frequency of time-shift adjustments for snapshots 1,2, and 3. In all cases, the frequency is less than two adjustments per minute.

Finally, we evaluate the impact of angle discretization precision on CASSINI's optimization formulation (Table 1). Intuitively, the execution time of a coarse-grained discretization is fast but such formulation misses interleaving opportunities, thereby finding imprecise rotation angles. Given that CASSINI's time-shifts are driven from rotation angles, a coarse-grained formulation leads to inaccurate time-shifts. On the other hand, having fine-grained precision leads to more accurate time-shifts at the expense of a longer execution time. Figure 18 demonstrates this trend and shows that using a precision of 5° is the sweet spot for achieving 100% accuracy for time-shifts while maintaining a low execution overhead.

6 Discussion and Limitations

Sharing with legacy datacenter workloads. We assume the ML training traffic is not sharing the network with non-ML legacy datacenter workloads, such as websearch, indexing, cloud, and storage. We believe this is reasonable because modern training clusters consist of custom-designed servers, each with dedicated NICs for training traffic (GPU NICs) and additional NICs for storage and other traffic (CPU NICs) [4, 45]. The CPU NICs are often connected through a separate fabric to carry storage and other control plane traffic. Our abstraction and time-shift values only affect the GPU NICs.

GPU multi-tenancy. For simplicity, we assume GPUs are dedicated resources for each job, and different jobs are not sharing the same GPU – this is not far from how many production clusters run today to ensure predictable and high-throughput training performance. Thus, our geometric abstraction only considers the network links as shared resources and allows the Down (Just Compute) phases of different jobs to overlap. Recent proposals have demonstrated the feasibility of multi-tenancy on GPUs [9, 67, 69, 70]. We note that capturing GPU multi-tenancy is possible by adding more constraints in our optimization formulation, but we omit the details for brevity.

Scaling. Scaling the number of GPUs on each server enables service providers to pack jobs within fewer servers, thereby reducing the chances of network congestion. In recent years, the compute requirements of DNN models are growing exponentially [3]. Training models across multiple servers is inevitable with growing model and dataset sizes. For example, large models like AlphaGo [61] and AlphaZero [60] are trained using hundreds to thousands of GPUs and TPUs. We expect CASSINI's gains to remain consistent for clusters with multiple GPUs per server, but we leave further investigation to future work.

CASSINI advocates placing jobs such that jobs with higher compatibility scores share network links. However, as the number of jobs sharing a network link increases, it becomes harder to interleave the communication demands, and the compatibility score reduces. CASSINI tries to avoid scenarios where jobs with low compatibility scores share a network link. We leave the study of the effect of the number of jobs sharing a network link on the compatibility scores for future work.

7 Related Work

Our work builds on several lines of related research.

Compute scheduling approaches. A large number of systems and techniques have focused on improving the performance of large-scale distributed ML workloads [13, 16, 19, 23, 24, 36, 42, 54, 63, 68, 71, 73]. Relevant to this paper, several papers aim to reduce communication overhead using smart scheduling techniques; e.g., Gandiva [69], Themis [40], Pollux [50], Tiresias [25], Shockwave [76], and Optimus [48]. These schedulers try to minimize network sharing by placing workers of the same job as close as possible to each other. However, these approaches do not consider interleaving the communication patterns of different training jobs when placing them on servers. CASSINI's contribution is complementary to these approaches by considering both the compute resources and the communication demands of different jobs

during scheduling. Moreover, CASSINI is designed as a pluggable module to augment these schedulers.

Multi-resource sharing. Recently, Muri [75] proposed a scheduling technique to interleave critical resources (e.g., GPU, CPU, network, storage) of DNN training jobs. Muri packs jobs that are being executed *on the same set of resources* into a group and interleaves their resource requirements using a Blossom-based scheduler. However, Muri's approach to resource interleaving only applies to jobs that share the same set of GPUs, CPUs, memory, and network resources.[2] Hence, Muri can interleave compute and communication phases of a set of jobs only if the jobs *are sharing the same set of GPUs*. In contrast, CASSINI is able to interleave compute and communication phases of different jobs, irrespective of which GPUs they occupy. For instance, Muri's algorithm is not applicable to interleave the resources of j_1 and j_2 in Figure 2(a), because j_1 is distributed between server$_1$ and server$_2$ while j_2 is distributed between server$_3$ and server$_4$; i.e., these two jobs do not belong to the same resource group in Muri's algorithm. Muri would have been able to interleave the resources if both j_1 and j_2 were distributed between all four servers. However, for many of the large models we use in our experiments, GPU-sharing is not possible because of the memory requirements of the model. Moreover, even with GPU sharing, in a large-scale cluster, cross-group network congestion is common. CASSINI is able to interleave the Up and Down phases of different jobs, without requiring them to share the same set of resources. Similarly, Synergy [44] has proposed a multi-resource interleaving scheduling approach by inferring the sensitivity of DNN jobs to GPU, CPU, and memory resources using optimistic profiling. Synergy improves the overall cluster efficiency by performing resource-sensitive allocations instead of a GPU-proportional allocation. However, Synergy's approach does not consider the network bandwidth as a resource and is unable to interleave the communication phases with other resources. In contrast, CASSINI's focus is on interleaving the network demand with the GPU resources. CASSINI is designed to augment both Muri and Synergy schedulers. Some previous studies have concentrated on the theoretical analysis of periodic tasks [22, 34]. However, these approaches exploit characteristics distinct from those inherent to distributed DNN training jobs.

Communication-aware scheduling. A variety of approaches have been developed to accelerate communication among ML training workers of the same job to reduce network overhead [2, 14, 23, 33, 45, 55, 65, 74] and to enable more efficient pipelining strategies [28, 47]. ByteScheduler [49] and Syndicate [41] accelerate ML training by scheduling and optimizing the order of communication operations between different GPU servers used by a training job. ByteScheduler overlaps compute and communication operations *within a training job*, while Syndicate provides a solution for planning and scheduling communication operations for large DNN training. Similarly, TACCL [56], BytePS [30], and CLOPT [74] improve the communication collective algorithms of large DNN models. BytePS seeks to find a balance between the Parameter Server [37] and Ring-AllReduce algorithms for synchronizing the gradients. TACCL proposes a communication collective algorithm for training large models with data and model parallelism. CLOPT co-optimizes network topology and communication schedules for ML training. These approaches optimize communication *within a training job*, however, they do not consider congestion and network sharing *across training jobs*. In contrast, CASSINI's approach is orthogonal to these techniques because CASSINI focuses on sharing the network resources *across different training jobs*.

Difference with prior workshop paper. A prior workshop paper [53] introduced the idea of using a geometric abstraction to achieve job compatibility at a single-link level. We extend this workshop paper in a few important ways. First, the workshop paper considers compute/communication interleaving at a high level and does not provide a concrete scheduling technique to achieve it. Specifically, it relies on an unfair congestion control protocol to achieve interleaving, but CASSINI does not require any changes to or assumptions about the congestion control protocol. Second, the workshop paper ignores the impact of cluster-level interleaving. Third, the workshop paper only considers the data parallelism paradigm, and its geometric abstraction does not generalize to model parallelism techniques. Finally, our optimization formulation, the Affinity graph abstraction, the design and implementation of the CASSINI module, and our formal arguments around correctness (Theorem 1) are all new contributions.

8 Conclusion

CASSINI is a simple but effective approach that can integrate with existing cluster schedulers to allow them to accommodate multiple ML jobs' network needs. We introduce a novel metric, called compatibility score, to rank different GPU placements when jobs compete on network links. Our evaluations show that CASSINI improves the average and tail completion time of jobs by up to 1.6× and 2.5×, respectively. Moreover, we show that CASSINI reduces the number of ECN marked packets by up to 33×.

Acknowledgements. We thank NSDI's anonymous reviewers and our shepherd, Xin Jin, for their valuable feedback. Thanks to Gautam Kumar, Frank Wang, Benoit Pit–Claudel, Venkat Arun, and Kapil Vaidya for helpful suggestions and discussions. The MIT-affiliated authors were supported in part by ACE and CUbiC, two of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, as well as NSF SHF-2107244, NSF ASCENT-2023468, NSF CAREER-2144766, NSF PPoSS-2217099, NSF CNS-2211382, and Sloan fellowship FG-2022-18504. Akella was supported by NSF grants CNS-2214015 and CNS-2207317, and by a gift from Cisco Research.

[2] Muri [75] states this limitation: "The algorithm avoids cross-group packing to minimize the packing overhead."

References

[1] MLPerf: A broad ML benchmark suite. https://mlperf.org/.

[2] Baidu, 2017. https://github.com/baidu-research/baidu-allreduce.

[3] AI and Compute, 2020. https://openai.com/blog/ai-and-compute/.

[4] Nvidia DGX SuperPOD, 2020. https://www.nvidia.com/en-us/data-center/dgx-superpod/.

[5] Data Center Bridging eXchange (DCBX), 2021. https://man7.org/linux/man-pages/man8/dcb-dcbx.8.html.

[6] Deep Learning Recommendation Model for Personalization and Recommendation Systems, 2021. https://github.com/facebookresearch/dlrm.

[7] DeepSpeed version of NVIDIA's Megatron-LM, 2021. https://github.com/microsoft/Megatron-DeepSpeed.

[8] NVIDIA A100 Tensor Core GPU, 2021. https://www.nvidia.com/en-us/data-center/a100/.

[9] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu. Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency. *SIGPLAN Not.*, 53(2):503–518, mar 2018.

[10] T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Comput. Surv.*, 52(4), aug 2019.

[11] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[12] M. A. Chang, A. Panda, D. Bottini, L. Jian, P. Kumar, and S. Shenker. Network evolution for dnns. *SysML*, 2018.

[13] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI'14*, pages 571–582, 2014.

[14] M. Cho, U. Finkler, D. Kung, and H. Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. SysML Conference, 2019.

[15] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengil, M. Liu, D. Lo, S. Alkalay, and M. Haselman. Accelerating persistent neural networks at datacenter scale. In *Hot Chips*, volume 29, 2017.

[16] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with cots hpc systems. In *International conference on machine learning*, pages 1337–1345, 2013.

[17] A. CONNEAU and G. Lample. Cross-lingual language model pretraining. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[18] A. K. Dash. VGG-16 Architecture. https://iq.opengenus.org/vgg16/.

[19] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[20] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[21] A. Gholami, A. Azad, K. Keutzer, and A. Buluç. Integrated model and data parallelism in training neural networks. *CoRR*, abs/1712.04432, 2017.

[22] J. Goossens. Scheduling of offset free systems. *Real-Time Systems*, 2003.

[23] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[24] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[25] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, Boston, MA, Feb. 2019. USENIX Association.

[26] S. Gupta. VGG-11 Architecture. https://iq.opengenus.org/vgg-11/.

[27] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[28] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *NeurIPS*, 2019.

[29] Z. Jia, S. Lin, C. R. Qi, and A. Aiken. Exploring hidden dimensions in accelerating convolutional neural networks. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2274–2283, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[30] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo. A unified architecture for accelerating distributed DNN training in heterogeneous gpu/cpu clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479. USENIX Association, Nov. 2020.

[31] C. Karakus, R. Huilgol, F. Wu, A. Subramanian, C. Daniel, D. Cavdar, T. Xu, H. Chen, A. Rahnama, and L. Quintela. Amazon sagemaker model parallelism: A general and flexible framework for large model training, 2021.

[32] A. Kaushik. VGG-19 Architecture. https://iq.opengenus.org/vgg19-architecture/.

[33] M. Khani, M. Ghobadi, M. Alizadeh, Z. Zhu, M. Glick, K. Bergman, A. Vahdat, B. Klenk, and E. Ebrahimi. Sip-ml: High-bandwidth optical network interconnects for machine learning training. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 657–675, New York, NY, USA, 2021. Association for Computing Machinery.

[34] M. Ladeira, E. Grolleau, F. Bonneval, G. Hattenberger, Y. Ouhammou, and Y. Hérouard. Scheduling Offset-Free Systems Under FIFO Priority Protocol. In M. Maggio, editor, *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, volume 231 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:19, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[35] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing. On model parallelization and scheduling strategies for distributed machine learning. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27, pages 2834–2842. Curran Associates, Inc., 2014.

[36] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich. Pytorch-biggraph: A large-scale graph embedding system. *CoRR*, abs/1903.12287, 2019.

[37] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. OSDI'14, pages 583–598. USENIX Association, 2014.

[38] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, aug 2020.

[39] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[40] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, Santa Clara, CA, Feb. 2020. USENIX Association.

[41] K. Mahajan, C.-H. Chu, S. Sridharan, and A. Akella. Better together: Jointly optimizing ML collective scheduling and execution planning using SYNDICATE. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 809–824, Boston, MA, Apr. 2023. USENIX Association.

[42] L. Mai, C. Hong, and P. Costa. Optimizing network performance in distributed machine learning. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, 2015. USENIX Association.

[43] L. Martin, B. Müller, P. J. O. Suárez, Y. Dupont, L. Romary, É. V. de la Clergerie, D. Seddah, and B. Sagot. Camembert: a tasty french language model. *CoRR*, abs/1911.03894, 2019.

[44] J. Mohan, A. Phanishayee, J. J. Kulkarni, and V. Chidambaram. Looking beyond gpus for dnn scheduling on multi-tenant clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, July 2022.

[45] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo, J. A. Yang, L. Gao, D. Ivchenko, A. Basant, Y. Hu, J. Yang, E. K. Ardestani, X. Wang, R. Komuravelli, C.-H. Chu, S. Yilmaz, H. Li, J. Qian, Z. Feng, Y. Ma, J. Yang, E. Wen, H. Li, L. Yang, C. Sun, W. Zhao, D. Melts, K. Dhulipala, K. Kishore, T. Graf, A. Eisenman, K. K. Matam, A. Gangidi, G. J. Chen, M. Krishnan, A. Nayak, K. Nair, B. Muthiah, M. khorashadi, P. Bhattacharya, P. Lapukhov, M. Naumov, L. Qiao, M. Smelyanskiy, B. Jia, and V. Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models, 2021.

[46] S. Narasimhan. NVIDIA Clocks World's Fastest BERT Training Time and Largest Transformer Based Model, Paving Path For Advanced Conversational AI, Aug. 2019. https://devblogs.nvidia.com/training-bert-with-gpus/.

[47] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP'19, pages 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[48] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[49] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 16–29, New York, NY, USA, 2019. Association for Computing Machinery.

[50] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 1–18. USENIX Association, July 2021.

[51] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving language understanding by generative pre-training. https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf.

[52] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. 2018. https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf.

[53] S. Rajasekaran, M. Ghobadi, G. Kumar, and A. Akella. Congestion Control in Machine Learning Clusters. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, HotNets '22, page 235–242, 2022.

[54] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik. Scaling distributed machine learning with In-Network aggregation.

In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, Apr. 2021.

[55] A. Sergeev and M. D. Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[56] A. Shah, V. Chidambaram, M. Cowan, S. Maleki, M. Musuvathi, T. Mytkowicz, J. Nelson, O. Saarikivi, and R. Singh. TACCL: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 593–612, Boston, MA, Apr. 2023. USENIX Association.

[57] C. J. Shallue, J. Lee, J. M. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl. Measuring the effects of data parallelism on neural network training. *CoRR*, abs/1811.03600, 2018.

[58] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. Hechtman. Mesh-tensorflow: Deep learning for supercomputers. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[59] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[60] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[61] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 2017.

[62] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[63] P. Sun, W. Feng, R. Han, S. Yan, and Y. Wen. Optimizing network performance for distributed dnn training on gpu clusters: Imagenet/alexnet training in 1.5 minutes. *arXiv preprint arXiv:1902.06855*, 2019.

[64] B. Wang, Q. Xu, Z. Bian, and Y. You. Tesseract: Parallelize the tensor parallelism efficiently. In *Proceedings of the 51st International Conference on Parallel Processing*. ACM, aug 2022.

[65] G. Wang, S. Venkataraman, A. Phanishayee, J. Thelin, N. Devanur, and I. Stoica. Blink: Fast and generic collectives for distributed ml. In *Conference on Machine Learning and Systems (MLSys 2020)*, March 2020.

[66] W. Wang, M. Khazraee, Z. Zhong, M. Ghobadi, Z. Jia, D. Mudigere, Y. Zhang, and A. Kewitsch. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 739–767, Boston, MA, Apr. 2023. USENIX Association.

[67] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, Renton, WA, Apr. 2022. USENIX Association.

[68] Z. Xia, Y. Zhou, F. Y. Yan, and J. Jiang. Genet: Automatic curriculum generation for learning adaptation in networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 397–413, New York, NY, USA, 2022. Association for Computing Machinery.

[69] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, Oct. 2018. USENIX Association.

[70] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548. USENIX Association, Nov. 2020.

[71] M. Yang, A. Baban, V. Kugel, J. Libby, S. Mackie, S. S. R. Kananda, C.-H. Wu, and M. Ghobadi. Using trio: Juniper networks' programmable chipset - for emerging in-network applications. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 633–648, New York, NY, USA, 2022. Association for Computing Machinery.

[72] S. Zagoruyko and N. Komodakis. Wide residual networks, 2016.

[73] H. Zhao, Z. Han, Z. Yang, Q. Zhang, F. Yang, L. Zhou, M. Yang, F. C. Lau, Y. Wang, Y. Xiong, and B. Wang. HiveD: Sharing a GPU cluster for deep learning with guarantees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 515–532. USENIX Association, Nov. 2020.

[74] L. Zhao, S. Pal, T. Chugh, W. Wang, P. Basu, J. Khoury, and A. Krishnamurthy. Optimal direct-connect topologies for collective communications, 2022.

[75] Y. Zhao, Y. Liu, Y. Peng, Y. Zhu, X. Liu, and X. Jin. Multi-resource interleaving for deep learning training. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 428–440, New York, NY, USA, 2022. Association for Computing Machinery.

[76] P. Zheng, R. Pan, T. Khan, S. Venkataraman, and A. Akella. Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 703–723, Boston, MA, Apr. 2023. USENIX Association.

[77] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery.

A Proof of Theorem 1

This section provides the proof of Theorem 1 (Correctness and Uniqueness Guarantee) in Section 4. To prove uniqueness we need to show that Algorithm 1 assigns a time shift value exactly once to every job $j \in J$ in the cluster with Affinity graph $G = (U, V, E)$. To prove the correctness, we need to show that:

$$\forall l, \forall (j_n, j_m) \in \{(j_i, j_k) | (j_i, l) \in E \text{ and } (j_k, l) \in E\} : \\ (t_{j_n} - t_{j_m}) \% p^l = (t^l_{j_n} - t^l_{j_m}) \% p^l \quad (6)$$

where p^l is the perimeter of the geometric abstraction for link l. In other words, to guarantee correctness, we prove that for every pair of jobs sharing a link, the difference in the time-shift values assigned by the algorithm is equal to the relative time-shift given by CASSINI's optimization formulation for that link.

We first use induction to prove that both the above statements are true for any connected and loop-free Affinity graph $G = (U_1, V_1, E_1)$, and later we extend this to a general Affinity graph with many connected sub-graphs.

Base case: First, we show that both statements hold for a graph G with only one link l. Algorithm 1 first selects one of the jobs j_1 connected to the link l and sets $t_{j_1} = 0$. Using its BFS traversal algorithm for all the other jobs j_i connected to l, Algorithm 1 sets the time shift as:

$$t_{j_i} = -t^l_{j_1} + t^l_{j_i}$$

As the algorithm uses BFS and visits each job exactly once, the time-shift value is computed and assigned exactly once for each job. This ensures that for a given job, there is a unique time-shift value computed by the algorithm.

To show correctness, we need to prove equation 6 for all job pairs connected to the link l. Say j_n and j_m are two jobs connected to the link l, then:

$$(t_{j_m} - t_{j_n}) \% p^l = (-t^l_{j_1} + t^l_{j_m} - (-t^l_{j_1} + t^l_{j_n})) \% p^l \\ = (t^l_{j_m} - t^l_{j_n}) \% p^l$$

This shows that the time shift assignments are correct for the base case.

Assumption Step: Let us assume that the two statements hold for every connected and loop-free Affinity graph having n links.

Induction step: We use the above assumption to prove that the two statements hold for a connected and loop-free Affinity graph having $n+1$ links. Let $G = (U_s, V_s, E_s)$ be the connected sub-graph with n links. Now, we create an affinity graph with $n+1$ links, by adding a new link l_n which is already connected to some set of jobs $J = \{j\}$. In order to get a connected and loop-free Affinity graph with $n+1$ links, l_n has to be connected to exactly one job $j_i \in U_s$. It has to be exactly one because having an edge with more than one job from the sub-graph G will form a loop, and not being connected with any of the jobs from the sub-graph G will make the Affinity graph disconnected. Let j_i be the job from subgraph G that is connected to l_n. Since j_i is the only path to reach the link l_n and the jobs J connected to the link, our algorithm 1 will reach link l_n through job j_i. Then, from the algorithm, the time assignments for the jobs in J are given by:

$$\forall j_k \in J, t_{j_k} = t_{j_i} - t^{l_n}_{j_i} + t^{l_n}_{j_k}$$

The uniqueness is guaranteed since BFS visits each job only once. From the assumption step, the correctness constraints for all the links in the subgraph G are assumed to be valid, so we have to only prove equation 6 for the jobs connected to l_n.

$$\forall (j_m, j_n) \in J, (t_{j_m} - t_{j_n}) \% p^l = (t_{j_i} - t^l_{j_i} + t^l_{j_m} - (t_{j_i} - t^l_{j_i} + t^l_{j_n})) \% p^l \\ = (t^l_{j_m} - t^l_{j_n}) \% p^l$$

This shows that both statements hold true for any Affinity graph with $n+1$ links. This concludes the induction proof. Hence, Algorithm 1 holds true for all connected and loop-free Affinity graphs.

Now, we extend to an Affinity graph of a cluster with multiple connected sub-graphs. Since our algorithm solves each connected sub-graph one by one and assigns a single time-shift value for each job in the sub-graph, uniqueness is guaranteed. For correctness, since there is no edge connecting jobs and links from different disjoint sub-graphs there are no constraints across disjoint graphs that need to be checked for correctness. Hence, this concludes the overall proof.

Example. As an example, traversing the affinity graph in Figure 8(b) results in the following unique time-shifts for j_1, j_2, and j_3:

$$t_{j_1} = 0 \ (\textit{reference point}) \quad (7)$$

$$t_{j_2} = (-t^{l_1}_{j_1} + t^{l_1}_{j_2}) \quad \text{mod} \quad \textit{iter_time}_{j_2} \quad (8)$$

Affinity graph path: $j_1 \rightarrow l_1 \rightarrow j_2$

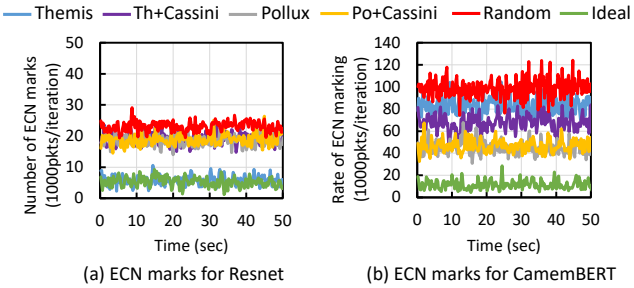
$$t_{j_3} = (-t^{l_1}_{j_1} + t^{l_1}_{j_2} - t^{l_2}_{j_2} + t^{l_2}_{j_3}) \quad \text{mod} \quad \textit{iter_time}_{j_3} \quad (9)$$

Affinity graph path: $j_1 \rightarrow l_1 \rightarrow j_2 \rightarrow l_2 \rightarrow j_3$

For the correctness of the algorithm, the graph should be loop-free. In CASSINI's design, we eliminate placement configurations that have loops. Themis allocates servers using an auction procedure, which involves multiple jobs in the cluster participating in the auction. This allows multiple possible placement configurations for the jobs participating in the auction. Hence, it is easy to find many loop-free placement configurations among them. Similarly, Pollux reallocates resources periodically, involving multiple jobs and creating many possible placement configurations.

DNN	Memory requirement (MB)	Batch size/GPU	Parallelization strategy	Type
VGG11 [26]	507	512-1800	Data Parallel	Vision
VGG16 [18]	528	512-1800	Data Parallel	Vision
VGG19 [32]	549	512-1800	Data Parallel	Vision
WideResNet101 [72]	243	256-1200	Data Parallel	Vision
ResNet50 [27]	98	256-1800	Data Parallel	Vision
BERT [20]	450	8-32	Data Parallel	Language
RoBERTa [39]	800	8-32	Data Parallel	Language
CamemBERT [43]	266	8-32	Data Parallel	Language
XLM [17]	1116	4-32	Data Parallel	Language
GPT1 [51]	650 - 9000	32-80	Model Parallel	Language
GPT2 [52]	1623- 27000	32-80	Model Parallel	Language
GPT3 [11]	1952-155000	16-48	Model Parallel	Language
DLRM [6]	890 - 1962	16-1024	Model Parallel	Recomm.

Table 3: DNN models used in our experiments.



(a) ECN marks for Resnet (b) ECN marks for CamemBERT

Figure 19: Number of ECN marked packets per iteration

B DNN Models

As mentioned in Section 5.1, we run our experiment with 13 popular DNN models: VGG11 [26], VGG16 [18], VGG19 [32], ResNet50 [27], WideResNet101 [72], BERT [20], RoBERTa [39], XLM [17], CamemBERT [43], GPT1 [51], GPT2 [52], GPT3 [11], and DLRM [6]. Table 3 summarizes the parameters of each model and batch sizes. Note that the batch sizes are provided as a range because the number of workers and hyper-parameters change during scheduling epochs. In particular, in different experiments, we select the batch size according to the hyper-parameters used in prior work [1, 40, 50, 54, 57]. The memory requirement of each model reflects the amount of memory each model occupies in the GPU memory. We adjust the model sizes for different models depending on the parallelization strategy.

C Number of ECN Marked Packets

Figure 19 plots the number of ECN marked packets per iteration for the models ResNet and CamemBERT. These measurements are from the experiment of Section 5.3. The ResNet model has relatively lower ECN marks in general than other models because ResNet has a smaller model size and requires less network bandwidth for its AllReduce phase.

Towards Domain-Specific Network Transport for Distributed DNN Training

Hao Wang[1], *Han Tian*[1], *Jingrong Chen*[2], *Xinchen Wan*[1], *Jiacheng Xia*[1], *Gaoxiong Zeng*[1]
Wei Bai[3,*], *Junchen Jiang*[4], *Yong Wang*[1], *Kai Chen*[1]
[1]*iSING Lab, Hong Kong University of Science and Technology*
[2]*Duke University,* [3]*Microsoft,* [4]*University of Chicago*

Abstract

Machine learning (ML) applications present rich characteristics to underlying network transport, yet little work has been done so far to systematically exploit these properties in transport design. This paper takes the initiative to pursue a domain-specific network transport, called MLT[1], for distributed DNN training that fully embraces several unique characteristics of machine learning.

At its heart, MLT employs three simple-yet-effective techniques to form a 3-step progressive scheme against long tail latency caused by transient packet drops and queueing. First, it leverages the *independencies among gradient updates* to enable per-packet load balancing to minimize network hotspots without worrying about packet re-ordering. Then, if hotspot arises, it performs priority queueing/dropping based on the *layers and magnitudes of gradients* to optimize the model convergence. Lastly, if drop occurs, it enables *bounded-loss tolerance*—a certain amount of gradient losses tolerated by the DNN training without affecting the model accuracy. MLT is readily deployable with commodity switches and imposes minimal modifications on various DNN training libraries (e.g., TensorFlow, MXNet and PyTorch) and communication routines (e.g., PS and Ring All-reduce). We show, via both testbed experiments and simulations, that MLT effectively optimizes network tail latency and delivers up to 62.2% better end-to-end training performance over prior work.

1 Introduction

Deep Neural Networks (DNNs) have been paramount to modern machine learning applications in computer vision (CV) [41, 57, 91] and natural language processing (NLP) [74, 99, 102]. However, training DNNs can be notoriously slow, due importantly to the sheer volumes of gradients that need to be frequently shuffled. While individual forward/backward propagations can be massively parallelized, typical DNN training still includes 100s of iterations, each ending up with shuffling massive gradients across 10s to 100s of workers, potentially causing severe worst-case congestion and tail latencies and slowing DNN training down to a crawl. Such communication bottlenecks have been witnessed in production clusters and reported in many literatures [33, 39, 50, 69, 78, 81, 100, 114].

To reduce the communication overhead in distributed DNN training, many solutions have been proposed recently. Some of them attempt to reduce traffic volume at application layer through gradient compression (e.g., sparsification [33, 66, 101] or quantization [12]), while others seek to overlap communication and computation through tensor partitioning and transmission scheduling [39, 49, 81]. However, while these solutions effectively optimize the *average* flow completion time (FCT) in general, they are susceptible to transient queueing or loss which could lead to severe *tail* FCT when network is under pressure. As we show in §2.2, by reducing the total traffic volume, gradient compression can reduce the average FCT, but fails to avoid the long tail FCT caused by sporadic packet loss in the network, which adversely affects the DNN training performance.

One plausible way to tackle this problem is to borrow the sophisticated datacenter network transport solutions such as DeTail [110], pFabric [15], Homa [77] and NDP [38], to name a few, to deep learning clusters. However, while these advanced solutions can potentially deliver near-optimal average or tail latencies, they are too complex by either re-factoring the whole network stack from physical layer, or requiring the support of specific switch ASICs (e.g., cutting payload [38]), or assuming non-blocking network cores, making them hard to deploy in practice.

In this paper, we ask: *can we design a simple, effective yet readily-deployable solution to the above problem?* Motivated by the special properties of machine learning (§3), we answer the question affirmatively by presenting MLT (§4), a domain-specific network transport that can optimize both average and tail FCTs of distributed DNN training by exploiting a series of ML-specific properties, in a progressive manner.

First, MLT exploits the property of inter-packet independency to perform per-packet load balancing, which minimizes network hotspots. Per-packet load balancing is highly desirable by almost all prior datacenter solutions with multi-pathing. However, none could really enjoy its benefit without paying considerable cost for packet re-ordering, with some of them backing off to flowlet as a compromise [13, 96]. Unlike traditional applications where one message often contains multiple packets (thus order-dependent), we note that messages in DNN training are essentially gradients, which are small enough (32bits or less) to allow multiple messages packed in one packet (thus order-free) (§3.1). By exploring this, MLT can effectively minimize network hotspots with perfect packet spraying without re-ordering concern.

Second, when hotspot arises, MLT performs priority

[*]Now with NVIDIA
[1]**MLT**: **M**achine **L**earning **T**ransport for AI-centric Networking (AICN).

queueing or dropping at the switch based on the importance of gradients in the packets. As elaborated in §3.3, we find that the importance of a gradient depends on its position (i.e., DNN layer) and magnitude. Typically, gradients of front layers are less important than back layers due to the popular pre-training technique used in CV and NLP applications: the front layers are often pre-trained on large/generic datasets and thus need fewer updates than the back layers [107]. Meanwhile, larger gradients are more important than smaller ones, because large gradients are critical for identifying the correlation between the intermediate features and the final prediction result.

Lastly, if packet drop occurs, MLT resorts to its final defense—bounded-loss tolerance—no retransmission up to a certain packet loss fraction allowed by the *approximate* SGD-based training, to protect against the long tail latency (§3.2). In contrast to today's all-or-nothing transport primitive (TCP or UDP), MLT supports a bounded-loss tolerance transport primitive: by intentionally ignoring packets (bounded by p) delayed or lost in the network without retransmission, MLT effectively cuts tail latency while still maintaining the same model accuracy. We note that while such loss-tolerance property has been used in prior work to reduce traffic volume [101] at the application layer which reduces the average FCT, we reuse it at the transport layer to cut the tail (§4).

We have implemented MLT (§5) with commodity switches, integrated it with TensorFlow, MXNet and PyTorch, and deployed it over our small-scale testbed with 64 RTX3090 GPUs. In our implementation, we only use basic switch functions such as ECN/RED and priority queueing, and build MLT transport protocol in user space without modifying kernel network stack. Through extensive testbed experiments and large-scale simulations (§6), we find that:

- Compared to the state-of-the-art in communication optimization of DNN training (BytePS [52]), MLT achieves up to 12.0%–62.2% training speedup across various DNN models while maintaining the same accuracy (§6.1).
- Compared to the PyTorch FSDP [115] in fine-tuning Large Language Models (LLMs) with fully sharded data parallelism [115], MLT achieves up to 35.5% speedup without affecting the fine-tuning process (§6.1).
- Compared to one of the best datacenter transports (NDP [38]), MLT delivers comparable network performance (in fact, 6.7%/10.3% lower average/tail FCTs) under realistic ResNet50-induced traffic, with no switch modification (§6.2).
- Each design component of MLT contributes effectively to the ultimate performance of MLT (§6.3).

We believe MLT showcases a first step towards an AI-centric networking design that systematically explores a series of ML-specific characteristics. While many questions, from theoretical understanding of more ML algorithmics/characteristics to practical real-world implementation/deployment, remain open (§7), our exploration in MLT has shown early promise of this direction and hopefully could inspire this community to think more on how to design AI-centric networking for ML applications by exploiting its special properties.

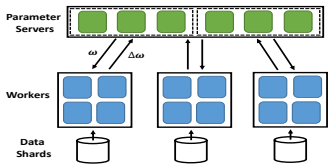


Figure 1: Data parallelism with Parameter Server

2 Distributed DNN Training

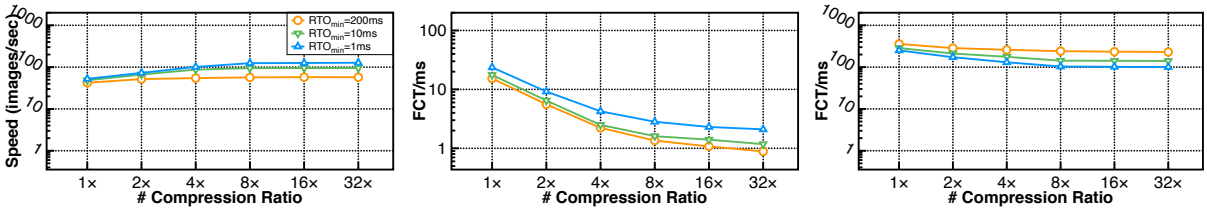
2.1 Communication Overhead

Training basics: DNNs learn intricate representations on large datasets. A training process refines a DNN model upon a dataset for many epochs, each consisting of multiple iterations. In each iteration: (1) The DNN model and a data partition (or mini-batch) are taken as input; (2) the mini-batch travels through the model from the first layer to the last layer, called *forward propagation (FP)*, and computes a loss; (3) with the loss derived, it computes the gradients backwards from the last layer to the first layer, called *backward propagation (BP)*; and finally (4) the gradients, which represent information acquired from the mini-batch, are used to update the model with optimization algorithms, normally Stochastic Gradient Descent (SGD) [21, 54, 82].

Parallelism schemes: There are mainly two types of parallelism schemes for distributed ML: Data Parallelism (DP) and Model Parallelism (MP). DP aims to faster batch processing speed while MP tries to achieve better memory efficiency. When a single GPU can accommodate a model, the mainstream approach is DP [52]. With the rise of LLMs, even the latest H100 [9] GPU cannot fit an entire model. As a result, researchers start to use MP. Some leverage pipelining to enhance computational efficiency (pipeline parallelism [78]), while others further divide a tensor across multiple GPUs (tensor parallelism [90]). Nevertheless, recent wisdom, e.g. , ZeRO [84] and FSDP [115], eliminates memory redundancies of DP and achieves comparable or better performance than MP in terms of memory efficiency, especially for models that are hard to evenly distribute [84]. Such DP approach can even achieve super-linear speedup [84].

Therefore, in this paper, we focus on data parallelism and leave others as future work (§7). To speed up with DP, mini-batches of training data are distributed across multiple machines (or workers) as shown in Figure 1. Different workers share the same global model and independently compute gradients (with FP and BP) on their respective mini-batches. Then, gradients from all workers are synchronized and aggregated to update the global model[2], using either Parameter Server

[2]There are three general approaches to model synchronization: Bulk Synchronous Parallelism (BSP) [95], Asynchronous Parallel (ASP) [86], and Stale Synchronous Parallel (SSP) [104]. Among them, BSP, in which all workers need to train on the same iterations, is preferred in production due



(a) Training Speed (for one worker) (b) Average Flow Completion Time (c) Tail Flow Completion Time

Figure 2: Time cost of training GoogleNet on ImageNet with different compression ratios and RTO_{min}s in our testbed. Note that the minimum RTO_{min} in Linux kernel is 1ms, and we also deep-dive microsecond level RTO_{min} with simulations in §6.3. However, please note that the realistic RTO_{min} in production is set typically at millisecond level even though the base RTT could be at microsecond level, mainly due to the reason explained in Footnote-3.

architecture [60] or collective routines like All-reduce [80]:

- *Parameter Server (PS)* [60] is a logically centralized key-value store. In each iteration, workers pull new model parameters from PS for training, and then push gradients to PS for model updating. PS enables flexible parameter synchronization pattern and is generally fault-tolerant.
- *All-reduce* [80] is a collective operation to sum up gradients of all workers. A popular implementation is Ring All-reduce [88], in which workers form a logical ring. Each worker receives a chunk of gradients from one neighbor, adds to its local copy, and sends the chunk to the other neighbor, until all the gradients are aggregated. Compared to PS, Ring All-reduce generates more uniform traffic pattern, but has higher synchronization overhead due to the smaller flow size and longer communication chain [27].

Communication overhead: The above gradient transmission and model synchronization demand efficient network transport [81]. In each iteration, each worker can send and receive model gradients with tens to thousands of MBs [41,57,91]. As a result, the network communication often takes a significant portion of the total training time. This overhead has been observed by many recent literatures [31, 33, 39, 50, 60, 63, 78, 81, 112]. For example, training AlexNet on 8 nodes requires more than 26Gbps bandwidth to avoid blocking computation of next iteration [112]. Recent measurement has also shown that communication accounts for 90% of the total training time over 32 GPUs [78]. Furthermore, as reported by a large online service provider, due to the communication overhead, the training performance is far from linear speed-up with an increasing number of GPU servers in many of their internal and publicly available training workloads [81].

2.2 Existing Solutions and Problems

There exist solutions to the above problems from different angles, however, they all have shortcomings.

Gradient sparsification and quantization: One direct way to optimize the communication overhead is to reduce traffic volume at application layer through gradient compression (e.g., sparsification [33, 66, 101] or quantization [12]). Specifically, gradient sparsification reduces network traffic by filtering near zero gradients, whereas gradient quantization represents the gradients with lower-precision floating point numbers to reduce traffic volume. While both approaches reduce overall traffic volume, they do not make communication completely immune to long-tail latency due to packet drops or queueing from micro-burstiness.

To show the problem, we train GoogleNet [93], a widely used model, on ImageNet [29] dataset with different compression ratios from 1× (original) to 32× (93.8%). Our testbed contains 64 RTX3090 GPUs connected by 25Gbps bandwidth. The detailed setup is in §6.1.1. For training, we adopted the PS architecture and set the number of servers equal to that of workers [81]. During the communication, workers send gradient updates simultaneously to workers or PSes, which may elicit a pathological phenomenon called incast [24] congestion. As RTO_{min}, short for minimum retransmission timeout, matters for the impact of incast [24], we conduct the experiments with different RTO_{min} values: $200ms$—the default value for Linux, $10ms$—a common setting of previous works [25], $1ms$—the minimum setting for Linux (release version) [24].[3] To exclude the overhead of gradient compression operations, we simulate the compression process by directly modifying the gradient's length. As reference, the computation time of GoogleNet is about $30ms$ for a worker in one iteration. To emulate a multi-job environment, we also run two concurrent background DNN training tasks.

Figure 2 shows the experimental results. One key observation is that the improvement on training speed is quite limited even with a significant gradient compression (Figure 2(a)), e.g., a 32× compression in traffic volume only translates to 1.36-2.4× improvement in training speed. The root cause behind is shown in Figure 2(b)(c): while the gradient compres-

to its simplicity and convergence properties over ASP/SSP [11, 35, 72, 78]. Furthermore, BSP also produces deterministic and reproducible results, which are needed in hyper-parameter tuning [103]. For these reasons, we focus on BSP in this paper and leave ASP/SSP for future exploration.

[3] For theoretical analysis, we also simulate the microsecond level RTO_{min} in §6.3, however, in practice, RTO_{min} in production clusters is typically set at millisecond level to avoid spurious retransmissions under large queue-buildups. Today's commodity switches use shared buffer management to improve burst tolerance, thus causing high queueing delay. For example, Broadcom Trident II [116] with 12MB shared buffer and 32 40G ports can cause up to $12MB/40Gbps = 2.4ms$ queueing delay per hop.

sion reduces the average FCT steadily from 15.3ms to 1.1ms, the tail FCT, which decides the ultimate communication time of one iteration, remains high. For large $RTO_{min} = 200ms$, one retransmission timeout costs at least 200ms delay, so the tail FCT is above 200ms. For smaller $RTO_{min} = 10ms$ or 1ms, we find that the actual RTO is not determined by RTO_{min}, and our measurement shows that the actual RTO is 48ms in case of 32× compression, which is largely caused by consecutive losses and spurious retransmissions. Consider the ∼30ms computation time, such long-tail latency compromises the overall training efficiency, undermining the benefit brought by gradient compression.

The reader may wonder if operators can deploy priority flow control (PFC) [1] to eliminate congestion loss, thus mitigating long tail latency shown above. However, many studies have shown that PFC causes a series of performance (e.g., congestion spreading and unfairness [116]) and management problems (e.g., PFC storm and even deadlock [36]). In a PFC-enabled network, a flow can be paused due to congestion that is not even on its path. Moreover, PFC cannot eliminate packet losses due to link failure, including both fail-stop failure and gray failure [48]. Essentially, the head-of-line blocking nature of PFC makes the network difficult to understand and manage, and we leave it for future exploration.

Computation/communication overlapping and scheduling: Most DNN training frameworks (e.g., MXNet, PyTorch and TensorFlow) and Poseidon [112] overlap communication with backward propagation (BP). Instead of waiting for BP on all DNN layers, they send gradients as soon as one DNN layer is processed, pipelining the gradient transmission of some layers with the gradient computation of other layers. Building on Poseidon, P3, TicTac [39] and ByteScheduler/BytePS [52,81] further overlap communication of the current iteration with the forward propagation (FP) of the next iteration through tensor partitioning and priority-based transmission scheduling. Despite being helpful, these solutions are still insufficient because, by design, such DNN structure-level overlappings do not directly solve the tail latency issue shown in Figure 2. Furthermore, they ignore the network as they are purely endhost-based solutions, but network switches are unaware of such application-level priorities when choosing which packets to queue or drop.

Advanced datacenter transport: One plausible way to tackle the tough tail latency could be introducing the sophisticated datacenter network transport solutions such as DeTail [110], pFabric [15], Homa [77] and NDP [38], etc., to deep learning clusters. These deliberately designed solutions can potentially deliver near-optimal average or tail latencies, which may effectively solve the issues in Figure 2. However, the downside is that these solutions are typically too complex, e.g., either re-factoring the whole network stack from physical layer, or requiring switch hardware modifications (e.g., cutting payload [38]), or assuming non-blocking network cores, making them hard to deploy in practice.

3 Observations and Opportunities

We seek an effective yet readily-deployable solution without much complexity. By exploiting the domain-specific properties of DNN training, we make the following observations which provide insights for our design.

3.1 Packets Are Order-Independent

Unlike many other applications, packets in DNN training can tolerate out-of-order delivery. This is because in traditional applications, the application data unit or message usually spans multiple packets, and therefore, ordering needs to be maintained among packets of a message. In contrast, in DNN training context, the message is just a gradient or parameter, which is typically represented as a 32-bit floating pointer number [60]. Therefore, multiple message can be packed within one packet, and packets can be interpreted independently. Such inter-packet order-independency provides an opportunity for packet-level load balancing in the network.

Meanwhile, the traffic in DNN training is predictable. To enhance communication efficiency, DNN training frameworks divide a tensor into buckets, e.g., the default maximum bucket size in PyTorch [79] is 25MB. To better overlap communication with computation, BytePS [52] implements tensor partitioning with a default size of 4MB. Such pre-determined maximum bucket or partition sizes facilitate tensor re-construction at the receiver end.

3.2 Packet Losses Are Bound-Tolerant

SGD-based DNN training is essentially an *approximation* algorithm which estimates better parameter values based on gradients calculated from mini-batches [21, 62]. It can tolerate loss for two main reasons: (1) gradient loss from workers results in dynamic mini-batch sizes instead of directly causing errors, as SGD updates the model with the average gradient values collected from workers, the expected gradient for update can be unbiased under certain random loss; (2) even if an error occurs, it will self-heal automatically. This is because in each round SGD recalibrates the gradient vector towards the optimal based on the current model weights, errors caused by loss in the earlier iterations will not be propagated to the latter ones. Recent work [108] has theoretically proven that, under random loss, this algorithm can converge within the same magnitude of iterations.

Before using this property, we try to understand it more precisely. We simulate the scenario of distributed SGD on different DNN models, and randomly set the gradients to be 0 with ratio p to simulate packet losses in the network. To quantify the impact, we first train the baseline (no loss) for 500 iterations, which is large enough to make the test accuracy almost constant for all the models and datasets. The final test accuracy value is set as Quality Target [4]. Then, we measure the rounds needed to reach the Quality Target for each ratio p. This approach is the same as MLPerf [4], a well-known ML benchmark. We repeat each measurement for 10 times

Bounded-loss ratio (p)	0%-1%		1%-2%	>2%
Model	GoogleNet [93] (0.7%)	LSTM [42] (0.8%)	ResNet34 [41] (1%)	Wide ResNet50 [109] (2.4%)
	AlexNet [57](0.8%)	VGG16 [91] (0.8%)	GRU [26] (1.2%)	ResNet50 [41] (2.4%)
	EfficientNetB0 [94] (0.8%)	VGG19 [91] (0.9%)	Wide ResNet101 [109] (1.3%)	ShuffleNetV2 [71] (2.5%)
	VGG13 [91] (0.9%)	ResNet18 [41] (0.9%)	DenseNet169 [47] (1.6%)	ResNet101 [41] (3.3%)

Table 1: The bounded-loss tolerance property for a wide range of DNN models. For LSTM and GRU models, we train them as NLP tasks using Wikitext-2 [75]; Other models are trained as CV tasks using ImageNet100 [7].

Model	GoogleNet		GRU	
Dataset	Cifar100	Caltech101	Wiki.-2	Wiki.-103
Grad. loss	0.8%	0.8%	1.2%	1.4%

Table 2: Loss-tolerant bounds for different datasets

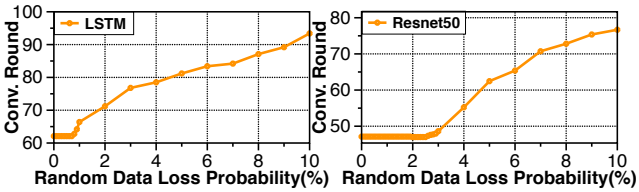


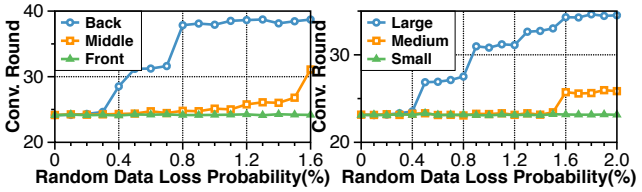
Figure 3: Impact of loss on model convergence: when the loss ratio is below 0.8% (for LSTM) and 2.5% (for ResNet50), the models converge with the same rounds to the same accuracies, indicating loss-tolerance.

and calculate the average. Figure 3 shows the example results of LSTM [42] and ResNet50 [41] trained on Wikitext-2 [75] and Cifar100 [56] datasets. We find that: (1) when the loss ratio is below 10%, we can achieve the Quality Target; and (2) when the loss ratio is below 0.8% (for LSTM) and 2.5% (for ResNet50), we can achieve this with *exactly the same rounds*. We further validate this feature across a wide range of DNN models using several general training datasets. Table 1 summarizes the results for these models, achieving the Quality Target with the same rounds.

We refer to this feature as *bounded-loss tolerance*: DNN training can tolerate a certain fraction p of data loss while still converging with the same iterations to the same accuracy. Despite our fine-grained profiling above, we note that the general property of tolerating gradient loss in DNN training is not new and has been explored in prior works [44, 101, 108]. However, while these existing solutions leverage it for reducing traffic volume [101] or designing reliable PS algorithms [108] at application layer to reduce average FCT, our paper reuses it at network transport layer to cut the tail (§4).

In addition, while different models may have different loss tolerance bounds, we observe that the bounds of one model for different datasets we used remain similar. We show a case in Table 2, in which the bounds for GoogleNet over Cifar100 [56] and Caltech101 [34] are the same, while the bounds for GRU over Wikitext-2 and Wikitext-103 [75] only differ by 0.2%. This enables us to profile the loss tolerance bound values for general DNN models[4].

[4]We note that in practice the bound of a model may vary if datasets differ greatly in some aspects. To explore the loss tolerance bound of a model on a large dataset, one practical way is to use the tolerance bound derived from a smaller sampled sub-dataset from the original dataset as an approximation. Through experiments, we find that the loss tolerance bounds remain almost the same between the sampled sub-dataset and the original dataset. We leave a full exploration of this as future work.



(a) Loss on different layers (b) Loss with different values

Figure 4: Gradient losses on different NN layers (a) and with different magnitudes (b) generate different impacts on model convergence.

3.3 Packets/Gradients Differ in Importance

We observe that different gradients have different impacts on DNN training, depending on their layer positions in DNNs [111], and their magnitudes [44].

First, for convergence, gradients of front layers are often less *important* than back layers due to the prevalent pre-training technique used in CV and NLP applications. Taking CV as an example, front layers extract low-level features such as edges and corners, whereas back layers learn more complex concepts like shapes of certain object. Due to the generality of low-level features across different tasks and datasets, people often pre-train the front layers of their DNNs on large and generic datasets, e.g., ImageNet [29], and then fine-tune them over the target dataset, which will accelerate the training and improve the model performance [107]. With per-training, the front layers extracting low-level features are generally well learned, thus need fewer updates than the back layers. Meanwhile, gradients of front layers are more *urgent* when pipelining strategies [50, 81] are employed, because the FP can start once the front-layer gradients are received.

To illustrate the impact of gradient loss on different layers, we train ResNet50 on Cifar100 [56] during which we randomly discard gradients from: the front layers (the first 20% layers), the middle layers (the middle 20% layers), and the back layers (the last 20% layers) with varying loss probabilities. We first trained the model without discarding gradients and found that the test accuracy converged to 93%, thus in our experiment we set 93% as the target accuracy. Results in Figure 4(a) show that front-layer gradients have a much higher bounded-loss tolerance than back-layer gradients. For instance, to maintain the same convergence speed and accuracy, we can only tolerate 0.3% gradient loss from the back layers but over 5% from the front layers.

Second, larger gradients are typically more *important* than smaller ones. This is because larger gradients are more effective for SGD to identify the correlations between the

features and the task than that of smaller ones. As a result, their losses have more negative impact on model accuracy [59]. Furthermore, larger gradients indicate bigger learning step sizes, thus having more impact on convergence speed.

To show the impact of gradient loss of different magnitudes, we again consider three scenarios: randomly dropping gradients among the smallest 20%, the medium 20%, and the largest 20% magnitudes with different loss probabilities. Results in Fig 4(b) show that dropping larger gradients has a much lower bounded-loss tolerance than that of dropping smaller ones. For example, to maintain the same convergence property, we can tolerate more than 20% loss of small gradients but only 0.4% of the large gradients.

4 The MLT Design

MLT is inspired by the above observations. In this section, we first introduce the key ideas of MLT (§4.1), and then present the detailed mechanisms (§4.2). We theoretically prove the convergence of MLT in Appendix A.

4.1 Key Ideas

1. Minimizing hotspots with *order-free* per-packet load balancing. Load balancing aims to eliminate hotspots by spreading traffic onto multiple paths. Ideally, it should be done at packet-level. However, to avoid costly packet reordering, current practice remains to work at the sub-optimal coarse granularity [13, 15, 19, 113]. Based on the observation in §3.1, packets of DNN training are order-free, which enables packet-level spreading without reordering concerns. Thus, MLT employs per-packet load balancing to minimize network hotspots (§4.2.1).

2. Gradient-aware packet queueing and dropping. While DNN training tolerates certain packet losses, the impact of losing different gradients differs as per the observation in §3.3: 1) in terms of DNN layer, front layer gradients are less important than back layer ones, whose dropping has less impact on model convergence; 2) in terms of magnitude, large gradients are more important than small ones, whose dropping has more impact on model accuracy. To respect them, when the switch queue is full and some packets have to be dropped, MLT enforces a gradient-aware selective dropping: 1) Packets carrying front layer gradients will be selected for dropping over those carrying back layer ones; 2) Packets carrying smaller gradients will be selected for dropping over those carrying larger ones.

Furthermore, as mentioned in §3.3, while gradients of front layers are less important for convergence, they are more urgent for training pipelining, since FP can start as soon as the front-layer tensors are received [81]. Therefore, in addition to selective dropping, we further enforce priority queueing to prioritize front-layer packets (§4.2.2).

3. Cutting tail latency with bounded-loss tolerance. As shown in §2.2, gradient compression does not completely solve the long tail latency issue. This is because network congestion (packet losses or queueing) may be caused not only by traffic volume, but also by traffic pattern, e.g., high fan-in burst in a short time.

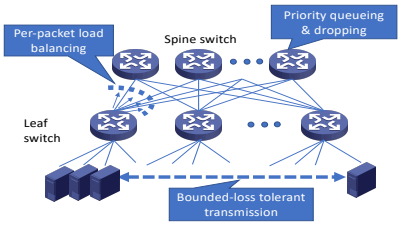


Figure 5: MLT Overview

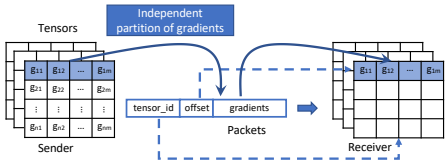


Figure 6: Tensor packing and unpacking

MLT exploits bounded-loss tolerance (§3.2) to address the tail latency. Currently, reliability in transport control is "all-or-nothing": TCP requires all packets to be received and thus may get blocked by a tiny fraction of packet losses waiting for retransmission; whereas UDP has no reliability guarantee. Neither suits for DNN training. Instead, MLT uses a bounded-loss tolerant transport protocol that tolerates up to a bound p fraction of packet losses without retransmission, thus effectively cutting the tail latency while not degrading the training outcome (§4.2.3).

4.2 Mechanisms

Figure 5 overviews MLT, with a 3-step workflow. First, data traffic is spread out onto multiple paths on a per-packet basis to minimize hotpots (**Idea 1**). Then, if hotspot arises, MLT performs priority queueing, followed by selective dropping in case of buffer overflow, based on gradient importance (**Idea 2**). Finally, if packet drops which may potentially trigger timeout, as a final defense, MLT enables bounded loss-tolerant data transmission to avoid long retransmission delay (**Idea 3**). These 3 steps work progressively to protect MLT against the long tail latency.

4.2.1 Order-free Per-packet Load Balancing

MLT performs fast start at end-hosts, and per-packet load balancing in the network. To ensure packet independency, MLT needs careful packet construction.

Order-free packet construction. MLT first performs tensor packing before transmission (Figure 6). Gradients in a tensor are divided into separated groups, which are then packed into packets with tensor ID, layer, and offset information. When a packet reaches the receiver, gradients in the packet can be put into the corresponding addresses according to the tensor ID and gradient offset. For lost gradients, we fill in the corresponding places with zeros. For lost parameters, we use the value of the previous iteration. This ensures that the MLT receiver can still re-construct the tensor even with some lost or reordered packets.

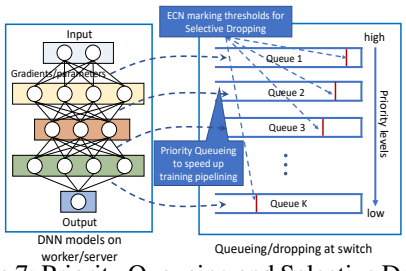


Figure 7: Priority Queueing and Selective Dropping

Per-packet load balancing. We consider two design options for per-packet load balancing. One is to leverage the switch side per-packet ECMP [43]. The other is to give end-hosts the control of multi-path routing [46]. This can be done by source routing or label switching. In our implementation of this paper, we follow the second option as source routing delivers more predictable performance by explicitly determining the path for each packet. With the aid of per-packet load balancing, MLT effectively minimizes hotspots to a great extent.

4.2.2 Gradient-aware Packet Queueing & Dropping

Congestion may still occur even if per-packet load balancing is employed. MLT performs priority queueing and selective dropping in case of queue buildup and buffer overflow, based on the impact of gradients/packets on DNN training (§3.3).

End-host packet tagging. MLT first tags packets based on the layers and magnitudes of gradients. To decide packet priority, a straightforward solution is to map packets of each layer to a unique priority. However, this is impractical as there are usually much more DNN layers than switch priorities (typically 8 [19]). To handle it, MLT evenly distributes all layers into available priorities: it tags the packets of i-th layer with $\frac{iP}{L}$, where L and P are total numbers of DNN layers and available switch priority queues, respectively.

To encode the magnitude information, we adopt a similar approach used in gradient sparsification [66]. Given all gradients in a packet, MLT calculates the mean value and compares it against a threshold to mark whether the packet is *important* or not (indicated by ECN field in packet header §5.2)[5]. By default, the threshold is set to median of all gradients in a tensor. To minimize overhead, we sample 5% of the gradients (inspired by [66] whose sample rate is <1%).

Switch queueing and dropping. Based on the tagging information, MLT switches perform priority queueing and selective dropping as shown Figure 7. For priority queueing, it maps packets of front layers to high priority queues, which speeds up the training pipelines. For selective dropping, it decides whether to drop a packet from two dimensions. On the *layer* level, to selectively drop front layer packets, the corresponding higher priority queues are set with lower dropping thresholds (more analysis in Appendix B). On the *magnitude* level, the switch drops packets based on the importance tags; less important packets get selectively dropped. Note that there is no need for complex deep packet inspection, and both priority queueing and selective dropping can be readily implemented with commodity switches by checking DSCP and ECN fields, respectively (§5.2).

[5]Note that the magnitude of gradient is a better importance indicator than that of the original parameter. If the original parameter is small yet the gradient is large, dropping the gradient may discard update information that builds new correlation between two features from "unrelated" to "related".

4.2.3 Bounded-loss Tolerant Data Transmission

With well-balanced traffic and deliberate queueing and dropping mechanisms, packet loss should happen rarely or, in case it happens, has limited impacts. However, in case of severe losses that may lead to timeouts, as a final resort, MLT enables a bounded-loss tolerance data transmission to avoid long retransmission latency.

Strawman design. To realize bounded-loss tolerant data transmission, a strawman solution is to let the receiver application start next iteration once it receives a certain fraction of gradients. However, this approach suffers from head-of-line (HoL) blocking due to in-order delivery nature of reliable transport protocols like TCP. Consider a 10-packet message where the second packet is lost. Though the transport layer gets 90% of data, it can only deliver the first packet up to the application as the lost second packet blocks the delivery of remaining packets. Consequently, application suffers from unnecessary retransmission delay to move forward.

The reader may wonder the feasibility of unreliable transport protocols such as UDP. While unreliable transport protocols do not have above HoL blocking problem, their best effort nature cannot guarantee the delivery of a certain portion of packets. In addition, they lack congestion control to prevent congestion collapse. Therefore, we decide to come up a new transport protocol that can guarantee the delivery of a certain fraction, say $(1-p)\%$, of packets.

Semi-reliable transmission. Before training, MLT first synchronizes among all the nodes with the following information: (1) loss-tolerant bound $p \in [0,1)$, the maximum tolerable loss fraction for a tensor; (2) ID and size of each tensor. Note that we can get this information before the training starts and tensor sizes are fixed for each of the iterations. Such information is transmitted via a reliable channel, e.g., TCP or RDMA RC (Reliable Connection) to ensure reliability.

When the correctly delivered data reaches the bound, MLT receiver sends a *stop* signal to stop the sender-side transmission. However, it is possible that the receiver may not obtain enough data after the sender transmitted all the packets for a round. To this end, after transmitting all the packets of a tensor, the sender sends a *probe* signal to the receiver to query the status. The receiver responds with a bitmap of received packets, and then the sender will re-transmit all missing packets. Such recovery process continues until the *stop* signal is received by the sender. Note that control packets such as *stop* and *probe* are transmitted over reliable channel.

Minimal rate control. In virtue of the loss-tolerance feature, MLT only requires a minimal rate control to avoid congestion

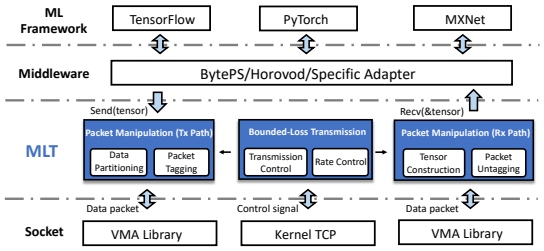


Figure 8: MLT end-host implementation overview

collapse. It uses the delay as the congestion signal and adopts a TIMELY-like [76] algorithm:

- Initially, flow starts at line rate. The sender encodes a timestamp in the data packet while the receiver periodically echoes back the timestamp via ACK. In our setting, the receiver sends one ACK for every ten arrival data packets.
- Every time the sender receives an ACK, it computes the current RTT and RTT gradient, then update the sending rate. The rule is: 1) if the current RTT is less than T_{low} or the RTT gradient is <0, it performs additive increase and sets the rate to $rate + \alpha$; 2) if the current RTT is larger than T_{high}, it performs multiplicative decrease and sets the rate to $rate \cdot (1 - \beta \cdot (1 - \frac{T_{high}}{current_rtt}))$. In our experiment, T_{low} is 12.5μs, T_{high} is 125μs, α is 40Mbps and β is 0.8.

5 Implementation

We build a MLT prototype with Mellanox LibVMA [73] and commodity switches, and integrate it into popular ML frameworks, e.g., Tensorflow [11], PyTorch [79] and MXNet [23].

5.1 End-host Network Stack

Overview. As shown in Figure 8, MLT is implemented as a shim layer running in user space between ML framework and socket interfaces. We provide a series of universal communication interfaces that can be integrated into various ML frameworks [11, 23, 79] and distributed training middleware systems (e.g., Horovod [88] and BytePS [52]).

Universal interfaces. We design and implement basic communication primitives for common application abstractions of ML frameworks [11, 23, 52, 79]. MLT provides an asynchronous operation and completion programming model which is similar to those used in high performance communication libraries such as libverbs and libfabric. It provides two basic communication APIs `dlcp_post_send(tensor, prio_func)` and `dlcp_post_recv (tensor, loss_bound)` and an API `dlcp_poll_cq (cq)` for completion notification. The tensor abstraction is widely used by almost all popular DNN frameworks such as TensorFlow and PyTorch.

MLT sender. At the sender side, a tensor is first partitioned into several MTU-sized (excluding header overhead) segments of gradients. Then we run the priority function `prio_func` to determine the per-segment priority. We use the DSCP field in the IP header to carry the priority value. The MLT header contained in UDP payload encodes the tensor identifier, length, offset and a sequence number.

MLT receiver. At the receiver side, `dlcp_post_recv` takes the tensor received and the loss-tolerant bound as input. Before data transmission, the sender and receiver do a rendezvous to allocate receiving buffer in advance. Upon receiving a new packet, the receiver copies its gradients to the pre-allocated memory according to its offset.

Data & signal transmission. We implement MLT using both UDP and TCP. Inspired by [40], we separate data transfers and control signals, and only provide full reliability for control signals (flow start/finish, retransmisison request, stop request/confirm) whose traffic size is much smaller. To ensure reliability, we use TCP to carry control signals. To minimize the loss of control signals, we reserve a separate priority for them at the switch. We find that control packets are rarely dropped in practice. To achieve high throughput, we adopt UDP in Mellanox LibVMA [73] (instead of Linux kernel), a high-performance user space network stack.

Retransmission. MLT implements selective acknowledgement (SACK) to manage retransmission. The buffer of a transmitting tensor is shared by both the application and MLT library until the corresponding completion is generated. Thus, there is no need to maintain an additional buffer for retransmission of unacknowledged packets.

RDMA Implementation feasibility. Today, RDMA is widely used to accelerate distributed ML training [52, 106]. The reader may wonder how to implement MLT in RDMA NIC. However, we did a feasibility study and noticed that today's RDMA NIC hardware still cannot provide enough programmability to implement complex transport functionalities efficiently. For example, even on the state-of-the-art NVIDIA ConnectX-7 NIC [8] which supports programmable congestion control, users still cannot modify packet retransmission logic inside the NIC. If we onload MLT's complex functionalities, e.g., semi-reliable transmission, to user space like previous work [53], we may lose the real benefits of hardware offloading. Therefore, we leave a full hardware implementation of MLT as future work.

5.2 Switch Configurations

We implement priority queueing and selective dropping using built-in functions of commodity switches.

Priority queueing (with DSCP). We enable strict priority queueing and classify packets into the corresponding priority queues based on the DSCP field [19, 22].

Selective dropping (with ECN). Current switching chips cannot *push out* packets that are already stored in the switch buffers. Therefore, we can only selectively drop packets at the ingress. To this end, inspired by Aeolus [45], we use RED/ECN function [18], which is supported by commodity switches. In current implementations, when the switch queue size exceeds the ECN marking threshold, the switch will mark the arrival ECN-capable packets and *drop* non-ECN-capable packets. Hence, to implement selective dropping, we tag the packets carrying

large gradients with ECN-capable at the sender side. To implement layer-wise selective dropping, e.g., packets from the front layers are easier to drop than that from the back layers, we set smaller ECN marking thresholds for higher priority queues.

5.3 ML Framework Integration

MLT can be directly integrated with ML frameworks such as TensorFlow [11], PyTorch [79], and MXNet [23] or indirectly integrated with some distributed training middleware systems such as Horovod [88] and BytePS [52]. Typically, ML frameworks have their own distributed training implementations. They tend to choose a specific RPC or messaging library and build an abstraction over it. For example, MXNet uses PS-Lite [60] and builds a key-value store over it, while PyTorch prefers collective communication API and can have multiple backends such as Gloo [32], MPI [28] or NCCL [5]. These communication abstraction layers decide which nodes are communicating with each other in one iteration and are usually built on the top of point-to-point communication primitives. To directly integrate MLT with ML frameworks, we only need to re-implement their communication abstraction layers using MLT's interfaces. To indirectly integrate MLT with distributed training middleware systems, we have to replace the point-to-point communication with MLT, and construct the All-reduce scheme or PS topology when necessary. As BytePS supports TensorFlow, PyTorch and MXNet, in our prototype, we integrate MLT into BytePS [52].

6 Evaluation

We evaluate MLT with a combination of testbed experiments and larger-scale simulations. The highlights include:

- In testbed experiments, we evaluate MLT across different DNN models, ML frameworks and synchronization paradigms (§6.1). Compared to the state-of-the-art work BytePS [52] or PyTorch FSDP [115], MLT achieves up to 62.2% (PS), 10.2% (Ring) and 35.5% (FSDP, for LLMs) training speedup without hurting the accuracy.
- In large-scale simulations, we compare MLT against various advanced datacenter transport protocols with realistic DNN training traffic (§6.2). MLT achieves 43.1%/91.8% lower average/tail FCTs over DCTCP [14] and 6.7%/10.3% lower average/tail FCTs over NDP [38] under ResNet50-induced traffic, without modifying switch hardware.
- Deep-dive into MLT design (§6.3) shows that each of its design components contributes effectively to its performance. We further quantify the impacts of loss-tolerant bounds and microsecond-level RTO_{min}.

6.1 Testbed Experiments

6.1.1 Experimental Setup

Testbed: Our testbed (Figure 9) has 8 physical GPU servers, each with 8 RTX3090 GPUs, 80 CPU cores (Intel Xeon Gold 5218R), 256GB memory, 2 Mellanox ConnectX5 100Gbps NICs, and 4 Mellanox SN2100 switches running

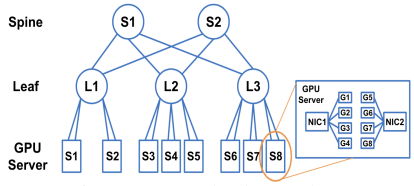


Figure 9: Testbed Topology

Onyx 3.7.1134 OS. It forms a leaf-spine topology with one spine switch and three leaf switches in physical. Each leaf switch has two 100Gbps links connecting to the spine switch, thus logically we have two spine switches. The three racks contain 2, 3, 3 GPU servers, respectively. Each server has two 100Gbps links connecting to the leaf switch. We further divide one physical servers into 8 docker containers, each with 1x GPU, 10x CPU cores, 32GB memory and 25Gbps virtual NIC[6]. Ultimately, we get a 64-node testbed with 2:1 to 3:1 over-subscription, a normal ratio in production [92].

Models and datasets: We use four models and two datasets in our experiments. Our models include three image classification tasks: VGG16 [91], ResNet50 [41] and GoogleNet [93] training on the synthetic data with the same image size as ImageNet [29], and one translation task: Transformer [98] training on SQuAD [85]. We run experiments on three ML frameworks: TensorFlow, PyTorch, MXNet with two parameter synchronization paradigms: PS (colocated and #servers = #workers) and Ring All-reduce.

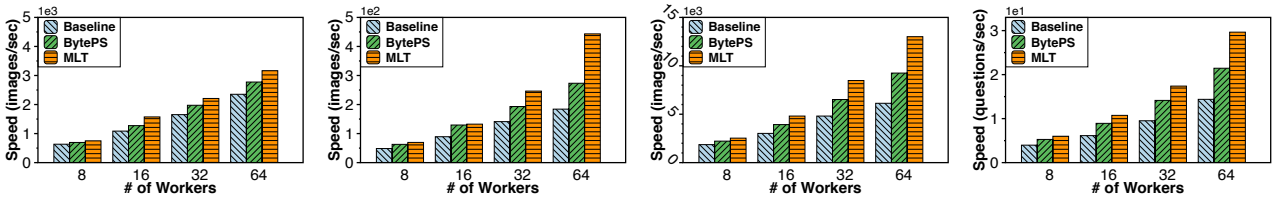
Baselines and metrics: We mainly compare MLT with the vanilla ML frameworks (baseline) and BytePS [52] with cross global barrier enabled. Basically, BytePS incorporates tensor partition and priority scheduling of ByteScheduler [81] and has better code robustness, thus representing a state-of-the-art in communication optimization of DNN training. For image classification models, we use the # of images processed per second as the speed metric, and for transformer we measure the # of questions processed per second [85].

Parameter settings: The batch sizes of VGG16, ResNet50, GoogleNet and Transformer are 32, 32, 32, and 10 samples (images or questions) per GPU, referring to the settings in [81]. Switches have 4MB shared memory, and 8 queues per port. We use DCTCP [14] as the transport protocol for baseline and BytePS. To ensure a fair comparison between MLT and BytePS, we open the multiple connections function [2] of BytePS, i.e., two connections per 25Gbps, to make sure it can saturate the bandwidth. RTO_{min} is $1ms$ and initial window size is 20. With selective dropping, in our experiments, we conservatively set loss-tolerant bound to 10% for MLT (see §6.3 for deep-dive on impact of loss-tolerant bounds).

6.1.2 Results

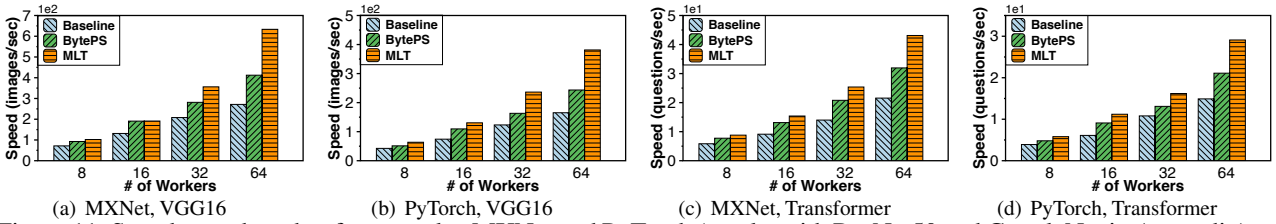
Overall, across different DNN models, ML frameworks and sync paradigms, MLT achieves remarkable training speedup over state-of-the-art. Our measurement also shows that the

[6] We use SR-IOV to separate the physical NIC. SR-IOV can achieve nearly the same performance as the non-virtualized environments [30].



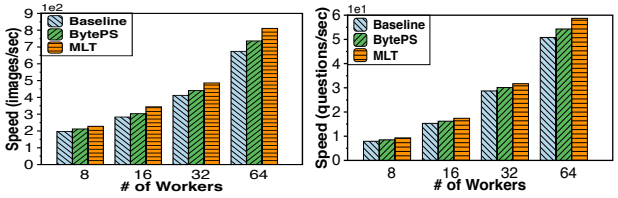
(a) ResNet50 (b) VGG16 (c) GoogleNet (d) Transformer

Figure 10: Speedup under different DNN models using TensorFlow under PS (all converge with the same epochs).



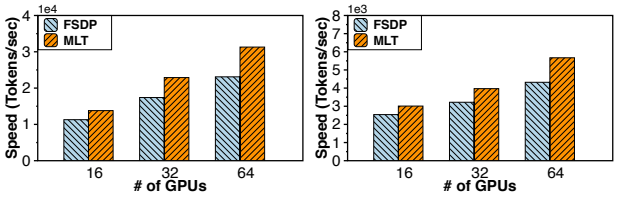
(a) MXNet, VGG16 (b) PyTorch, VGG16 (c) MXNet, Transformer (d) PyTorch, Transformer

Figure 11: Speedup under other frameworks: MXNet and PyTorch (results with ResNet50 and GoogleNet in Appendix).



(a) VGG16 (b) Transformer

Figure 12: Speedup under Ring All-reduce.



(a) T5-3B (b) T5-11B

Figure 13: Speedup under Large Language Models.

additional CPU cost introduced by MLT is <1%. Note that our results below are all based on the condition that *the models converge to the same accuracy with the same iterations.*

Speedup under different DNN models: Figure 10 shows performance of MLT over baseline and BytePS on four DNN models using TensorFlow under PS. From the figure, we make the following three observations. First, MLT performs the best of all on all models. Specifically, MLT outperforms BytePS by 14.1%–62.2% and baseline by 34.5%–141%, across the four models. The main reason is that the default reliable transport is sensitive to packet losses, which may trigger timeouts and cause millisecond-level delay. During the training, we observed around 0.15% packet losses from the buffer counting function provided by our switch [3]. Second, the improvement of MLT becomes more significant as the # of workers increases. As we can see, MLT outperforms BytePS by 9.34% to 10.7% when the # of workers is 8, and by 14.1% to 62.2% when it increases to 64. This is expected because as the network pressure grows higher, packet losses become more frequent. Third, MLT achieves more speedup in VGG16 than other models. As shown in Figure 10(a), MLT achieves up to 62.2% speedup in VGG16, more than that in other models. The reason is that VGG16 is communication-bound and it has the largest communication-to-computation ratio.

Speedup under different ML frameworks: Figure 11 shows the performance of MLT over the baseline and BytePS in PyTorch and MXNet by training VGG16 and Transformer under PS (ResNet50 and GoogleNet in Appendix-C, Figure 22). Note that the native PyTorch Distributed Data Parallel does not support PS, so we use the modified version [6] as the baseline for PyTorch. We observe similar trends as above in TensorFlow. Overall, we find that MLT outperforms BytePS by 12.0%–53.6% and 15.3%–56.6% in MXNet and PyTorch respectively, and they both achieve the best speedup with VGG16 due to the same reason explained above. This experiment shows that MLT can deliver persistent performance improvement across different ML frameworks.

Speedup under Ring All-reduce: The above experiments all use PS, which is widely used in both academia and industry [50, 52, 60, 81], especially for the large and sparse models [61]. Here, we further evaluate the performance of MLT under Ring All-reduce. Please note that BytePS [52] by default does not support the Ring All-reduce communication, we adapt it to support Ring. Figure 12 shows the results of VGG16 and Transformer (ResNet50 and GoogleNet in Appendix-C, Figure 23). In all cases, we see clear performance improvement with MLT, though not significant. Compared to BytePS, MLT achieves 6.89%–10.2% improvement; in comparison, BytePS achieves 5.97%–8.21% improvement over the baseline. As expected, we observe less speedup in Ring All-reduce than that in PS, and the key reason is that: packet loss is rare in Ring All-reduce, thus it is immune from long tail latency caused by timeout. Nevertheless, MLT still delivers better performance over BytePS due to the fine-grained per-packet load balancing and reduction of data volume in transmission with bounded loss tolerance.

Speedup under Large Language Models (LLMs): Recently, LLMs like ChatGPT [10] have gained popularity and received considerable attention from the community. To support training or fine-tuning of LLMs, a popular approach is to divide the parameters, gradients, and optimizer states equally

into each GPU's memory during the data parallelism. Before computing a layer of a model, an all-reduce communication is performed to ensure each GPU possesses a full copy of the current layer. This approach reduces GPU memory footprint through frequent communication. Notable implementations of this approach include Zero Redundancy Optimizer (ZeRO) [84] and Fully Sharded Data Parallel (FSDP) [115].

To evaluate MLT under LLMs, we replace the inter-server communication module of the PyTorch FSDP [115] (using NCCL [5]) with MLT. For the communication of optimizer states that are used for gradient calculation, i.e., gradient momentums and historical gradient values, we treat them the same as gradients. Considering that each GPU does not store the full copy of parameters, we cannot use the past parameters to replace the missing new parameters. To handle this issue, we mark all parameter packets as important packets. This does not significantly alter the proportion of important packets, as the optimizer states dominate the traffic volume in FSDP [84, 115]. Given the limitation of compute power of our testbed, we chose the HuggingFace T5 model [83], a popular transformer-based open-source LLM, 3/11-billion versions and fine-tune them on the WikiHow dataset [55] for the text summarization task.

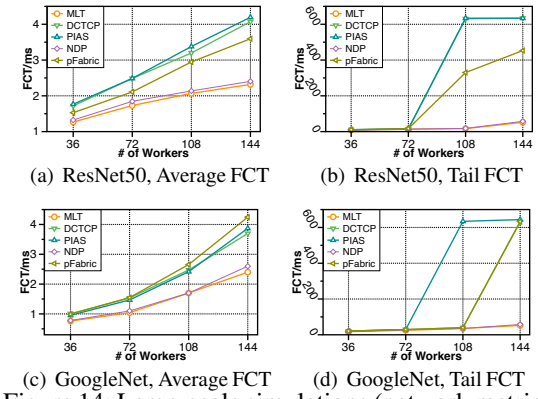
Figure 13 shows the results. In our experiment, we find that for the T5-11B model, 8 RTX3090s are insufficient to accommodate it, so we show the results with 16, 32 and 64 GPUs respectively. Compared to the PyTorch FSDP, MLT achieves an improvement of 22.1%-35.5% for T5-3B and 18.5%-31.2% for T5-11B. We observe more speedup compared to Figure 12 with the same Ring All-reduce setting, this is because FSDP involves more frequent communication and a larger volume of data transfer for LLMs. Moreover, we monitor the model loss changes during the fine-tuning process and find that MLT does not affect the model convergence.

6.2 Large-scale Simulations

6.2.1 Simulation Setup

Topology and traffic: As [15, 19], we choose a leaf-spine topology with 4 core, 9 ToR switches and 144 hosts. Each ToR switch is connected to 16 hosts using 100Gbps links and 4 core switches using 4×100Gbps links. The base RTT between two servers (4 hops) is $24\mu s$. Each switch port has 512KB buffer. For network traffic, we use the realistic workloads derived from training ResNet50 and GoogleNet under PS. We evenly distribute workers and parameter servers across all the racks, with the ratio of 3:1. We also obtain the computation time and tensor sizes from our testbed.

Schemes compared: We use DCTCP [14] as the baseline, as it is widely used in production. We also compare MLT with PIAS [19], pFabric [15], NDP [38] (simulated on htsim). TCP initial window is 10, and ECN marking threshold is 65 packets [14]. RTO_{min} is set to 10ms by default [20, 97] (We also ran simulations with 5ms and 1ms RTO_{min}, and observed similar trends). DupACKs is 3 for DCTCP and PIAS, and DelayAck is disabled. DCTCP and PIAS use per-flow ECMP while pFabric uses per-packet ECMP.



(a) ResNet50, Average FCT (b) ResNet50, Tail FCT

(c) GoogleNet, Average FCT (d) GoogleNet, Tail FCT

Figure 14: Large-scale simulations (network metrics).

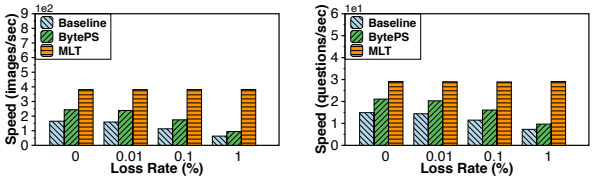
6.2.2 Results

Figure 14 shows the average and tail FCTs of MLT versus other schemes for ResNet50 and GoogleNet induced traffic at varying network scales. In general, MLT delivers the best performance. For ResNet50, MLT achieves up to 43.1%, 44.8%, 35.5%, 6.7% lower average FCT and 91.8%, 91.8%, 88.6%, 10.3% lower tail FCT compared to DCTCP, PIAS, pFabric and NDP, respectively. For GoogleNet, MLT delivers up to 35.1%, 38.0%, 43.5%, 7.7% lower average FCT and 91.7%, 91.9%, 91.7%, 8.7% lower tail FCT over these schemes, respectively. From the above results, we make the following observations:

- *MLT preforms the best in all settings.* MLT achieves the best performance in all workloads and network scales. The main reason is that all the other algorithms suffer from packet loss and retransmission to ensure reliability, while MLT tolerates certain packet loss.
- *MLT significantly reduces the tail FCT.* Relative to average FCT, MLT reduces the tail more significantly. The reason is that retransmission timeout greatly increases the tail of other algorithms, while MLT tolerates packet loss and is free of retransmission timeout. We also note that NDP can achieve similar performance as MLT. However, it requires special switch hardware and is not readily deployable.
- *The speedup of MLT is more notable as the network scale increases.* In general, MLT reduces FCT more significantly at larger scale. The reason is that, as the scale increases, prior solutions experience more packet loss, thus performance degradation, whereas MLT is immune to overhead by packet loss with the bounded-loss tolerance.

6.3 Deep Dive

MLT under gray failure of links [testbed]: Fault-tolerance and reliability are crucial for distributed training, especially for large models. We find that the loss tolerance feature of MLT can be a good solution in the gray failure of network links, i.e., hard-to-detect but persistent link random packet loss, which may be caused by subtle hardware malfunctions, firmware bugs, or environmental interference [48]. A measurement work in Microsoft categorizes a network problem when the loss rate is larger the 0.1% [37]. Here we evaluate the performance of



(a) VGG16 (b) Transformer

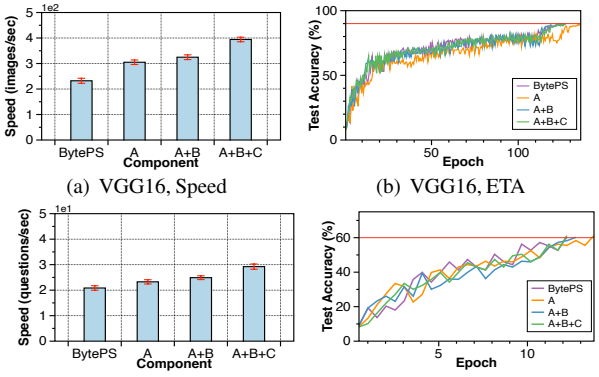
Figure 15: Speedup under gray failure of links.

MLT under gray failure on our 64 GPUs testbed. We assume that a worker-to-switch link has a gray failure, and emulate packet loss by adding random packet dropping to the mapped NICs by using the tc qdisc tool in linux. The loss rate is set to 0, 0.01%, 0.1% and 1% in the experiment and we use PyTorch as the framework. Figure 15 shows the results of VGG16 and Transformer. Figure 21 shows ResNet50 and GoogleNet in Appendix-C. We can see that the baseline and BytePS start to suffer from noticeable performance degradation (12.2%-31.5% and 15%-28.3%) when the loss rate is 0.1% and the degradation is significant (25.7%-61.7% and 31.2%-61.3%) when the loss rate is 1%. As a comparison, MLT maintains its performance in all the loss scenarios.

Design component effectiveness [testbed]: MLT consists of three main components: A) bounded-loss tolerant data transmission, B) gradient-aware queueing and dropping, and C) order-free per-packet load balancing. We now look into the effectiveness of each component. To do so, we set BytePS [52] as the baseline, and start MLT with component A and then gradually add B and C one by one (setting I). Then, we start with component C (using UDP with the minimal rate control of MLT) and gradually add B and A (setting II). We measure the speeds of training ResNet50, Transformer, VGG16 and GoogleNet under PyTorch with 64 GPUs, and we repeat the experiment 10 times and compute the mean and standard error. Correspondingly, we also record the convergence curves (epoch-to-accuracy, ETA). The results of VGG16 and Transformer are shown in Figure 16 for setting I and Figure 17 for setting setting II (ResNet50 and GoogleNet in Appendix-C, Figure 24 and Figure 25). For CNN models, we use top-1 accuracy; for Transformer, we use exact match (EM) as the test accuracy metric.

From the results, we see that each component contributes effectively to the overall performance. Specifically, in setting I, compared to BytePS, MLT with component A only can improve the training speed (left figures). While it requires a bit more rounds to converge to the same accuracy (right figures, due to the packet loss as we set the loss bound as 10%), we find that the end-to-end training time still improves (decided by both unit speed and # of epochs). Then, with component B added, the ETA of MLT is significantly improved (almost as good as BytePS) as the training speed increases. The main reason is due to selective dropping which preserves important gradients from being dropped. Finally, after incorporating component C, MLT maintains its good ETA while improving remarkably in training speed. This is because of the better network utilization brought by perfect load balancing.

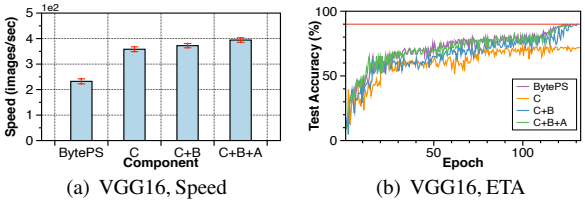
In setting II, spreading gradients out at packet granularity (C)



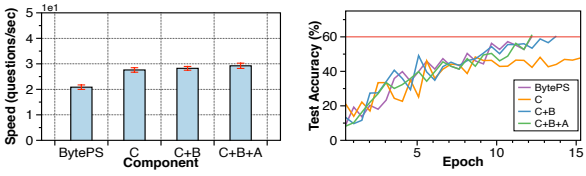
(a) VGG16, Speed (b) VGG16, ETA

(c) Transformer, Speed (d) Transformer, ETA

Figure 16: Effectiveness of design components (I).



(a) VGG16, Speed (b) VGG16, ETA

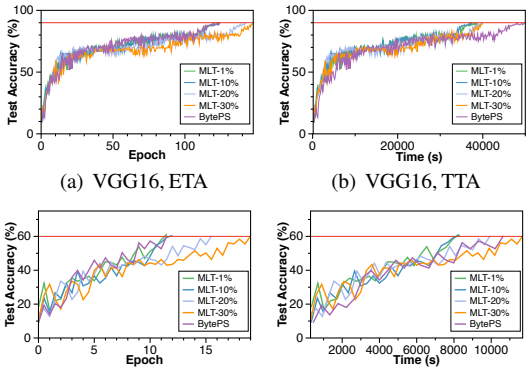


(c) Transformer, Speed (d) Transformer, ETA

Figure 17: Effectiveness of design components (II).

can effectively improve the training speed, but requiring more epochs to converge and may reduce the accuracy. After adding gradient-aware packet queueing and dropping (B), the speed is sightly increased and convergence is significantly improved, only needs a few more epochs than the baseline. The reason is that priority queueing can speed up the communication-computation pipeline and selective dropping can protect more important gradients. Finally, with the bounded-loss tolerance transmission (A), it can further speedup the training and achieving the same convergence speed and accuracy as the baseline.

Impact of loss-tolerant bounds [testbed]: We inspect the convergence behaviors of MLT with different loss-tolerant bounds by training VGG16 and Transformer in Figure 18 (ResNet50 and GoogleNet in Appendix-C, Figure 26). We set the bound as 1%, 10%, 20% and 30%, respectively. Figure 18(a)(c) show curves of ETA. We can see that with 1% and 10% loss-tolerant bounds, the curves are all almost in line with BytePS; with 20% and 30% loss-tolerant bounds, it requires more epochs to converge to the same accuracy. We also measured the bounds of other widely-used models with MLT in Table 5 of Appendix-C. The results indicate that, with the optimization of gradient-aware dropping, MLT can tolerate more packet loss than pure random loss shown in Table 1. This motivates us to set the loss-tolerant bound as 10% in our testbed experiment (§6.1). Figure 18(b)(d) show results of time-to-accuracy (TTA). Compared to BytePS, MLT converges faster under all



(a) VGG16, ETA (b) VGG16, TTA

(c) Transformer, ETA (d) Transformer, TTA

Figure 18: Impact of loss-tolerant bounds.

loss-tolerant bounds, except for MLT-30% under Transformer. Meanwhile, we find that 1% and 10% loss-tolerant bounds take almost the same time to converge. This implies that we might not need to struggle with fine-tuning the loss-tolerant bound in order to achieve satisfactory performance. However, the relationship between the model architecture and its loss-tolerant bounds remains an open question for future study.

Impact of microsecond RTO$_{min}$ [simulation]: As a complement to testbed, we measure the tail FCT when RTO_{min} is <1ms using simulation. Specifically, we choose three RTO_{min}s: 25μs which is close to our 24μs base RTT, and two intermediate values: 100μs and 500μs. We conduct the simulations with GoogleNet-induced traffic under different compression ratios, and show the results in Figure 19. From the figure, we make two observations. First, while the RTO_{min} is sub-ms level, the tail FCT is still around 10ms or above (MLT can reduce it to 4.12ms). This is due to the same reason as explained in §2.2—tail FCT is mainly decided by large actual RTO amplified by consecutive packet losses and spurious retransmissions. For example, we observe 4.2KB spurious retransmission from the tail flow (the size is 127.4KB) and 0.97% packet loss under 25μs RTO_{min} and 32× compression. MLT can reduce the tail to 4.12ms. Second, the curves turn to flat after a certain compression ratio, which implies that further compressing the gradient volume does not help to reduce the tail FCT. This is also in line with our observation in §2.2. As a final remark, while we showcase the μs-level RTO_{min} here, we remind readers that in production RTO_{min} is typically set at ms-level.

7 Open Questions and Discussion

MLT is, by no means, a *full stop* to ML-specific transport. We have left a series of open questions from theoretical aspects of ML characteristics to practical implementation/deployment throughout the paper. Here, we discuss a few more. Due to the space limit, we move the discussion of "co-existing with non-DNN traffic", "flexible selective dropping" and "MLT vs SRD" to the Appendix-D.

MLT for other parallelism schemes. While MLT is mainly designed for data parallelism, its core ideas may provide insights for designing network transport for other schemes, i.e., model/pipeline/tensor parallelism [51]. Some MLT mechanisms can be directly used. For example, the order-free

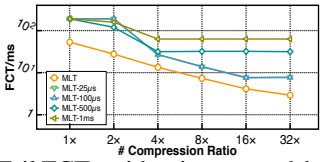


Figure 19: Tail FCTs with microsecond-level RTO_{min}.

pre-packet load balancing remains effective for activations and propagated gradients in model parallelism. Others may need to be modified to adapt to the nature of model parallelism. For example, packets in transmission for model parallelism are primarily used for FP/BP computations, which makes their priorities/loss-tolerance properties different from the model synchronization/update phases in data parallelism. We leave the theoretical analysis of this for future study.

Multi-tenant cloud environment. In multi-tenant cloud environments, e.g., EC2 and Azure, we will not be able to manipulate the underlying network protocols and switches. For example, the traffic may be encapsulated inside a tunnel (like VXLAN) and switches cannot access those fields to perform priority queueing or dropping. Furthermore, the cloud providers typically do not allow tenants to access the queues of the underlying network for security reasons. Thus, we acknowledge that MLT does not work for such scenarios.

8 Related Work

Besides the closely related works discussed in §2.2, there exist some other solutions to improve the communication of DNN training. For example, RDMA [17, 67] and NCCL [5] provide higher bandwidth to speedup tensor transmission. BlueConnect [27] and PLink [70] design novel communication patterns with network topology awareness for gradient synchronization at each iteration for better performance and robustness. GPipe [49] and PipeDream [78] overlap communication with computation in model parallelism. More recently, SwitchML [87] and ATP [58] leverages in-network aggregation to reduce the communication overhead in the network. Note that these works are orthogonal to MLT.

9 Conclusion

This paper presented MLT, a domain-specific network transport exploiting the special properties of machine learning to optimize distributed DNN training. MLT consists of three key ideas: 1) order-free per-packet load balancing, 2) gradient-aware packet queueing and dropping, and 3) bounded-loss tolerant data transmission. Extensive testbed experiments and simulations have shown the promise of MLT.

Acknowledgement

We thank the anonymous NSDI reviewers and our shepherd Costin Raiciu for their feedback and suggestions. This work is supported in part by Hong Kong RGC TRS T41-603/20R, GRF 16213621, ITF ACCESS, NSFC 62062005, Key-Area Research and Development Program of Guangdong Province (2021B0101400001), and the Turing AI Computing Cloud (TACC) [105]. Kai Chen is the corresponding author.

References

[1] Ieee. 802.11qbb. priority based flow control https://1.ieee802.org/dcb/802-1qbb/, 2011.

[2] Byteps best practice: https://github.com/bytedance/byteps/blob/master/docs/best-practice.md?plain=1#L26, 2020.

[3] Mellonax switch: https://www.mellanox.com/products/ethernet-switches, 2020.

[4] Mlperf training results resnet50: https://mlperf.org/training-results-0-6, 2020.

[5] Nvidia collective communications library: https://www.nvidia.com/en-us/data-center/nvlink/, 2020.

[6] Pytorch distributed data parallel supporting ps: https://github.com/bytedance/byteps/blob/master/docs/DistributedDataParallel.md, 2020.

[7] Imagenet-100: https://www.kaggle.com/datasets/ambityga/imagenet100, 2022.

[8] Connectx-7: https://nvdam.widen.net/s/csf8rmnqwl/infiniband-ethernet-datasheet-connectx-7-ds-nv-us-2544471, 2023.

[9] Nvidia h100 tensor core gpu: https://www.nvidia.com/en-us/data-center/h100/, 2023.

[10] Openai chatgpt: https://chat.openai.com, 2023.

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

[12] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *NIPS*, 2017.

[13] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 503–514, 2014.

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

[15] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *SIGCOMM*. ACM, 2013.

[16] Arnold O Allen. *Probability, statistics, and queueing theory*. Academic press, 2014.

[17] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. Empowering azure storage with rdma. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, 2023.

[18] Wei Bai, Kai Chen, Li Chen, Changhoon Kim, and Haitao Wu. Enabling ecn over generic packet scheduling. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 191–204. ACM, 2016.

[19] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 455–468, 2015.

[20] Wei Bai, Shuihai Hu, Kai Chen, Kun Tan, and Yongqiang Xiong. One more config is enough: Saving (dc) tcp for high-speed extremely shallow-buffered datacenters. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 2007–2016. IEEE, 2020.

[21] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT'2010*. Springer, 2010.

[22] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. Scheduling mix-flows in commodity datacenters with karuna. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 174–187, 2016.

[23] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[24] Yanpei Chen, Rean Griffith, Junda Liu, Randy H Katz, and Anthony D Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 73–82, 2009.

[25] Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. Catch the whole lot in an action: Rapid precise packet loss notification in data center. In *11th*

{*USENIX*} *Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 17–28, 2014.

[26] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[27] Minsik Cho, Ulrich Finkler, and David Kung. Blueconnect: Novel hierarchical all-reduce on multi-tired network for deep learning. In *Proceedings of the Conference on Systems and Machine Learning (SysML)*, 2019.

[28] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The mpi message passing interface standard. In *Programming environments for massively parallel distributed systems*, pages 213–218. Springer, 1994.

[29] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[30] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480, 2012.

[31] Nikoli Dryden, Naoya Maruyama, Tim Moon, Tom Benson, Andy Yoo, Marc Snir, and Brian Van Essen. Aluminum: An asynchronous, gpu-aware communication library optimized for large-scale training of deep neural networks on hpc systems. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2018.

[32] Facebook. Gloo: Collective communications library with various primitives for multi-machine training., 2020.

[33] Jiawei Fei, Chen-Yu Ho, Atal Narayan Sahu, Marco Canini, and Amedeo Sapio. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *SIGCOMM*, 2021.

[34] Li Fei-Fei, Rob Fergus, and Pietro Perona. Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. *Computer Vision and Pattern Recognition Workshop*, 2004.

[35] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[36] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 202–215. ACM, 2016.

[37] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.

[38] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42, 2017.

[39] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

[40] E. He, J. Leigh, O. Yu, and T. A. Defanti. Reliable blast udp : predictable high performance bulk data transfer. In *Proceedings. IEEE International Conference on Cluster Computing*, pages 317–324, 2002.

[41] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[42] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[43] Christian Hopps et al. Analysis of an equal-cost multi-path algorithm. Technical report, RFC 2992, November, 2000.

[44] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R Ganger, Phillip B Gibbons, and Onur Mutlu. Gaia: Geo-distributed machine learning approaching lan speeds. In *NSDI*, 2017.

[45] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A building block for proactive transport in datacenters. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures,*

and protocols for computer communication, pages 422–434, 2020.

[46] Shuihai Hu, Kai Chen, Haitao Wu, Wei Bai, Chang Lan, Hao Wang, Hongze Zhao, and Chuanxiong Guo. Explicit path control in commodity data centers: Design and applications. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 15–28, 2015.

[47] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[48] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 150–155, 2017.

[49] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019.

[50] Anand Jayarajan, Jinliang Wei, Garth A. Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. In *Proceedings of Systems and Machine Learning (SysML)*, 2019.

[51] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[52] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed {DNN} training in heterogeneous gpu/cpu clusters. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 463–479, 2020.

[53] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 1–16, 2019.

[54] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[55] Mahnaz Koupaee and William Yang Wang. Wikihow: A large scale text summarization dataset. *arXiv preprint arXiv:1810.09305*, 2018.

[56] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The cifar-10 dataset, 2014.

[57] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[58] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M Swift. Atp: In-network aggregation for multi-tenant learning. In *NSDI*, pages 741–761, 2021.

[59] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[60] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[61] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. *Advances in Neural Information Processing Systems*, 27:19–27, 2014.

[62] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670, 2014.

[63] Y. Li, J. Park, M. Alian, Y. Yuan, Z. Qu, P. Pan, R. Wang, A. Schwing, H. Esmaeilzadeh, and N. S. Kim. A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–188, 2018.

[64] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Advances in Neural Information Processing Systems*, pages 2737–2745, 2015.

[65] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 5330–5340, 2017.

[66] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[67] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K Panda. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[68] Yuanwei Lu, Guo Chen, Larry Luo, Kun Tan, Yongqiang Xiong, Xiaoliang Wang, and Enhong Chen. One more queue is enough: Minimizing flow completion time with explicit priority notification. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[69] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: a rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 41–54, 2018.

[70] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. Plink: Efficient cloud-based training with topology-aware dynamic hierarchical aggregation. In *Proceedings of the 3rd MLSys Conference*, 2020.

[71] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV)*, pages 116–131, 2018.

[72] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612*, 2018.

[73] Mellanox. Messaging accelerator (vma), 2019.

[74] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182*, 2017.

[75] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[76] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. Timely: Rtt-based congestion control for the datacenter. In *SIGCOMM'15*. ACM, 2015.

[77] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

[78] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[79] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[80] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[81] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29. ACM, 2019.

[82] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 1999.

[83] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[84] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[85] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 2016.

[86] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

[87] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701*, 2019.

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

[89] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE Micro*, 40(6):67–73, 2020.

[90] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[91] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[92] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 45(4):183–197, 2015.

[93] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[94] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[95] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[96] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 407–420, 2017.

[97] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G Andersen, Gregory R Ganger, Garth A Gibson, and Brian Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. *ACM SIGCOMM computer communication review*, 39(4):303–314, 2009.

[98] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[99] Subhashini Venugopalan, Marcus Rohrbach, Jeffrey Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. Sequence to sequence-video to text. In *Proceedings of the IEEE international conference on computer vision*, pages 4534–4542, 2015.

[100] Xinchen Wan, Hong Zhang, Hao Wang, Shuihai Hu, Junxue Zhang, and Kai Chen. Rat - resilient allreduce tree for distributed machine learning. In *4th Asia-Pacific Workshop on Networking*, APNet '20, page 52–57, New York, NY, USA, 2020. Association for Computing Machinery.

[101] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization. In *NIPS*, 2018.

[102] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[103] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 595–610, 2018.

[104] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data*, 2015.

[105] Kaiqiang Xu, Xinchen Wan, Hao Wang, Zhenghang Ren, Xudong Liao, Decang Sun, Chaoliang Zeng, and Kai Chen. Tacc: A full-stack cloud computing infrastructure for machine learning tasks. *arXiv preprint arXiv:2110.01556*, 2021.

[106] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. Fast distributed deep learning over rdma. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–14, 2019.

[107] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.

[108] Chen Yu, Hanlin Tang, Cedric Renggli, Simon Kassing, Ankit Singla, Dan Alistarh, Ce Zhang, and Ji Liu. Distributed learning over unreliable networks. In *International Conference on Machine Learning*, pages 7202–7212. PMLR, 2019.

[109] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[110] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. Detail: reducing the flow completion time tail in datacenter networks. In *SIGCOMM*. ACM, 2012.

[111] Chiyuan Zhang, Samy Bengio, and Yoram Singer. Are all layers created equal? *arXiv preprint arXiv:1902.01996*, 2019.

[112] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 181–193, 2017.

[113] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. Resilient datacenter load balancing in the wild. In *SIGCOMM*. ACM, 2017.

[114] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. Is network the bottleneck of distributed training? In *Proceedings of the Workshop on Network Meets AI & ML*, pages 8–13, 2020.

[115] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

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

Appendix

A Convergence Analysis of MLT

This section formally presents a convergence analysis for MLT, essentially a distributed machine learning process with priority/selective dropping on gradients. Note that prior work [108] has already proven the comparable convergence rate of distributed learning with random dropping (i.e., unreliable network with independent and equivalent packet drop probability p for each message). Here, on top of [108], we further extend the convergence proof from random dropping to priority/selective dropping. The notations used here are shown in Table 3.

$\|\cdot\|$	l_2 norm for vectors
$\|\cdot\|_F$	the Frobenius norm of matrices
n	number of workers
m	number of servers
γ	model learning rate
p	packet dropping ratio

Table 3: Definitions and notations

The distributed optimization problem is defined as:

$$\min_{\vec{x}} f(\vec{x}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\vec{x}), \tag{1}$$

where n is the number of workers, $f_i(\vec{x}) = \mathbb{E}_{\xi\, D_i} F_i(\vec{x}, \xi)$ represents the expected loss function F over D_i, the local data distribution of worker i. At each iteration, every worker performs SGD on a random chosen subset of dataset $D_t^{(i)}$:

$$G_t^{(i)} = \nabla F_i\left(X_t^{(i)}, D_t^{(i)}\right).$$

$X_t^{(i)}$, $G_t^{(i)}$ and $D_t^{(i)}$ denotes the model weights, generated gradients and training data of worker i at iteration t respectively.

Before sending the gradients, every worker i divides the gradients into m equal blocks:

$$G_t^{(i)} = \left((G_t^{(i,1)})^\intercal, (G_t^{(i,2)})^\intercal, \ldots, (G_t^{(i,m)})^\intercal\right).$$

When sending gradients $G_t^{(i)}$, some blocks may be dropped because of the networking condition and priority dropping. For each blocks, the gradients on every workers are collected and averaged by parameter server:

$$\widetilde{G}_t^j = \frac{1}{|N_t^{(j)}|}\sum_{i \in N_t^{(j)}} G_t^{(i,j)},$$

where $\widetilde{G}_t^j$ denotes the averaged gradients of block j at iteration t, and $N_t^{(j)}$ denotes the number of workers whose blocks j are successfully averaged at iteration t.

After averaging gradients, the parameter server updates the corresponding weight block using SGD algorithm and returns them back to each workers for their local updates. For workers that fail to receive the averaged block, they just use the original gradients. Formally, the updated gradients on worker i is:

$$X_{t+1}^{(i)} = \left((X_{t+1}^{(i,1)})^\intercal, (X_{t+1}^{(i,2)})^\intercal, \ldots, (X_{t+1}^{(i,m)})^\intercal\right),$$

where

$$X_{t+1}^{(i,j)} = \begin{cases} X_t^{(i,j)} - \gamma \widetilde{G}_t^j, & i \in \widetilde{N}_t^{(j)} \\ X_t^{(i,j)} - \gamma G_t^{(i,j)}, & i \notin \widetilde{N}_t^{(j)}. \end{cases}$$

$\widetilde{N}_t^{(j)}$ denotes the set of workers to which the averaged block j is successfully sent at iteration t.

For the algorithm, we make the following assumptions commonly used for analyzing stochastic optimization algorithms [64, 108].

Assumption 1. *We make the following commonly used assumptions:*

1. ***Lipschitzian gradient**: The gradient function $\nabla f_i(\cdot)$ is L-Lipschitian, which means*

$$\|\nabla f_i(\vec{x}) - \nabla f_i(\vec{y})\| \leq L\|\vec{x}-\vec{y}\|$$

2. ***Bounded gradient**: The variance of stochastic gradient is bounded for every worker i and any $\vec{x}$.*

$$\mathbb{E}_{\xi\, D_i}\|\nabla F_i(\vec{x};\xi) - \nabla f_i(\vec{x})\|^2 \leq \sigma^2, \forall i, \forall \vec{x}$$
$$\frac{1}{n}\sum_{i=1}^{n}\|\nabla f_i(\vec{x}) - \nabla f(\vec{x})\|^2 \leq \xi^2, \forall i, \forall \vec{x},$$

3. ***Start from 0**: For simplicity, we assume $X_1 = 0$ w.l.o.g.*

With arbitrary packet dropping policy, the updated gradients on each worker can always be represented as the linear combination of local gradients.

$$X_{t+1}^{(i,j)} - X_t^{(i,j)} = G_t^{(\cdot,j)} W_t^{(j)},$$

where

$$G_t^{(\cdot,j)} := \left((G_t^{(1,j)})^\intercal, (G_t^{(2,j)})^\intercal, \ldots, (G_t^{(i,j)})^\intercal\right).$$

$W_t^{(j)}$ is the coefficient matrix. And $\left[W_t^{(j)}\right]_{m,k}$ denotes the coefficient of worker m's gradients received by worker k after one update step. $\left[W_t^{(j)}\right]_{m,k} = 0$ means worker m's gradient block j is not received by worker k, which may be dropped either before or after the averaging during the communication with the parameter server.

[108] shows $W_t^{(j)}$ satisfies the following properties under uniformly random dropping environment:

$$\mathbb{E}[W] = \alpha_1 I_n + (1-\alpha_1) A_n \tag{2}$$
$$\mathbb{E}[W_t^{(j)} W_t^{(j)\intercal}] \alpha_1 I_n + (1-\alpha_1) A_n \tag{3}$$
$$\mathbb{E}[W_t^{(j)} A_n W_t^{(j)\intercal}] = \alpha_2 I_n + (1-\alpha_2) A_n \tag{4}$$

for some constants α_1 and α_2 satisfying $0 < \alpha_2 < \alpha_1 < 1$. While [108] considers the algorithm where workers perform the averaging operation, the properties also hold for dedicated parameter server setting. Also, as MLT adopts priority dropping mechanism, $(\alpha_1^{(j,t)}, \alpha_2^{(j,t)})$ varies in different blocks j and iterations t. To adopt the convergence proof in [108] for MLT, we use $\alpha_{1_{max}}, \alpha_{2_{max}}$ instead, which denotes the maximum value of $\max_{j,t} \alpha_1^{(j,t)}$ and $\max_{j,t} \alpha_2^{(j,t)}$ across all workers and iterations and preserve the validity of the proof. Thus we can get the following theorem:

Theorem 1. *(Convergence of MLT). Under Assumption 1, choosing learning rate γ to be small enough satisfying $1 - \frac{6L^2\gamma^2}{(1-\sqrt{\beta_{max}})^2} > 0$, MLT have the following convergence rate:*

$$\begin{aligned}
&\frac{1}{T}\sum_{t=1}^{T}\left(\mathbb{E}\|\nabla f(\overline{\vec{x}_t})\|^2 + (1-L\gamma)\mathbb{E}\|\overline{\nabla} f(X_t)\|^2\right) \\
&\leq \frac{2f(\vec{0}) - 2f(\vec{x}^*)}{\gamma T} + \frac{\gamma L \sigma^2}{n} + 4\alpha_{2_{max}} L\gamma(\sigma^2 + 3\xi^2) \\
&+ \frac{2\alpha_{2_{max}} L\gamma + L^2\gamma^2 + 12\alpha_{2_{max}} L^3\gamma^3)\sigma^2 C_1}{(1-\sqrt{\beta_{max}})^2} \\
&+ \frac{3(2\alpha_{2_{max}} L\gamma + L^2\gamma^2 + 12\alpha_{2_{max}} L^3\gamma^3)\xi^2 C_1}{(1-\sqrt{\beta_{max}})^2},
\end{aligned} \tag{5}$$

where

$$\nabla f(\overline{\vec{x}_t}) = \nabla f\left(\frac{1}{n}\sum_{i=1}^{n}\vec{x}_t^{(i)}\right)$$
$$\overline{\nabla} f(X_t) = \sum_{i=1}^{n}\nabla f_i(\vec{x}_t^{(i)})$$
$$\beta_{max} = \max_{j,t}(\alpha_1^{(j,t)} - \alpha_2^{(j,t)})$$
$$C_1 = \left(1 - \frac{6L^2\gamma^2}{(1-\sqrt{\beta_{max}})^2}\right)^{-1}.$$

It can be inferred from the definitions that $\beta = 1$ if and only if the dropping probability of the gradient block is 1, which may cause the bound to be infinity. In MLT we can make the assumption that no gradient block has dropping probability equal to 1, since the magnitude of gradients varies among different iterations.

By choosing appropriate learning rate $\gamma = \frac{(1-\sqrt{\beta_{max}})^2}{6L + 3(\sigma+\xi)\sqrt{\alpha_{2_{max}} T} + \frac{\sigma\sqrt{T}}{\sqrt{n}}}$, we can get

$$\begin{aligned}
\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}\|\nabla f(\overline{\vec{x}_t})\|^2 &\leq \frac{(2f(\vec{0}) - 2f(\vec{x}^*) + L)\sigma}{\sqrt{nT}(1-\sqrt{\beta_{max}})} \\
&+ \frac{(2f(\vec{0}) - 2f(\vec{x}^*) + L)(\sigma+\xi)}{1-\sqrt{\beta_{max}}}\sqrt{\frac{\alpha_{2_{max}}}{T}} \\
&+ \frac{L^2(\sigma^2+\xi^2)}{(\frac{T}{n} + \alpha_{2_{max}} T)\sigma^2 + \alpha_{2_{max}} T\xi^2} \\
&+ \frac{(2f(\vec{0}) - 2f(\vec{x}^*)L}{T}
\end{aligned} \tag{6}$$

We can see from Equation 6 that the dominant term in the convergence rate ($O(1/\sqrt{nT})$) is consistent with prior works for both centralized SGD and decentralized SGD [65, 108], which theoretically prove that MLT will converge with the same order of iterations as the previous vanilla SGD methods.

B Thresholds Setting of Selective Dropping

This section presents the formulation to derive the optimal thresholds for the selective dropping mechanism (RED/ECN setting) in §4.2.2 by leveraging the queueing theory [16]. As a guidance from the analysis, to find the thresholds, we need to know the size of each layer of the model, and measure the additional rounds to model convergence caused by the loss of small/large gradients in each layer. Note that our analysis does not yet provide optimal thresholds for specific models, we leave the additional rounds measurement and optimal thresholds calculation as the future work. The notations used here are shown in Table 4.

N	number of switch queues
B	size of switch buffer
S_i	ECN/RED threshold of queue i
L_i	length of queue i
λ	packet arrival rate
μ	packer service rate
M	number of model layers
S_i^m	size of layer i in the model
θ	ratio of small gradients in the model
$f_i^S(.)/f_i^L(.)$	additional convergence rounds cost by 1% small/large gradient loss in layer i

Table 4: Definitions and notations

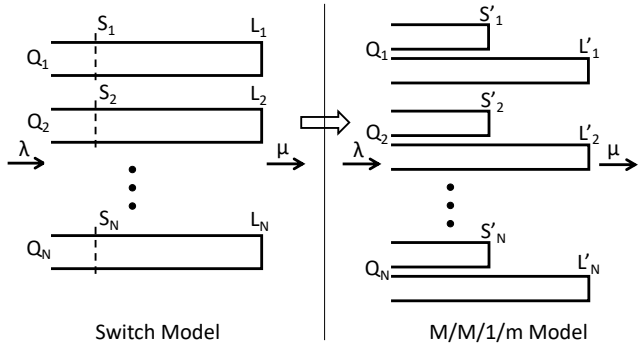


Figure 20: Problem formulation

Problem formulation: Figure 20 shows the mathematical modeling of MLT switch. All flows come with the arrival rate of λ, and enter the corresponding queues. For each queue, if the sum of queueing packets' size is larger than S_i, all small gradients packets come to this queue will be discarded. If the sum is larger than L_I, all packets will be discard. To simplify the analysis and take the advantage of M/M/1/m Model in queueing theory, we split each queue into two, the first is for small gradients only and the second is for large gradients only. The dropping thresholds for each one are S_i' and L_i'. We can easily represent S_i and L_i with S_i' and L_i': $S_i = S_i'/\theta, L_i = S_i' + L_i'$.

Then, we deduce the value of S_i' and L_i'. The arrival rate for one queue depends on the corresponding layers' packet arrival rate, for simplicity, we assume the rate is proportional to the size of the layer. Therefore, the arrival rate for queue i is $\frac{\lambda S_i^m}{S}$, and for the small gradients' queue, the value is $\theta\frac{\lambda S_i^m}{S}$, for the large one, is $(1-\theta)\frac{\lambda S_i^m}{S}$.

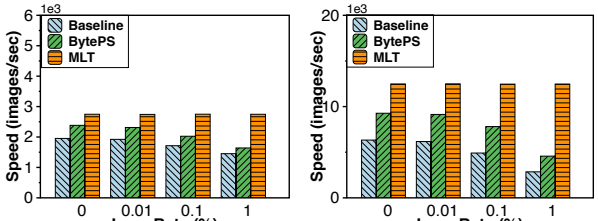
The service rate for one queue is determined on its priority, it is serviced only when the higher priority queues are idle, for the highest priority queue Q_1, the service rate is $\mu_1 = \mu$, the idle time is $1-\rho_1$, where $\rho_1 = \lambda/\mu$, for queue Q_2, the service rate is $\mu_2 = (1-\rho_1)\mu$. Generally, the idle time for queue Q_i is $1-\rho_i$, where $\rho_i = \lambda_i/\mu_i$ and the service rate is $\mu_i = \Pi_{k=0}^{i-1}(1-\rho_k)\mu$.

Supposed the service rate for small/large gradients is proportional to the size, thus, the service rate for small/large gradients queue in queue Q_i are $\mu_i^S = \theta\mu_i$ and $\mu_i^L = (1-\theta)\mu_i$ respectively. Therefore the idle time are $\rho_i^S = \lambda_i^S/\mu_i^S = \lambda_i/\mu_i = \rho_i = \rho_i^L$.

Suppose the losses of small and large gradients in each layer affect the convergence rounds independently, our goal is to minimize the impact of gradients' loss to model convergence, that is to find the optimal S_i and L_i to minimize the loss function $\sum_{i=1}^{N}\left(f_i^S(r_i^S) + f_i^L(r_i^L)\right)$, where r_i^S and r_i^L are the loss ratio for small/large gradients in queue Q_i. There are a lot of existing solutions to solve the optimization problem, e.g. gradient descent. Here, we only need to calculate the value of r_i^S and r_i^L. In fact, for each queue, we can treat it as a typical M/M/1/m model in queueing theory, specially, one FIFO queue with finite capacity. Previous work [16] has derived the calculation formula of loss rate, that is $\frac{\rho^m - \rho^{m+1}}{1-\rho^{m+1}}$, where ρ is the idle time of the queue. Therefore, we have: $r_i^S = [(\rho_i)^{\theta S_i} - (\theta\rho_i)^{S_i+1}]/[1-(\rho_i)^{\theta S_i+1}]$, $r_i^L = [(\rho_i)^{L_i-\theta S_i} - (\rho_i)^{L_i-\theta S_i+1}]/[1-(\rho_i)^{L_i-\theta S_i+1}]$, then we express the loss function in terms of known parameters and thresholds (S_i, L_i), after solving this optimization problem, we obtain the optimal thresholds.

C Supplemental Experiments

Due to the space limitation, in the main part of the paper, we only show the results of VGG16 and Transformer in Figure 11, 12, 16, 17 and 18. Here, we append the results of ResNet50 and GoogleNet in Figure 21, 22, 23, 24, 25 and 26. Please note that the results/trends embodied in these figures are consistent with that in the main part of the paper.



(a) ResNet50 (b) GoogleNet

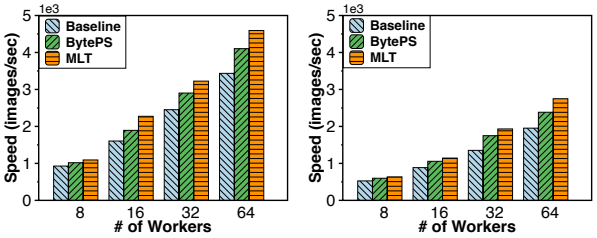
Figure 21: Speedup under gray failure of links.

D Open Questions and Discussion (Cont'd)

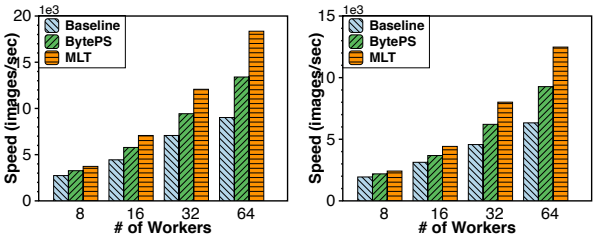
Co-existing with non-DNN traffic. This paper focuses on AI-centric networking (AICN) dedicated to DNN training. However, in datacenter with multi-purpose traffic, our implementation of MLT should mitigate its intrusion on non-DNN

Model	GoogleNet [93] (13%)	LSTM [42] (11%)	ResNet34 [41] (14%)	Wide ResNet50 [109] (17%)
	AlexNet [57](18%)	VGG16 [91] (17%)	GRU [26] (12%)	ResNet50 [41] (16%)
	EfficientNetB0 [94] (17%)	VGG19 [91] (18%)	Wide ResNet101 [109] (16%)	ShuffleNetV2 [71] (16%)
	VGG13 [91] (16%)	ResNet18 [41] (18%)	DenseNet169 [47] (16%)	ResNet101 [41] (17%)

Table 5: The loss-tolerant bounds measurement with MLT, under the condition that the models converge with the same iterations to the same accuracy. We use the same models and settings with Table 1.

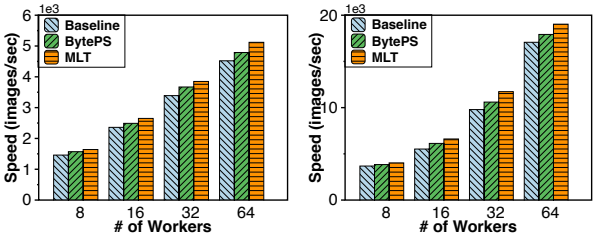


(a) MXNet, ResNet50 (b) PyTorch, ResNet50



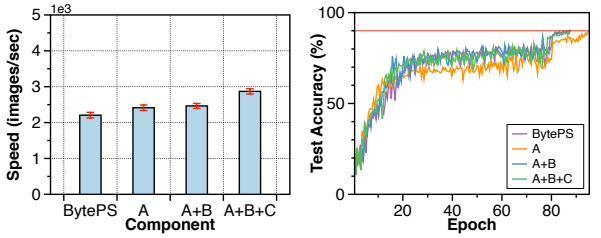
(c) MXNet, GoogleNet (d) PyTorch, GoogleNet

Figure 22: Speedup with MXNet and PyTorch.

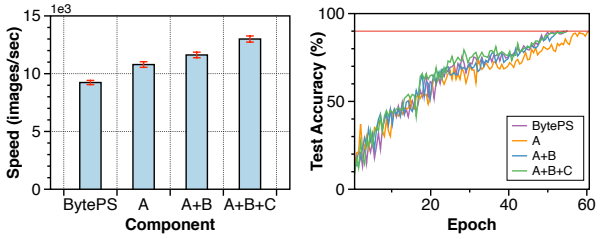


(a) Ring All-reduce, ResNet50 (b) Ring All-reduce, GoogleNet

Figure 23: Speedup with Ring All-reduce

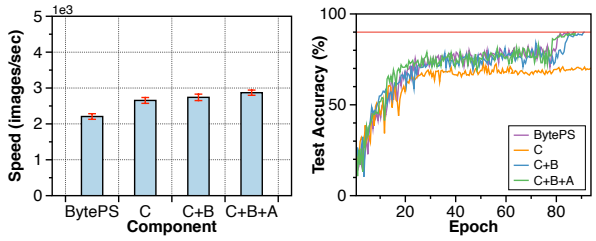


(a) ResNet50, Speed (b) ResNet50, ETA

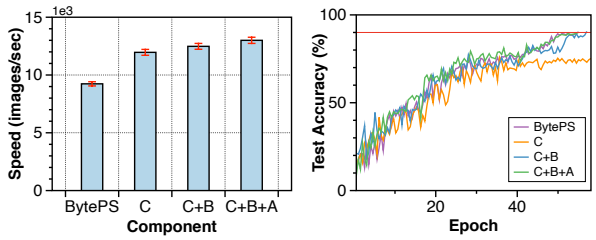


(c) GoogleNet, Speed (d) GoogleNet, ETA

Figure 24: Effectiveness of design components (I).

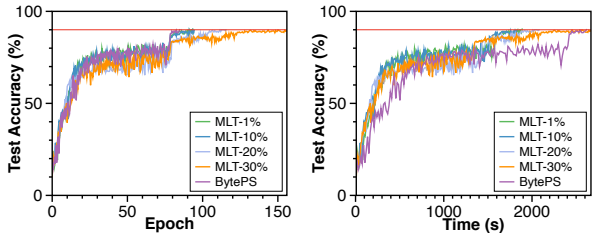


(a) ResNet50, Speed (b) ResNet50, ETA

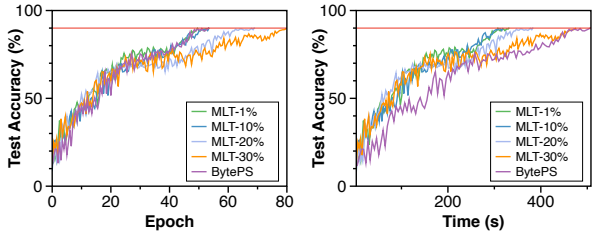


(c) GoogleNet, Speed (d) GoogleNet, ETA

Figure 25: Effectiveness of design components (II).



(a) ResNet50, ETA (b) ResNet50, TTA



(c) GoogleNet, ETA (d) GoogleNet, TTA

Figure 26: Impact of loss-tolerant bounds.

traffic and ensure bandwidth fair-sharing. A straightforward idea is to separate DNN and non-DNN traffic into different queues and perform fair-queueing between them. One issue to consider is that current commodity switches only have a limited # of queues (typically 8), so the # of queues dedicated for MLT become even smaller. For priority queueing, it is not a concern because the recent design [68] of using two priority queues to emulate many fine-grained priorities can be borrowed: the insight is that we only need to privilege the highest priority flows; MLT can leverage such idea and

only place the foremost layers into the highest priority queue each time. On the other hand, it may reduce the granularity of selective dropping of gradients across DNN layers, as this relates to the # of physical queues available to MLT.

Flexible selective dropping. We consider the gradients of back layers more *important* under the assumption of pre-training (§3.3). As researchers in ML area are improving the model architectures and training algorithms, the loss tolerance across model parameters may vary in the future. MLT allows a flexible selective dropping policy through tagging. One can simply modify the gradient tagging policy at end-hosts to fit new features in the training process.

MLT vs SRD. AWS SRD [89] is a hardware transport protocol that also uses packet spraying to avoid hotpots. It provides reliable but out-of-order delivery. To be used by general-purpose applications, SRD still relies on a messaging layer above it to restore orders. In contrast, MLT exploits ML-specific properties to provide semi-reliable delivery without the need of packet order restoration.

Swing: Short-cutting Rings for Higher Bandwidth Allreduce

Daniele De Sensi
Sapienza University of Rome

Tommaso Bonato
ETH Zurich

David Saam
RWTH Aachen University

Torsten Hoefler
ETH Zurich

Abstract

The allreduce collective operation accounts for a significant fraction of the runtime of workloads running on distributed systems. One factor determining its performance is the number of hops between communicating nodes, especially on networks like torus, where a higher number of hops implies multiple messages being forwarded on the same link, thus reducing the allreduce bandwidth. Torus networks are widely used on systems optimized for machine learning workloads (e.g., Google TPUs and Amazon Trainium devices), as well as on some of the Top500 supercomputers. To improve allreduce performance on torus networks we introduce *Swing*, a new algorithm that reduces the number of hops between communicating nodes by *swinging* between torus directions. Our analysis and experimental evaluation show that Swing outperforms by up to 3x existing allreduce algorithms for vectors ranging from 32B to 128MiB, on different types of torus and torus-like topologies, regardless of their shape and size.

1 Introduction and Motivation

Allreduce is a collective operation used to aggregate vectors among a set of nodes and to distribute the aggregated result back to them. Allreduce is widely used to perform distributed gradient aggregation when training deep learning models [10]. Studies have shown that it can account for up to 40% of the total training time [32, 45, 51] and between 19% and 30% of the total core hours in MPI jobs running on production supercomputers [14].

In this work, we design a new allreduce algorithm optimized for torus-like networks. Torus networks are widely used, both on systems optimized for running machine learning (ML) workloads and on some of the top supercomputers [2, 11] (e.g., Fugaku uses a 6D torus [4]). Although torus networks are characterized by a lower bisection and global bandwidth compared to other topologies (e.g., Clos), their simplicity and lower cost allow running some workloads such as ML training in a more cost-effective way, since their communication is often arranged as a 3D logical torus [26].

Seen from a different perspective, torus networks trade off a lower cost for a lower bisection bandwidth, which, however, is enough to train most ML models efficiently [26]. This is the reason why many systems optimized for ML training rely on torus-like topologies. These include, for example, Google TPUs [31] (a TPU v5p pod connects $\sim$ 9,000 chips on a 3D torus [15]), Amazon Trainium devices [7] (16 chip on a 2D torus), Graphcore IPU-POD [24] (connecting 64 chips on a 2D torus), Enflame [40] (2D torus).

Researchers proposed several allreduce algorithms [6, 29, 50], and the most performing one depends on a combination of vector size, number of nodes, and physical topology [27,49]. Those algorithms perform a predefined number of steps and, at each step, each node sends and receives data to and from some predetermined nodes. Different trade-offs exist between the number of steps to perform (more critical for allreduce on small vectors) and the total number of bytes it transmits (more relevant for larger allreduce). However, a third factor that must be considered when designing a new collective algorithm is the number of hops between communicating nodes [38,44,49,50]. This is particularly relevant on networks that do not provide full bisection bandwidth such as torus, since the higher the number of hops, the higher the number of flows sharing the same links.

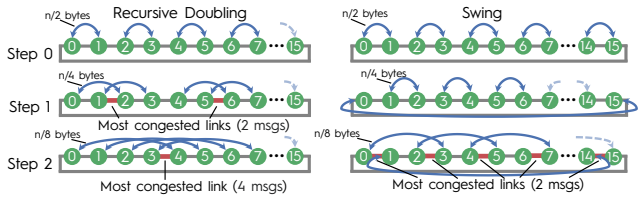


Figure 1: First three steps of the recursive doubling and Swing allreduce algorithms on a 1D torus with 16 nodes.

We show the importance of the number of hops in the allreduce through an example. In Fig. 1, we show a 16 node 1D torus (we only show a subset of the nodes since the commu-

nications are symmetric). We assume minimal (i.e., shortest path) routing and we show the communications performed by the bandwidth-optimal recursive doubling algorithm [50] (also known as *Rabenseifner* algorithm [41], which we describe more in detail in Sec. 2.3.3), and by the *Swing* algorithm (that we propose in this work).

Both algorithms perform the same number of steps (we show only the first three for simplicity). We denote with n the number of bytes of the allreduce vector. In the first step, in both algorithms, each node r sends $n/2$ bytes to node $q = r$ XOR 1 (and receives $n/2$ bytes from it). In the second step, however, in the recursive doubling, each node r sends $n/4$ bytes to node $q = r$ XOR 2 (two hops distant), whereas in the Swing algorithm, each node still sends $n/4$ bytes of data, but with the other neighbor (one hop distant).

Although both algorithms transmit the same number of bytes, two different messages cross the same link in the recursive doubling. For example, two messages cross the link between nodes 1 and 2 and that between nodes 5 and 6. As a consequence, in the worst case all nodes *transmit data at most at half the bandwidth* of the link between 1 and 2, thus slowing down the entire allreduce operation. Instead, in the Swing algorithm, each node can still transmit at full bandwidth because, in this example, in the second step each link is crossed at most by one message per direction. Something similar also happens in the third step. Indeed, when using Swing at most two messages cross each link instead of the four messages crossing the link between nodes 3 and 4 in recursive doubling.

It is thus clear how even if two different algorithms transmit the same number of bytes and perform the same number of steps, they might have different performance, depending on the network characteristics and the distance between communicating nodes. In this example, we have shown an extreme case using a 1D torus. However, similar effects can happen *on any topology that does not provide full bisection bandwidth*.

Although some algorithms (i.e., ring [38,50] and bucket [9, 29,44]) avoid this problem by having each node communicate with its neighbors only, they perform more steps (linear in the number of nodes) and are thus not well-suited for small- and medium-sized vectors. Those are the sizes that, however, are practically used in most machine learning [35] and HPC [14] workloads. Indeed, larger allreduce are split into smaller ones to overlap better computation and communication, especially more when using 3D parallelism in machine learning training [10].

This work makes the following contributions:

- We design a new allreduce algorithm called *Swing*, which performs a logarithmic number of steps and transmits the minimal number of bytes while reducing the distance between communicating nodes compared to other known algorithms designed for small- and medium-sized vectors (Sec. 3 and Sec. 4).
- We evaluate Swing on different torus and torus-like topologies (e.g., HammingMesh [26] and HyperX [3,20]), by comparing it with the best state-of-the-art algorithms (Sec. 5). Our evaluation shows that *Swing outperforms the other existing algorithms for allreduce on vectors ranging from 32B to 128MiB on different torus-like topologies, and regardless of their shape and size*. We show that Swing outperforms the best-known algorithm up to 2.2x on square torus with 4,096 to 16,384 nodes and up to 3x on rectangular tori and HyperX with 4,096 nodes.

2 Background

2.1 Targeted Collectives

We briefly introduce the *reduce-scatter* and *allgather* collectives since for medium and large vectors, Swing allreduce algorithm executes a reduce-scatter followed by an allgather (similarly to the Rabenseifner algorithm [41]). In the reduce-scatter, the compute nodes reduce vectors (one per node) using a reduction operation (e.g., addition) and shard the resulting vector across all the nodes. In the allgather, each node provides a vector that they concatenate and distribute to all the nodes.

If the vector contains a number of elements larger or equal than p, the nodes can run the allreduce as a reduce-scatter followed by an allgather [50]. I.e., they aggregate vectors coming from all the nodes and distribute back the resulting vector. Although for space reasons we mainly target the allreduce, Swing can also be used for performing reduce-scatter and allgather collectives, as well as any other collective operation where recursive doubling or binomial tree can be used (e.g., broadcast and reduce) [49].

2.2 Notation and Model

We consider D-dimensional tori of size $\{d_0, d_1, \ldots, d_{D-1}\}$, and we denote with p the number of nodes in the torus, i.e., $p = d_0 \cdot d_1 \cdot \ldots \cdot d_{D-1}$. We assume the collectives run on p nodes, that ranks are mapped to nodes linearly and, without loss of generality, to have one process (or *rank*) per node. We assume that each node in the network has $2 \cdot D$ ports, that each link is bidirectional, and that each node can send $2 \cdot D$ messages and receive $2 \cdot D$ messages concurrently (one send and receive per port). We also assume the network forward packets using minimal adaptive routing, and that does not have any hardware support to accelerate collective operations (e.g., in-network aggregation [6, 18, 23]).

To support the description of the different algorithms, we model their performance with the commonly used latency and bandwidth model [5, 16, 39]. We model the communication time $T(n)$ to send n bytes in a point-to-point communication as $T(n) = \alpha + n\beta$, where α represents the *latency* (i.e., the time for the first byte to reach the destination), and β the time

Name	Description
D	Number of torus dimensions
$d_0,\ldots,d_{D-1}$	Size of each dimension
n	Size of the vector to reduce
p	Number of nodes in the network
Λ	Latency deficiency
Ψ	Bandwidth deficiency
Ξ	Congestion deficiency
$\delta(s)$	Number of hops between 2 communicating nodes at step s
$\rho(s)$	$\sum_{i=0}^{s}-2^i$
$\pi(r,s)$	The node with which node r communicates at step s

Table 1: Variables and functions used in our modeling.

to transmit a single byte (it can be seen as the inverse of the *bandwidth*). When modeling collective operations involving data reduction, researchers also consider an additional γ term to model the aggregation cost. To avoid burdening the notation, we do not model this term explicitly since Swing is no worst than the other algorithms in that regard and most implementations overlap the aggregation with the communication [37, 52].

Collective operations involve multiple communication steps. Previous works proved that the allreduce requires at least $\log_2 p$ steps and the transmission of at least $2\frac{p-1}{p}n \approx 2n$ bytes of data [9]. Hence, the optimal allreduce time can be modeled as $T(n) = \alpha\log_2 p + \beta 2n$. However, because each node has $2D$ ports, bandwidth-optimal algorithms distribute the data equally across all the ports [9, 26, 29, 44], and we can model the allreduce time as $T(n) = \alpha\log_2 p + \beta\frac{n}{D}$ In practice, however, algorithms have some *deficiency* compared to optimal, either in the latency or the bandwidth terms (or in both).

We consider three different type of deficiencies: i) *latency deficiency* (Λ) i.e., how much the latency is higher than the optimal; ii) *algorithmic bandwidth deficiency* (Ψ), i.e., how many more bytes does the algorithm transmit; iii) *congestion bandwidth deficiency* (Ξ) i.e., what is the bandwidth slowdown caused by multiple messages of the same collective being forwarded on the same link (as discussed in Sec. 1). We can see deficiencies as multiplicative factors that denote how much an algorithm is distant from the optimal (e.g., a latency deficiency of one means that the algorithm is latency optimal), and we can thus model the allreduce time as:

$$T(n) = \log_2 p \cdot \alpha \cdot \Lambda + \frac{n}{D}\beta \cdot \Psi \cdot \Xi \qquad (1)$$

For brevity, we refer to Ψ as *bandwidth deficiency* and to Ξ as *congestion deficiency*. While Λ and Ψ only depend on the algorithm, the congestion deficiency Ξ depends on the network. To simplify the notation and the discussion, in the following, we are only going to model the three deficiencies Λ, Ψ, and Ξ. We summarize the variables we use in our modeling in Table 1.

2.3 State-of-the Art Algorithms

In the following, we review and model the main allreduce algorithms for multidimensional torus, and we summarize in Table 2 their deficiencies, as well as those of the Swing algorithm.

2.3.1 Hamiltonian Rings

Ring allreduce algorithm [38, 50] runs a reduce-scatter followed by an allgather. Each node splits its data into p equally sized blocks. For both the reduce-scatter and the allgather, the algorithm performs $p-1$ steps. Nodes are arranged in a ring, and at each step, each node sends a block to its right neighbor and receives a block from its left neighbor. Because the algorithm performs $2(p-1) \approx 2p$ the latency deficiency is $\Lambda = \frac{2p}{\log_2 p}$.

The algorithm sends $\approx 2n$ bytes (n in the reduce-scatter and n in the allgather). On multiport networks (assuming $2D$ ports), it splits the data into $2D$ parts (of $n/2D$ bytes each) and runs one ring algorithm on each part (each sending and receiving to and from a different port). Since it sends a minimal number of bytes and uses all the ports, the algorithm has no bandwidth deficiency ($\Psi = 1$).

Moreover, because each node only communicates with neighbors on a 1D torus, the algorithm does not have congestion deficiency since each link is used by at most one communication in each direction. In the version optimized for the 2D torus, the four rings that run in parallel are mapped into two edge-disjoint Hamiltonian cycles (one per direction) [26] so that each link is still used by at most one communication per direction (thus $\Xi = 1$). To our knowledge, this algorithm does not work for $D > 2$. Moreover, the algorithm can build the two edge-disjoint Hamiltonian cycles on an $r \times c$ 2D torus only if $r = c \cdot k,\ k \geq 1$ and the greatest common divisor between r and $c-1$ is 1, which limits the applicability of the algorithm.

2.3.2 Latency-Optimal Recursive Doubling

The latency-optimal recursive doubling algorithm [50] executes $\log_2 p$ steps (thus it has a latency deficiency $\Lambda = 1$). At each step s (we denote steps starting from 0), each node r sends its vector to node $q = r$ XOR 2^s (assuming p is a power of 2) and receives q's vector, which aggregates with its own before moving to the next step. When running on a torus, if the size of each dimension is a power of two, it can keep a shorter distance between communicating nodes by communicating in a different dimension at each step, as shown in Fig. 2.

Each node transmits $n\log_2 p$ bytes of data. To our knowledge, no multiport versions of this algorithm exist, and we model its bandwidth deficiency as $\Psi = D\log_2 p$. Each node communicates with D nodes at distance 2^i (one per dimension), with $0 \leq i < \frac{\log_2 p - 1}{D}$. For this algorithm, the number

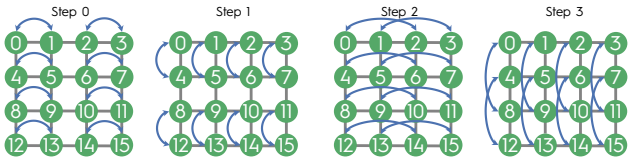


Figure 2: Example of recursive doubling on a 4*x*4 torus. Wrap-around links are not shown.

ALGORITHM	LAT. DEF. (Λ)	BAND. DEF. (Ψ)	CONG. DEF. (Ξ) D=2	D=3	D=4
RING	$\frac{2p}{\log_2 p}$	1	1		
REC.DOUB. (L)	1	$D\log_2 p$	$2D\sqrt[D]{p}$		
REC.DOUB. (B)	2	$2D$	$\frac{2^D-1}{2^D-2}$		
BUCKET	$\frac{2D\sqrt[D]{p}}{\log_2 p}$	1	1		
SWING (L)	1	$D\log_2 p$	$\frac{4}{3}D\sqrt[D]{p}$		
SWING (B)	2	1	1.19	1.03	1.008

Table 2: Algorithms deficiencies on D-dimensional torus with $D \geq 2$. (L) and (B) stands for latency-optimal and bandwidth-optimal (*-optimized* for recursive doubling) respectively.

of messages forwarded on the most congested link is equal to the distance between communicating nodes[1]. We can thus estimate the congestion deficiency as the sum of the distance between communicating nodes over all the steps, i.e., $\Xi = D\sum_{i=0}^{\frac{\log_2 p-1}{D}} 2^i \leq 2D\sqrt[D]{p}$.

Non-power-of-two If the size of a dimension is not a power of two, some extra steps are needed. One possible solution consists in reducing the number of nodes to the largest power-of-two $p' < p$ [8, 12]. Before starting the allreduce, each node in the range $(p', p-1)$ sends its data to a node in the range $(0, p'-1)$. Then, the first p' nodes run the allreduce and, when completed, send the reduced data to the nodes in the range $(p', p-1)$. This algorithm increases all the deficiencies compared to the case where p is a power of two, but we do not model this explicitly for brevity.

2.3.3 Bandwidth-Optimal Recursive Doubling

The classic bandwidth-optimal recursive doubling algorithm (also known as Rabenseifner algorithm [50]) performs an allreduce as a reduce-scatter followed by an allgather. The reduce-scatter and the allgather use recursive doubling (each performing $\log_2 p$ steps). Differently from the latency-optimal algorithm, each node divides the data into p blocks $\{b_0, \ldots, b_{p-1}\}$, each of size $\frac{n}{p}$.

[1]The only exception to this is the last step in each dimension because each node can reach its peer with two different minimal paths. For example, in Fig. 2, in step 2 node 0 can send half of the packets directed to 2 to the right and half to the left on the wrap-around link (not shown). However, this is negligible for large enough networks.

At each step, the reduce-scatter halves the size of the transmitted data and doubles the distance between communicating nodes. Thus, each node transmits n bytes of data in the reduce-scatter. The allgather works similarly but reverses the communication pattern, doubling the size of the transmitted data at each step and halving the distance between communicating nodes. The allreduce executes $2\log_2 p$ steps ($\Lambda = 2$), and transmits $2n$ bytes of data.

Sack et al. optimized the algorithm for torus networks [44], similarly to what we described for the latency-optimal algorithm, to reduce its congestion deficiency to $\Xi = \frac{2^D-1}{2^D-2}$ for $D > 1$ [44]. However, to our knowledge, no multiport versions of this algorithm exist, and its bandwidth deficiency is $\Psi = 2D$. Hence, for torus networks we consider this algorithm as *bandwidth-optimized* rather than *bandwidth-optimal*.

Non-power-of-2 If the number of nodes p is not a power of 2, the algorithm performs some extra steps [42, 50] that increase the latency and reduce the bandwidth (because transmit extra data). Allreduce implementations can use either similar techniques like those described for the latency-optimal recursive doubling or more sophisticated ones like the *3-2 elimination* technique [42] (which increases the bandwidth deficiency to $3/2$). However, we found no reference of adaptations to torus networks.

2.3.4 Bucket Algorithm

To simplify the exposition, we first describe the algorithm for a $a \times a$ 2D torus (with $a \cdot a = p$) [9]. Each node runs a ring reduce-scatter with the other $a-1$ nodes on the same row. This requires $a-1$ steps, and each node transfers $n\frac{a-1}{a}$ bytes. Then a ring reduce-scatter with the other $a-1$ nodes on the same column, but only on the data already reduced at the previous step (of size $\frac{n}{a}$). Then, each node runs an allgather with all the nodes on the same column and then with all those on the same row.

On a $D-$dimensional torus, the algorithm performs D reduce-scatter followed by D allgather (each on $\sqrt[D]{p}$ nodes on a square torus). Because it runs $2D\sqrt[D]{p}$ steps, the latency deficiency is $\Lambda = \frac{2D\sqrt[D]{p}}{\log_2 p}$. To use all the $2 \cdot D$ ports, the algorithm splits the data into $2 \cdot D$ parts and concurrently runs $2 \cdot D$ bucket algorithms (one for each part) [29, 44]. Since the algorithm sends the minimal number of bytes and uses all the ports evenly, the bandwidth deficiency is $\Psi = 1$. Each of the $2 \cdot D$ bucket algorithms starts from a different dimension and direction so that, at each step, each link is used by at most one ring per direction (i.e., the congestion deficiency is $\Xi = 1$).

2.3.5 Other Approaches

Topology-Specific Algorithms Researchers proposed several allreduce algorithms [9, 30, 33, 41, 43, 49], some of which optimized for specific topologies [6, 22, 36]. In this work, we

focus on those explicitly designed for torus networks, since they are characterized by a lower congestion deficiency.

Automatic Generation of Collective Algorithms Some approaches use linear programming formulations for finding the best collective algorithm given a network specification and the size of the collective [13, 48]. However, this requires solving an NP-hard problem that grows exponentially with scale. Finding a solution for 128 nodes requires up to 11 hours [48], and a new solution might need to be found when changing the number of nodes or the size of the collective. This makes generating collective algorithms for large systems like the 9,000 nodes Google's TPU v5p pod [15] challenging or even impossible. On the contrary, Swing can seamlessly run on any number of nodes. Moreover, unlike Swing, some of these solutions do not explicitly model the congestion deficiency.

Topology Reconfiguration and In-Network Compute Other solutions improve allreduce performance by reconfiguring the network topology according to the specific traffic pattern [51]. Swing is orthogonal to these approaches and, by reducing the network congestion, can make the expensive network re-configurations less frequent. Last, some algorithms exploit in-network compute capabilities of programmable switches [17, 21, 34, 46] to aggregate data directly in the network, reducing network traffic and improving performance. However, unlike Swing, these solutions require specific switches to be deployed in the network, whereas Swing can seamlessly run on any network.

3 Swing Design

By analyzing the algorithms we described, we observe different tradeoffs. The latency-optimal recursive doubling has the lowest latency deficiency. It is thus more suited to small allreduce, where the number of steps executed by the algorithm, rather than the total number of transmitted bytes, dominates the runtime. On the other hand, ring and bucket algorithms are characterized by the lowest bandwidth and congestion deficiency, and we expect them to perform better on large allreduce. The bandwidth-optimized recursive doubling lies somewhere in between since it has a higher bandwidth and congestion deficiency but a lower latency deficiency and would perform better for medium-sized vectors.

With the Swing algorithm, we aim at designing an algorithm with a congestion deficiency Ξ lower than the bandwidth-optimized recursive doubling algorithm by reducing the distance between communicating nodes. We also aim to reduce bandwidth deficiency Ψ by using all the $2D$ available ports. To simplify the exposition, we first discuss the design of the bandwidth-optimal Swing algorithm on a 1D torus, assuming the number of nodes p is a power of 2 (Sec. 3.1). Then, we extend it to any number of nodes (Sec. 3.2), and describe its design for tors with more than one dimension (Sec. 4).

3.1 Algorithm Design

3.1.1 Bandwidth-optimal Algorithm

We describe in the following the design of the Swing algorithm, and we formally prove its correctness in Appendix A. The bandwidth-optimal Swing algorithm runs a reduce-scatter followed by an allreduce. In the reduce scatter, at step s (starting from 0), each node r communicates with a node $\pi(r,s)$ such that:

$$\pi(r,s) = \begin{cases} r+\rho(s) \bmod p, & \text{if } r \text{ is even} \\ r-\rho(s) \bmod p, & \text{if } r \text{ is odd} \end{cases} \tag{2}$$

Where $\rho(s) = \sum_{i=0}^{s}(-2)^i = \frac{1-(-2)^{s+1}}{3}$. This selection of the communicating peer leads to a communication pattern like the one shown in Figure 1 for a 16 nodes 1D torus. We observe how, at each step, the communicating peer of each node *swings* from left to right and vice versa (hence the algorithm's name). *Intuitively, unlike recursive doubling, each node reaches distant nodes in fewer hops by short-cutting the ring*.

More precisely, at each step, each node communicates with a node at a distance $\delta(s)$, with:

$$\delta(s) = |\rho(s)| = \left|\sum_{i=0}^{s} -2^i\right| = \frac{2^{s+1}-(-1)^{s+1}}{3} \leq$$
$$\leq \frac{2^{s+1}+1}{3} < 2^s + \frac{1}{3}$$

Because $\delta(s)$ is always a natural number, we have $\delta(s) \leq 2^s$ (it is strictly smaller for $s > 1$). Hence, Swing has a lower congestion deficiency than recursive doubling because of the lower distance between communicating nodes (we estimate precisely the congestion deficiency in Sec. 4).

For simplicity, we first describe the reduce-scatter algorithm using only one port and extend it to use all the $2D$ ports. In the reduce-scatter each node splits data into p blocks $\{b_0, b_1, \ldots, b_{p-1}\}$, each of size $\frac{n}{p}$. Each node r runs $\log_2 p$ steps, communicating at each step s with the node $\pi(r,s)$ and halving the size of the transmitted data. At the end of the reduce-scatter, we want each node r to have the fully aggregated block b_r.

To do so, data transmitted from r to q includes the block b_q, plus all the blocks that q will transmit to other nodes in the subsequent steps. The allgather works similarly, but each node selects its peer in the reverse order, thus communicating first with the more distant ones. In the first step, each node r sends its block b_r, doubling the transmitted data's size at each step (data transmitted from r to q includes all the blocks that r

gathered until step s). Because the algorithm performs $2\log_2 p$ steps, its latency deficiency is $\Lambda = \frac{\log_2 p}{2}$. Because it transmits the minimal number of bytes and uses all the ports (as we will show in Sec. 4), its bandwidth deficiency is $\Psi = 1$. We estimate the congestion deficiency in Sec. 4 when describing the algorithm for torus with more than one dimension.

We summarize the algorithm in Listing 1 for reduce-scatter (the algorithm for allgather is analogous). The function `get_rs_idxs` computes the indexes of the data blocks that a given node `r` must send at step `step`, and relies on the function `pi` we defined in Eq. 2. Then, the `reduce_scatter` function executes $\log_2 p$ steps, and at each step, computes the bitmaps `blocks_s` and `blocks_r` denoting the blocks of data that must be sent and received. Last, it sends and receives those blocks.

```
def get_rs_idxs(r, step, p, blocks):
  if step >= log2(p): return
  for s in range(step, int(log2(p))):
    peer = pi(r, s, p)
    # Set to 1 the node I directly reach
    blocks[peer] = 1
    # and those that it will reach
    get_rs_idxs(peer, s+1, p, blocks)

def reduce_scatter(r, p, data):
  for s in range(0, int(log2(p))):
    blocks_s = blocks_r = [0]*p
    dest = pi(r, s)
    get_rs_idxs(r, s, p, blocks_s)
    get_rs_idxs(peer, s, p, blocks_r)
    # Send blocks where blocks_s[i]=1,
    # recv blocks where blocks_r[i]=1
    sendrecv(dest, data, blocks_s, block_r)
```

Listing 1: Swing reduce-scatter pseudocode.

We can transmit non-contiguous data using, for example, MPI *indexed* datatypes. However, because communicating non-contiguous data can introduce overhead [25, 47], in the allreduce, we logically remap the blocks (i.e., without any actual memory movement) so that each node sends contiguous data. Indeed, even if the algorithm shuffles the block in the reduce-scatter, they are eventually placed again in the proper order in the buffers by the allgather. Moreover, by sending contiguous data, we also guarantee that the algorithm works with non-commutative reduction operators [42, 49].

3.1.2 Latency-optimal Algorithm

The latency-optimal Swing algorithm uses the same communication pattern as the bandwidth-optimal one, but instead of running a reduce-scatter followed by an allgather, at each step each node exchanges its entire vector with that of its peer (similarly to the latency-optimal recursive doubling). The algorithm only requires $\log_2 p$ steps ($\Lambda = 1$) but transmits $n \cdot \log_2 p$ bytes ($\Psi = D\log_2 p$ because the algorithm uses all the ports). We estimate the congestion deficiency in Sec. 4 when describing the algorithm for torus with more than one dimension.

3.2 Non-power-of-two Nodes

When p is even but not a power of 2, some nodes can receive the same block of data twice (one of which in the last step, see Appendix A.2). Thus, in that case, it is enough for each node not to send the same data block twice. Because no extra data is sent compared to the power of two cases, deficiencies do not increase.

If p is odd, we run the algorithm on $p-1$ nodes, while node $p-1$ at each step sends $(p-1)/2^s$ of its blocks to the corresponding $(p-1)/2^s$ nodes (and receiving from those its block). We show this through an example in Fig. 3 for a 1D torus with 7 nodes (we only show the first two steps). The first 6 nodes run the algorithm for even nodes as usual. At step 0 the last node sends (and receives) $\frac{n}{7}$ bytes to nodes 0, 1, and 2. At step 1, the last node sends $\frac{n}{7}$ bytes to node 3 and 4, and in the last step $\frac{n}{7}$ bytes to node 5. This slightly increases the bandwidth deficiency (by a $1/p$ additive factor).

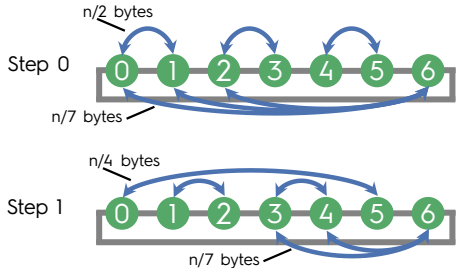


Figure 3: First 2 step of the Swing algorithm on a 1D torus with 7 nodes.

4 Design for Multidimensional Tori

In Sec. 3.1, we described the design of the Swing algorithm for 1D torus. We now discuss how to extend it to square (Sec. 4.1) and non-square (Sec. 4.2) torus with more than one dimension.

4.1 Square Tori

Like the recursive doubling algorithms optimized for tori (discussed in Sec. 2.3.2 and Sec. 2.3.3), in the Swing algorithm (both the latency- and the bandwidth-optimal) each node communicates on one dimension at a time. Formally, at step s, each node communicates on the dimension $\omega(s) = s \bmod D$. We define with $\sigma(s) = \lfloor \frac{s}{D} \rfloor$ the step of the algorithm relative to a specific dimension. For example, on a 2D torus, the third step of the algorithm is the second step executed in the

first dimension (i.e., because we count steps starting from 0, $\sigma(2) = \lfloor 3/2 \rfloor = 1$).

We denote the coordinates of a node with $(a_0, a_1, \ldots, a_{D-1})$. Then, at step s, each node communicates with a node with the same coordinates, except for the coordinate $a_{\omega(s)}$. If $a_{\omega(s)}$ is even, the coordinate $a_{\omega(s)}$ is then replaced with $(a_{\omega(s)} + \delta(\sigma(s))) \bmod d_{\omega(s)}$ (if odd, we flip the sign before $\delta(\sigma(s))$).

To use all the $2 \cdot D$ ports, Swing splits the data into $2 \cdot D$ parts and runs one allreduce on each. To avoid increasing the congestion deficiency, we must guarantee that, at each step, each of these $2 \cdot D$ collectives use different links. Swing runs D of these collectives (that we call *plain* collectives), each starting from a different dimension, using the algorithm described above. Swing runs the remaining D collectives (which we call *mirrored* collectives) with the same approach but starting from the opposite direction than that of the corresponding *plain* collective. By doing so, each of the $2 \cdot D$ allreduce uses a different port at each step.

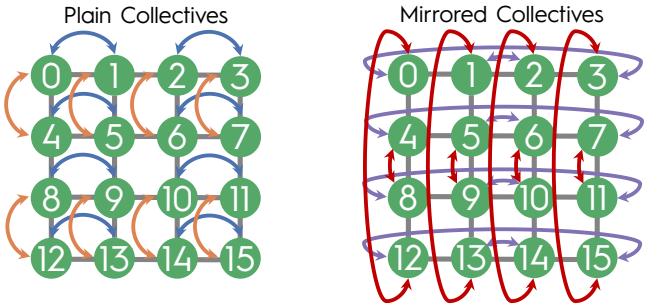


Figure 4: First step of the Swing algorithm on a 4x4 torus.

We show this through an example in Figure 4 on a 4×4 torus by showing the first step of the Swing algorithm. Node 0 runs a plain collective on the horizontal dimension exchanging data with node 1, and one in the vertical dimension with node 4. The two mirrored allreduce inverts the directions, exchanging data with node 3 in the horizontal dimension and node 12 in the vertical one.

Whereas both latency and bandwidth deficiencies are unaffected by the number of dimensions, congestion deficiency decreases with the number of dimensions due to increased bisection bandwidth. Intuitively, the more dimensions the torus has, the more communications the algorithm does with nodes at a closer distance before moving to higher distances. Because Swing halves the data size at each step, when a node needs to communicate with a distant node, data size becomes smaller and is thus less affected by congestion.

We can model the bandwidth term of the Swing allreduce as:

$$\frac{n}{2D}\beta \sum_{s=0}^{\log_2(p)-1} \frac{\delta(\sigma(s))}{2^{s+1}}$$

I.e., at each step s Swing halves the size of the data and, because the distance between communicating nodes at step s is $\delta(\sigma(s))$, there is at least one link shared by $\delta(\sigma(s))$, proportionally increasing the time to transmit one byte.

We can estimate the congestion deficiency by dividing this quantity by $(n\beta)/D$. Instead of deriving a hardly readable closed form, we report in Table 2 the values for different dimensions and for $p \to \infty$ (since the congestion deficiency increases with p). If $D \geq 3$, the Swing algorithm has a congestion deficiency $\Xi < 1.003$ (i.e., lower than 3%). To estimate the latency-optimal version's congestion deficiency, we sum the distances over all the steps, similar to what we did in Sec. 2.3.2 for the latency-optimal recursive doubling algorithm. I.e.:

$$\Xi = D \sum_{s=0}^{\frac{\log_2(p)-1}{D}} \delta(s) \leq \frac{4}{3} D \sqrt[D]{p}$$

We want to remark that the state-of-the-art latency-optimal and bandwidth-optimized recursive doubling algorithms described in Sec. 2.3.2 and Sec. 2.3.3 only use one port. In principle, we could extend them to use $2D$ ports by using the same approach we used for Swing, running D plain and D mirrored collectives. However, as we show in Sec. 5.1, they will perform strictly worse than Swing. Indeed, while mirroring decreases their bandwidth deficiencies, their congestion deficiencies are still higher than that of Swing due to the higher distance between communicating nodes.

4.2 Non-Square Tori

If not all the dimensions have equal size, the algorithm completes all the steps in one dimension while there are still steps to execute in other dimensions. If d_{min} is the smallest dimension, for the first $D \cdot \log_2(d_{min})$ the algorithm behaves exactly like in a $d_{min} \times \ldots \times d_{min}$ torus. After that, no data is sent anymore on that dimension, and the algorithm proceeds on the remaining ones. However, from that point on, it does not use all the available ports.

Indeed, since data is not transmitted anymore into one of the dimensions, the ports on that dimension are not used. If d_{min} is large enough, this has a limited impact because the size of the transmitted data decreases after each step. We show this through an example in Fig. 5 on a 2×4 torus where to not clutter the figure, we report only the communications performed by node 0. In step 2 (the last step), all the 4 collectives communicate on the horizontal dimension since each node has already reached all the nodes in their column.

While there is no difference in latency and bandwidth deficiency, the congestion deficiency increases compared to a square torus. In practice, it will be somewhere between that of an equivalent 1D torus (in the worst case) and of a perfectly square D-dimensional torus with the same number of nodes. The actual congestion deficiency depends on the relative differences between the dimensions. We denote with d_{min}

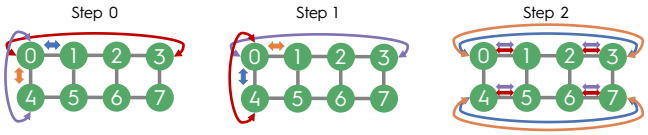


Figure 5: Multiport Swing collective on a 2x4 torus.

the smallest dimension and with d_{max} the largest one and we consider, as a worst case, a $d_{min} \times d_{min} \times \ldots \times d_{max}$ torus.

The algorithm uses all the ports for the first $D \cdot \log_2(d_{min})$ steps and behaves as it runs on a D-dimensional torus with d_{min}^D nodes. After that, it used only two ports for the remaining steps, behaving as it runs on a 1D torus, but starting from step $s = \log_2 d_{min}$ and on data of size $\frac{n}{2 \cdot 2^{D \log_2 d_{min}}} = \frac{n}{2 d_{min}^D}$. We denote the congestion deficiency of this second phase as:

$$\Xi_Q = \frac{1}{2 \cdot d_{min}^D} \sum_{s=\log_2 d_{min}}^{\log_2(d_{max})-1} \frac{2^{s+1} - (-1)^{s+1}}{3 \cdot 2^{s+1-\log_2 d_{min}}} \approx$$
$$\approx \frac{1}{6 \cdot d_{min}^D} \sum_{s=\log_2 d_{min}}^{\log_2(d_{max})-1} 2^{\log_2 d_{min}} = \quad (3)$$
$$= \frac{(\log_2(d_{max}/d_{min}))}{6 \cdot d_{min}^{D-1}}$$

We can then approximate the congestion deficiency for rectangular tori by summing the one for square tori to the one in Eq.3 (that is 0 for square tori). Generally, the higher the ratio between the largest and the smallest dimension, the more steps the algorithm executes not using all the available $2D$ ports. Nevertheless, we show in Sec. 5.2 that Swing still outperforms state-of-the-art algorithms by up to 3x, except on very large allreduce ($\geq$ 128MiB).

5 Experimental Evaluation

We evaluate the performance of the Swing algorithm on several torus and torus-like networks by comparing it with the best state-of-the-art algorithms described in Sec. 2.3. We implemented all these algorithms in the *Structural Simulation Toolkit* (SST [1]), a packet-level network simulator. We simulate networks with 400Gb/s links with 100ns latency and 300ns of per-hop packet processing latency [19, 28].

Since each node has $2D$ ports, the maximum injection bandwidth of a node is $2 \cdot D \cdot 400Gb/s$. Also, in all the plots, we show the *goodput*, i.e., how many bytes are reduced per time unit. Because the allreduce needs to send at least twice the number of bytes in the vector [9], the peak goodput is half the injection bandwidth (i.e., $D \cdot 400Gb/s$).

In the following, we analyze the performance on 2D square (Sec. 5.1) and rectangular tori (Sec. 5.2), higher-dimensional tori (Sec. 5.3), and other torus-like topologies (Sec. 5.4). Eventually, we summarize the results (Sec. 5.5).

5.1 Performance on 2D Square Torus

In Fig. 6, we show the performance evaluation on a 64x64 2D torus with 4,096 nodes. In the main plot, we show the goodput of the allreduce for different vector sizes, with each line representing a different algorithm. For the Swing algorithm, for each size we only report the best between the latency- and bandwidth-optimal versions, and we annotate the point where we switch from the latency-optimal to the bandwidth-optimal algorithm with a large dot. We do something similar for recursive doubling as well.

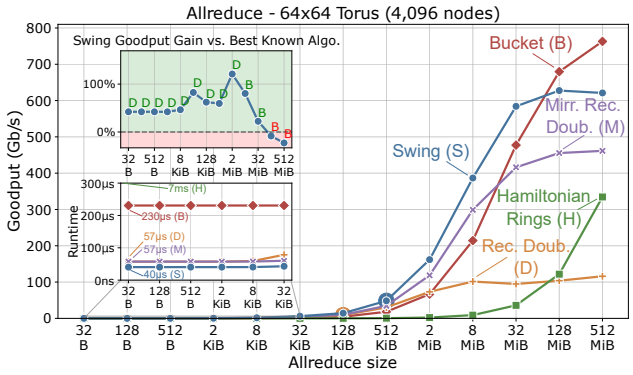


Figure 6: Goodput of allreduce algorithms on a 64x64 2D Torus topology with 4,096 nodes. The small plot in the bottom left reports runtime for small allreduce (ranging from 32B to 32KiB). The runtime for 32B allreduce is annotated using the same one-letter labels as in the main plot. The top left inner plot shows the goodput gain of Swing compared to the best-known state-of-the-art algorithm, and the letters on top of each datapoint denote the name of the best-known algorithm.

The zoomed-in plot in the bottom left shows the runtime of each algorithm for allreduce on small vectors (from 32B to 32KiB). We also denote the runtime for 32B allreduce for each algorithm, using the same one-letter labels used in the main plot. Because the runtime of the ring algorithm for small vectors is orders of magnitude larger than the other algorithms, it is always out of scale (the time is nevertheless annotated on the top of the small plot).

Last, the top left plot shows, for each allreduce size, the goodput gain of Swing compared to the best-known algorithm as a function of the allreduce size. For example, a 100% gain denotes Swing is 2x faster than the best state-of-the-art algorithm. The letter at each data point represents the name of the best-known algorithm.

For completeness, in this plot, we also show our improved version of recursive doubling, which uses all the ports (denoted as *Mirrored Recursive Doubling*) using the same *plain* and *mirrored* allreduce technique used by Swing (and described in Sec. 4). Swing consistently outperforms our mirrored recursive doubling at any size due to the lower conges-

tion deficiency, and we thus exclude it from the comparison and from the subsequent results.

By analyzing the results, we observe that Swing outperforms all the other allreduce algorithms for vectors ranging from 32B to 32MiB due to the lower latency deficiency compared to the ring and bucket algorithms and the lower bandwidth deficiency compared to the latency-optimal recursive doubling algorithm. We observe more than 2x improvement over the recursive doubling algorithm for 2MiB allreduce.

The bucket algorithm performs better than Swing starting from 128MiB due to its lower congestion deficiency, which compensates for the higher latency deficiency on large vectors. This is instead not the case for the ring algorithm, characterized by a higher latency deficiency than the bucket algorithm. Moreover, we observe that on 512MiB, Swing achieves around 77% of the peak goodput, which is what we would expect from our modeling. Indeed, a congestion deficiency of 1.19 on a 2D torus (see Table 2) means Swing can reach at most 81% of peak goodput.

By analyzing the two small inner plots, we observe up to 50% improvement for small vectors ($\leq$ 32KiB) compared to the latency-optimal recursive doubling algorithm. This is partly due to the lower congestion deficiency, but mostly to the shorter distance between communicating nodes, which reduces the latency α (although we did not explicitly model it so as not to burden the notation). We observe the highest goodput gains (around 120%) for 2MiB vectors. This is indeed the sweet spot where the recursive doubling algorithm performs poorly, and the performance of the bucket and ring algorithms is still severely affected by their higher latency deficiency.

5.1.1 Scaling

We then analyze the performance of Swing on 2D torus of different sizes. We show in Fig. 7 the goodput gain of Swing over the best-known algorithm at each allreduce size and for networks ranging from 64 to 16,384 nodes. We observe that Swing outperforms state-of-the-art algorithms regardless of the network size, for up to 32MiB allreduce.

Moreover, we can see that the maximum Swing gain increases when increasing the network size. Indeed, the larger the network, the larger the impact of latency deficiency on performance. As a consequence, both the bucket and ring algorithm performance decreases when increasing the number of nodes (and the Swing gain increases).

Nevertheless, the bucket algorithm outperforms Swing for large vectors ($\geq$ 128MiB). Indeed, when increasing the network size, Swing congestion deficiency increases. However, as estimated in Table 2 and as shown in the figure, on 2D torus, we expect at most a negative gain of around 20% (i.e., a peak bandwidth of around 80%) regardless of network size.

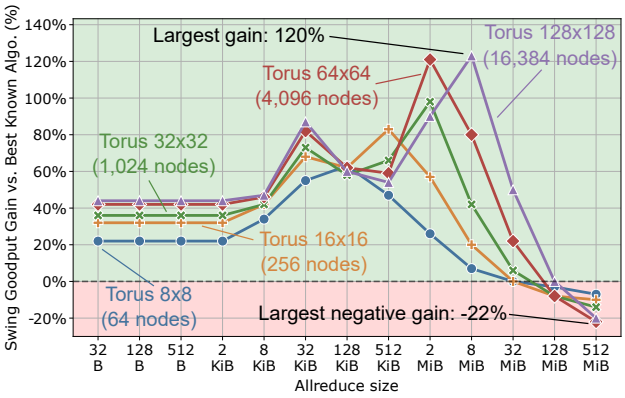


Figure 7: Swing goodput gain on square torus networks ranging from 64 to 16,384 nodes.

5.1.2 Bandwidth Impact

To analyze Swing performance for different network bandwidths, we show in Fig. 8 the goodput gain for 8x8 torus networks with bandwidth ranging from 100 Gb/s to 3.2 Tb/s. We observe consistent gains over the best-known state-of-the-art algorithm regardless of the network bandwidth.

For low bandwidths, the relative impact of bandwidth and congestion deficiencies on performance is higher, and the gain of Swing over recursive doubling for small messages increases. At higher bandwidth, the relative impact of congestion deficiency is lower, and the maximum gain of Swing for small allreduce decreases. At the same time, however, Swing is not outperformed anymore by the bucket algorithm for large allreduce. For example, on 3.2 Tb/s networks, Swing outperforms all the other algorithms even for 512MiB allreduce.

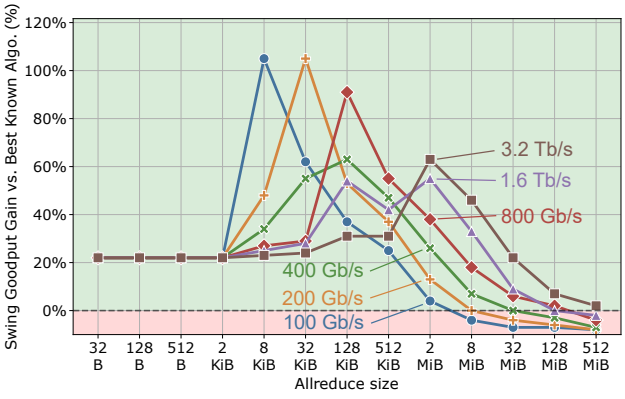


Figure 8: Swing goodput gain on 8x8 torus networks with network bandwidth ranging from 100 Gb/s to 3.2 Tb/s.

5.2 Performance on Rectangular Tori

As discussed in Sec. 4.2, the congestion deficiency of the Swing algorithm increases if the torus dimensions are not all equal, proportionally to the ratio between the sizes of the largest and smallest dimension. The ring algorithm is instead unaffected by the shape of the torus, because the Hamiltonian rings span over all the nodes (as long as the conditions discussed in Sec. 2.3.1 are satisfied).

On the other hand, the shape of the torus negatively impacts the bucket algorithm's latency deficiency. Indeed, if some torus dimensions are larger than the others, some of the $2D$ concurrent collectives might move from dimension i to dimension $i+1$, whereas there are still collectives running on dimension $i+1$. We show this through an example in Fig. 9, where at step 1, one bidirectional ring moves from the vertical to the horizontal dimension while the other bidirectional ring is still running.

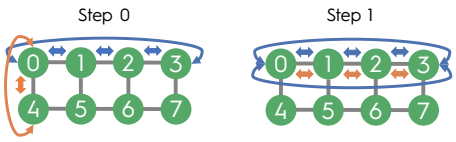


Figure 9: First steps of the Bucket algorithm on a 2x4 torus.

This is usually detrimental to performance, and it is better if all the collectives synchronously move from one dimension to the next one [44]. Thus, each step is completed only after all the collectives running on the largest dimension are completed. Hence, if d_{max} is the size of the largest dimension, the latency deficiency becomes $\Lambda = \frac{2D \cdot d_{max}}{\log_2 p}$. Bandwidth and congestion deficiencies are instead unaffected. In a nutshell, this means that the latency deficiency of the bucket algorithm is the same as that for a $d_{max} \times d_{max} \cdots \times d_{max}$ torus.

For these reasons, we show in Fig. 10 the goodput of the different algorithms for different torus networks, all with 1,024 nodes but with different rectangular shapes. First, we observe that, as expected, the ring algorithm is unaffected by the shape of the torus, and outperforms both the bucket and Swing algorithms for allreduce larger than 512MiB. On the other hand, the latency deficiency of the bucket algorithm increases proportionally to the ratio between the largest and the smallest dimensions, reducing its performance for small and medium vectors. This is visible, for example, by analyzing how the goodput for large allreduce decreases when moving from a 64x16 to a 256x4 torus.

Last, Swing performance decreases compared to a square torus due to its higher congestion deficiency for rectangular torus networks. Nevertheless, we observe that Swing still outperforms the other algorithms up to 32MiB regardless of the network shape (up to 3x on the 128x8 and 256x4 torus).

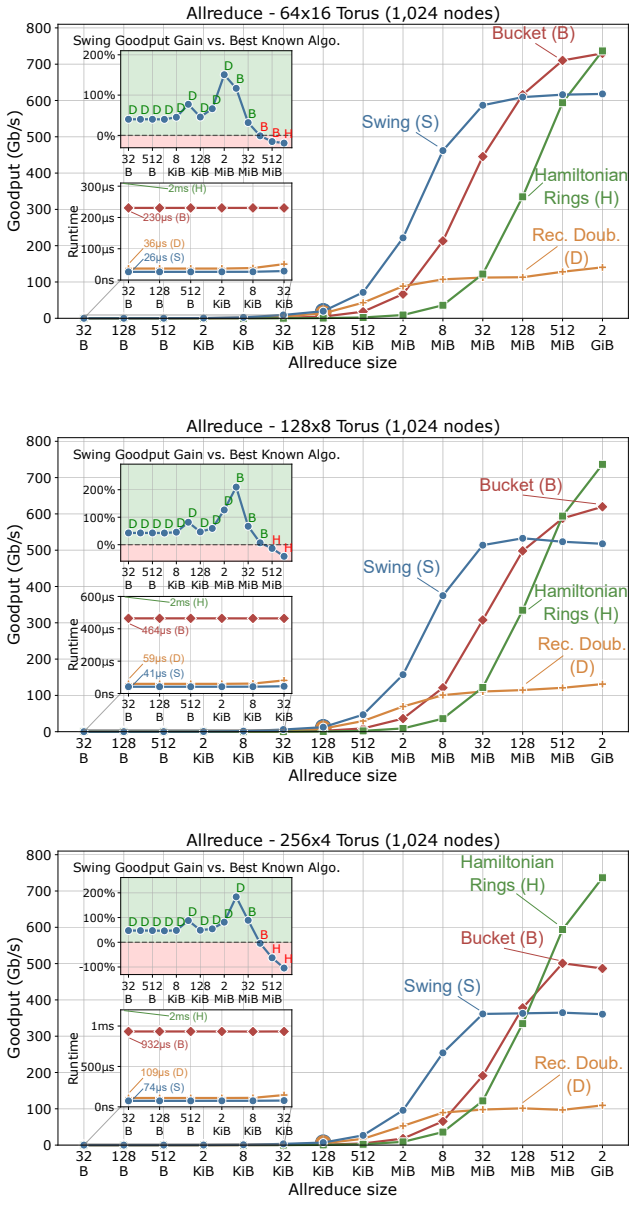


Figure 10: Goodput on 2D torus with 1,024 nodes and different rectangular shapes: 64x16, 32x8, and 256x4.

5.3 Performance for 3D and 4D Torus

As discussed in Sec. 4 and summarized in Table 2, the performance of the allreduce algorithm for multidimensional torus also depends on the number of dimensions. Thus, we evaluate the performance of the different allreduce algorithms on 8^2, 8^3, and 8^4 torus networks.

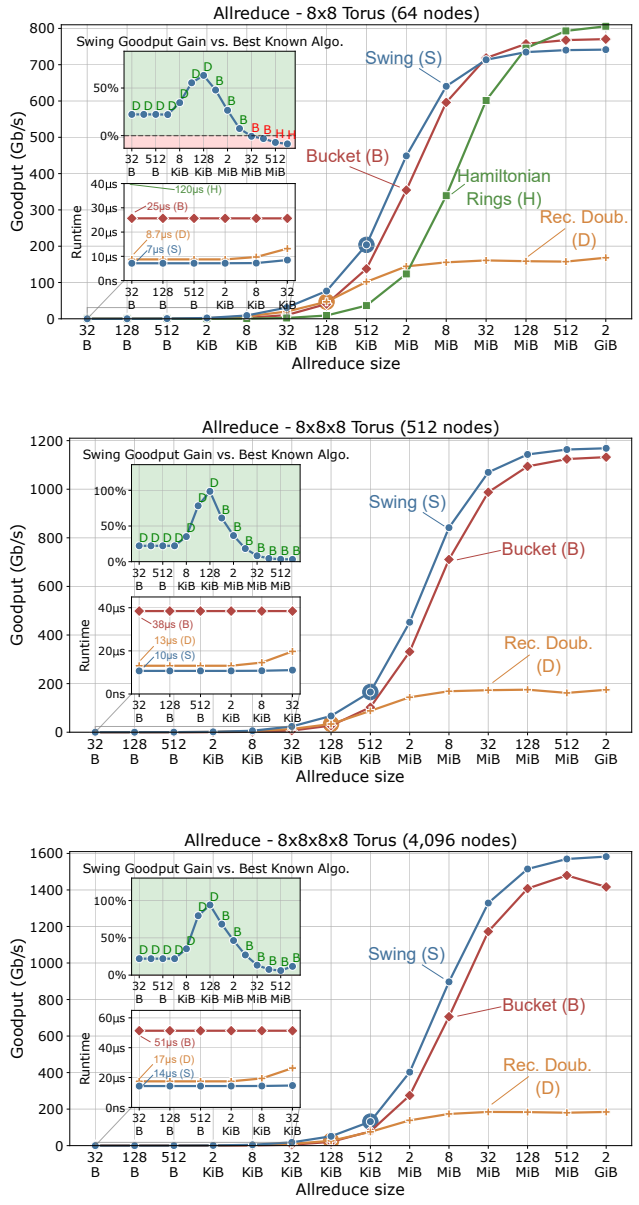


Figure 11: Goodput on higher-dimensional torus networks: 2D 8x8, 3D (8x8x8), and 4D (8x8x8x8).

We report the evaluation result in Fig. 11. We do not include the Hamiltonian ring algorithm in the 3D and 4D torus results since it only works for 2D torus networks. When increasing the number of dimensions, the goodput gain of Swing increases because, as shown in Table 2 and discussed in Sec. 4, the congestion deficiency drops to 3% on 3D torus and to 0.8% on 4D torus. Consequently, *for 3D and 4D torus networks, Swing outperforms by up to 2x all existing algorithms on allreduce ranging from 32B to 2GB.*

5.4 Performance on Torus-Like Topologies

Some topologies like HammingMesh [26] and HyperX [3,20] extend torus by adding additional links, thus increasing the network bisection bandwidth. Seen from a different perspective, those extra links allow distant nodes to communicate crossing fewer hops, decreasing Swing congestion deficiency.

5.4.1 Performance on HammingMesh

HammingMesh [26] groups nodes into square *boards*. Each board is a 2D mesh, and nodes on the same column (or row) located at the edge of the boards are connected together using fat trees. Due to its higher performance and flexibility compared to a torus a similar topology is used, for example, to interconnect TPUv4 devices [31]. Because of the extra links, the congestion deficiency of Swing on a HammingMesh is lower than that on a 2D torus. Moreover, for a fixed number of nodes, having smaller boards increases the number of extra (fat tree) links and, thus, decreases the congestion deficiency.

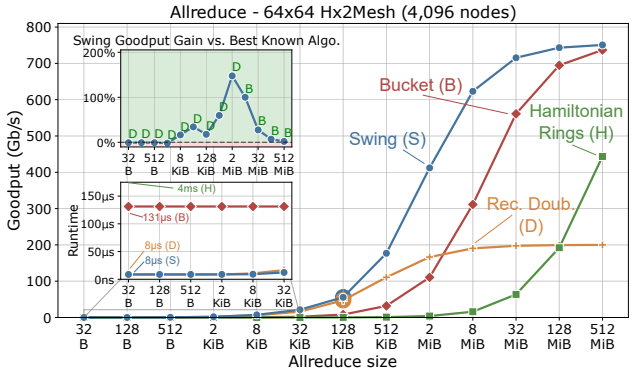


Figure 12: Goodput on a 4,096 nodes Hx2Mesh.

We show in Fig. 12 the performance of the different algorithms for a Hx2Mesh network with 4,096 nodes (2x2 boards arranged in a 32x32 configuration). For such configuration, Swing outperforms the state-of-the-art algorithms at any size, up to 2.5x for 2MiB allreduce. Moreover, because of the lower congestion deficiency, we observe how the peak Swing performance is higher compared to a 2D torus with the same number of nodes (Fig. 6). Last, we also observe a runtime reduction for all the algorithms for small vectors, since nodes on the same board on HammingMesh are connected through PCB traces, with lower latency than optical network cables.

In Fig. 13 we instead report the results for a HammingMesh with the same number of nodes (4,096), but using 4x4 boards (4x4 boards arranged in a 16x16 configuration). This configuration is a middle point between the torus topology and the Hx2Mesh since it has more extra links than a torus, but fewer than a Hx2Mesh. Due to fewer extra links compared to Hx2Mesh, on Hx4Mesh Swing has a higher congestion deficiency, as we can see starting from 128MiB allreduce.

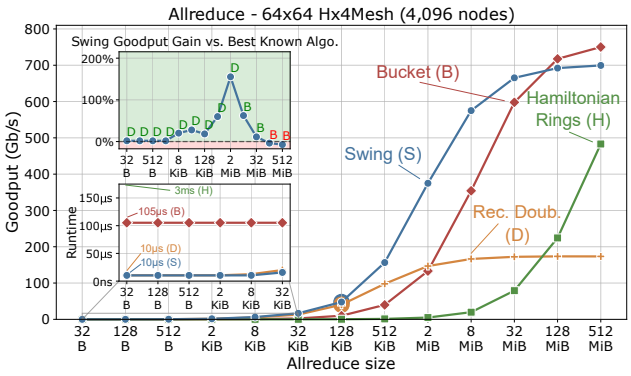


Figure 13: Goodput on a 4,096 nodes Hx4Mesh.

5.4.2 Performance on 2D HyperX

Last, we report in Fig. 14 the performance on a 4,096 2D HyperX topology [3] (which can be seen as a HammingMesh with 1x1 boards [26]). HyperX connects each node to every node in the same row and column. Because in Swing each node communicates only with nodes on the same row (or the same column), on HyperX Swing does not experience any congestion deficiency. We indeed observe from the plot that Swing outperforms all the other algorithms at any allreduce size. Moreover, the maximum goodput gain increases from 2.5x in Hx2Mesh and Hx4Mesh, to 3x in HyperX.

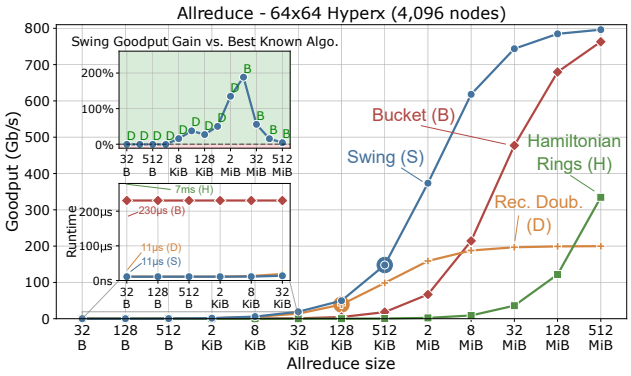


Figure 14: Goodput on a 4,096 nodes HyperX.

5.5 Summary

Last, we summarize all the presented results in Fig. 15, where we report the distribution of Swing goodput gain over the best algorithm at each message size for the different scenarios we analyzed. We show data for allreduce sizes up to 512MiB since these are the sizes practically used in existing HPC [14] and machine learning workloads [35] and larger allreduce are usually split into smaller ones to overlap communication and computation better.

For each box, the triangle shows the median across the allreduce sizes. The left and right sides of the box represent the first ($Q1$) and third ($Q3$) quartile, respectively. The left whisker denotes the smallest point larger than $Q1 - 1.5 \cdot (Q3 - Q1)$, whereas the right whisker the biggest point smaller than $Q3 + 1.5 \cdot (Q3 - Q1)$. Empty dots outside the whiskers are considered outliers.

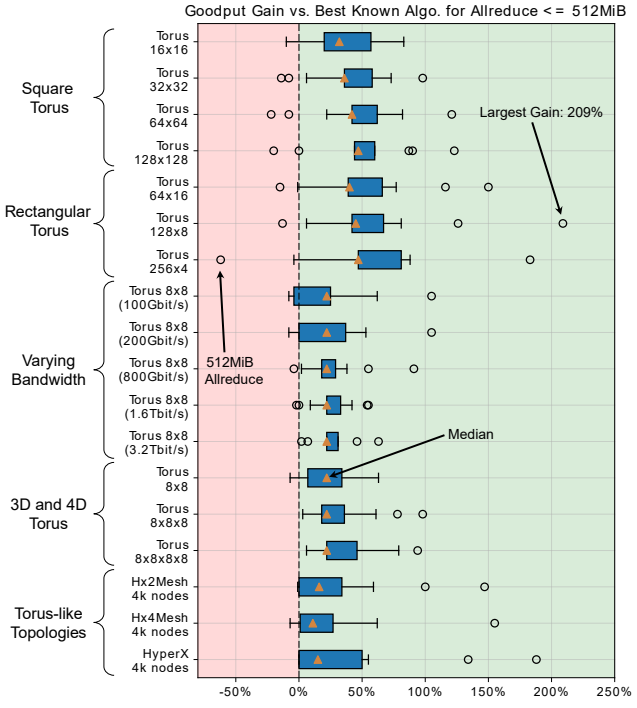


Figure 15: Summary of Swing goodput gain on different topologies.

First, by observing the performance on square torus networks, we see that Swing median and maximum goodput gain increases with the network size (going from 8x8 to 128x128 torus). As discussed in Sec. 5.1, the maximum negative gain also increases with network size, but is at most around 20% and only occurs for large allreduce ($\geq$ 128MiB).

For rectangular torus networks, Swing median and maximum goodput gain increase proportionally to the ratio between the maximum and minimum-sized dimension (i.e., go-

ing from 64x16 to 256x4 torus), up to 3x for 128x8 torus. For 256x4 torus, Swing performs around 60% worse than the ring algorithm on 512MiB allreduce.

Increasing the network bandwidth has a positive effect on Swing goodput gain. Although the maximum goodput gain decreases, for higher bandwidths, Swing outperforms all the state-of-the-art algorithms at any allreduce size. Regardless of the bandwidth, the median Swing goodput gain across different allreduce sizes stands around 25%.

Swing goodput gain also increases with the number of dimensions (moving from 8x8 to 8x8x8x8 torus) since each node has additional one-hop distant nodes. Because in the reduce-scatter nodes communicate first with their neighbors, and halve the data size at each step, Swing can further reduce the transmitted data size before nodes need to communicate with more distant ones. On one side, this increases the maximum goodput gain up to 2x. On the other side, it outperforms all the other algorithms at any size.

Last, Swing provides consistent performance gains even on other torus-like topologies such as HammingMesh and HyperX. When comparing HammingMesh with a torus with the same number of nodes (i.e., 64x64), we observe higher goodput gains on HammingMesh, due to the extra links compared to the torus, which helps in reducing Swing congestion deficiency. Also, on both Hx2Mesh and HyperX, Swing outperforms the other algorithms regardless of the allreduce size.

Overall we observe a median goodput ranging between 20% and 50%, and a maximum goodput gain of 3x. This underlines the advantages of using the Swing allreduce algorithm on torus and torus-like networks of any shape and number of dimensions and at any network bandwidth.

6 Discussion

Extension to Other Collectives Although in this work we focus on allreduce (and, indirectly, on reduce-scatter and allgather), a similar approach can be adopted for other collective operations. Namely, Swing can replace the recursive doubling algorithm for all those collectives where it is used (e.g., broadcast and reduce).

Routing Impact In all our experiments, we used minimal adaptive routing. I.e., packets are forwarded on the least congested shortest path. Because in all the analyzed algorithms, each node only communicates with others on the same row/column, traffic is evenly distributed across links. Sending packets on non-shortest paths would unnecessarily increase network traffic and decrease performance.

Swing Performance on Full-Bandwidth Topology On full-bandwidth topologies (e.g., non-blocking fat trees), both Swing and recursive doubling will not have any congestion deficiency, and we expect them to have the same performance.

7 Conclusions

Due to the relevance of torus networks in high-performance machine learning systems, in this work, we presented *Swing*, a new allreduce algorithm for torus networks. To motivate Swing design, we modeled the latency, bandwidth, and congestion deficiencies, of Swing and of the best state-of-the-art algorithms. Our modeling highlights the shortcomings of existing allreduce algorithms for torus networks, especially for small- and medium-sized allreduce.

We then presented the Swing design. Swing performs a logarithmic number of steps and transmits a minimal number of bytes. To reduce the impact of the torus low bisection bandwidth, Swing shortcuts the torus, reducing the distance between communicating nodes and, thus, the congestion deficiency. Last, we extensively evaluated Swing performance and compared it against the best state-of-the-art algorithms, for different node counts, network bandwidths, shapes, number of dimensions, and topologies. Our evaluation shows improvements up to 3x on all practically used allreduce sizes.

Acknowledgments

We thank the anonymous NSDI reviewers and our shepherd, Philip Levis, for their constructive feedback. This work was supported by Sapienza University under the SEED-2022 and "Progetti Grandi 2023" funding schemes; by UrbanTwin, an ETH Board Joint Initiatives project; and by EuroHPC-JU under grant agreement RED-SEA, No 055776. Daniele De Sensi is a member of the *Gruppo Nazionale Calcolo Scientifico - Istituto Nazionale di Alta Matematica* (GNCS-INdAM).

References

[1] Helgi Adalsteinsson, Scott Cranford, David A. Evensky, Joseph P. Kenny, Jackson Mayo, Ali Pinar, and Curtis L. Janssen. A simulator for large-scale parallel computer architectures. *Int. J. Distrib. Syst. Technol.*, 1(2):5773, apr 2010.

[2] N.R. Adiga, G. Almasi, G.S. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A.A. Bright, J. Brunheroto, C. Cascaval, J. Castanos, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T.M. Cipolla, P. Crumley, K.M. Desai, A. Deutsch, T. Domany, M.B. Dombrowa, W. Donath, M. Eleftheriou, C. Erway, J. Esch, B. Fitch, J. Gagliano, A. Gara, R. Garg, R. Germain, M.E. Giampapa, B. Gopalsamy, J. Gunnels, M. Gupta, F. Gustavson, S. Hall, R.A. Haring, D. Heidel, P. Heidelberger, L.M. Herger, D. Hoenicke, R.D. Jackson, T. Jamal-Eddine, G.V. Kopcsay, E. Krevat, M.P. Kurhekar, A.P. Lanzetta, D. Lieber, L.K. Liu, M. Lu, M. Mendell, A. Misra, Y. Moatti, L. Mok, J.E. Moreira,

B.J. Nathanson, M. Newton, M. Ohmacht, A. Oliner, V. Pandit, R.B. Pudota, R. Rand, R. Regan, B. Rubin, A. Ruehli, S. Rus, R.K. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B.D. Steinmacher-Burow, K. Strauss, C. Surovic, R. Swetz, T. Takken, R.B. Tremaine, M. Tsao, A.R. Umamaheshwaran, P. Verma, P. Vranas, T.J.C. Ward, M. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. Krolak, C.T. Li, T. Liebsch, J. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. Wait, J. Wittrup, M. Bae, K. Dockser, L. Kissel, M.K. Seager, J.S. Vetter, and K. Yates. An overview of the bluegene/l supercomputer. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 60–60, 2002.

[3] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S. Schreiber. Hyperx: Topology, routing, and packaging of efficient large-scale networks. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, 2009. Association for Computing Machinery.

[4] Yuichiro Ajima, Takahiro Kawashima, Takayuki Okamoto, Naoyuki Shida, Kouichi Hirai, Toshiyuki Shimizu, Shinya Hiramoto, Yoshiro Ikeda, Takahide Yoshikawa, Kenji Uchida, and Tomohiro Inoue. The tofu interconnect d. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 646–654, 2018.

[5] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. Loggp: Incorporating long messages into the logp model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[6] George Almási, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, José E. Moreira, B. Steinmacher-Burow, and Yili Zheng. Optimization of mpi collective communication on bluegene/l systems. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, page 253–262, New York, NY, USA, 2005. Association for Computing Machinery.

[7] Amazon. AWS Trn1 Architecture. https://awsdocs-neuron.readthedocs-hosted.com/en/v2.3.0/general/arch/neuron-hardware/trn1-arch.html, 2023. Accessed: 06-Mar-2023.

[8] AMOTZ BAR-NOY, SHLOMO KIPNIS, and BARUCH SCHIEBER. An optimal algorithm for computing census functions in message-passing systems. *Parallel Processing Letters*, 03(01):19–23, 1993.

[9] M. Barnett, R. Littlefield, D.G. Payne, and R. Vandegeijn. Global combine algorithms for 2-d meshes with wormhole routing. *Journal of Parallel and Distributed Computing*, 24(2):191–201, 1995.

[10] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis, 2018.

[11] Brett Bode, Michelle Butler, Thom Dunning, Torsten Hoefler, William Kramer, William Gropp, and Wen-mei Hwu. *The Blue Waters Super-System for Super-Science*, pages 339–366. 11 2017.

[12] JEHOSHUA BRUCK and CHING-TIEN HO. Efficient global combine operations in multi-port message-passing systems. *Parallel Processing Letters*, 03(04):335–346, 1993.

[13] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 62–75, New York, NY, USA, 2021. Association for Computing Machinery.

[14] Sudheer Chunduri, Scott Parker, Pavan Balaji, Kevin Harms, and Kalyan Kumaran. Characterization of mpi usage on a production supercomputer. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 386–400, 2018.

[15] Google Cloud. Enabling next-generation ai workloads: Announcing tpu v5p and ai hypercomputer, 2024.

[16] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, jul 1993.

[17] Daniele De Sensi, Edgar Costa Molero, Salvatore Di Girolamo, Laurent Vanbever, and Torsten Hoefler. Canary: Congestion-aware in-network allreduce using dynamic trees. *Future Generation Computer Systems*, 152:70–82, 2024.

[18] Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefler. Flare: Flexible in-network allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[19] Daniele De Sensi, Salvatore Di Girolamo, Kim H. McMahon, Duncan Roweth, and Torsten Hoefler. An In-Depth Analysis of the Slingshot Interconnect. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC20)*, Nov. 2020.

[20] Jens Domke, Satoshi Matsuoka, Ivan R. Ivanov, Yuki Tsushima, Tomoya Yuki, Akihiro Nomura, Shin'ichi Miura, Nie McDonald, Dennis L. Floyd, and Nicolas Dubé. Hyperx topology: First at-scale implementation and comparison to the fat-tree. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Jiawei Fei, Chen-Yu Ho, Atal N. Sahu, Marco Canini, and Amedeo Sapio. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 676–691, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Guangnan Feng, Dezun Dong, and Yutong Lu. Optimized mpi collective algorithms for dragonfly topology. In *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, New York, NY, USA, 2022. Association for Computing Machinery.

[23] Richard L. Graham, Lion Levi, Devendar Bureddy, Gil Bloch, Gilad Shainer, David Cho, George Elias, Daniel Klein, Joshua Ladd, Ophir Maor, Ami Marelli, Valentin Petrov, Evyatar Romlet, Yong Qin, and Ido Zemah. Scalable hierarchical aggregation and reduction protocol (sharp)tm streaming-aggregation hardware design and evaluation. *High Performance Computing*, 12151:41 – 59, 2020.

[24] Graphcore. Graphcore ipu-pod system, 2024.

[25] William Gropp, Ewing Lusk, and Deborah Swider. Improving the performance of mpi derived datatypes. In *Proceedings of the Third MPI Developer's and User's Conference*, pages 25–30. Citeseer, 1999.

[26] Torsten Hoefler, Tommaso Bonato, Daniele De Sensi, Salvatore Di Girolamo, Shigang Li, Marco Heddes, Jon Belk, Deepak Goel, Miguel Castro, and Steve Scott. Hammingmesh: A network topology for large-scale deep learning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022.

[27] Torsten Hoefler and D. Moor. Energy, Memory, and Runtime Tradeoffs for Implementing Collective Communication Operations. *Journal of Supercomputing Frontiers and Innovations*, 1(2):58–75, Oct. 2014.

[28] Torsten Hoefler, Duncan Roweth, Keith Underwood, Bob Alverson, Mark Griswold, Vahid Tabatabaee, Mohan Kalkunte, Surendra Anubolu, Siyan Shen, Abdul Kabbani, Moray McLaren, and Steve Scott. Datacenter ethernet and rdma: Issues at hyperscale, 2023.

[29] Nikhil Jain and Yogish Sabharwal. Optimal bucket algorithms for large mpi collectives on torus interconnects. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, page 27–36, New York, NY, USA, 2010. Association for Computing Machinery.

[30] Andreas Jocksch, Noé Ohana, Emmanuel Lanti, Eirini Koutsaniti, Vasileios Karakasis, and Laurent Villard. An optimisation of allreduce communication in message-passing systems. *Parallel Computing*, 107:102812, 2021.

[31] Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings, 2023.

[32] Benjamin Klenk, Nan Jiang, Greg Thorson, and Larry Dennison. An in-network architecture for accelerating shared-memory multiprocessor collectives. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 996–1009, 2020.

[33] Dmitry Kolmakov and Xuecang Zhang. A generalization of the allreduce operation, 2020.

[34] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761. USENIX Association, April 2021.

[35] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, aug 2020.

[36] Junchao Ma, Dezun Dong, Cunlu Li, Ke Wu, and Liquan Xiao. Paard: Proximity-aware all-reduce communication for dragonfly networks. In *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable*

Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), pages 255–262, 2021.

[37] Open MPI. Open MPI Allreduce. https://github.com/open-mpi/ompi/blob/main/ompi/mca/coll/base/coll_base_allreduce.c, 2023. Accessed: 06-Mar-2023.

[38] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distrib. Comput.*, 69(2):117–124, feb 2009.

[39] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G.E. Fagg, E. Gabriel, and J.J. Dongarra. Performance analysis of mpi collective operations. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 8 pp.–, 2005.

[40] The Next Platform. The ai training chip tencent has an eye on, 2021.

[41] Rolf Rabenseifner. Optimization of collective reduction operations. In Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2004*, pages 1–9, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[42] Rolf Rabenseifner and Jesper Larsson Träff. More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 36–46, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[43] Martin Ruefenacht, Mark Bull, and Stephen Booth. Generalisation of recursive doubling for allreduce: Now with simulation. *Parallel Computing*, 69:24–44, 2017.

[44] Paul Sack and William Gropp. Collective algorithms for multiported torus networks. *ACM Trans. Parallel Comput.*, 1(2), feb 2015.

[45] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[46] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling Distributed Machine Learning with In-Network Aggregation. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, Apr 2021.

[47] Timo Schneider, Fredrik Kjolstad, and Torsten Hoefler. MPI datatype processing using runtime compilation. In Jack J. Dongarra, Javier García Blas, and Jesús Carretero, editors, *20th European MPI Users's Group Meeting, EuroMPI '13, Madrid, Spain - September 15 - 18, 2013*, pages 19–24. ACM, 2013.

[48] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. Taccl: Guiding collective algorithm synthesis using communication sketches, 2022.

[49] Rajeev Thakur and William D. Gropp. Improving the performance of collective operations in mpich. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 257–267, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[50] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, feb 2005.

[51] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 739–767, Boston, MA, April 2023. USENIX Association.

[52] J. Worringen. Pipelining and overlapping for mpi collective operations. In *28th Annual IEEE International Conference on Local Computer Networks, 2003. LCN '03. Proceedings.*, pages 548–557, 2003.

A Correctness Proof

In this section we first prove the correctness of the Swing algorithm when the number of nodes is a power of two (Sec. A.1) and then extend it for an arbitrary number of nodes (Sec. A.2).

A.1 Power of Two Number of Nodes

For clarity reasons, we prove the correctness of the latency-optimal algorithm (both the latency- and bandwidth-optimal algorithms perform the same communication pattern, although nodes transmit different data). To prove the algorithm's correctness, we need to prove that the data transmitted by each node eventually reaches all the other nodes. For example, on a 1D torus with p nodes, at step 0, node 0 communicates with node 1 (which aggregates the received data with its own). At step 1, node 0 communicates with node

$-1 \bmod p = p-1$, whereas node 1 communicates with node 2. Thus, we can say that at step 1 the data sent from 0 reached nodes $\{1, 2, p-1\}$ (2 has been reached *indirectly* through node 1).

Because the number of reached nodes doubles at each step, and because we perform $\log_2 p$ steps, the data sent from any given node would eventually reach $p-1$ nodes. We need, however, to prove that those $p-1$ nodes are distinct (i.e., that the data sent by each node reach every other node exactly once and is thus never aggregated twice). To do so, we need first to prove a few lemmas.

Lemma A.1. *$\rho(s)$ and $\delta(s)$ are odd $\forall s \in \mathbb{N}$.*

Proof. $(-2)^i$ is odd for $i = 0$, and even for $i > 0$. The sum of even numbers with an odd number is odd. □

Lemma A.2. *If r is even, $\pi(r,s)$ is odd, and vice versa.*

Proof. An even node r communicates at step s with node $\pi(r,s) = r + \rho(s) \bmod p$. Because p is a power of two (thus even), and $\rho(s)$ is odd (Lemma A.1), $\pi(r,s)$ is odd. Vice versa, odd nodes communicate with even nodes. □

If a node r communicates at step s with a node $q = \pi(r,s)$, and q communicates with a node $z = \pi(q,h)$ at step $h > s$, we say that s indirectly reached node z. Because even nodes only communicate with odd nodes (and vice versa), if r is even, we can rewrite:

$$z = \underbrace{\overbrace{(r + \rho(s) \bmod p)}^{q=\pi(r,s)} - \rho(h) \bmod p}_{\pi(q,h)} = r + \rho(s) - \rho(h) \bmod p$$

I.e., the sign behind $\rho(s)$ alternates between positive and negative, starting from positive. In general, an even node r can reach through a sequence of k steps $\{s_0 < s_1 < s_2 < \ldots < s_{k-1}\}$ a node q, with:

$$q = r + \rho(s_0) - \rho(s_1) + \rho(s_2) - \ldots \bmod p = \\ = (r + \sum_{i=0}^{k-1} -1^i \rho(s_i)) \bmod p$$

The same applies if r is odd, by replacing -1^i with -1^{i+1}.

Lemma A.3. *Even nodes reach (directly or indirectly) odd nodes through an odd number of steps k. Odd nodes reach (directly or indirectly) even nodes through an odd number of steps k.*

Proof. This stems from Lemma A.2. If r is even and k is odd, then $q = (r + \sum_{i=0}^{k-1} -1^i \rho(s_i)) \bmod p$ is odd because $\rho(s)$ is always odd. Similarly, if r is odd and k is odd, q is even. □

Lemma A.4. *Given k integers $\{e_0 < e_1 < \ldots < e_{k-1}\}$, with $e_{k-1} \leq \log_2(p) - 1$, then $-p < \sum_{i=0}^{k-1} (-2)^{e_i} < p$.*

Proof. We have $\sum_{i=0}^{k-1} (-2)^{e_i} \leq \sum_{i=0}^{k-1} 2^{e_i} < 2^{e_{k-1}+1} \leq p$. Similarly, $\sum_{i=0}^{k-1} (-2)^{e_i} \geq -\sum_{i=0}^{k-1} 2^{e_i} > -(2^{e_{k-1}+1}) \geq -p$. □

Theorem A.5. *On a 1D torus, if a node r at step s communicates with node $\pi(r,s)$ (defined in Eq. 2), it will reach (directly or indirectly) all the other $p-1$ nodes in $log_2(p)$ steps (with p power of two).*

Proof. We need to prove that, a unique sequence of k steps $\{s_0 < s_1 < \ldots s_{k-1}\}$ exists by which a given node r reaches a node q. We prove this by contradiction, and we will prove it by assuming that r is even and q is odd (the proof for the other cases is analogous and only requires changing the signs before the ρ terms). Assume that there are two different sequences of steps $\{s_0 < s_1 < \ldots s_{k-1} \leq \log_2(p) - 1\}$ and $\{t_0 < t_1 < \ldots t_{h-1} \leq \log_2(p) - 1\}$ of k and h steps respectively (both k and h are odd from Lemma A.3), so that:

$$q = r + \rho(s_0) - \rho(s_1) + \rho(s_2) - \ldots + \rho(s_{k-1}) \bmod p = \\ = r + \rho(t_0) - \rho(t_1) + \rho(t_2) - \ldots + \rho(t_{h-1}) \bmod p \quad (4)$$

By expanding the first of the two sequences we have:

$$q = r + \sum_{i=0}^{s_0} (-2)^i - \sum_{i=0}^{s_1} (-2)^i + \ldots + \sum_{i=0}^{s_{k-1}} (-2)^i \bmod p \\ = r + \sum_{i=0}^{s_0} (-2)^i + \sum_{i=s_1+1}^{s_2} (-2)^i + \ldots + \sum_{i=s_{k-2}+1}^{s_{k-1}} (-2)^i \bmod p$$

By expanding similarly the second assignment in Eq. 4, we have that the two sequences exist if:

$$\sum_{i=0}^{s_0} (-2)^i + \ldots + \sum_{i=s_{k-2}+1}^{s_{k-1}} (-2)^i \equiv \\ \equiv \sum_{i=0}^{t_0} (-2)^i + \ldots + \sum_{i=t_{k-2}+1}^{t_{h-1}} (-2)^i \pmod p \quad (5)$$

From Lemma A.4, we know that both sides are in the range $(-p, p)$. Thus, the two sides are congruent only if: i) they have the same sign and are equal, or; ii) they have different signs, and by summing p on the negative side, we get the positive side. Since each side is the sum of distinct powers of -2, case i) is only possible if the two sequences of steps are equal. To prove that case ii) is impossible, let us consider the case where the left side is negative (the other case is analogous). Because $p = 2^a$ for some $a \in \mathbb{N}$, and because $2^a = (-2)^a$ (if a is even[2]), Eq. 5 becomes:

$$\sum_{i=0}^{s_0} (-2)^i + \ldots + \sum_{i=s_{k-2}+1}^{s_{k-1}} (-2)^i + (-2)^a = \\ = \sum_{i=0}^{t_0} (-2)^i + \ldots + \sum_{i=t_{k-2}+1}^{t_{h-1}} (-2)^i$$

[2] If a is odd, $p = 2^a = (-2)^{a+1} + (-2)^a$, and the same considerations still hold.

However, because both sides are sums of distinct powers of -2, they can be equal only if the two sequences of steps are equal, which implies that there must be a t_i such that $t_i = a$. This is impossible because the number of steps can be at most equal to $\log_2(p) - 1 = a - 1$.

We thus proved by contradiction that there are no two sequences of steps leading to the same node and that, at each step, each node reaches only nodes that it did not already reach. □

A.2 Non-Power of Two Number of Nodes

The correctness proof in Sec. A.1 assumes p is a power of 2 (needed by the last part of Theorem A.5). If p is not a power of 2, the theorem only holds until the second-last step. If p' is the largest power of 2 smaller than p, in the second last step, the data sent by each node reached $p' - 1$ nodes. Thus, we need to guarantee that in the last step: i) no nodes receive data it already received; ii) each node reaches the remaining $p - p' - 1$ nodes. To guarantee property i), it is enough for each node to pre-compute the blocks b_i it will send at each step and if it would send a block twice, send that only in the last step.

To guarantee property ii), it is enough to prove that $\pi(r,s) = \pi(g,s) \Leftrightarrow g = s$. Indeed, if no two nodes reach the same node in the last step, then each node has reached each other node once. First, if r and g are both even, we need to prove that $r + \rho(s) \equiv g + \rho(s) \pmod{p}$. This implies $r - g \equiv 0 \pmod{p}$. However, because we have $r < p$ and $g < p$, this is only possible if $r = g$. The proof for odd r and g is analogous. Then, if r is even and g is odd, we have $r + \rho(s) \equiv g - \rho(s) \pmod{p}$, which implies $2\rho(s) \equiv g - r \pmod{p}$. However, we know from Lemma A.1 that $\rho(s)$ is always odd, and thus $2\rho(s)$ is even. Because r is even and g is odd, $g - r$ is odd. Thus, if p is even, this can only hold for $r = g$.

When p is even but not a power of 2, it is enough for each node not to send the same data block twice, thus not increasing the deficiency compared to the power of two case.

Last, if p is odd, we might have $\pi(r,s) = \pi(g,s)$ even if $r \neq g$, and also Lemma A.2 does not hold anymore. Consequently, some nodes might not reach all the nodes and, at a given step, might receive data from more nodes simultaneously, thus decreasing the performance. For this reason, we run the algorithm on $p - 1$ nodes, with the *"odd"* node sending data directly to each of the other nodes.

LitePred: Transferable and Scalable Latency Prediction for Hardware-Aware Neural Architecture Search

Chengquan Feng[1*] Li Lyna Zhang[2‡] Yuanchi Liu[1] Jiahang Xu[2] Chengruidong Zhang[2]
Zhiyuan Wang[1] Ting Cao[2] Mao Yang[2] Haisheng Tan[1‡]

[1]University of Science and Technology of China [2]Microsoft Research

Abstract

Hardware-Aware Neural Architecture Search (NAS) has demonstrated success in automating the design of affordable deep neural networks (DNNs) for edge platforms by incorporating inference latency in the search process. However, accurately and efficiently predicting DNN inference latency on diverse edge platforms remains a significant challenge. Current approaches require several days to construct new latency predictors for each one platform, which is prohibitively time-consuming and impractical.

In this paper, we propose LitePred, a lightweight approach for accurately predicting DNN inference latency on new platforms with minimal adaptation data by transferring existing predictors. LitePred builds on two key techniques: (i) *a Variational Autoencoder (VAE) data sampler* to sample high-quality training and adaptation data that conforms to the model distributions in NAS search spaces, overcoming the out-of-distribution challenge; and (ii) *a latency distribution-based similarity detection method* to identify the most similar pre-existing latency predictors for the new target platform, reducing adaptation data required while achieving high prediction accuracy. Extensive experiments on 85 edge platforms and 6 NAS search spaces demonstrate the effectiveness of our approach, achieving an average latency prediction accuracy of 99.3% with less than an hour of adaptation cost. Compared with SOTA platform-specific methods, LitePred achieves up to 5.3% higher accuracy with a significant 50.6× reduction in profiling cost. Code and predictors are available at https://github.com/microsoft/Moonlit/tree/main/LitePred.

1 Introduction

Hardware-aware Neural Architecture Search (NAS) has achieved remarkable success in automating the design of hardware-friendly deep neural networks (DNNs) in many tasks [8, 9, 22, 36, 51, 53, 56, 57]. This holds particular importance for crafting models suited to resource-limited edge platforms, such as mobile phones. However, to design low-latency models for diverse edge platforms, it remains of significant challenge to estimate accurately the inference latency of numerous models, which depends on multiple factors such as hardware, inference frameworks, and data precision [52].

Direct on-device measurement is expensive and impractical due to the very large search space of possible models. Consequently, many works [6, 7, 9, 18, 31, 54] have been proposed to predict the inference latency based on the given model architecture. However, these methods face two limitations. First, they typically rely on random sampling and require a significant amount of training data to achieve accurate predictions. This process is time-consuming, requiring *several days of data collection for a single platform* [18, 54]. Second, existing approaches focus on platform-specific prediction without considering the latency change due to various dynamics such as new hardware, various inference frameworks or new versions, and different data precision. We take nn-Meter [54], a cutting-edge approach, as an example. By developing latency predictors for the Xiaomi12 CPU platform using Onnxruntime (ORT) [35], it achieves an impressive 99.8% accuracy on the MobileNetV3 search space [8, 22]. However, the predictor drops to *0 accuracy* when the hardware is switched to a Xiaomi11 CPU. Thus, these approaches require rebuilding predictors for every new platform, which is prohibitively expensive and limits the applicability of hardware-aware NAS. This is especially challenging given the diverse edge platforms, which include a large number of mobile devices (e.g., 8318 heterogeneous smartphones [17]) and various inference frameworks (e.g., TFLite [20], ORT, OpenVINO [25], NCNN [60]).

Although a few platform-agnostic prediction works [30, 33] have attempted to address the high cost, they encounter various limitations. HELP [30] trains a latency predictor with meta learning [29, 47], but it conducts model-level prediction, which requires expensive redesigning and retraining of the meta predictor for new NAS search spaces. OneProxy [33] trains a latency monotonicity model to predict model latency rankings on different platforms. However, it only predicts latency rankings on new platforms, rather than the actual

*Work was done during the internship at Microsoft Research
‡Corresponding authors, (lzhani@microsoft.com and hstan@ustc.edu.cn)

values, which is often mandatory in practical deployments.

In this work, we propose LitePred, a lightweight approach for accurately predicting inference latency in hardware-aware NAS that eliminates the expensive rebuilding process required for new platforms. To handle the diverse model graphs in different NAS search spaces, LitePred performs kernel-level prediction and computes the model latency as the sum of the predicted latencies of all kernels [‡]. The key idea of LitePred is to identify the most similar pre-existing latency predictors for each kernel on new platform, and then finetune them with just a few adaptation samples to achieve high prediction accuracy. This makes LitePred a cost-effective solution for predicting latency, eliminating the bottleneck in hardware-aware NAS.

At the core of LitePred lies the principle that *knowledge from a pre-existing latency predictor for one platform can be transferred to new platforms that share similarities*. This is based on the fact that latency depends on key factors and their relationships, which can be learned by an accurate predictor. When a new platform shares similarities with a previous one, we can transfer the existing knowledge and adapt the predictor to capture new dynamics. For example, when using the same TFLite version 2.1, we can transfer a latency predictor trained on a Xiaomi11 CPU to a Pixel 6 CPU. By reusing the knowledge learned for TFLite 2.1, we only need a few adaptation data to learn behaviors on the Pixel 6 CPU.

LitePred maintains a knowledge pool of existing latency predictors, each being a 16-layer Multilayer Perceptron (MLP) network. Initially, it constructs base predictors for warmup platforms and stores them in the pool. When targeting a new platform, LitePred detects its kernels and identifies the most similar predictor for each kernel from the pool. These predictors are then finetuned with a small amount of adaptation data from the new platform to achieve accurate predictions.

LitePred faces two technical challenges. **(1)** First, constructing initial latency predictors and finetuning existing predictors require effective data collection. However, latency-dominant kernels, such as Conv and DWConv kernels, exhibit a multi-dimensional joint distribution in NAS search spaces, with dimensions that are highly correlated. This makes random and adaptive data sampling methods [54] ineffective due to the out-of-distribution problem, resulting in a large amount of useless training data and low accuracy. Direct sampling from search spaces can cause data leakage and limit the generalization ability in new NAS search spaces. **(2)** Second, the similarity between existing predictors and new platforms greatly affects the number of adaptation samples and the final prediction accuracy. However, the high diversity between different edge platforms, many of which are black boxes, poses a challenge in effectively detecting the most similar predictor.

VAE data sampler. To address the first challenge, we leverage the concept of Variational AutoEncoder (VAE) [27] and introduce the VAE data sampler. It consists of an encoder-decoder network, where the encoder compresses the multi-dimensional joint distribution in the search spaces into a latent space with a multivariate Gaussian distribution [19]. The decoder then reconstructs the Gaussian distribution back into the original distribution. By sampling from this distribution and decoding the data using the decoder, we can generate new training data that conforms to the original multi-dimensional distribution, addressing the out-of-distribution problem while preserving generalizability in new model search spaces.

Similar predictor detection. To identify the most similar pre-existing latency predictors for a target platform, we propose a *latency distribution-based similarity detection* method. The key idea is to compare the real latency distribution on the target platform with the predicted latency distributions by pre-existing latency predictors. For each kernel, we create a small representative set reflecting platform-specific optimizations. Then, we compute the distribution similarity by calculating the Kullback-Leibler divergence [13] between the real and predicted latencies for each latency predictor in the knowledge pool. The predictor with the highest similarity is selected.

We extensively evaluate LitePred on 85 edge platforms, including 10 hardware, 10 CPU frequencies, 5 commonly used edge inference frameworks, and 2 data precisions (FP32 and INT8). Before LitePred, platform-specific approaches take several days of data collection for a single platform, limiting evaluations to only a few platforms. To the best of our knowledge, we are the first to evaluate on such a wide range of platforms, with requiring $\sim$1 hour of measurements on each platform.

We summarize our key contributions as follows:

- We propose LitePred, the first to transfer pre-existing latency predictors and achieve accurate latency prediction on new edge platforms with a profiling cost of less than 1 hour.
- We introduce two key techniques: a VAE data sampler to collect effective multi-dimensional data and a latency-distribution similarity detector that identifies the most similar pre-existing latency predictors for black-box platforms.
- Extensive experiments on various platforms and 6 NAS search spaces demonstrate LitePred's effectiveness, achieving 99.3% accuracy and outperforming state-of-the-art latency prediction baselines. LitePred significantly improves accuracy on new edge platforms compared to platform-agnostic baselines, with up to a 77.5% improvement, and achieves up to +5.3% higher accuracy against platform-specific baselines while reducing profiling cost by 50.6×.
- By integrating LitePred with NAS, we discover better models with superior accuracy and lower latency than MobileNets. Our models surpass MobileNetV2, achieving an impressive up to 4.4% higher accuracy on ImageNet.

[‡]A kernel represents an execution unit, it may be either a single primitive operator or a fusion of multiple fused operators, similar to nn-Meter [54].

(a) Same device equipped with various inference frameworks					
Xiaomi11	TFLite 2.1	TFLite 2.7	ORT	NCNN	Mindspore
	289.4ms	271.8ms	451.8ms	165.2ms	209.2ms
(b) Different devices under the same inference framework					
TFLite2.1	Xiaomi11CPU	Xiaomi11GPU	Xiaomi12CPU	Pixel5CPU	Pixel5CPU*
	289.4ms	36.3ms	244.4ms	300.1ms	568.6ms
(c) Different data precision under the same platform					
Xiaomi11	TFLite 2.1		TFLite 2.7		
	FP32	INT8	FP32	INT8	
	289.4ms	75.4 ms	271.8ms	77.8ms	

Table 1: The vastly different inference latency of ResNet50 on diverse platforms. *: we set a lower CPU frequency.

2 Background and Motivations

2.1 Factors that impact the latency

To investigate factors impacting model latency on diverse edge platforms, we start by conducting measurements. We monitor different platforms, varying devices, CPU frequency, inference frameworks, and data precision. Our sample model is ResNet50, and we deploy it on each platform to measure the inference latency. The results in Table 1 show that model latency is heavily dependent on the following factors:

Devices. It has been widely observed that the same model can exhibit varying inference latencies on different devices. As shown in Table 1(b), ResNet50 runs 8.0× faster on a GPU than a CPU on the Xiaomi 11. Surprisingly, even among mobile phones with ARM CPUs, latency differences for the same model can exceed 10%.

Inference frameworks: The effectiveness of framework optimizations and how well they align with the underlying hardware can significantly impact the overall latency. As shown in Table 1(a), on Xiaomi11 CPU, deploying ResNet50 with NCNN yields a 1.7×, 1.6×, 2.7× and 1.3× faster speed than TFLite 2.1, TFLite 2.7, ORT and Mindspore Lite [24], respectively. Notably, even using the same TFLite framework, different versions can introduce significant latency differences.

CPU frequency. In addition to the device type and inference frameworks, CPU frequency has a significant impact on inference latency, as shown in Table 1(b). Our experiments on the Pixel5 demonstrates that even minor changes in CPU frequency (2.2 GHz to 1.9 GHz) can substantially affect latency.

Data precision. Finally, edge platforms support different data precisions, which lead to different memory and computation costs, thereby affecting inference latency. Our experiments focus on the widely supported FP32 and INT8, as shown in Table 1(c). When we switch the data precision of ResNet50 from FP32 to INT8, we observe a 3.8× and 3.5× reduction in latency in TFLite 2.1 and TFLite 2.7, respectively.

In summary, these results indicate that *any alteration in the device, framework, CPU frequency, or data precision within a platform can significantly affect model inference latency.*

2.2 Challenges to platform-specific prediction

However, these findings pose a significant challenge to current platform-specific latency prediction approaches [6, 7, 18, 31, 32, 54]. Trained predictors cannot be directly applied to new platforms, requiring a costly rebuilding process for adaptation.

Predicted Platform		New Platforms			
Xiaomi12 CPU, ORT		Xiaomi11 CPU, ORT		Xiaomi12 CPU, TFLite 2.1	
Accuracy	RMSE	Accuracy	RMSE	Accuracy	RMSE
99.84%	12.2 ms	**0%**	**57.6 ms**	**0%**	**220.6 ms**

Table 2: Directly apply nn-Meter's predictors on new platform results in a poor prediction accuracy of 0%.

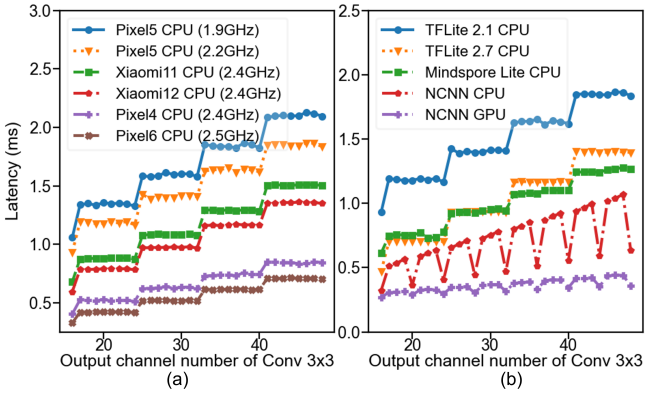


Figure 1: (a) Under the same TFLite 2.1, mobile ARM CPUs exhibit a similar staircase latency pattern; (b) On Xiaomi11, various frameworks exhibit similar latency patterns. Config: HW=56, Stride S=1, Input channel C_{in}=16.

Poor prediction accuracy on new platforms. nn-Meter [54] leads in device-specific latency prediction. By building predictors with 42k training data on Xiaomi12 CPU with ORT, it achieves an impressive 99.8% accuracy in predicting MobileNetV3 search space [8]. However, such highly accurate predictors perform poorly on new platforms. Table 2 shows a significant drop in accuracy to 0 when transitioning to Xiaomi11 CPU or switching to TFLite 2.1 framework.

The reason is that the objective of device-specific latency prediction is to minimize the regression errors between the predicted latency, y', and the actual latency, y. As a result, when there are significant changes in the actual latency, y, on a new platform, these device-specific predictors experience a large regression error and become unreliable.

Expensive rebuilding cost. The accuracy drop to 0 mandates an expensive rebuild of the platform-specific latency prediction method, demanding substantial training data collection on the new platform. Unfortunately, this data collection process can be extremely time-consuming, taking several days to complete for a single platform. For instance, nn-Meter usually collects around 42k kernel data points, requiring 2.5 days on a Google Pixel 4 CPU. Such high cost makes platform-specific approaches unfeasible and unscalable for handling the vast number of edge devices and various inference frameworks.

2.3 Opportunities

Observations and insights. Although numerous factors can impact model latency, we observe that similar latency behavior patterns persist across diverse platforms. This is primarily because many inference frameworks share a common goal of

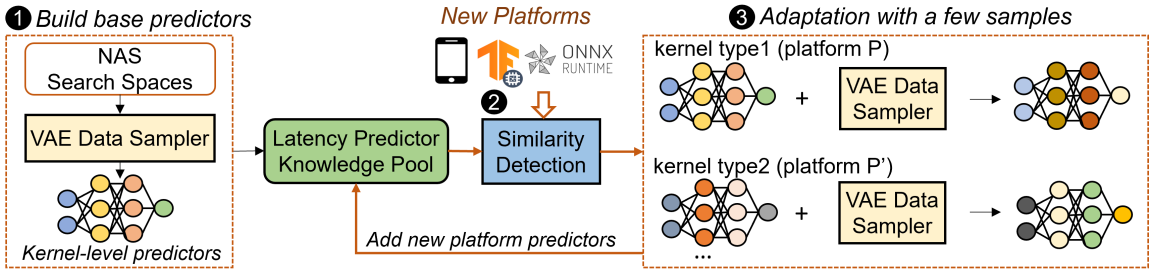


Figure 2: The overview of LitePred. (1) LitePred starts with building accurate base predictors for warmup platforms and stores them in a knowledge pool. (2) For a new platform, we match the most similar latency predictors for each kernel from knowledge pool through similarity detection. (3) We adapt identified latency predictors with a few samples from the new platform.

maximizing the utilization of underlying device resources to minimize DNN inference latency. Consequently, some critical optimization techniques are commonly applied across these frameworks. An example is operator fusion, which combines multiple operators into a single one without storing intermediate results in memory. Moreover, many frameworks leverage Neon optimizations [3] specifically designed for ARM CPUs. TFLite and Mindspore employ CPU kernels optimized for the ARM Neon instruction set, while NCNN goes a step further with assembly-level optimization for ARM Neon.

To showcase this, we conduct two experiments. In the first experiment, we keep the inference framework fixed at TFLite 2.1 and measure the latency of Conv 3x3 on various mobile phones. As shown in Fig. 1(a), despite significant absolute latency differences among various mobile CPUs, they all exhibit the same staircase latency pattern as output channels increases. In the second experiment, we explore whether similar patterns exist between various inference frameworks. Fig. 1(b) show that TFLite 2.1, TFLite 2.7 and Mindspore Lite all exhibit a very similar staircase latency pattern on CPU, while NCNN CPU and NCNN GPU display a different step pattern.

We thus reveal two key observations: *(i): mobile hardware of the same type exhibit similar latency patterns; (ii) despite varying optimizations and implementations, there exist inference frameworks display similar latency patterns.*

Opportunities. These observations motivate us to develop platform-transferable latency predictors that can eliminate the need for a costly rebuilding process when introducing new devices or frameworks in the target platform. Our insight is to train latency predictors to well learn the latency patterns for a specific platform and then transferring the shared common knowledge to a new platform. By finetuning the latency predictor with a few training samples to adapt to any new factors in the platform, we can achieve high prediction accuracy.

3 System Design

Overview. The observations in Section § 2 motivate LitePred, a lightweight approach for predicting the latency of arbitrary DNN models, through transferring pre-existing latency predictors with minimal adaptation cost across diverse edge platforms. Fig. 2 illustrates the system overview.

To start, we select a few edge platforms (e.g., 5) as warmup ones and build a set of precise kernel-level latency predictors from scratch (§ 4). Utilizing a VAE data sampler for high-quality training data collection, these predictors learn accurate latency patterns for each warmup platform and are stored in a "knowledge pool". When targeting a new platform, we perform kernel detection [54] to identify all possible kernels. Then, we identify the most similar latency predictor for each kernel type from the knowledge pool and finetune them swiftly using VAE data sampler with just a few adaptation data (§ 5). We use these finetuned latency predictors to predict kernel latencies, and sum them up as the final model latency.

Design choice. Our ultimate objective is to effectively serve hardware-aware NAS on a wide range of diverse edge platforms with LitePred. Specifically, LitePred aims to accurately predict the inference latency of any DNN models on the target platform within NAS search spaces. Guided by this goal, we design the system based on the following principles.

• *LitePred predicts latencies for DNN models in NAS search spaces*. In theory, DNN models could be arbitrary, resulting in a vast number of possibilities that complicate latency prediction. Yet, many models are inferior in accuracy and therefore disregarded. In our work, we focus on top-tier model search spaces in NAS, crafted by AI experts and known for their success in finding accurate models. We collect 5 CNN NAS search spaces and 1 vision transformer NAS search space for our final evaluation of latency prediction. These search spaces include a vast number of 10^{20} high-quality models.

• *The accuracy of initial base latency predictors is crucial for successful transfer across devices.* Our approach starts by training dedicated, accurate latency predictors for warmup platforms, which learn the latency patterns resulting from various inference optimizations. These predictors serve as a source of knowledge for transferring to new platforms. If the base predictor fails to achieve high accuracy, transferred predictors will have inferior performance. For example, a more accurate predictor (90% base accuracy) outperforms an inferior one (67.1% base accuracy) with 1.6× less adaptation data to achieve the same finetuned accuracy.

• *LitePred detects the most similar pre-existing latency predictors.* Instead of randomly selecting a predictor from the

knowledge pool, LitePred uses a similarity-based approach to identify the most similar predictor for a new platform. This allows for the reuse of learned knowledge and leads to reduced adaptation data samples and lower measurement costs. Intuitively, the most similar predictor needs the least amount of adaptation data and delivers the best prediction accuracy.

• *We predict latency at the kernel-level to simplify predictors transfer to new platforms.* While HELP [30] and One-Proxy [33] use model graphs to predict latency, LitePred takes a kernel-level latency approach. We divide a model into kernels and sum up their predicted latencies as the model latency, since kernels are sequentially executed on edge platforms. This is advantageous as kernels are the basic scheduling units in inference frameworks, covering framework optimizations (e.g., operator fusion) and specific kernel algorithms (e.g., Winograd convolution [28]). Latency prediction at the kernel level simplifies cross-platform predictor detection and requires less adaptation data than model-level prediction.

Technical challenges. LitePred faces two major technical challenges: *(i) Given the exponentially large NAS search spaces, how to efficiently collect high-quality data for training base predictors and finetuning pre-existing predictors? (ii) How to find most similar predictors and adapt them on new platforms with minimal profiling costs?*

4 Build Accurate Base Latency Predictors

In this section, we introduce our approach for training base latency predictors on warmup platforms and propose VAE data sampler to efficiently collect high-quality training data.

4.1 NAS search spaces and the challenges

We begin by studying model distributions in NAS search spaces. Then, we discuss the challenges in data collection.

NAS search space collection. We collect 5 widely-used CNN and 1 vision transformer NAS search space, including OFA-MobileNetV3 [22], ProxylessNAS [9], OFA-ResNet [8], BigNAS [53], FBNetV3 [14] and AutoFormer [10]. These search spaces are of high quality (i.e., models have the potential for high accuracy) and specifically tailored to edge-regime DNNs. In total, the 6 search spaces contain an impressive of 10^{20} different models, representing a vast prediction scope.

Multi-dimensional distribution of kernel configurations. As elaborated in Section § 3, LitePred builds latency predictors at kernel level. To construct these kernel latency predictors, we need to collect a large amount of training data consisting of configuration and latency pairs for each kernel type. However, this presents a practical challenge due to the vast configuration space of common CNN model kernels, such as Conv+bn+relu. This kernel has five primary configuration dimensions (*input height and width HW, kernel size K, strides S, input channels C_{in}, and output channels C_{out}*), resulting in a large number of possible configurations. Profiling on-device latency for every possible configuration on every target platform can be prohibitively expensive.

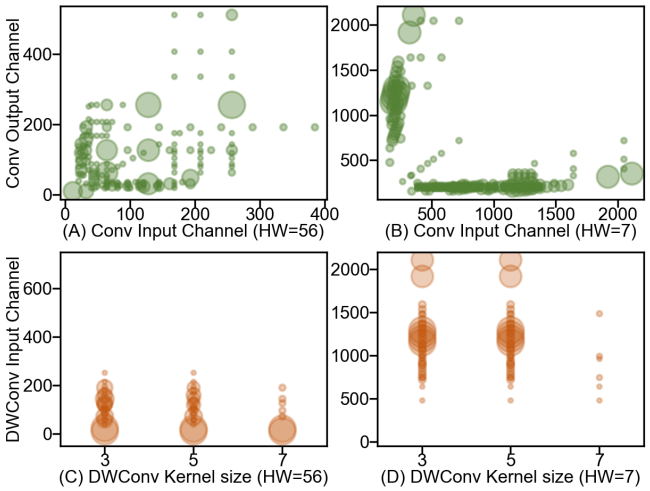


Figure 3: The multi-dimensional distributions of Conv/DW-Conv configurations in 5 widely-used NAS search spaces. (AB): C_{in} vs. C_{out} exhibit different patterns under different HW. (CD): The kernel size KS and C_{in} of DWConv also exhibit different distributions under different HW. Larger circle size indicates that the configurations have larger frequency.

Fortunately, we observe that each kernel configuration displays a unique and smaller multi-dimensional joint distribution in NAS search space. To illustrate this, we collect all the configurations $(HW, K, S, C_{cin}, C_{out})$ for Conv and DWConv kernels from 5 CNN search spaces. As shown in Fig. 3(AB), the values of C_{in} and C_{out} of Conv kernels under different HW display distinct distributions. Fig. 3(CD) demonstrates that kernel size KS also has a different distribution under varying channel numbers C_{in} and input size HW. *This allows us to gather training data conforming to the distribution, rather than collecting data for all possible configurations.*

Challenges in collecting high-quality training data. However, gathering training data aligned with the multi-dimensional distribution poses a challenge. Random sampling can result in the collection of irrelevant data and lower accuracy. nn-Meter introduces an adaptive data sampler that constructs a probability distribution per dimension and performs independent sampling. However, since the dimensions are highly correlated (Fig. 3), independent sampling can still trigger out-of-distribution issues. While uniform sampling from search spaces resolves this, it introduces data leakage problems by revealing evaluation data during training, limiting the generalizability to new search spaces. Therefore, we call for a new sampling approach. In next section, we introduce VAE data sampler to address all these challenges.

4.2 Efficient VAE data sampler

To collect high-quality training and adaptation data, we divide the process into two specific tasks: *(i)* learning the multi-dimensional joint distribution of different kernels, and *(ii)* generating new data based on the distribution. In this section, we take inspiration from the concept of Variational Autoencoders (VAE) [27] and propose a VAE data sampler.

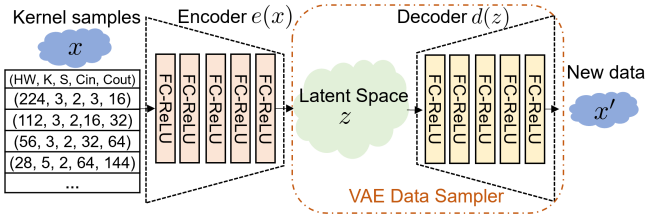


Figure 4: Our VAE data sampler for collecting training data.

VAEs are a type of generative neural network, originally popularized for their applications in image and text generation tasks [40, 41, 43]. They learn to represent complex data distributions by encoding input data into a lower-dimensional latent space and then decoding it back to the original space. VAEs comprise two primary components: an encoder network that maps input data to the latent space, and a decoder network that reconstructs the input data from this latent space.

Inspired by the encoding and decoding process of VAEs, we propose training an encoder-decoder VAE model to learn the multi-dimensional distribution for each kernel. Then, we utilize the trained decoder as our data sampler, generating new samples that adhere to each kernel's distribution within the NAS search space. Next, we will discuss the application of VAEs for sampling data to train kernel latency predictors.

VAE model design. Fig. 4 illustrates the design of our VAE model, which consists of a 5-layer Linear encoder and a 5-layer Linear decoder with a hidden dimension of 256. The latent space dimension size is empirically set to 128. For each kernel type (e.g., Conv+bn+relu), we carefully sample several configurations as input x, representing its distribution in search spaces. The encoder maps input x to the latent space z, which is assumed to follow a multivariate Gaussian distribution [19]. The decoder then maps from the latent space back to the input space, generating data x' that adhere to the same distribution as the original input x. Note that x and x' follow the same distribution, but with minimal values overlap.

Input data x. To train the VAE model, it is crucial to collect representative configurations from search spaces, which characterizes the distribution for each kernel and acts as the target distribution for the input data x, which the VAE aims to learn.

However, collecting such data is non-trivial, as many latency-dominated configurations have low frequencies in NAS search spaces. For instance, Conv and DWConv kernels' latency is significantly larger in the first 3 layers, contributing up to 44% of model latency. Accurate prediction of these configurations is crucial for precise latency prediction. However, they represent only 1.52% in NAS search spaces, with most configurations in the middle layers. Naive uniform sampling can result in the omission of latency-dominating data.

We tackle this challenge by performing data normalization on latency-dominated large kernels. We assign a larger weight to Conv and DWConv related kernels. Also, since the number of kernel configurations differs for each search space, we collect the x proportionally based on their relative quantities.

Training the VAE model. After collecting the representative distribution x for each kernel type, the next step is to train the VAE model that learns the multi-dimensional distribution for each kernel and generates new configurations x' based on it. The training objective aims to minimize the distance between the decoder output x' and the encoder input x.

We follow the original VAE work [27] and use two loss functions: a Mean Squared Error (MSE) reconstruction loss [2] and a Kullback-Leibler (KL) divergence [1]. The reconstruction loss measures the difference between the decoder output x' and the original input data x. The KL divergence measures the difference between the encoder output distribution in the latent space and the standard normal distribution. We train the VAE model to minimize the total loss function. After the training is done, the decoder is able to map the latent space z back to the original kernel configuration distribution.

Generating kernel configuration data via the decoder. Once the VAE is trained, we can employ the decoder to generate new kernel configuration data x' that closely aligns with the distribution x in NAS search spaces. As illustrated in Fig. 4, to generate N configuration data for Conv+bn+relu kernel, we first sample N vectors from the latent space, which follows a multivariate Gaussian distribution. We then pass these N vectors through the decoder, resulting in the generation of N new configurations for Conv+bn+relu. Note that the original decoded configuration data x' are continuous values. We apply a straightforward round-to-nearest strategy, mapping continuous values to the closest valid discrete value.

4.3 Build transferable base latency predictors

The above VAE data sampler efficiently samples high-quality training data for each kernel. This section outlines designing latency predictor models that accurately predict latency on warmup platforms using the collected data. Moreover, these predictors are designed for easy transfer to new platforms.

Conventional platform-specific prediction methods [6, 7, 11, 54] typically rely on decision-tree-based machine learning regressors to create latency predictors. For example, models like RandomForest and XGBoost regression are often used, which fit the training data by minimizing the difference between actual and predicted latency values. However, these decision-tree-based regressors require a complete retraining process, making it impossible to reuse fitted predictors for transferring to new platforms.

To achieve accurate and transferable latency prediction, we introduce a DNN model. DNN models have demonstrated strong performance in transfer learning tasks [39, 59]. Specifically, our latency model is a 16-layer Multilayer Perceptron (MLP) network, a small model with $\sim$1 million parameters.

We now describe the training process. For each kernel, we measure the latency of VAE sampled kernel configurations on warmup platforms. The prediction features are kernel configurations, FLOPs, and parameter size, while the corresponding inference latency is the label. Our training objective is to minimize the Mean Absolute Percentage Error (MAPE)

loss between the predicted and actual latency values. We use MAPE loss because we follow previous work [18, 54], which uses ±10% accuracy as the evaluation metric. This metric calculates the percentage of kernels with predicted latency within ±10% error. Minimizing MAPE loss aligns with our goal of maximizing ±10% accuracy.

LitePred trains base predictors for five randomly selected warmup platforms listed in Table 3 and stores them in the knowledge pool, as illustrated in Fig. 2. In the following section, we will describe our method for reusing and transferring these existing predictors to new platforms.

5 Transfer Predictors to New Platforms

For a new edge platform, LitePred addresses the challenge of identifying the most similar kernel predictors from the knowledge pool. It then transfers these pre-existing predictors to the new platform using minimal adaptation data.

5.1 Similar predictor detection for each kernel

Identifying the most similar latency predictor for each kernel on new platforms is challenging as most edge platforms are black boxes. OneProxy [33] suggests using Spearman's Rank Correlation Coefficient (SRCC) to assess the statistical dependence between latency rankings of models. However, this method has limitations. First, the SRCC is based on the total model latency, which might not accurately identify the most similar platform. In contrast, our experiments shows that the most similar platform predictor varies by the kernel type. Second, it depends on random model sampling to evaluate platform similarity, which is typically sparse and fails to reflect specialized optimizations on the target platform.

Overview of our approach. In our work, we introduce a lightweight latency-distribution based similarity detection approach. Instead of focusing on the similarity of the entire model, we detect similarity at the kernel level since on-device optimizations are usually implemented at this level. *A pre-existing kernel latency predictor is considered similar for the target platform if its predicted latency displays a similar distribution to the real latency.* To achieve this, we design a small set of representative kernel configurations that capture both kernel distribution and platform-specific optimizations, rather than randomly sampling or relying solely on the VAE data sampler. This enables us to identify the most similar latency predictors more effectively while keeping the approach lightweight.

Fig. 5 illustrates the overall process. We design a small set of representative configurations for each kernel, and measure the actual latency of under these configurations on the platform, denoted as Y_r. For each latency predictor in the knowledge pool, we predict the latency of these configurations as Y_p. We then calculate the similarity score using KL divergence between the actual latency Y_r and predicted latency Y_p. Finally, we return the predictor with lowest KL divergence (i.e., the highest similarity score).

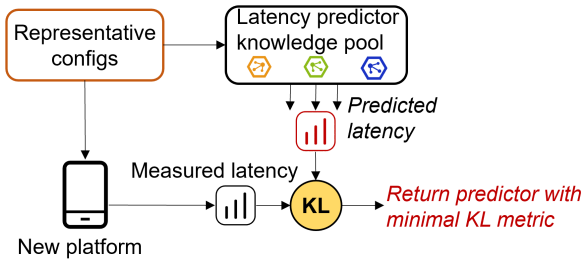


Figure 5: Our proposed similarity detection to identify the most similar predictor for each kernel type on new platforms.

Representative data for computing similarity. The effectiveness and cost of our similarity detection technique heavily rely on the quality of representative data. We now introduce how we design it for computing predictor similarity between diverse platforms. We consider two types of kernel configurations: *(i)* configurations in the search spaces that reflect the underlying distribution; and (ii) specifically designed configurations that capture the latency patterns on the target platform.

Type (i) data collection employs our VAE data sampler. To collect type (ii) data that reflects specific on-device optimizations, we adopt a simple but effective fine-grained approach. This is inspired by Fig. 1, where it reveals that device-specific staircase latency patterns necessitate dense profiling to uncover. In contrast, VAE and random samplers exhibit sparse coverage, which can distort the profiled latency patterns.

Specifically, we select an initial configuration from search spaces per kernel. We fix all dimensions except one, and perform fine-grained sampling on that dimension. We sample 16 continuous points in the channel number dimension and enumerate {1, 3, 5, 7} in the kernel size dimension. For example, for Conv+bn+relu kernel with the configuration (56, 3, 2, 16, 16), we fix the HW, K, S and C_{in} dimensions, and generate 16 continuous points (e.g., 16 to 32) in the output channel dimension. Then, we fix the HW, S, C_{in}, and C_{out} dimensions, and enumerate {1, 3, 5, 7} in the kernel size dimension.

Similarity score. To compute the similarity of latency distributions between each pre-existing predictor and a target platform, we use the Kullback-Leibler (KL) divergence, a widely used statistical metric for comparing probability distributions. First, we obtain the real kernel latency for representative configurations as Y_r. Next, for each pre-existing predictor i that is trained or finetuned for a different platform, we obtain its predicted latencies Y_p^i under the representative configurations. Finally, we compute the KL divergence between the probability distributions P and Q that correspond to Y_p^i and Y_r, respectively. The KL divergence is defined as follows:

$$D_{KL}(P||Q) = \sum_{y_r \in Y_r, y_p \in Y_p} P(y_p) \log \frac{P(y_p)}{Q(y_r)}$$

For each kernel, the predictor with the lowest KL divergence is chosen for adaptation to the new target platform. Note that the cost of our similarity detection method comes from the profiling overhead of the representative configurations, which typically takes less than 10 minutes.

5.2 New platform adaptation

After identifying the most similar latency predictors for all kernels, we introduce the method for adapting these predictors to the new platform through a finetuning process.

Our method, illustrated in Fig. 2, begins by sampling adaptation configurations on the target platform using a VAE data sampler and measuring their latencies. We then finetune the identified predictors using the sampled data, updating their weights to improve accuracy on the new platform.

Increasing adaptation data typically improves prediction accuracy but raises profiling costs. Our experimental results indicate that a small number of samples (less than 500) can achieve high accuracy when finetuning on a new platform. This is because we identified very similar latency predictors, which greatly reduces the required amount of adaptation data. As a result, our approach is significantly more cost-effective than traditional platform-specific prediction methods.

6 Implementation

Training the VAE. For large kernels such as Conv+bn+relu and DWConv+bn+relu, we collect 2000 configurations to train the VAE model as described in Sec.§ 4.2. For smaller kernels such as Squeeze&Excitation (SE) [23] and bn+relu, we collect 500 configurations. Our VAE model is implemented based on the Pytorch implementation [44]. We train the model for 2k epochs. During each training step, we randomly sample a batch of 16 configurations based on their frequency. We use the Adam optimizer [26] with a learning rate of 0.001 and decay rates of (0.9, 0.999).

Build base latency predictors with VAE decoder. After completing VAE training, we utilize the learned latent space and the decoder to gather training data for constructing base latency predictors for each kernel type. Specifically, to sample N configurations, we generate N 128-dimensional vectors from multivariate Gaussian distribution and feed them into the decoder. The N decoder outputs are used as training data. We sample up to N=10k configurations for Conv and DWConv related kernels and N=1000 for small kernels.

For each kernel type, we generate the corresponding model graph based on the sampled configurations. Then, we measure the inference latency on 5 random warmup platforms to gather training data for building base latency predictors. Each predictor is trained for 350 epochs using the AdamW optimizer with a cosine learning rate scheduler, and the initial learning rate is set to 0.001. The training cost is feasible. On an Nvidia RTX 2080Ti, it takes 26 minutes to train latency predictors for large kernels and only 6 minutes for small kernels.

Transfer to new platforms. For a new platform, we first conduct kernel detection [54] to identify all possible kernels. Then, we perform similarity detection for each kernel to find the most similar predictor from the knowledge pool. Specifically, we identify similar predictors for Conv and DWConv kernels using 400 representative data. For small kernels, we reuse the detected platform for Conv kernels and use the corresponding predictor, as small kernels usually have same platform detection results as Conv kernels.

Device	CPU	GPU	CPU Frequency
Pixel 4	Qualcomm Snapdragon 855	Adreno 640	2.4GHz, 2.1GHz
Pixel 5	Qualcomm Snapdragon 765G	Adreno 620	2.2GHz, 1.9GHz
Pixel 6	Google Tensor SoC	Mali-G78	2.5GHz, 2.2GHz
Xiaomi 11	Qualcomm Snapdragon 888	Adreno 660	2.4GHz, 2.1GHz
Xiaomi 12	Qualcomm Snapdragon 8 Gen 1	Adreno 730	2.4GHz, 2.1GHz
Inference engines	TFLite 2.1, TFLite 2.7, NCNN, Mindspore Lite, Onnxruntime		
Precision	FP32, INT8		

Table 3: Our **85** evaluated platforms, including 10 different hardware and CPU frequencies, 5 popular inference frameworks on edge and 2 data precision.

To finetune predictors, we use our VAE data sampler to generate a few configurations and measure their latency. We sample 100 configurations for small kernels and 500 for large kernels like Conv and DWConv. If accuracy is unsatisfactory, we iteratively sample 500 more data points per round until the desired accuracy is reached. During finetuning, we use detected kernel weights as initial weights and finetune for 300 epochs. We follow the same training settings as the base predictor, but with a smaller learning rate of 0.0005.

7 Evaluation

7.1 Experiment Setup

Platforms. We evaluate LitePred on a wide variety of edge platforms, as detailed in Table 3. Our evaluation includes 10 mobile hardware (CPU and GPU), 5 popular inference engines for edge devices, and 2 data precision options (FP32 and INT8 for CPU devices). Besides, we test each CPU device under two frequencies. In total, we evaluate 85 different platforms, a significantly larger number than previous works. This large-scale evaluation is achieved by our proposed lightweight, scalable, and transferable latency prediction paradigm. Without it, the cost of such an evaluation would be extremely expensive.

Evaluation datasets. To evaluate the effectiveness of LitePred in hardware-aware NAS, we build latency datasets using the 6 high-quality NAS search spaces (ref Section 4.1). We randomly sample 4k models from each search space and measure their latency on our 85 platforms. In total, the dataset contains 1.86 million model and latency pairs.

For each platform, we measure the model inference latency by performing a warmup of 10 inference runs and then calculating the average latency over 50 subsequent inference runs. For CPU platforms, we set the CPU frequency to the target value and measure the corresponding latency. For INT8 precision, we measure the INT8 latency on TFLite platforms, as TFLite has good support for this precision. Specifically, we use TFLite's official tools [46] to quantize models to INT8 precision and follow the standard process to measure latency.

Comparison baselines. We compare LitePred with state-of-the-art latency prediction methods by implementing two strong baselines: (1) nn-Meter [38, 54], a platform-specific method; and (2) HELP [30], a platform-agnostic method.

Metrics. We evaluate three metrics: Root Mean Square Error (RMSE), prediction accuracy, and profiling cost. RMSE quantifies the errors between predicted and real latency. Since RMSE can be highly influenced by the range of real latency values, we also report prediction accuracy. Following related works [18, 54] , we measure prediction accuracy as the percentage of models whose predicted latency error falls within $\pm 5\%$ and $\pm 10\%$ of the real measured latency. For profiling cost, we report both the number of sampled kernels and the total on-device measurement time cost.

7.2 Key findings

Before presenting detailed results, we summarize our valuable findings and insights gained from extensive experimentation:
(i): Despite differences in hardware, software inference engine implementations, data precision, and CPU frequencies among edge platforms, LitePred successfully transfers pre-existing latency predictors and achieves 99.3% accurate latency prediction across diverse edge platforms.
(ii): On most edge platforms, LitePred requires $<$1 hour of adaptation profiling cost. Compared to state-of-the-art platform-specific baselines, it achieves up to 5.3% higher accuracy and reduces profiling costs by 50.6×.
(iii): The detected most similar predictor's corresponding platform varies depending on the kernel type. (Table 5).
(iv): The data required for adaptation depends on the similarity between the detected platform predictor and the target platform. A highly similar predictor requires minimal data, while less similar predictors demand more data for finetuning.
(v): By enabling hardware-aware NAS, LitePred facilitates the development of efficient DNN models for various edge platforms. Remarkably, our searched models outperform MobileNetV2 by up to 4.4% in accuracy on the ImageNet dataset.

7.3 Evaluation on diverse edge platforms

7.3.1 Comparison with baseline methods

We demonstrate the effectiveness of LitePred by comparing with state-of-the-art latency prediction baselines.
Setup. We select 4 out of 85 platforms for comparison and use the higher frequency of the CPU device. By default, we use FP32 data precision unless stated otherwise. Since HELP performs model-level latency prediction, a separate meta latency predictor needs to be designed and trained for each search space. For simplicity, we choose MobileNetV3 search space, which HELP already supports, as the evaluation dataset.

For nn-Meter, we use the official code [38] to sample training data and train kernel latency predictors for each target platform. For HELP, we initially adapt its meta predictor using 10 randomly sampled models from target platform. However, the accuracy is poor. Therefore, we add two edge platforms (i.e., Pixel5 CPU with MindSpore, and Xiaomi11 GPU with NCNN) to retrain the meta latency predictor for better prediction. During evaluation, we increase the number of adaptation models and allow the profiling cost to be the same as ours. We refer to this improved implementation as HELP*.

Platform	Method	Train Data	Cost	RMSE	Prediction Acc ±5%	Prediction Acc ±10%
Xiaomi11 CPU Mindspore	HELP	10 models	12.44s	6.6 ms	11.5%	22.5%
	HELP*	1030 models	0.35h	4.1 ms	39.3%	48.7%
	nn-Meter	234997 kernels	16.23h	0.8 ms	78.0%	98.9%
	Ours	4800 kernels	**0.35h**	**0.4 ms**	**95.4%**	**100%**
Xiaomi11 CPU NCNN	HELP	10 models	10.87s	9.5 ms	15.4%	23.0%
	HELP*	3000 models	0.88h	6.7 ms	37.1%	49.1%
	nn-Meter	169305 kernels	20.17h	0.4 ms	96.4%	100%
	Ours	11400 kernels	**0.88h**	**0.3 ms**	**99.5%**	**100%**
Pixel 5 GPU TFLite 2.7	HELP	10 models	2.66s	1.2 ms	13.9%	28.0%
	HELP*	2500 models	0.62h	0.8 ms	51.6%	61.1 %
	nn-Meter	104996 kernels	7.94h	0.8 ms	37.7%	95.8%
	Ours	11900 kernels	**0.62h**	**0.3 ms**	**95.9%**	**99.9%**
Pixel 5 GPU NCNN	HELP	10 models	16.41s	12 ms	7.9%	16.8%
	HELP*	2100 models	0.96h	7.5 ms	33.5%	50.8 %
	nn-Meter	397384 kernels	48.60h	1.6 ms	52.2%	94.7%
	Ours	17400 kernels	**0.96h**	**0.9 ms**	**92.6%**	**100%**

Table 4: LitePred outperforms both state-of-the-art platform-specific and platform-agnostic baselines by achieving higher prediction accuracy with significantly lower sampling costs.

Results and analysis. Table 4 summarizes the comparison results. LitePred consistently outperforms both platform-specific and platform-agnostic prediction baselines by achieving higher prediction accuracy and lower RMSE, all while requiring significantly lower sampling costs. Compared to platform-agnostic methods, LitePred conducts much more precise latency prediction on new platforms, with over 92% of models whose predicted latency error falls within $\pm 5\%$ of the real latency. In contrast, HELP achieves only 12% accuracy on average, and the improved HELP* achieves 40.4%.

Furthermore, LitePred achieves much higher $\pm 5\%$ prediction accuracy than platform-specific baseline while reducing prediction costs to within 1 hour. Compared to nn-Meter, LitePred speedups the prediction cost by 46.4×, 22.9×, 12.8× and 50.6× on the four platforms, respectively. These results demonstrate the remarkable effectiveness of LitePred in terms of both prediction accuracy and efficiency.

7.3.2 Transfer to diverse new platforms

We now evaluate the effectiveness of LitePred in latency prediction on a wider range of new platforms by transferring pre-existing predictors. Due to space limit, we select 14 different edge platforms, covering a variety of hardware types, frequencies, data precisions, and inference frameworks.
Setup. We conduct two experiments to demonstrate that LitePred can adapt well to any new edge platforms. **(a)**: we select the most similar kernel predictors from knowledge pool, regardless of the hardware or inference engines for the detected predictors. **(b)**: To demonstrate that LitePred can transfer latency predictors across different hardware and inference engines, we exclude the predictors with the same inference engines as target platform from the knowledge pool and select the most similar predictors from the remaining ones.
Results and analysis. Table 5 summarizes the detailed results, including the detected similar kernel predictors, the required data and time cost for adaptation, and the prediction accuracy over our benchmark dataset on each target platform.

Platform	Similar Platforms		#Adaptation		Prediction Accuracy	
	Conv kernel	DWConv kernel	Data	Cost	±5%Acc	±10%Acc
(a) Selecting most similar kernel predictors from the whole knowledge pool						
Xiaomi11CPU, ORT	Xiaomi12 CPU, ORT	Xiaomi12 CPU, ORT	1400	0.48h	90.5%	98.9%
Pixel5 GPU, NCNN	Xiaomi11 GPU, NCNN	Xiaomi11 GPU, NCNN	17400	0.96h	84.3%	99.1%
Xiaomi11 CPU, MindSpore	Pixel5 CPU, MindSpore	Xiaomi12 CPU, MindSpore	4800	0.35h	90.4%	99.9%
Xiaomi11 GPU, TFLite 2.7	Xiaomi12 GPU, TFLite 2.7	Xiaomi12 GPU, TFLite 2.7	11000	0.17h	83.7%	98.6%
Xiaomi11 CPU, NCNN	Xiaomi11 CPU, MindSpore	Pixel5 CPU, NCNN	11400	0.88h	80.3%	98.9%
Pixel6 CPU, TFLite 2.1	Xiaomi12 CPU, TFLite 2.1	Xiaomi12 CPU, TFLite 2.1	3500	0.16h	79.4%	100%
Pixel5 CPU, TFLite 2.7	Xiaomi11 CPU, TFLite 2.7	Xiaomi11 CPU, TFLite 2.7	3400	0.13h	79.6%	99.2%
Xiaomi12 CPU, TFLite 2.7, INT8	Xiaomi11 CPU, ORT	Pixel5 GPU, TFLite 2.7	3100	0.05h	95.7%	100%
(b) Similarity detection of kernel predictors *Excluding* same inference frameworks						
Xiaomi11 CPU, ORT	Pixel5 CPU, MindSpore	Pixel5 GPU, NCNN	2400	0.72h	84.2%	99.2%
Xiaomi12 GPU, TFLite 2.7	Pixel5 GPU, NCNN	Xiaomi12 CPU, MindSpore	16100	0.22h	79.4%	98.7%
Xiaomi11 CPU, Mindspore	Pixel5CPU, TFLite 2.7	Pixel5GPU, TFLite 2.7	9700	0.80h	98.1%	99.2%
Pixel5 GPU, NCNN	Xiaomi12 CPU, TFLite 2.1	Xiaomi11 CPU, ORT	18500	1.73h	86.5%	99.3%
Xiaomi12 CPU, TFLite 2.1, low Freq	Xiaomi11 GPU, NCNN	Pixel5 CPU, MindSpore	1800	0.18h	94.7%	100%
Xiaomi12 CPU, TFLite 2.1	Pixel4 CPU, TFLite 2.7	Pixel5 CPU, MindSpore	1800	0.10h	97.6%	99.9%

Table 5: Transferable latency prediction of LitePred on diverse new platforms. LitePred accurately predicts the inference latency of models across five different CNN NAS search spaces, with minimal adaptation cost (0.05 to 1.73 hours) on new platforms.

Table 5 demonstrates that LitePred achieves superior accuracy in predicting latency on diverse new edge platforms equipped with varying hardware, inference engines, data precision, and frequencies. Remarkably, we achieve an average of 99.3% transfer accuracy, with 87.0% of models having prediction errors within a negligible 5% margin.

Furthermore, not only does LitePred accurately predict inference latency on unseen new platforms, but it also requires only ∼4,000 adaptation data points for finetuning all kernel predictors across most of the platforms we evaluated. This leads to <1 hour of measurement overhead, which is a significant improvement over platform-specific methods that typically require 1-3 days of measurement for a single platform.

In Table 5, we observe that more adaptation data is needed for finetuning kernel predictors on GPUs with NCNN/TFLite 2.7 and CPUs with NCNN. This is due to two main reasons. First, models generate 4 new kernels in the NCNN framework: Conv+bn+swish/hswish and DWConv+bn+swish/hswish, requring the use of similar predictors and more adaption data points. Second, edge GPU platforms typically require more adaptation data for Conv and DWConv related kernels due to specific optimizations on various GPUs, making it challenging to find a highly similar platform. Thus, more data is needed for the predictor to learn these optimizations.

7.4 Ablation study

We now conduct ablation studies to assess the effectiveness of each of our techniques.

The effectiveness of VAE data sampler. An effective data sampler is crucial for improving prediction accuracy and avoiding useless data. To evaluate the effectiveness of our VAE data sampler, we compare it with the state-of-the-art adaptive data sampler proposed in nn-Meter on 4 platforms.

Platform	Method	Conv Acc.	DWConv Acc.
Xiaomi11 CPU MindSpore	Adaptive data sampler	84.9%	52.8%
	VAE data sampler	**91.4%**	**93.6%**
Xiaomi11 CPU NCNN	Adaptive data sampler	81.5%	95.5%
	VAE data sampler	**88.8%**	**98.3%**
Pixel5 GPU TFLite 2.7	Adaptive data sampler	61.6%	86.7%
	VAE data sampler	**76.7%**	**89.1%**
Pixel5 GPU NCNN	Adaptive data sampler	65.6%	79.6%
	VAE data sampler	**87.1%**	**81.7%**

Table 6: Under the same sampling budget of 10k data points, VAE data sampler outperforms state-of-the-art methods with achieving much higher latency prediction accuracy.

We sample 10k configurations using both data samplers to train the latency predictor for Conv and DWConv kernels.

Table 6 presents a comparison of the achieved prediction accuracy. Our VAE data sampler consistently outperforms the adaptive data sampler, achieving much higher prediction accuracy on all four platforms. This is because we can sample configurations that conform to the multi-dimensional distribution in NAS search spaces. In contrast, the adaptive data sampler is constrained to align solely with individual dimensions, leading to the out-of-distribution issue.

The effectiveness of our similarity detection technique. We evaluate whether our method detects the most similar latency predictor for target platform. We set up two baselines: *(i)* OneProxy [33], choosing the predictor with the highest SRCC metric by comparing predicted and real latency rankings for random kernel configurations, and *(ii)* Random, selecting a predictor from the knowledge pool randomly. For the selected kernel predictor, we use VAE data sampler to generate the same adaptation data for finetuning. A more similar predictor is expected to have higher accuracy after finetuning on the

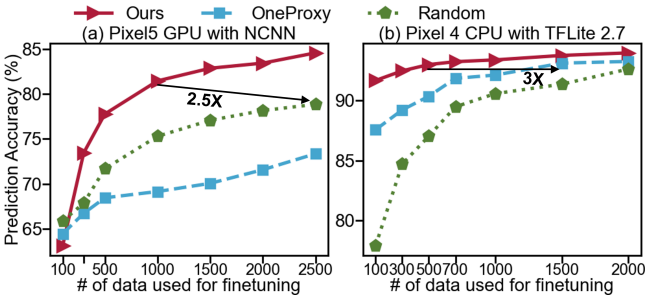


Figure 6: Finetuned accuracy of Conv kernel on new platforms. By our similarity detection, we achieve higher accuracy with 2.5× less adaptation data than baseline methods.

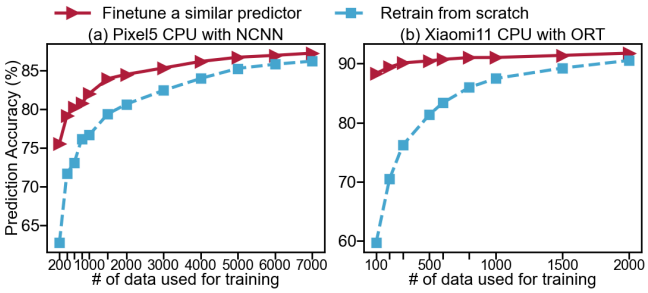


Figure 7: Prediction accuracy of Conv kernel. Finetuning a similar pre-trained predictor yields higher accuracy than training a new predictor from scratch on the same platform.

new platform and require less adaptation data.

Fig. 6 compares the finetuned accuracy of Conv kernel predictor on two platforms. Our method consistently identify the most similar predictors for the target platform, resulting in significantly higher accuracy across varying amounts of adaptation data. When achieving same accuracy, our detected predictors require 2.5× and 3× less adaptation data than those required by random selection and OneProxy, respectively.

The effectiveness of finetuning pre-existing predictors. With sufficient time, it's possible to train kernel latency predictors from scratch for a new platform using VAE data sampler. Our experiment shows that finetuning a similar pre-trained predictor yields higher prediction accuracy with less training data. As shown in Fig. 7, finetuning leads to faster convergence and better accuracy. This finding is consistent with the intuition of our work, as a similar predictor can already capture some latency behaviors for the target platform.

Transfer cost analysis. We now analyze the transferring costs in Table 5. Our results show that major adaptation costs come from finetuning Conv and DWConv kernels. Conv finetuning uses **51.1%** of the adaptation data, and DWConv uses **32.5%**. Small kernels need only 100 points, while SE kernels need slightly more, ranging from 300 to 700 points.

We further analyze the impact of varying amounts of adaptation data on prediction accuracy. We select two platforms: Pixel5 GPU with NCNN, which required the largest amount of data, and Xiaomi12 CPU with TFLite 2.1, which only required 1800 samples. The results, shown in Fig. 8, indicate

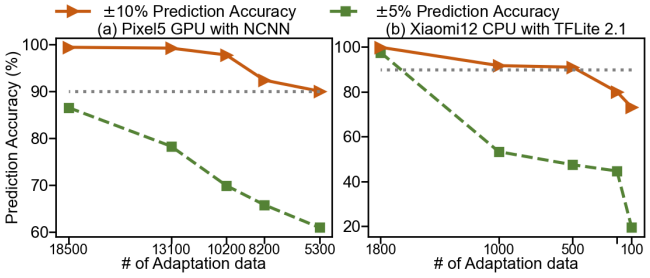


Figure 8: Model-level prediction Acc. vs. data amounts.

a large drop of ±5% accuracy when reducing the adaptation data, followed by a more gradual decline at ±10% accuracy. If we set a relaxed threshold of 90% ±10% accuracy as acceptable number, then we can reduce the amount of adaptation data by 3.5× and 3.6× for the two platforms, respectively.

7.5 Hardware-aware NAS with LitePred

We now showcase how LitePred effectively supports hardware-aware NAS in finding accurate, low-latency DNNs for various edge platforms. We integrate LitePred with a state-of-the-art NAS approach, called OFA [8], and conduct latency-constrained search for 4 different edge platforms. It is worth noting that we test 2 edge GPU platforms, which are rarely evaluated in current hardware-aware NAS due to the challenges of accurately obtaining latency on them. Specifically, we search 5k model architectures for each given latency constraint and select the model with highest validation accuracy on ImageNet 2012 dataset [15]. We then evaluate the test accuracy of the final model and measure the on-device latency.

Table 7 compares the best searched model accuracy with MobileNetV2 [42] and MobileNetV3 [22], which are state-of-the-art lighweight CNNs designed for edge platforms. Results show that OFA with LitePred delivers better models than MobileNets, achieving higher accuracy on ImageNet and lower latency on 4 diverse edge platforms. Our searched models surpass MobileNetV2 by up to 4.4% higher ImageNet accuracy. Our results proves LitePred's value as a tool for hardware-aware NAS in designing higher accuracy models that meet specific latency constraints on diverse edge platforms.

7.6 LitePred on Transformer models

We demonstrate LitePred's generalization ability on other DNN architectures by assessing transfer latency prediction on vision transformers [10, 16, 48]. We test on mobile CPUs with TFLite platforms, which have good support for transformer models. We build latency predictors for five vision transformer sub-modules: MultiHeadAttention, PatchEmbedding, MLP, LayerNormalization, and Linear layers. Specifically, TensorFlow's optimizations for MultiHeadAttention remove the need for fine-grained kernel detection.

Table 8 shows the latency prediction accuracy for AutoFormer [10] search space. The high transferable prediction accuracy showcases LitePred's effectiveness in transferring latency predictors for vision transformers across edge platforms with varying devices, data precision, and CPU frequencies.

Method	Pixel6 CPU, TFLite 2.1		Xiaomi11 CPU, Mindspore		Pixel5 GPU, NCNN		Xiaomi11 GPU, TFLite 2.7	
	Top1 Acc. ↑	Latency ↓	Top1 Acc. ↑	Latency ↓	Top1 Acc. ↑	Latency ↓	Top1 Acc. ↑	Latency ↓
MobileNetV2	72.0	45.9ms	72.0	22.7ms	72.0	31.3ms	72.0	5.0 ms
OFA [8] + LitePred	**76.4**	**44.4 ms**	**73.4**	**21.8 ms**	**75.3**	**31.1 ms**	**75.8**	**5.0 ms**
MobileNetV3× 0.75	73.3	29.7ms	73.3	24.4ms	73.3	34.4ms	73.3	4.0 ms
OFA [8] + LitePred	**74.8**	**29.6 ms**	**74.5**	**23.9 ms**	**76.0**	**34.3 ms**	**74.4**	**3.9 ms**
MobileNetV3	75.2	37.2ms	75.2	33.4ms	75.2	30.3ms	75.2	4.7ms
OFA [8] + LitePred	**75.5**	**36.8 ms**	**75.6**	**33.2 ms**	**75.6**	**29.8 ms**	**75.5**	**4.6 ms**

Table 7: Hardware-aware NAS search results on ImageNet 2012 dataset [15]. By integrating LitePred into OFA, we achieve superior accuracy compared to MobileNets across various edge platforms.

Platform	Similar Platform	Time Cost	Prediction Acc ±5%	Prediction Acc ±10%
Xiaomi11 CPU TFLite 2.7	Xiaomi11 CPU TFLite 2.1	0.05h	100%	100%
Xiaomi12 CPU TFLite 2.1	Pixel5 CPU TFLite 2.7, LowFreq	0.08h	83.9%	99.9%
Xiaomi12 CPU TFLite 2.7, INT8	Pixel5 CPU TFLite 2.7, LowFreq	0.02h	41.4%	99.9%

Table 8: Transferable latency prediction of LitePred on vision transformer NAS search space [10].

8 Related Works

Platform-specific latency prediction. Most previous latency prediction approaches [6, 7, 18, 31, 32, 54] are designed to build platform-specific latency predictors. Notable examples include BRP-NAS [18] and nn-Meter [54]. BRP-NAS [18] uses graph convolutional networks (GCN) to train a GCN latency predictor. However, it requires the entire model graph as input, which makes it necessary to redesign and retrain the model for new hardware-aware NAS search spaces, leading to expensive and time-consuming processes. nn-Meter [54] tackles this challenge by building kernel-level predictors. However, all these approaches require significant time costs and efforts to rebuild the predictors for new platforms, as they rely on platform-specific data collection and predictor training.

Platform-adaptive latency prediction. Recently, a few works propose to construct platform-agnostic latency predictors [30, 34, 37]. HELP [30] builds a meta latency predictor that incorporates hardware embeddings. However, meta-training requires numerous latency measurements on a wide variety of heterogeneous platforms and faces challenges in generalizing to new unseen devices. OneProxy [34] exploits latency monotonicity across diverse devices to predict DNN latency rankings on new unseen platforms. However, many hardware-aware NAS approaches [8, 9, 45, 56, 57] require actual latency values rather than rankings.

Hardware-aware NAS. Hardware-aware NAS approaches [8, 9, 49, 50, 53, 55, 57] aim to design efficient DNN models that balance accuracy and latency. However, the vast model search space makes latency measurement costly. Most NAS works [21, 49, 50] use FLOPs as an efficiency metric, but it's an inaccurate proxy for latency. Recent works such as ProxylessNAS [9] and OFA [8] employ a layer-wise latency predictor, but ignore latency changes caused by graph optimizations. Also, most of the evaluated platforms are limited to cloud platforms. LitePred provides rapid and accurate latency predictions on diverse edge platforms, facilitating the design of more efficient DNN models for edge environments.

9 Discussion

Comparison with Cost Models in DNN Compilers. Many deep learning compilers [4, 5, 11, 12, 58] build cost models to predict the execution time of different code implementations on a given hardware platform. They typically rely on complex feature engineering to build decision-tree-based regression models. For instance, TVM [11] employs XGBoost to make predictions based on a diverse set of features, including memory access and data reuse ratio, along with embedded features like AST. However, as many edge inference frameworks are closed source, these code-based methods are infeasible. LitePred differentiates itself from these approaches by predicting model latency solely based on the model configurations.

Generalization Ability. Currently, LitePred focuses on predicting inference latency for CNNs and vision transformers on commercial edge platforms. If a new edge platform is significantly different from our knowledge pool, it may be necessary to sample more adaptation data and train like starting from scratch. Our approach can be easily extended to other model types, such as language transformers. Generalization to cloud platforms has not been validated due to potential concurrency in kernel execution. We leave this as future work.

10 Conclusion

In this work, we propose LitePred, a lightweight latency prediction approach that can accurately predict the inference latency of DNN models on a new edge platform based on a small amount of extra measurements. LitePred incorporates a VAE data sampler to collect high-quality training and adaptation data. By transferring the most similar pre-existing latency predictors, LitePred achieves accurate predictions with an adaptation cost of less than 1 hour. Extensive experiments on 85 edge platforms and 6 NAS search spaces demonstrate the effectiveness of LitePred, achieving an impressive prediction accuracy of 99.3% and a remarkable 50.6× reduction in profiling cost compared with state-of-the-art baselines.

11 Acknowledgement

We are thankful to the anonymous NSDI reviewers and our shepherd, Hong Xu, for their constructive feedback. The work of Chengquan Feng, Yuanchi Liu, Zhiyuan Wang and Haisheng Tan are supported by the National Science Foundation of China under Grant No. 62132009.

References

[1] Pytorch kl-divergence loss. https://pytorch.org/docs/stable/generated/torch.nn.KLDivLoss.html?highlight=kl+divergence.

[2] Pytorch mseloss. https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html.

[3] ARM. An introduction to the armv8 instruction sets. https://developer.arm.com/documentation/den0024/a/An-Introduction-to-the-ARMv8-Instruction-Sets, 2019.

[4] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, et al. A deep learning based cost model for automatic code optimization. *Proceedings of Machine Learning and Systems*, 3:181–193, 2021.

[5] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, 2019.

[6] Noureddine Bouhali, Hamza Ouarnoughi, Smail Niar, and Abdessamad Ait El Cadi. Execution time modeling for cnn inference on embedded gpus. Association for Computing Machinery, 2021.

[7] Halima Bouzidi, Hamza Ouarnoughi, Smail Niar, and Abdessamad Ait El Cadi. Performance prediction for convolutional neural networks on edge gpus. In *Proceedings of the 18th ACM International Conference on Computing Frontiers*, CF '21, page 54–62. Association for Computing Machinery, 2021.

[8] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *ICLR*, 2020.

[9] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2019.

[10] Minghao Chen, Houwen Peng, Jianlong Fu, and Haibin Ling. Autoformer: Searching transformers for visual recognition. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 12270–12280, 2021.

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, 2018.

[12] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31, 2018.

[13] THOMAS M. COVER and JOY A. THOMAS. *Elements of Information Theory*. 1991.

[14] Xiaoliang Dai, Alvin Wan, Peizhao Zhang, Bichen Wu, Zijian He, Zhen Wei, Kan Chen, Yuandong Tian, Matthew Yu, Peter Vajda, and Joseph E. Gonzalez. Fbnetv3: Joint architecture-recipe search using neural acquisition function. *CoRR*, abs/2006.02049, 2020.

[15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*, pages 248–255. IEEE Computer Society, 2009.

[16] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

[17] DroidChart. You can currently find 8318 smartphones from 238 brands, 2023.

[18] Lukasz Dudziak, Thomas Chau, Mohamed Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas Lane. Brp-nas: Prediction-based nas using gcns. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems (Neurips)*, volume 33, pages 10480–10490. Curran Associates, Inc., 2020.

[19] Nathaniel R Goodman. Statistical analysis based on a certain multivariate complex gaussian distribution (an introduction). *The Annals of mathematical statistics*, 34(1):152–177, 1963.

[20] Google. Tensorflow lite. https://tensorflow.google.cn/lite/. Accessed 2022-12-14.

[21] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling.

In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XVI 16*, pages 544–560. Springer, 2020.

[22] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3. In *ICCV*, 2019.

[23] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.

[24] Huawei. Mindspore. https://www.mindspore.cn/lite/. Accessed 2022-12-14.

[25] Intel. Deploy high-performance deep learning inference, openvino. https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html, 2019.

[26] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[27] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[28] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4013–4021, 2016.

[29] Hayeon Lee, Eunyoung Hyung, and Sung Ju Hwang. Rapid neural architecture search by learning to generate graphs from datasets. *ArXiv*, abs/2107.00860, 2021.

[30] Hayeon Lee, Sewoong Lee, Song Chong, and Sung Ju Hwang. Hardware-adaptive efficient latency prediction for nas via meta-learning. *Advances in Neural Information Processing Systems*, 34:27016–27028, 2021.

[31] Zhuojin Li, Marco Paolieri, and Leana Golubchik. Predicting inference latency of neural architectures on mobile devices. Association for Computing Machinery, 2023.

[32] Liang Liu, Mingzhu Shen, Ruihao Gong, Fengwei Yu, and Hailong Yang. Nnlqp: A multi-platform neural network latency query and prediction system with an evolving database. 2022.

[33] Bingqian Lu, Jianyi Yang, Weiwen Jiang, Yiyu Shi, and Shaolei Ren. One proxy device is enough for hardware-aware neural architecture search. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 5(3):1–34, 2021.

[34] Bingqian Lu, Jianyi Yang, Weiwen Jiang, Yiyu Shi, and Shaolei Ren. One proxy device is enough for hardware-aware neural architecture search. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 5(3):1–34, 2021.

[35] Microsoft. onnxruntime, 2022.

[36] Bert Moons, Parham Noorzad, Andrii Skliar, Giovanni Mariani, Dushyant Mehta, Chris Lott, and Tijmen Blankevoort. Distilling optimal neural networks: Rapid search in diverse spaces. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 12229–12238, 2021.

[37] Saeejith Nair, Saad Abbasi, Alexander Wong, and Mohammad Javad Shafiee. Maple-edge: A runtime latency predictor for edge devices. In *CVPR EVW workshop*, pages 3660–3668, 2022.

[38] Microsoft Research nn Meter Team. nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices, 2021.

[39] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[40] Jialun Peng, Dong Liu, Songcen Xu, and Houqiang Li. Generating diverse structure for image inpainting with hierarchical vq-vae. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10775–10784, 2021.

[41] Ali Razavi, Aaron Van den Oord, and Oriol Vinyals. Generating diverse high-fidelity images with vq-vae-2. *Advances in neural information processing systems*, 32, 2019.

[42] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[43] Stanislau Semeniuta, Aliaksei Severyn, and Erhardt Barth. A hybrid convolutional variational autoencoder for text generation. *arXiv preprint arXiv:1702.02390*, 2017.

[44] A.K Subramanian. Pytorch-vae. https://github.com/AntixK/PyTorch-VAE, 2020.

[45] Chen Tang, Li Lyna Zhang, Huiqiang Jiang, Jiahang Xu, Ting Cao, Quanlu Zhang, Yuqing Yang, Zhi Wang, and Mao Yang. Elasticvit: Conflict-aware supernet training for deploying fast vision transformer on diverse mobile devices. *arXiv preprint arXiv:2303.09730*, 2023.

[46] Tensorflow. Post-training integer quantization. https://www.tensorflow.org/lite/performance/post_training_integer_quant, 2023.

[47] Sebastian Thrun and Lorien Y. Pratt. Learning to learn: Introduction and overview. In Sebastian Thrun and Lorien Y. Pratt, editors, *Learning to Learn*, pages 3–17. Springer, 1998.

[48] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Herve Jegou. Training data-efficient image transformers and distillation through attention. In *International Conference on Machine Learning*, volume 139, pages 10347–10357, July 2021.

[49] Dilin Wang, Chengyue Gong, Meng Li, Qiang Liu, and Vikas Chandra. Alphanet: Improved training of supernet with alpha-divergence. In *ICML*, 2021.

[50] Dilin Wang, Meng Li, Chengyue Gong, and Vikas Chandra. Attentivenas: Improving neural architecture search via attentive sampling. In *Conference on Computer Vision and Pattern Recognition*, 2021.

[51] Hanrui Wang, Zhanghao Wu, Zhijian Liu, Han Cai, Ligeng Zhu, Chuang Gan, and Song Han. HAT: hardware-aware transformers for efficient natural language processing. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 7675–7688. Association for Computational Linguistics, 2020.

[52] Xudong Wang, Li Lyna Zhang, Yang Wang, and Mao Yang. Towards efficient vision transformer inference: A first study of transformers on mobile devices. In *Proceedings of the 23rd Annual International Workshop on Mobile Computing Systems and Applications*, 2022.

[53] Jiahui Yu, Pengchong Jin, Hanxiao Liu, Gabriel Bender, Pieter-Jan Kindermans, Mingxing Tan, Thomas Huang, Xiaodan Song, Ruoming Pang, and Quoc Le. Bignas: Scaling up neural architecture search with big single-stage models. In *ECCV*, 2020.

[54] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, page 81–93, New York, NY, USA, 2021. ACM.

[55] Li Lyna Zhang, Youkow Homma, Yujing Wang, Min Wu, Mao Yang, Ruofei Zhang, Ting Cao, and Wei Shen. Swiftpruner: Reinforced evolutionary pruning for efficient ad relevance. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, pages 3654–3663, 2022.

[56] Li Lyna Zhang, Xudong Wang, Jiahang Xu, Quanlu Zhang, Yujing Wang, Yuqing Yang, Ningxin Zheng, Ting Cao, and Mao Yang. Spaceevo: Hardware-friendly search space design for efficient int8 inference. *arXiv preprint arXiv:2303.08308*, 2023.

[57] Li Lyna Zhang, Yuqing Yang, Yuhang Jiang, Wenwu Zhu, and Yunxin Liu. Fast hardware-aware neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2020.

[58] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[59] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. A comprehensive survey on transfer learning. *Proceedings of the IEEE*, 109(1):43–76, 2020.

[60] Nihui Zuo Zhang. Tencent: Ncnn: a high-performance neural network inference computing framework optimized for mobile platforms. https://github.com/Tencent/ncnn/, 2019.

Harmonic: Hardware-assisted RDMA Performance Isolation for Public Clouds

Jiaqi Lou[1]* Xinhao Kong[2]* Jinghan Huang[1] Wei Bai[3]† Nam Sung Kim[1] Danyang Zhuo[2]

University of Illinois Urbana-Champaign[1] *Duke University*[2] *Microsoft*[3]

Abstract

Performance isolation is essential for sharing resources in multi-tenant public clouds. Compared with traditional kernel-based networking, RDMA presents unique challenges especially because RDMA NIC's complex microarchitecture resources are often hidden from users. Current RDMA isolation methods overlook these microarchitecture resources, leading to insufficient performance isolation. Consequently, a faulty/malicious tenant can exploit these microarchitecture resources to compromise well-behaved tenants' network performance. In this paper, we introduce the first microarchitecture-resource-aware RDMA performance isolation solution for public clouds, Harmonic. It consists of two key components designed to be conscious of the RDMA NIC's microarchitectural resources: (1) a programmable intelligent PCIe switch (prototyped with FPGA) and (2) an RDMA-friendly rate limiter. At runtime, these two components allow us to accurately monitor and modulate the RDMA NIC resource usage per tenant. We evaluate Harmonic with a state-of-the-art RDMA performance isolation test suite (Husky) and a popular in-memory database application (Redis). We demonstrate that Harmonic can not only successfully pass Husky but also provide Redis with 1.4× higher throughput than the best alternative isolation solution.

1 Introduction

The Remote Direct Memory Access (RDMA) technology has been widely deployed in modern clouds to improve network performance. First-party workloads in clouds, such as storage [11, 17], heavily rely on RDMA to achieve high throughput, low latency, and high CPU efficiency. A natural next step for cloud providers is to bring RDMA's benefits to their public cloud tenants. Unfortunately, this has not yet come true because RDMA was initially designed for high-performance computing, lacking adequate multi-tenancy support.

One of the key missing components for bringing RDMA to public clouds is *performance isolation*. Without proper performance isolation, a buggy or malicious tenant can affect the RDMA performance of other tenants, and even conduct side-channel attacks through the RDMA network [29, 52, 54, 55]. Although network performance isolation has been extensively studied in the past decades [10, 12, 19, 20, 25, 32, 53], recent work has highlighted that prior RDMA performance isolation solutions are insufficient for public clouds [29]. An RDMA NIC (RNIC) has microarchitecture resources, such as on-NIC cache and on-NIC processing units that significantly affect RDMA performance [26, 27, 29]. However, all existing performance isolation solutions are agnostic to the contention of these microarchitecture resources among tenants, providing insufficient performance isolation when the microarchitecture resources are exhausted. For example, RDMA traffic that keeps generating expensive ATOMIC requests can exhaust the on-NIC processing units and drastically reduce the RDMA performance of other tenants [29, 46].

The goal of this paper is to explore the possibility of building a microarchitecture-resource-aware solution for RDMA performance isolation. Our high-level approach is as follows: we monitor the usage of RDMA resources (including microarchitecture resources) per tenant, and then modulate it accordingly to provide isolation. Yet, realizing our approach faces two challenges:

(C1) Accurately measuring per-tenant RNIC resource usage. RDMA traffic bypasses the kernel, which makes it hard to intercept and monitor the RDMA traffic in system software. Moreover, RNICs today only expose limited aggregate statistics, such as RNIC cache miss rates and total PCIe bandwidth consumption, without the capability of identifying the specific tenant causing this resource usage.

(C2) Finding an appropriate rate limit enforcement entry point. System software is not a viable rate limit enforcement point because most RDMA operations bypass the control of cloud providers. Commodity RNICs also do not provide rich rate enforcement features. For example, no current RNIC provides a mechanism to limit a tenant's rate of specific RDMA operations (*e.g.*, ATOMIC), and cloud providers cannot feasi-

*Jiaqi Lou and Xinhao Kong contributed equally to this research.
†Wei Bai is now with NVIDIA.

bly modify existing RNICs to incorporate these new features. We also cannot simply drop excessive packets at the RNIC, because packet losses can significantly degrade RDMA performance [21, 31, 60, 64].

Our key approaches to address the above challenges are outlined below. First, we make a PCIe switch serve as a sweet spot for measuring the RDMA resource usage of tenants at runtime. This choice is motivated by the following reasons. All RDMA traffic goes through the PCIe bus, allowing us to intercept all RDMA behaviors. More importantly, RDMA pins all RDMA-related objects (*e.g.*, payloads and other metadata) in the host DRAM. Thus, the physical address to tenant/object mapping is fixed. This enables us to correlate a PCIe transaction with a specific tenant and associated RDMA behaviors by mapping the transaction's target physical memory address to the RDMA objects.

To tackle the second challenge, we repurpose the rate limiters in RNIC hardware for our performance isolation. Modern commodity RNICs employ many rate limiters for congestion control purposes. These rate limiters react to network congestion feedback and reduce rates accordingly. We therefore can proactively inject an appropriate amount of congestion feedback to targeted tenants, to limit their rates when we need to limit their RDMA resource usage.

Applying our insights above, we develop Harmonic, the first hardware/software co-design solution for RDMA performance isolation that takes RNIC microarchitecture resources into account without requiring changes to applications. To measure the RDMA resource usage of tenants at runtime, we implement an FPGA-based Programmable Intelligent PCIe Switch (PIPS) in Harmonic. We extend existing RNIC kernel drivers to a Harmonic kernel driver to obtain the aforementioned physical memory address to tenant/object mappings. PIPS connects the RNIC to the host, and monitors the RDMA traffic of each tenant using the mappings provided by the Harmonic kernel driver. We implement a Harmonic daemon to repurpose the rate limiters in the RNIC hardware. Most, if not all, commodity RNICs support DCQCN [64] as congestion control algorithm [13, 24, 40]. The Harmonic daemon therefore can generate and send Congestion Notification Packet (CNP), the congestion feedback in DCQCN, to rate-limit targeted tenants for our performance isolation purpose. The Harmonic daemon limits tenants' rates based on PIPS's monitoring results. To make performance isolation more practical for public RDMA clouds, we also extend the existing RDMA performance abstraction to include a set of RDMA-specific resources, such as the number of QPs and the RDMA request rate.

We use Harmonic to enhance an NVIDIA ConnectX-6 Dx 25 Gbps NIC and evaluate Harmonic with the state-of-the-art RDMA performance isolation test suite, Husky [29], and a popular in-memory database application, Redis over RDMA [63]. We compare Harmonic with other performance isolation solutions, including hardware Single Root I/O Virtualization (SR-IOV), separate hardware queues, and Justitia [62]. Our evaluation results show that Harmonic successfully provides stronger performance isolation under various types of resource contention. This results in improving the throughput of Redis by up to 1.4×, compared to the state-of-the-art isolation solutions. To the best of our knowledge, Harmonic is the first RDMA performance isolation solution that can pass the Husky test suite [29].

Lastly, current Harmonic supports 25 Gbps RNICs, limited by the speed of the PCIe physical layer (PCIe PHY) in our commodity FPGA development board[1]. A deployable solution for high-speed RNIC will require adopting our proposed techniques in the future RNIC design. While Harmonic serves as a prototype, it demonstrates the viability of RDMA performance isolation for public clouds and can act as a benchmark for future implementations. Our design presented in this paper is currently being integrated into one leading technology enterprise's next-generation RNIC design.

2 Background

2.1 Remote Direct Memory Access

RDMA enables user applications to directly interface with RNIC by offloading network stack processing to RNIC hardware. RDMA enables low-latency, CPU-efficient networking at high bandwidth, and it is increasingly deployed at datacenters [11, 17, 21]. For example, Bai et al. [11] demonstrated that more than 70% of traffic in Azure is RDMA.

Figure 1 shows the four key components (*i.e.*, userspace libraries, kernel drivers, RNIC firmware, and RNIC ASIC) in a modern commodity RDMA system from a top-down perspective. The first component that user applications interact with is *userspace libraries*. Applications invoke APIs provided by these libraries to issue data verb and control verb operations. For example, applications call control verbs to allocate necessary objects such as queue pair (QP), completion queue (CQ), and memory region (MR). Applications thereby issue data verbs to send RDMA network traffic, such as RDMA WRITE requests to directly write remote host's memory. In a typical RDMA system, control verbs are first processed by RDMA *kernel drivers*. Kernel drivers usually conduct a few checks (*e.g.*, parameter validation) and construct a command to send to the RNICs. In the RNIC, a small piece of software or microcode embedded into hardware device memory will process these commands and return results to the kernel drivers, such as the newly created QP [42]. This software on the RNIC is known as the *RNIC firmware*. When the RNIC firmware processes control verbs, *RNIC ASIC* is also involved since many hardware status may be updated. For example, RNIC

[1] We find that the state-of-the-art FPGA whose PCIe PHY can be configured in PCIe switch upstream/downstream mode only has 8-lane edge connector after several rounds of communication with our FPGA manufacturer, Xilinx. This is also confirmed by Xilinx's public information [3, 4], but it can support any RNICs with any speed offered by the PCIe PHY. We discuss the scalability of our solution to higher speed in §7.

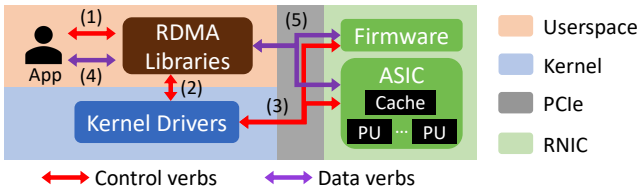


Figure 1: RDMA workflow.

has on-NIC cache to store QP contexts [27], which can be accessed and updated when the RNIC firmware handles QP creation or destruction.

Data verbs are directly passed to RNIC hardware without involving kernel drivers (or any system software), which is known as kernel bypass. For example, when applications call `ibv_post_send` to issue an RDMA SEND request, userspace libraries prepare work queue entries (WQEs) in send/recv queues. Each entry in the queues corresponds to a data verb. The libraries then notify the RNIC hardware that there is a WQE to process. Specifially, the libraries may ring the doorbell of the corresponding QP (*i.e.*, write a specific register) on the RNIC, triggering RNIC hardware to DMA read those WQEs from the host and start to process. When processing data verbs, RNIC firmware may also be involved under some scenarios, such as handling an error triggered by a data verb.

There are three types of resources in RNICs:

(R1) Traditional network resources. They include network bandwidth and packet processing capacity, indicated by bits per second (BPS) and packets per second (PPS), respectively.

(R2) RDMA-specific architectural resources. They comprise the number of QPs and request rates of different verbs (*e.g.*, ATOMIC, WRITE, and SEND) that applications can directly operate on.

(R3) RDMA-specific microarchitecture resources. They encompass the PCIe bandwidth, on-NIC cache and on-NIC processing units that are vendor-specific. These resources are not exposed to applications and can be neither monitored nor controlled precisely [31].

2.2 RDMA Performance Isolation

RDMA has already been successfully adopted in accelerating first-party workloads such as storage [11, 17]. The next question is whether these RDMA advantages can be extended to third-party workloads in the public cloud. RDMA performance isolation for public clouds is important, as customers primarily choose RDMA for workloads with demanding performance requirements. Without proper performance isolation, a faulty or malicious tenant could detrimentally impact the performance of other tenants [29].

To design a performance isolation solution, one key question is: *what's the abstraction of network performance?* The conventional wisdom is that a cloud provider should guarantee network bandwidth, measured by BPS, to a virtual machine (VM) or container. For example, Amazon Web Service (AWS) provides an 30 Gbps guarantee for its m7gd.16xlarge instance and Azure offers a 40 Gbps guarantee for its D96as_v5 VM series [7, 35]. This is done by limiting the available network bandwidth to the remaining VMs co-located on the same host.

In this paper, we argue that this conventional wisdom does not work for an RDMA network. The aforementioned microarchitecture resources make RDMA performance isolation different from that on traditional TCP/IP networks. In RDMA, most verb processing tasks are offloaded to the RNIC firmware and RNIC hardware. RNICs leverage their internal resources to support these offloaded functionalities. Not considering these resources results in performance isolation designs that are insufficient to be used in public clouds. One of the empirical evidences is that Husky [29], a prior work, has already shown that no mature RDMA performance isolation solution exists. Therefore, a comprehensive RDMA performance isolation solution for the public cloud has to consider various types of interference on RNIC's microarchitecture resources, which can occur when multiple tenants contend for access to these resources.

Static partitioning versus dynamic resource usage modulation. In general, there are two approaches to achieve performance isolation when sharing resources. Our paper explores the dynamic resource usage modulation approach, which is to monitor and control each tenant's resource usage. The other approach is to statically partition every resource and assign partitioned resources to each tenant. We did not explore the static partitioning approach for two reasons. First, RNIC microarchitecture resources (*e.g.*, NIC caches) are crucial for applications' performance. We have observed many prior works in RDMA application design to use these resources efficiently in order to avoid resource exhaustion [15, 26, 27, 31]. Static partitioning of these resources may cause catastrophic performance penalties for RDMA applications. Second, commodity RNICs currently do not support static resource partitioning, and exploring this approach thus requires building an RNIC from scratch, which is beyond the scope of our research. Our goal is to design a prototype that shows feasibility for deployment, and we thus choose to build our system around commodity RNICs.

2.3 Design Space for Monitoring and Controlling Tenant RDMA Resouce Usage

Two key questions arise for monitoring and controlling tenants' RDMA resource usage: (1) where should the cloud provider monitor per-tenant resource usage, and (2) where should the provider enforce resource usage?

The answers to these two questions depend on the deployment model of RDMA, *i.e.*, how RDMA is virtualized. Figure 2 shows the ownership (*i.e.*, owned by tenants or cloud providers) of RDMA system components in typical RDMA virtualization schemes. In the bare-metal scenario, tenants own the entire physical host, including userspace libraries and RDMA kernel drivers. They can even modify and upgrade RNIC firmware as needed [41]. Cloud providers have

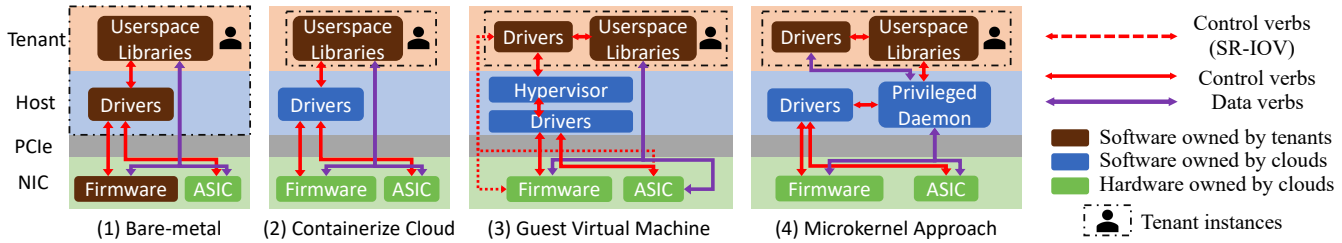


Figure 2: RDMA virtualization schemes.

limited observability and control over both data and control verbs in this scenario. However, RNIC isolation is not a pressing concern as one tenant exclusively occupies the entire machine.

In containerized clouds, each tenant owns a container, and the host OS manages all containers. In this setup, a tenant owns its container instance, including userspace libraries. The tenant's data verbs therefore fully bypass the cloud provider's control. However, drivers and hardware components are still controlled by the cloud provider, allowing them to implement management features. For instance, cloud providers can monitor and regulate RDMA control verbs by incorporating the necessary logic into kernel drivers.

In guest virtual machine (VM) clouds, each tenant owns a VM, running on top of the hypervisor. There are several approaches to expose an RNIC to guest VM. A widely adopted approach is to use Single Root Input/Output Virtualization (SR-IOV). With SR-IOV, multiple virtual instances of the RNIC, referred as Virtual Functions (VFs), are allocated on a physical RNIC. These VFs can be attached to VMs, allowing applications within the VM to directly interact with and utilize the RNIC. The control verbs and data verbs generated by guest VM applications bypass the hypervisor completely. HyV [49] and MasQ [22] employ hybrid virtualization techniques to expose RDMA to guest VMs. They introduce backend drivers within the hypervisor, requiring guest VM drivers to communicate with these backend drivers for processing tenants' control verbs. The hypervisor operates control verbs on the RNICs on behalf of these tenants. Meanwhile, tenants within the guest VMs have the capability to directly transmit data verbs to the RNIC, bypassing the guest kernel and the hypervisor. This ensures native RDMA performance for tenant applications. In these guest VM scenarios, cloud providers typically retain ownership of the hardware components, while the ownership of kernel drivers may vary depending on the specific scheme being employed.

Another virtualization scheme adopts a microkernel-like approach. It forces all tenants to talk to a privileged daemon to use RDMA, such as Freeflow [28] and mRPC [14]. In this scenario, tenants send both control verbs and data verbs to this privileged daemon. The daemon, in turn, initiates the actual RDMA APIs to execute these verbs and subsequently provides the results back to the tenants. This design grants cloud providers comprehensive control over all aspects but comes with the trade-off of additional performance overhead.

Existing solutions' observability and enforcement entry point. To summarize, except for bare-metal environment and virtualization only using SR-IOV, control verbs can be monitored and controlled by cloud providers in kernel drivers, hypervisor backend or privileged daemon. However, data verbs cannot be easily observed or regulated. In containerized cloud (2) or guest VM (3) scenarios, data verbs completely bypass cloud provider's control. Justitia [62], an RDMA performance isolation solution, requires tenants to use its customized *userspace* libraries. However, a malicious tenant can easily bypass or alter the libraries, circumventing the intended isolation. For the microkernel approach, even if we add performance isolation features into a microkernel service, it is still challenging to accurately monitor and regulate data verbs, especially for one-sided operations. For example, RDMA one-sided operations (*e.g.*, WRITE and READ) completely bypass the responder's CPU and therefore cannot be intercepted by the privileged daemon easily.

3 Harmonic Overview

We develop Harmonic, the first RDMA performance isolation solution for public clouds that considers RDMA microarchitecture resources. Our design incorporates three key ideas.

We first introduce an RDMA-specific performance abstraction tailored for public clouds. Currently, cloud providers provide tenants with network abstractions based on BPS or PPS. Unfortunately, such metrics fall short of capturing the varied sets of resources RDMA operations use. RDMA supports various verbs as its primitives, and these verbs demand distinct resource usage. For example, let's consider an 8-byte RDMA ATOMIC compare-and-swap (CAS) request and an 8-byte RDMA SEND request. Both generate identical network traffic in terms of bits and packets, yet the ATOMIC request consumes more NIC processing cycles [27, 29], thus incurring a higher cost. Our performance abstraction considers the RDMA-specific architectural resource capacities allocated to each tenant, such as the number of QPs, CQs, MRs, and the total MR size. It is worthwhile to note that our abstraction does not include RDMA-specific microarchitecture resources, because these resources are vendor-specific and cannot be directly controlled by tenants.

The second pillar of our design ideas is to perform runtime hardware-based measurements of per-tenant RDMA resource

consumption. Since RDMA data verbs bypass the kernel, resource measurement requires direct hardware involvement. In RDMA networks, a tenant's resource consumption is tightly coupled with its verb behaviors. Therefore, by intercepting and analyzing these verbs, we can gain precise insights into the resource consumption of that particular tenant. However, we cannot directly observe verb behaviors on the inter-host network, *i.e.*, Ethernet (for standard RoCEv2 deployment). This limitation arises because many RNIC resource usage behaviors would be opaque if we only monitor packets sent and received by the RNIC. For instance, the RNIC initiates PCIe transactions to retrieve entries from DRAM when its cache entries are exhausted. This RNIC activity incurs both cache miss and extra consumption of PCIe bandwidth—a crucial microarchitectural resource—but remains undetected on the Ethernet. We argue that we need to observe this *within* the host. We find PCIe switch as a sweet spot to enable this runtime measurement feature for two reasons. First, all RDMA traffic goes through PCIe bus, allowing us to capture all tenants' verb behaviors including the host memory address to be accessed in the PCIe Transaction Layer Packet (TLP) header. Second, RDMA requires all RDMA-related objects (*e.g.*, payloads, QPs, CQs) to be pinned in the host DRAM. This indicates the physical address to objects/tenants mapping is fixed and we can monitor tenant's verb behaviors by monitoring which addresses are accessed. Therefore, we can simply parse the TLP header to extract the address field and match it with the mapping, without looking into the large volume of PCIe TLP payloads. There is no existing PCIe switch supporting this functionality. We therefore build an FPGA-based Programmable Intelligent PCIe Switch to prototype this runtime measurement feature. The analogy of this PCIe switch is a programmable switch (*e.g.*, P4-based Tofino switch) in the traditional computer network. The difference is that we design the switch to run on PCIe bus instead of Ethernet. Observing verb behaviors directly allows us to not only measure the network resource consumption (*e.g.*, BPS) but also gauge the utilization of RDMA-specific microarchitecture resources, including PCIe and RNIC processing capacities.

Our third idea is to repurpose the RNIC's congestion control mechanism to facilitate RDMA-friendly rate limiting. Given the kernel and CPU bypass characteristics of RDMA, traditional software-based rate limiters are off the table due to the CPU overheads and the additional latency. Software-based rate limiters are also ineffective in limiting the data receiver side when one-sided operations are used. Moreover, RDMA deployment stems from a lossless network, and current RNICs cannot consistently ensure optimal retransmission performance across all scenarios [21, 31, 64]. Therefore, simply discarding excessive RDMA packets in hardware [16] can cause RDMA performance degradation and is not an option. Our key observation is that we already have a native hardware rate limiting mechanism implemented in modern commodity RNICs for congestion control purposes (*i.e.*, DCQCN [64]). These rate limiters react to network congestion feedback, known as congestion notification packets (CNPs) in DCQCN, to reduce the rate of RDMA connections. We can re-purpose these rate limiters for performance isolation purpose by proactively generating and sending CNPs to modulate the RDMA resource usage per tenant. While this method does consume some processing cycles (CPU cycles in our prototype), the overheads are considerably reduced compared to software-based rate limiters (§6.5).

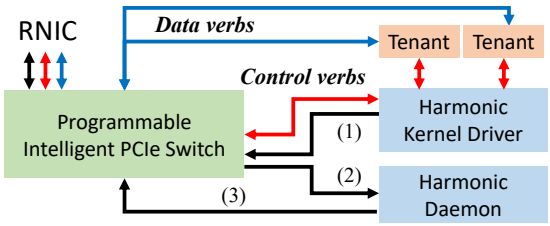


Figure 3: Harmonic overview.

Harmonic's deployment model and workflow. Harmonic assumes that the cloud provider owns the RDMA kernel drivers to intercept control verbs. This is standard for containerized RDMA clouds, para-virtualized VM clouds and microkernel virtualization clouds. We didn't consider the RDMA virtualization scheme that solely depends on SR-IOV, and we show SR-IOV itself is not enough to provide performance isolation (§6.3). In summary, Harmonic can handle the (2), (3) and (4) scenarios in Figure 2. We do not consider the virtualization scenario (1) because RNIC isolation is unnecessary on the bare-metal setting. We implement our prototype with a temporary focus on scenario (2), but it should be easily generalized to both (3) and (4) because we only rely on the modification to the RDMA kernel drivers without touching other system software components.

Figure 3 presents the system architecture of Harmonic. Harmonic has two main components: the Harmonic daemon and the Programmable Intelligent PCIe Switch (PIPS) with Harmonic kernel driver. Harmonic kernel driver is a modified version of the standard RDMA kernel driver that keeps track of control verbs issued per tenant and (1) generates the address-to-tenant/object mappings to PIPS. PIPS not only forwards RDMA traffic as a regular PCIe switch, but intercepts PCIe traffic to keep track of data verbs issued per tenant as well. Harmonic daemon is a privileged process that runs on the host OS or hypervisor. It (2) polls tenant's data verb behavior statistics from the PIPS and (3) sends congestion feedback packets to each tenant to modulate their RDMA resource usage. All these components are trusted and will not be tempered by the tenants.

Harmonic's benefits. Harmonic has several key benefits compared to existing RDMA performance isolation solutions. First and most important, Harmonic takes microarchitecture resource usage into account and thus provides stronger isolation. Harmonic observes both data and control verbs, in the meantime, restricts tenant resource usage correspondingly.

Table 1: An example for RDMA performance abstraction.

Name	# of QPs	# of WQEs	# of MRs	# of CQs	# of CQEs	MR Size	BPS	DRPS	CRPS	Prio
Alice	128	16384	128	16	8192	2 GB	10 Gbps	30 Mrps	1 Krps	0

This is different from simply observing network bandwidth usage. Second, Harmonic requires no modification of applications. There's no need to adjust application libraries, allowing for straightforward integration with application binaries. Third, our approach delivers native RDMA performance for public cloud usage. Applications' data verbs continue to bypass system software entirely, and the only latency overhead comes from the PCIe switch, which is minimal (§6.5).

Harmonic's performance abstraction. Our performance abstraction includes a set of metrics that enable tenants to accurately describe their expected RDMA network performance needs. At the same time, it allows us to design the performance isolation mechanisms to guarantee the metrics to tenants. In addition to the conventional BPS metric, our performance abstraction considers per-tenant RDMA-specific resources, including the number of QPs, CQs, MRs, and the total MR size. Application developers have direct control over these RDMA-specific architectural resources, because they directly interface these resources in the application source code. The resources in our abstraction are also vendor-agnostic: they are specified as part of the verb library [2], which work across different vendors' RNICs. It is important to note that our performance abstraction intentionally excludes explicit consideration of RNIC microarchitecture resources, such as on-NIC cache and NIC processing units. These components are vendor-specific and generally opaque to RDMA developers.

Moreover, our performance abstraction includes the typical resources other performance isolation solutions use, such as Request Per Second (RPS). We categorize RPS into data verbs RPS (DRPS) and control verbs RPS (CRPS) as they serve different purposes. While a more granular categorization of DRPS into sub-types such as ATOMIC RPS or SEND RPS is conceivable, we have chosen to opt for a normalized RPS, balancing precision with user-friendliness. The analogy is that CPU vendors use cycles instead of instructions per second as the performance metric because instructions can have variable lengths. To illustrate, Table 1 presents an example detailing the guaranteed metrics for a tenant within this framework. Let us assume one ATOMIC request consumes the resources equivalent to 3 SEND requests. Alice, with 30M DRPS, therefore can achieve up to either 10M ATOMIC requests per second or 30M SEND requests per second. It should be noted that DRPS and BPS guarantees are offered in a mutually exclusive "OR" fashion. For instance, a tenant consistently posting SEND requests with large message sizes will encounter BPS throttling before reaching the DRPS limit. Next, we present the design and implementation details of Harmonic that uses the above performance abstraction to provide RDMA performance isolation.

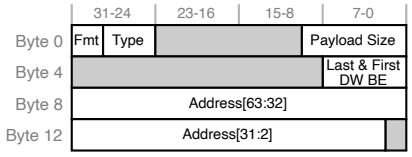


Figure 4: TLP header format where the gray blocks represent unused fields for PIPS. DW BE denotes dword byte enable.

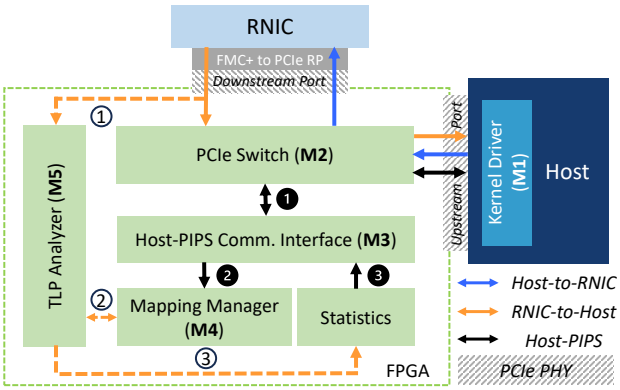


Figure 5: Programmable Intelligent PCIe Switch (PIPS) internal architecture. The dash line indicates asynchronous TLP analysis, decoupled with PCIe switch forwarding path.

4 Programmable Intelligent PCIe Switch

To monitor tenant's verbs behavior through PCIe, we develop a Programmable Intelligent PCIe Switch (PIPS) that can forward PCIe Transaction Layer Packets (TLPs) at line rate and perform real-time RDMA-centric inspections. Given address-to-object/tenant mappings captured in kernel driver, we extract the physical address of the host memory from the RNIC-issued DMA read/write TLP header (Address field in Figure 4) and utilize it to identify both the object and the tenant associated with this TLP. This capability enables us to accurately measure per-tenant RDMA resource utilization.

We build PIPS using AMD/Xilinx Versal VCK190 Evaluation FPGA board with 4K lines of RTL Verilog code and various AMD/Xilinx IPs (Intellectual Property Core). PIPS has five Modules (Figure 5): **(M1)** kernel driver, **(M2)** PCIe switch, **(M3)** host-PIPS communication interface, **(M4)** mapping manager, and **(M5)** TLP analyzer. The kernel driver maintains latest address-to-object/tenant mappings. The PCIe switch routes TLPs to their corresponding destinations. Host-PIPS communication interface and mapping manager handle the synchronization of address-to-object/tenant mappings between host and PIPS while collecting RDMA traffic statistics. The TLP analyzer inspects the TLP headers of RNIC-initiated DMA read/write requests and matches them with the address-to-object/tenant mappings.

4.1 PCIe Configuration and Routing Logic

The PCIe switch **(M2)** is the key component of Harmonic. It consists of routing logic and two instances of Xilinx Versal ACAP Integrated Block for PCI Express IPs [8]. The PCIe PHYs in the two instances are configured as PCIe switch's upstream port and downstream port, respectively. Figure 5 demonstrates Harmonic architecture: the upstream port is directly connected to the host using the PCIe edge connecter of FPGA, and the downstream port leverages the FMC+ expansion connector with a PCIe Root FMC+ plug-in module [23] to be connected to RNIC.

4.2 Address-to-Object/Tenant Mappings

Maintaining real-time address-to-object/tenant mappings in PIPS is essential for precisely monitoring RDMA resource usage per tenant. These mappings can change when applications create, delete, or modify objects. The change of the mappings is triggered by control verbs posted by RDMA applications, which are processed by the kernel driver **(M1)**. Therefore, we modify a legacy NVIDIA RNIC kernel driver (*e.g.*, `mlx5_ib.ko` and `ib_uverbs.ko`) to track address-to-object/tenant mappings. We use container's process ID as tenant ID. When a tenant calls a control verb, the Harmonic kernel driver first traverses the process tree in the kernel to find the tenant ID. It then records a mapping entry for this control verb behavior, including tenant ID, the type (*e.g.*, QP creation), the size and start physical address of the object. The RNIC kernel has already translated the virtual addresses for these RDMA objects to physical addresses for its DMA purpose, and we can directly use these translated physical addresses to populate our mapping entries. For application payloads, we also record the payload registered flags (*e.g.*, ATOMIC enabled). This information helps us determine the type of payload regions accessed by tenants in PIPS. The kernel driver is responsible for updating address-to-object/tenant mappings on PIPS by embedding an operation code in mapping entry to signal insert or delete operations to PIPS. We show detailed format and contents of both address-to-object/tenant mapping and statistics entry in Appendix B.

4.3 Mapping Synchronization and Management

We obtain address-to-object/tenant mappings from the kernel driver and then utilize the host-PIPS communication interface **(M3)** and the mapping manager **(M4)** to continuously update and manage the most up-to-date mappings in PIPS. This is crucial for later use by the TLP analyzer.

The host-PIPS communication interface receives and parses the MMIO write requests from host to update address-to-object/tenant mappings in the PIPS mapping manager (❶, ❷). Out of performance (*i.e.*, achieving real-time monitoring) and implementation complexity considerations, the mapping manager employs a hashing-based mechanism and maintains a hierarchical mapping storage system, consisting of a first-level (L1) direct-map scheme and a second-level (L2) linked-list slot pool. The mapping manager utilizes a double-hash strategy and leverages two distinct hash functions for calculating the hash values of the address field as the indexes to L1 and L2, respectively. Note that L2 is only used when collision happens in L1. In this case, each mapping entry in L1 is treated as the head of a linked list, with the remaining entries being stored in L2 linked-list slot pool. In addition to mapping management, the host-PIPS communication interface also generates completion TLPs with associated statistics as payload, when the host polls RDMA traffic statistics through MMIO read requests (❸).

4.4 Efficient TLP Analyzer

The TLP analyzer **(M5)** is responsible for extracting the target physical address in TLP headers from RNIC-issued DMA read/write requests (①). When a TLP arrives at PIPS, it duplicates the TLP and sends one copy to the TLP analyzer for analysis, while simultaneously forwarding the original TLP to its destination. In addition, the TLP analyzer implements an efficient search engine to collaborate with the mapping manager, which can perform `search` operation (②) in parallel with `insert` and `delete` operations, taking the hash value of physical address in TLP header as search key. Since the hash collision rate is low, the average search time is only 7 cycles including the latency for interconnection and updating statistics.Upon a mapping search hit, the TLP analyzer computes the statistics entry offset based on `TID`, `flags`, and `type` found in retrieved mapping entry, along with the direction of current TLP (*i.e.*, RNIC DMA read/write Host). Then it updates the statistics entry at this determined offset (③). With this approach, PIPS maintains an accurate record of both the access count and the volume of bytes accessed for each object and tenant, while simultaneously identifying the type and flag associated with the accessed memory.

5 RDMA-friendly Rate Limiting

Harmonic daemon is responsible for modulating per tenant's resource usage. It achieves this by employing two distinct rate limiting techniques for data verbs and control verbs.

5.1 Data Verbs Rate Limiting in Harmonic Daemon

The Harmonic daemon takes the proactive approach of creating and injecting Congestion Notification Packets (CNPs) to control tenants' rate. Because commodity RNICs automatically generate CNPs within the ASIC without providing an interface to users, the Harmonic daemon forges CNPs and sends them to the data sender side of tenants. Forging CNP needs the source and destination IP addresses as well as the remote QP number (QPN). Harmonic daemon obtains this information during the setup of connections. When tenants create or modify QPs, these control verbs are intercepted by Harmonic kernel driver. Subsequently, Harmonic kernel driver sends an event to notify Harmonic daemon that a new connection is set up, including both IP addresses and the QPN.

Harmonic daemon decides which tenant should be paced and at what specific rate. Harmonic daemon first keeps polling statistics collected by the PCIe switch through MMIO reads. These statistics include BPS and RPS of various types of RNIC-initiated DMA requests, such as fetching WQEs, fetching QP context, and writing payload into host memory. Then, Harmonic daemon calculates per tenant NIC BPS, PCIe BPS, DRPS consumption, as well as cache miss frequency based on the collected statistics. It sums up tenant's DMA accesses to various types of payloads (*e.g.*, WRITE) to calculate NIC BPS and DRPS, and sums up tenant's DMA accesses to various types of RDMA metadata (*e.g.*, QP contexts) to calculate cache miss frequency. For DRPS, we normalize different types of RDMA requests into the same unit, based on an estimated cost ratio for various types of data verbs. We conduct an offline profiling to estimate this cost ratio by running a set of micro-benchmarks. We run `perftest` [1] to send requests of minimal sizes in a batch to measure the maximum rate of different data verbs, and the 1/rate is the cost. In practice, we normalized WRITE and SEND operations to 1 unit, READ to 1.1 unit and ATOMIC to 3 units. Given the accurate resource usage per tenant, Harmonic daemon next compares each tenant's current usage and its allocation. Harmonic daemon directly uses NIC BPS and DRPS from tenant's profile (*e.g.*, Table 1), and calculates tenant's PCIe allocation using dominant resource fairness model [18]. Harmonic daemon analyzes guarantee profiles of all tenants on the same host and identifies the dominant resource among them. Then Harmonic daemon distributes the PCIe bandwidth based on the allocation of this dominant resource. For example, given a network capacity as 25 Gbps bandwidth and 30M DRPS, let us assume tenant A needs 15 Gbps bandwidth and 10M DRPS and tenant B needs 5 Gbps and 15M DRPS. The dominant resource therefore is bandwidth for tenant A ($\frac{2}{3}$) and DRPS for tenant B ($\frac{1}{2}$). We next allocate the available PCIe bandwidth to tenants A and B following the proportion of 4:3 (*i.e.*, $\frac{2}{3}/\frac{1}{2}$).

When a tenant uses more BPS/DRPS/PCIe bandwidth than its allocation, we send CNPs to data sender ends of this tenant's connections. Harmonic daemon currently applies a simple strategy to compute the CNP rate. Harmonic sends 1-4 CNPs in a batch after T_i intervals (in microseconds) to manage tenants' rate. Equation 1 shows how interval is updated based on the measured rate and target rate. We use two heuristic parameters T_{min} and T_{basic} in practice. T_{min} is a minimal interval threshold to avoid excessively frequent adjustments, which could lead to unstable rate or even cause performance anomaly. T_{basic} serves as a multiplier, reflecting the intrinsic response sensitivity to resource overuse. Tuning these values can adjust the strictness of policy, as a small T_{basic} punishes tenants that overuse resources more strictly.

$$T_i = max(T_{min}, T_{basic} * (1.0 - \frac{R_{current} - R_{target}}{R_{target}}) \quad (1)$$

We specially handle on-NIC cache resources due to their unique characteristics. While we can measure tenant's cache miss statistics by tracking the number of PCIe access to those metadata (*e.g.*, QP context), we do not set a cache miss threshold for each tenant. This decision is because a higher cache miss rate in one tenant does not necessarily indicate an excessive use of cache resources. Instead, we monitor overall RNIC cache contention and slow down tenants accordingly. When Harmonic daemon observes severe cache misses, Harmonic starts to slow down tenants with the lowest priority. For tenants with the same priority, we slow down them using the dominant resource fairness policy mentioned above. We acknowledge that there are alternative policies, such as monitoring a tenant's active QPs/MRs as the basis for rate-limiting decisions. However, we find that our straightforward policy is already effective in providing isolation when cache contention arises.

5.2 Control Verb Rate Limiting in Harmonic Drivers

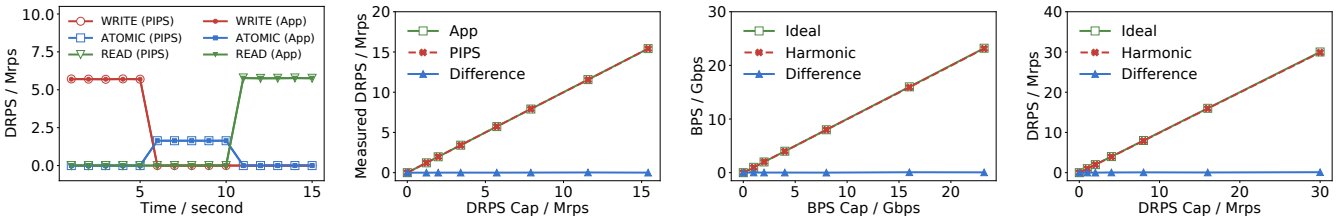
Control verbs rate limiter first needs to limit the capacity of control verbs (akin to in-flight packets) for each tenant, including the maximum number of QPs and MRs allowed per tenant. We record tenants' control verbs guarantee profiles as a linked list in Harmonic kernel driver. When a new tenant is created, we invoke Harmonic kernel driver to register a new control profile and insert it to the linked list. Whenever this tenant calls a control verb, Harmonic driver checks its current resource usage and the profile, determining if this control verb should be rejected or not.

We also need to limit the rate for control verbs to prevent tenants from excessively updating hardware status. Frequent updates have the potential to induce RNIC cache thrashing, as discussed in prior work [29]. We record timestamps for each tenant in our defined structure when they issue control verbs. When a tenant calls a control verb, we compare the current timestamp and the previously recorded timestamps. If the tenant is making control verb calls at a rate that exceeds their allocated rate, we introduce a sleep delay. We choose to slow down tenants through sleep instead of returning an explicit error. This way, Harmonic remains transparent to tenants. If we directly return errors to applications, it would necessitate error code checks and retries in applications.

6 Evaluation

6.1 Testbed Setup

There are two servers in our testbed, each equipped with one NVIDIA ConnectX-6 Dx (CX-6) 25 Gbps RNIC. Our FPGA-based programmable PCIe switch supports up to PCIe Gen 4 with 8 lanes with up to 128 Gbps PCIe bandwidth. Nevertheless, there are no NVIDIA 100 Gbps RNICs that support PCIe Gen 4 with 8 lanes. We therefore use Harmonic to enhance our CX-6 25 Gbps RNIC. RNICs of two hosts are directly connected without a network switch. The BPS capacity of our RDMA endhost is 25 Gbps. We use the standard RDMA benchmark tool, `perftest` [1], to measure the DRPS capacity,



(a) Measurement of operation types (b) Measurement of RPS (c) Control of BPS (d) Control of RPS

Figure 6: Measurement and control of RDMA traffic. App denotes the performance metrics as reported by `perftest`.

and the result is ∼30 M DRPS.

Both servers are running Ubuntu 20.04. Harmonic kernel driver is built upon MLNX_OFED-5.8.1.1.2.1 [42], with a total of 658 lines of C code modifications. Harmonic daemon is implemented in C/C++ with a total of 2537 lines of code.

6.2 Measurement and Control of RDMA Resources

We first use microbenchmarks to demonstrate that Harmonic can accurately measure tenants' verbs behaviors and limit their resource usage. We let a tenant run different data verbs workload from `perftest` in three time periods. It generates WRITE traffic, ATOMIC traffic, and READ traffic, each for 5 seconds. We record the DRPS measured by the `perftest` per second and compare it with the request rate measured by PIPS. As shown in Figure 6a, PIPS successfully identifies the types of data verbs and measures the request rates of each workload accurately. Figure 6b shows that Harmonic also accurately measures tenants' behaviors across different request rates.

Next, we evaluate our CNP-based RDMA-friendly rate limiter. We use `perftest` to generate workloads that extensively consume BPS and DRPS resources. We let Harmonic to set different capacity for these two resources. We measure the achieved BPS/DRPS and compare them with the capacity. Figure 6c and Figure 6d show that our rate limiter can accurately control a tenant's BPS and DRPS. It is worthwhile to note that we observe that Harmonic daemon can react to resource overuse within one millisecond. As discussed in Equation 1, a stricter T_{basic} or T_{min} leads to fast reaction (*i.e.*, a few hundreds of microseconds) while it may also hurt overall performance. In practice, we set T_{basic} to 500 us and T_{min} to 200 us, which we find already sufficient to enforce isolation.

6.3 Harmonic End-to-end Evaluation

We use the state-of-the-art RDMA performance isolation test suite, Husky [29], to perform end-to-end evaluation of Harmonic. Husky includes a set of victim traffic patterns that are sensitive to different types of resource contention, and four sets of attacker traffic patterns that exhaust four types of resources: RNIC BPS, RNIC processing capacity, RNIC cache, and RNIC PCIe bandwidth. We observe that the reliable connection retransmission attack described in Husky (Section 3.3) that exhausts RNIC processing capacity has already been fixed in the latest NIC firmware, and the RNIC control verbs cache attack only has a negligible effect on 25 Gbps RNIC. Harmonic passes all other Husky's tests with a tolerance level $\alpha = 20\%$, indicating a tenant's traffic will be no less than 80% of its guarantee in the worst case, which is substantially better than all existing solutions. For most tests, Harmonic effectively safeguards tenants to achieve their guarantees (*i.e.*, less than 5% difference). We next use a set of typical workloads from Husky as the case study to demonstrate why existing solutions fail and how Harmonic satisfies tenants' guarantee.

For each case study, we also compare our results with three baselines: (1) SR-IOV, which allocates individual virtual function (VF) for each tenant to use [47]; (2) Separate hardware traffic class (HW TC), which is supported by modern RNICs to dedicate a RNIC traffic class to specific tenant for quality-of-service (QoS) control and performance isolation [45]; (3) Justitia, a recent software-based isolation solution for RDMA networks [62]. Note that Justitia requires tenants to use specific userspace libraries, so a malicious tenant can circumvent Justitia's control by not using these Justitia's libraries. However, we still want to evaluate Justitia's isolation mechanism (which includes its rate limiter design).

We allocate two tenants, named Alpha and Beta, on the same pair of hosts. Our RDMA hosts support up to 25 Gbps and ∼30 M DRPS. We thus set our isolation goal to be that both Alpha and Beta are guaranteed with 12.5 Gbps and 15 M DRPS. For each attack, we let Alpha run a Husky victim traffic that is sensitive to a specific type of resource, and we let Beta run an attacker traffic targeting the specified resource. Results are shown in Figure 7 to Figure 10. *No Interference* means running Alpha or Beta alone with no isolation enabled, and *No Protection* means Alpha and Beta are running together without any isolation. For a fair comparison, we configure SR-IOV and HW TC to assign one virtual function or traffic class to each tenant, respectively. Justitia currently only supports fair share and does not provide any QoS guarantee. SR-IOV and HW TC only support RNIC bandwidth (*i.e.*, BPS) guarantee. We therefore configure each virtual function with 12.5 Gbps for SR-IOV. HW TC currently does not support floating-point rate configuration, and we thus configure both tenants with 12 Gbps guarantee and reserve 1 Gbps for potential traffic burst. For each figure, we use a *dashed red line* to denote the guarantee, a *gray dashed line* to de-

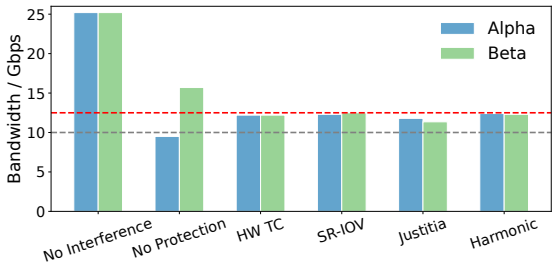


Figure 7: NIC bandwidth contention.

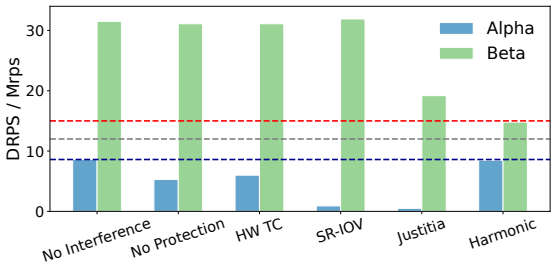


Figure 8: Processing capacity contention.

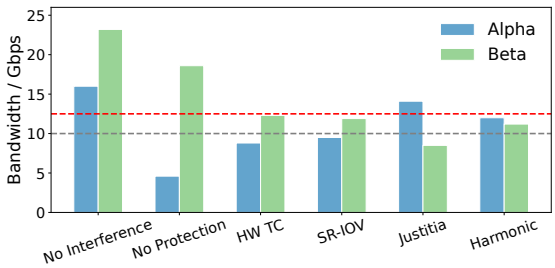


Figure 9: RNIC cache contention.

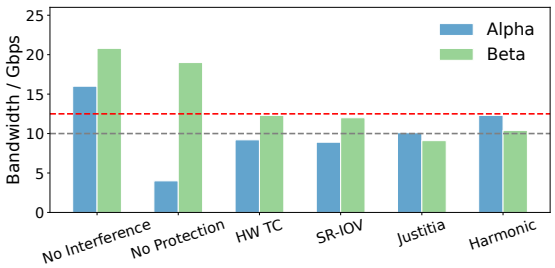


Figure 10: PCIe bandwidth contention.

note the tolerance bar (10 Gbps and 12 Mrps). We use a *blue dash line* in Figure 8 to show the victim performance since it is originally smaller than the guarantee and should not be affected.

RNIC BPS contention. We first conduct BPS contention experiment. Alpha sets up a single connection and keeps sending 64KB WRITE verbs, and Beta sets up 16 connections and keeps sending 4KB WRITE verbs in a batch. Both Alpha and Beta consume almost all the RNIC BPS when running alone. When running together without isolation, Beta occupies more BPS since it has more connections. When isolation is enabled, we observe that all existing solutions and Harmonic successfully satisfy all tenant's guarantees. This shows that RNIC BPS is accurately monitored and controlled by all the existing solutions and Harmonic.

Processing capacity contention. We let Alpha run a throughput-sensitive Husky victim, which uses 36 connections and keeps issuing 64-byte messages. Beta uses 64 connections to keep generating expensive 8-byte ATOMIC traffic to exhaust RNIC processing capacity. We normalize DRPS based on our profiling results, which show that a single ATOMIC operation costs roughly three times as WRITE operations. Figure 8 shows the DRPS for Alpha and Beta. Note that Alpha does not use all its traffic demand, so the isolation goal is that Alpha's performance should not be affected when Beta joins (shown as the blue dashed line). When no isolation is enabled, Beta's ATOMIC workloads exhaust the RNIC processing capacity and cause Alpha's performance to drop by 38%. HW TC does not react to this attack effectively because Beta only consumes a small amount of BPS (i.e., 6.7 Gbps), which is substantially lower than the rate limit. Beta's rate therefore is not paced by HW TC, and Beta exhausts the RNIC processing capacity. When SR-IOV is enabled, Beta's rate is not reduced as well, and we observe Alpha's rate drops. Since SR-IOV implementation details are not publicly available, our best guess is that this workload may cause some scheduler issues in SR-IOV implementation. Even though Justitia considers processing capacity in its design, it is agnostic to the type of verbs and thus performs even worse. It does reduce Beta's ATOMIC traffic, but Alpha's performance is even more severely degraded. This is because Justitia treats these verbs equivalently without accounting for the actual resource consumption of expensive ATOMIC verbs. Our observation of existing solutions is aligned with Husky's results. Harmonic carefully considers the expensive costs of ATOMIC requests and limits Beta's rate accordingly, reserving adequate processing capacities to achieve Alpha's guarantee while satisfying Beta's requirement.

RNIC cache contention. We let Alpha run a Husky victim that is sensitive to on-NIC cache contention, which keeps generating 8-byte WRITE requests in batches across 512 different memory regions. Beta runs a Husky attacker that uses 4 connections to repeatedly issue single 512-byte WRITE request to 16K different memory regions to exhaust the on-NIC cache resources. Figure 9 shows that when cache contention occurs, the available RNIC BPS is less than 25 Gbps. Even though both SR-IOV and HW TC reduce Beta's BPS consumption to less than 12 Gbps, Alpha's performance is only improved by a minimal extent. We suspect that under severe cache contention, the effectiveness of SR-IOV and HW TC is also affected. For example, the severe cache miss may also slow down the SR-IOV and HW TC scheduling process. Though the current design of Justitia is cache agnostic, it successfully satisfies Alpha's guarantee while leading to a drastic drop in Beta's performance, making it not satisfy the guarantee even with a 20% tolerance level. This is probably because Justitia identifies both applications as throughput-sensitive applications and schedules them equivalently, while Alpha

issues smaller messages in batch and therefore occupies more Justitia's tokens. Harmonic detects the cache contention and measures the available BPS. It then allocates the reduced available BPS to Alpha and Beta fairly. This makes both Alpha and Beta achieve the guarantee within the tolerance level. Note that the strict guarantee is impractical in this case because the bottleneck is the RNIC cache.

PCIe contention. Though our testbed supports up to $\sim$ 64 Gbps PCIe bandwidth. We configure our FPGA-based PCIe switch to only support 32 Gbps PCIe bandwidth for our 25 Gbps RNIC. This $\frac{PCIe\ BW}{RNIC\ BW} = \frac{32}{25}$ ratio emulates scenarios for higher speed RNICs (*e.g.*, $\frac{128}{100}$ and $\frac{256}{200}$), where PCIe bandwidth can be one of the bottlenecked microarchitecture resources. We let Alpha run the same application as in the Cache contention case. We let Beta run the PCIe attack in Husky, which keeps sending 257-byte WRITE that triggers several DMAs to maximize PCIe consumption. As shown in Figure 10, the available RNIC BPS therefore is capped by the PCIe bandwidth and is substantially smaller than 25 Gbps. Both SR-IOV and HW TC successfully reduce Beta's rate and improve Alpha's performance, but to a limited extent. The key reason is that given the same amount of RNIC BPS consumption, Beta consumes more PCIe bandwidth than Alpha and should be paced more in this situation. Similar to what is observed in the above cache contention scenario, Justitia reduces Beta more because of its larger message size and only satisfies Alpha's guarantee. Harmonic's hardware monitor allows us to accurately track each tenant's PCIe bandwidth consumption and allocate PCIe bandwidth accordingly based on tenants' guarantee. For example, each tenant is allowed to consume half of the PCIe bandwidth (*i.e.*, roughly 16 Gbps) in this situation.

6.4 Performance Isolation for End-to-End Applications

We evaluate how Harmonic provides performance isolation for a real application. We use an RDMA-based Redis [63] as our tenant workload. We use the same Husky attack workloads described in the previous section as attackers. Similarly, our isolation goal is to enforce fair share resource allocation between the Redis application and the attackers. We also enable both SR-IOV and HW TC as a comparison. We do not evaluate Justitia's performance for two reasons: (1) Jusitia needs application modification to fully support its isolation and is not secured for real cloud deployments; (2) Justitia does not support READ operations in the latest drivers.

Redis over RDMA implements an RDMA backend transport to accelerate Redis key-value store and has been large-scale deployed in industry [63]. We use this redis-benchmark application to generate 1KB get and set workloads, and measure its average application QPS. This benchmark can achieve about 450K QPS, consuming 4.2 Gbps BPS and 1.2 Mrps DRPS. This is less than its performance guarantee, so the goal is that Redis's performance should not be affected by any attacker. We then run those four types of attacks without

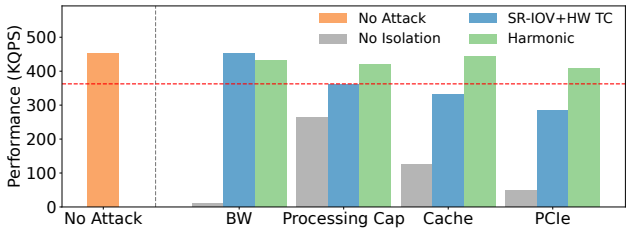


Figure 11: Performance of Redis over RDMA across different attack types and isolation schemes.

isolation, with SR-IOV + HW TC, and with Harmonic. As shown in Figure 11, all four types of attack successfully exhaust specific types of RDMA resources and cause a drastic Redis performance drop. When isolation is enabled, we observe that both SR-IOV + HW TC and Harmonic successfully provide protection against an attacker that tries to exhaust network bandwidth. Though SR-IOV + HW TC does not consider processing capacity, Redis achieves its guarantee under processing capacity contention with SR-IOV + HW TC. This is probably because Redis workload is more robust to the processing capacity contention. However, SR-IOV + HW TC fails to provide sufficient isolation when cache or PCIe bandwidth is contended. Harmonic proactively monitors these microarchitecture resource contention and applies rate limit according to per tenant's usage. Harmonic therefore successfully maintains Redis's performance within the tolerance level when on-NIC cache or PCIe bandwidth is under contention performing 1.3x$\sim$1.4x better than the combination of two state-of-the-art isolation solutions.

6.5 Overhead Analysis

Hardware and PCIe costs. Our hardware cost analysis based on the implementation report from AMD Vivado [9] shows that PIPS, with an internal reference clock frequency set at 250 MHz, consumes 8,571 LUTs (*i.e.*, 0.95% of VCK190 FPGA LUT resources), and 554 BRAMs (*i.e.*, 57.29% of VCK190 FPGA BRAM resources) mainly used to store host mapping entries.

We also measure the cost of PCIe bandwidth for updating mappings and collecting statistics between the host and PIPS. RDMA application does not frequently invoke control verbs during data transmission, the mapping updates consumption is therefore negligible. During our evaluation of various Husky's attack workloads, we observe that the mapping updates only consume no more than 8 Mbps (0.025%) extra PCIe bandwidth. The extra PCIe bandwidth consumption caused by polling statistics is determined by the polling frequency. In practice, we poll these statistics every 100 us and we find it already sufficient to achieve an accurate rate control and enforce performance isolation. The per tenant PCIe bandwidth consumption is 64 Mbps for host-to-switch direction and 76.8 Mbps for switch-to-host direction, which can be comfortably accommodated by the bandwidth slack between PCIe and RNIC line rate. The detailed calculation and

Table 2: Network performance overhead.

	Latency (us)		Max. Bandwidth (Gbps)	Max. Throughput (Mrps)
	64B	64KB		
Baseline	3.3	50.4	23.0	28.1
Harmonic	5.6	52.6	22.8	28.1

analysis is in Appendix C. Harmonic daemon currently only consumes 33.5% of a single CPU core and scales with negligible CPU usage increment. The CPU usage is determined by the frequency of polling statistics.

Network performance overheads. We run microbenchmarks using `perftest` to measure the latency, achieved bandwidth, and request throughput with and without Harmonic to analyze network overheads introduced by Harmonic. For brevity, we show the results of RDMA READ in Table 2. We demonstrate the latency penalty under different packet sizes in Figure 12. While there is a marginal increase in latency overhead with larger packet sizes, Harmonic adds less than 2 us to the round-trip latency across all packet sizes. This is mainly because our PCIe switch is implemented in FPGA, which is less performant than traditional ASIC-based PCIe switches on the host and SmartNICs. Furthermore, employing PCIe extender card and FMC+ to PCIe root module for the purpose of full-system operation can also incur additional latency. Note that our monitoring feature is decoupled with the PCIe switch forwarding functionality, so the monitoring feature does not contribute to this overhead at all. Besides, Harmonic introduces only negligible drops in network bandwidth or request throughput.

To summarize, Harmonic's overhead is negligible for high-speed RDMA networks. Additionally, we believe that the monitoring and rate limiting functions inherently should be integrated into future generations of RNICs. Overheads, such as the extra PCIe consumption and the FPGA's latency, will be further eliminated when these functions are implemented within RNIC's ASIC. For example, NVIDIA Bluefield-2 SmartNIC has an embedded PCIe switch that routes RDMA traffic among RNIC ASIC, embedded ARM CPUs, and host [44], and only introduces nanosecond-level latency overhead [59]. We therefore believe Harmonic's PIPS overhead can also be mostly eliminated by implementation in ASIC and being integrated into RNIC.

7 Discussion

Scaling to higher-speed network. We believe our solution is scalable to 100/200 Gbps RNICs because the overhead of Harmonic (*i.e.*, FPGA resources usage, extra PCIe bandwidth consumption) does not increase with higher network capacity. The concerns may fall on whether TLP analyzer can keep up with higher PCIe bandwidth and whether the mapping manager scales to store more mapping entries. Our TLP analyzer can handle higher PCIe bandwidth, as the average search and update time of our design for one mapping entry is 7 cycles at 250MHz frequency. This can be even further minimized with increased parallelism. The architecture of the mapping manager can easily be extended to a multi-hashing hierarchy, thereby facilitating the storage of a greater number of mappings with only a marginal increase in search time. Additionally, concepts from match-action table of P4 switch and more advanced mapping management like binary search or Cuckoo hashing can be implemented on top of PIPS to further reduce memory overheads. In the meanwhile, the scalability concerns are notably mitigated when considering an ASIC implementation with optimized logic interconnections and resource utilization while running at a higher frequency.

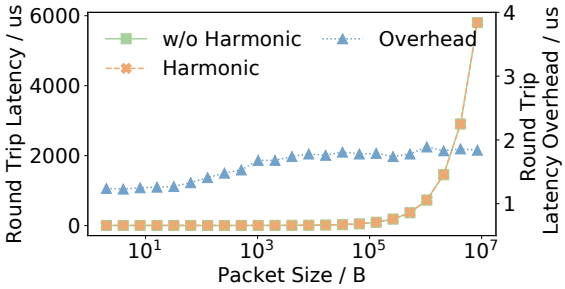


Figure 12: Latency overhead across different packet sizes. The green and orange lines present the absolute round trip latency with left y-axis when packet size differs. The blue line demonstrates the round trip latency overhead is less than 2 us using the right y-axis.

Is our performance abstraction easy for users to understand? Our performance abstraction is more complex than traditional performance abstraction which only considers network bandwidth. We believe this is necessary because RDMA network is indeed more complex and application developers are already interacting with this performance abstraction when developing RDMA programs [26, 27]. We only extend the abstraction to include more architectural resources that users can directly control, such as the number of QPs. These extended metrics are no difference from the number of vCPUs or the size of memory in today's cloud VMs specifications. We believe developers should be aware of these resources in order to write performant and predictable RDMA applications.

Deployability of Harmonic. Harmonic requires both hardware and software modifications to existing clouds. From the perspective of hardware, Harmonic uses PIPS as a prototype to measure per-tenant RDMA resource consumption at runtime. In practice, the best implementation entry point should be within the RNIC regarding performance and hardware costs. One leading technology enterprise is currently integrating part of our designs into their next-generation RNIC. In terms of software, Harmonic needs to have the full control of the RDMA kernel drivers to manage control verbs for all tenants. Containerized clouds have already provided such control since all tenants are sharing the same kernel managed by cloud operators. Harmonic software therefore can be deployed in containerized clouds without any barriers. In VM-based clouds, native SR-IOV does not support to manage control verbs for tenants since guest kernel drivers

can directly communicate with RNICs. Extra modification to both guest kernel drivers and hypervisors are required to deploy Harmonic in these scenarios. Existing solutions such as HyV [49] and MasQ [22] have already virtualized all control verbs involving hypervisor in VM-based clouds. This provides feasible entry points to integrate Harmonic's software features into these solutions.

RDMA-friendly rate limiting. We currently repurpose RNIC's native rate limiters to modulate tenants' RDMA resource usage by sending CNPs. This achieves efficiency and is transparent to applications, but we acknowledge that sending CNPs from software may not be the best approach in the future. For example, a transient network congestion may affect the accuracy of such rate limiting mechanism. Emerging RNIC features such as programmable congestion control (PCC) [43] allow customized congestion control algorithms. This potentially provides a more straightforward and accurate way to leverage RNIC's rate limiter for performance isolation purposes. For example, a data receiver can send specialized packets to specify the maximal sending rate that the data sender can enforce, similar to TCP receive window.

Generality of Harmonic. Harmonic currently targets at RDMA performance isolation, focusing on the bottlenecked RNIC microarchitecture resources. We believe Harmonic can also be leveraged for other scenarios besides RDMA networks. For example, multiple I/O devices (*e.g.*, GPU and NIC) may be connected to the same PCIe switch and thereby contend on PCIe and memory bus resources [30]. Harmonic can also be adapted to isolate resources among different I/O devices and hence manage the complex intra-host network.

8 Related Works

Understanding microarchitecture resources in RNICs. Research community has already started to study the hardware resources in RNICs. Existing works focus on how to avoid certain performance anomalies caused by NIC resources from the application layer [15, 26, 27, 29, 38]. Husky [29] discusses the definition of RNIC microarchitecture and conducts a holistic study on how different RDMA operations make use of on-NIC microarchitecture resources. Kalia et al. [27] provides guidelines for writing efficient high-performance RDMA programs. These works target understanding or optimizing the RDMA programs and the usage of some specific RNIC microarchitecture resources, but they do not provide RDMA performance isolation.

RNIC design. Several works have been conducted to optimize RNIC design [34, 37, 57, 58]. SRNIC [58] modifies both protocols and RNIC architecture to improve on-NIC memory efficiency and utilization for better scalability. IRN [37] proposes to enable fast loss recovery on NIC to avoid reliance on lossless fabrics. These works contribute to improving RDMA performance. However, our work targets at provide performance isolation for multi-tenant RDMA clouds.

Understanding intra-host communication. Intra-host communication has received increasing attention in research communities [5, 6, 30, 33, 39, 61]. Breaking Band [61] leverages an expensive commercial PCIe analyzer to get a system-level PCIe latency breakdown. Min [36] implements a simple soft PCIe switch to obtain CPU-GPU communication patterns at 10 Gbps. Neugebauer et al. [39] analyze the PCIe theoretical model and study how PCIe affects network performance. Harmonic targets a different angle. It sniffers intra-host communication traffic to monitor RDMA network behaviors for RDMA performance isolation.

Performance isolation and QoS. Previous research [12, 20, 25, 32, 50, 51] has already provided software-based solutions implemented on the endpoints (hosts) and achieved performance isolation and QoS, by ensuring VM-pair level bandwidth guarantee. However, centering around the TCP/IP kernel network stack, they mainly focus on the bandwidth contention of the network fabric (*e.g.*, switch, router, etc.) and provide pure software solutions to the narrow problem. PicNIC [32] uses the number of CPU cycles spent on the packet processing as a criterion of NIC contention for TCP/IP networks. Harmonic is an orthogonal and complementary research work, with a focus on performance isolation on the RDMA-capable endhost. An end-to-end network performance isolation solution requires isolation mechanisms in different components of the network, including both inter-host network bandwidth and RDMA NIC resources on the endhost. Harmonic provides a microarchitecture-resource-aware solution for the RDMA NIC resource isolation in addition to traditional network bandwidth.

9 Conclusion

We propose the first RDMA performance isolation solution for public clouds, Harmonic, that is aware of microarchitecture resources. Harmonic consists of an FPGA-based programmable intelligent PCIe switch to measure per-tenant RDMA resource usage and an RDMA-friendly rate limiter to modulate RDMA resource per tenant. Harmonic requires no application modification. We evaluate Harmonic using the state-of-the-art test suite for RDMA performance isolation. Our evaluation results show that Harmonic delivers strong RDMA performance isolation in a multi-tenant public cloud setting, compared to all the existing solutions.

Acknowledgement

We thank Xilinx for their technical support. We also thank our shepherd Yizhou Shan and other anonymous reviewers for their insightful feedback. Our work is supported in part by grants from NSF (CNS-2238665 and CCRI-CISE 2213808) and IBM IIDAI, and gifts from Adobe, Amazon, Meta, Intel, and IBM.

References

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

[2] RDMA Core Userspace Libraries and Daemons. https://github.com/linux-rdma/rdma-core/.

[3] Advanced Micro Devices. How to implement pcie switch on Ultrascale. https://support.xilinx.com/s/question/0D54U00006cnZ2rSAE/how-to-implement-pcie-switch-on-ultrascale?language=en_US.

[4] Advanced Micro Devices. Use Ultrascale+ PCIe Integrated block as an endpoint PCI to PCI Bridge device. https://support.xilinx.com/s/question/0D54U00006hwlreSAA/use-ultrascale-pcie-integrated-block-as-an-endpoint-pci-to-pci-bridge-device?language=en_US.

[5] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. Understanding Host Interconnect Congestion. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks (HotNets)*, pages 198–204, 2022.

[6] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. Host Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 275–287, 2023.

[7] Amazon. Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/.

[8] AMD/Xilinx. Versal Adaptive SoC Integrated Block for PCI Express LogiCORE IP Product Guide. https://docs.xilinx.com/r/en-US/pg343-pcie-versal.

[9] AMD/Xilinx. Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html.

[10] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end Performance Isolation Through Virtual Datacenters. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 233–248, 2014.

[11] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering Azure Storage with RDMA. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 49–67, 2023.

[12] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards Predictable Datacenter Networks. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 242–253, 2011.

[13] Broadcom. Introduction to Thor Congestion Control for RoCE. https://docs.broadcom.com/doc/NCC-WP1XX.

[14] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. Remote Procedure Call as a Managed System Service. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 141–159, 2023.

[15] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*, pages 1–14, 2019.

[16] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 51–66, 2018.

[17] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When Cloud Storage Meets RDMA. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 519–533, 2021.

[18] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 323–336, 2011.

[19] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C Snoeren. SmartNIC Performance Isolation with FairNIC: Programmable Networking for the Cloud. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 681–693, 2020.

[20] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *Proceedings of the 6th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 1–12, 2010.

[21] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 202–215, 2016.

[22] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. MasQ: RDMA for Virtual Private Cloud. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 1–14, 2020.

[23] HiTech Global. PCI Express Gen4 Root FMC+ Module. https://hitechglobal.us/index.php?route=product/product&path=18_85&product_id=273.

[24] Intel. Intel® Ethernet 800 Series Linux Flow Control. https://edc.intel.com/content/www/us/en/design/products/ethernet/800-series-linux-flow-control-configuration-guide-for-rdma-use-c/congestion-management-tuning-parameters/.

[25] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 297–311, 2013.

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

[27] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, pages 437–450, 2016.

[28] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 113–126, 2019.

[29] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R Lebeck, and Danyang Zhuo. Understanding RDMA Microarchitecture Resources for Performance Isolation. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 31–48, 2023.

[30] Xinhao Kong, Jiaqi Lou, Wei Bai, Nan Sung Kim, and Danyang Zhuo. Towards a Manageable Intra-Host Network. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 206–213, 2023.

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

[32] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, and Amin Vahdat. PicNIC: Predictable Virtualized NIC. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 351–366, 2019.

[33] Qiang Li, Qiao Xiang, Derui Liu, Yuxin Wang, Haonan Qiu, Xiaoliang Wang, Jie Zhang, Ridi Wen, Haohao Song, Gexiao Tian, Chenyang Huang, Lulu Chen, Shaozong Liu, Yaohui Wu, Zhiwu Wu, Zicheng Luo, Yuchao Shao, Chao Han, Zhongjie Wu, Jianbo Dong, Zheng Cao,

Jinbo Wu, Jiwu Shu, and Jiesheng Wu. From RDMA to RDCA: Toward High-Speed Last Mile of Data Center Networks Using Remote Direct Cache Access. *arXiv preprint arXiv:2211.05975*, 2023.

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

[35] Microsoft. Azure Virtual Machine series. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/series/.

[36] Seung Won Min. *Fine-grained memory access over I/O interconnect for efficient remote sparse data access*. PhD thesis, 2022.

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

[38] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a Feather Flock Together: Scaling RDMA RPCs with Flock. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 212–227, 2021.

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

[40] NVIDIA. DCQCN PARAMETERS. https://enterprise-support.nvidia.com/s/article/dcqcn-parameters.

[41] NVIDIA. Firmware Burning Tools (MFT). https://docs.nvidia.com/networking/category/mft.

[42] NVIDIA. MLNX_OFED InfiniBand/VPI. https://docs.nvidia.com/networking/category/mlnxofedib.

[43] NVIDIA. NVIDIA ConnectX-6 DX Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectX-6-dx-datasheet.pdf.

[44] NVIDIA. NVIDIA MELLANOX BLUEFIELD-2 Datasheet. https://network.nvidia.com/files/doc-2020/pb-bluefield-2-smart-nic-eth.pdf.

[45] NVIDIA. Quality of Service (QoS). https://docs.nvidia.com/networking/pages/viewpage.action?pageId=107485812.

[46] NVIDIA. Security Bulletin: NVIDIA ConnectX - April 2023. https://nvidia.custhelp.com/app/answers/detail/a_id/5459/~/security-bulletin%3A-nvidia-connectx---april-2023.

[47] NVIDIA. Single Root IO Virtualization (SR-IOV). https://docs.nvidia.com/networking/pages/viewpage.action?pageId=107485951.

[48] PCI-SIG. PCI Express® Base Specification Revision 4.0. https://pcisig.com/specifications.

[49] Jonas Pfefferle, Patrick Stuedi, Animesh Trivedi, Bernard Metzler, Ionnis Koltsidas, and Thomas R. Gross. A Hybrid I/O Virtualization Framework for RDMA-Capable Network Interfaces. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, page 17–30, 2015.

[50] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C Mogul, Yoshio Turner, and Jose Renato Santos. Elasticswitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 351–362, 2013.

[51] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C Snoeren. Cloud Control with Distributed Rate Limiting. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 337–348, 2007.

[52] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMArk: Bypassing RDMA Security Mechanisms. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, pages 4277–4292, 2021.

[53] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the Data Center Network. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 309–322, 2011.

[54] Konstantin Taranov, Benjamin Rothenberger, Daniele De Sensi, Adrian Perrig, and Torsten Hoefler. NeVerMore: Exploiting RDMA Mistakes in NVMe-oF Storage Applications. In *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2765–2778, 2022.

[55] Shin-Yeh Tsai, Mathias Payer, and Yiying Zhang. Pythia: Remote Oracles for the Masses. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, pages 693–710, 2019.

[56] VITA. FPGA Mezzanine Card Plus (FMC+) Standard. https://www.vita.com/fmc.

[57] Xizheng Wang, Guo Chen, Xijin Yin, Huichen Dai, Bojie Li, Binzhang Fu, and Kun Tan. StaR: Breaking the Scalability Limit for RDMA. In *Proceedings of the IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–11, 2021.

[58] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, Tianhao Wang, Weicheng Ling, Kejia Huo, Pingbo An, Kui Ji, Shideng Zhang, Bin Xu, Ruiqing Feng, Tao Ding, Kai Chen, and Chuanxiong Guo. SRNIC: A Scalable Architecture for RDMA NICs. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2023.

[59] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing Off-path SmartNIC for Accelerating Distributed Systems. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 987–1004, 2023.

[60] Zhuolong Yu, Bowen Su, Wei Bai, Shachar Raindel, Vladimir Braverman, and Xin Jin. Understanding the Micro-Behaviors of Hardware Offloaded Network Stacks with Lumina. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 1074–1087, 2023.

[61] Rohit Zambre, Megan Grodowitz, Aparna Chandramowlishwaran, and Pavel Shamis. Breaking Band: A Breakdown of High-Performance Communication. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP)*, pages 1–10, 2019.

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

[63] Zhenwei Pi. Redis Over RDMA Implementation. https://github.com/redis/redis/pull/11182.

[64] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 523–536, 2015.

A Harmonic Prototype Setup

We present the prototype setup of Harmonic in Figure 14. PIPS is implemented on an AMD Versal VCK190 FPGA board, connecting to the host system with a PCIe extender card. We connect RNIC with PIPS using an FMC+ expansion connector because the FPGA board does not contain a PCIe root connector interface. FMC+ is built upon FPGA Mezzanine Card (FMC) standard [56] which is a versatile and widely adopted standard for high-performance interfacing FPGAs with external devices.

B Entries for Mappings and Statistics

We illustrate the address-to-object/tenant mapping and statistics entry format in Figure 13. We explain how each field is derived to offer a comprehensive understanding of the mapping mechanism in Harmonic.

B.1 Address-to-Object/Tenant Mappings

There are mainly three types of objects (*i.e.*, memory regions, queue structures, and RDMA metadata) in RDMA. All these objects will be pinned in the host physical memory after creation, and the RNIC will maintain virtual-to-physical address mappings to DMA these objects.

The first type of object is application's memory region (MR). Applications register these MRs through `ibv_reg_mr`, which is processed by `mlx5_ib.ko` drivers in our NVIDIA testbed. We modify `mlx5_ib.ko` to record the starting physical address, the process ID (PID) of the caller, the size and the memory flags (*e.g.*, `IBV_ACCESS_REMOTE_WRITE`, which allows remote write) of this region. Note that we use container's process ID as tenant ID (TID).

The second type of objects is queue structures, including send/receive queues, completion queues, and the doorbell (memory mapped registers) for these queues. When an application initiates RDMA data verbs, the memory is accessed by the RNIC to fetch WQEs from send/receive queues or write completion queue entries (CQE) to completion queues. The memory for these objects is allocated and pinned during the allocation of these projects, such as `mlx5_ib_create_qp` in `mlx5_ib.ko`. Similarly, we record the PID of the caller, and the address and size of these objects.

The third type of object is RDMA metadata managed by RNIC driver and firmware, including QP contexts and memory translation/protection tables. When other two types of objects (*e.g.*, a QP) are created, the firmware reserves a few pinned pages and allocates metadata (*e.g.*, a QP context) in the pinned pages. We record the information on these pinned pages in a similar fashion as described above.

Figure 13a shows our unified entry to update such address-to-object/tenant mappings to our PCIe switch. Note that the

Opcode: 127-112	Rsvd: 111-96	Size: 95-65	Flags: 64-63	Addr: 62-15	TID: 14-3	Type: 2-0

(a) Address-to-Object/Tenant Mapping Entry Format.

# Byte: 63-31	# Access: 30-1	Direction: 0

(b) Statistics Entry Format.

Figure 13: Address-to-Object/Tenant Mapping and Statistics Entry Formats.

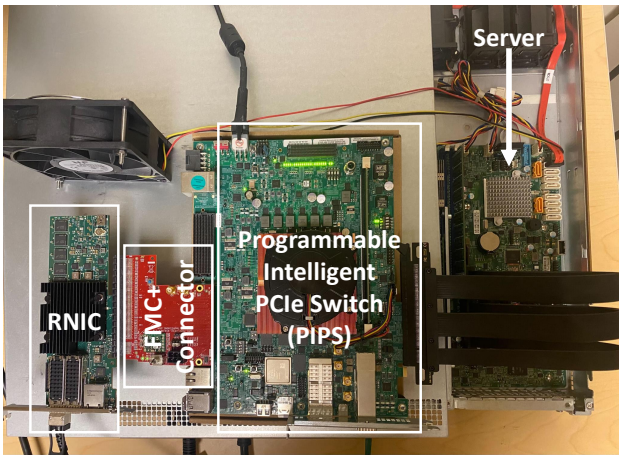


Figure 14: Programmable Intelligent PICe Switch (PIPS) Prototype.

most significant 16 bits of address-to-object/tenant mapping entry together serve as an operation code that notifies PIPS to either insert or delete the entry in PIPS. We reserve the second 16-bit field considering the possibility of other customization demands. Our modified drivers will fill in the remaining five fields and expose these entries to Harmonic daemon through system files.

B.2 Per-tenant RDMA Statistics

We store the monitored RDMA resource statistics in a 128-bit structured entry as shown in Figure 13b. As discussed in §4, the TLP analyzer leverages the physical address enclosed in TLP headers to search and retrieve the corresponding address-to-object/tenant mapping entry from which we identify the object and tenant associated with the TLPs. Then we collect and record PCIe bandwidth consumption, number of PCIe transactions, direction of TLPs, accessed memory type, and other information from TLP headers (Figure 4) in per-tenant statistics entries.

C Harmonic PCIe Overhead Computation

PIPS maintains 40 statistic entries per tenant and each entry is 8-byte. We issue PCIe read request to read these statistics from PIPS. For a PCIe read request, the minimum PCIe protocol overhead is 20 bytes [48]. Upon receiving the read request, PIPS responds with a completion packet (*i.e.*, Completion TLP), containing 8-byte payload and a 16-byte PCIe protocol overhead. Therefore, for a single statistics read, it consumes 800-byte for host-to-PIPS direction 960-byte for PIPS-to-host direction in total.

Assuming Harmonic daemon polls the statistics every N milliseconds. The extra PCIe bandwidth consumed therefore is $\frac{1000}{N} * 8 * 800 = \frac{6.4}{N}$ Mbps for the host-to-PIPS direction and $\frac{7.68}{N}$ Mbps for PIPS-to-host direction. Harmonic currently poll statistics every 100 us, which consumes 64 Mbps and 76.8 Mbps for these two directions. This overhead is less than 0.25% of the total PCIe bandwidth. Together with the extra PCIe bandwidth consumed by updating mappings, the overall PCIe bandwidth overhead of Harmonic is below 0.31% which can be comfortably accommodated by the existing 21.87% bandwidth slack between PCIe and RNIC line rate. Note that we assume our PCIe limit as 32 Gbps, which has the same network-to-PCIe capacity ratio as higher speed networks (*e.g.*, 100 and 200 Gbps). This means that the PCIe overhead of our solution remains negligible with a higher network speed. Not to mention that this PCIe overhead only depends on the number of tenants and the frequency of polling, independent on network bandwidth.

LDB: An Efficient Latency Profiling Tool for Multithreaded Applications

Inho Cho[1] Seo Jin Park[2] Ahmed Saeed[3] Mohammad Alizadeh[1] Adam Belay[1]
[1] *MIT CSAIL* [2] *University of Southern California* [3] *Georgia Tech*

Abstract

Maintaining low tail latency is critical for the efficiency and performance of large-scale datacenter systems. Software bugs that cause tail latency problems, however, are notoriously difficult to debug. We present LDB, a new latency profiling tool that aims to overcome this challenge by precisely identifying the specific functions that are responsible for tail latency anomalies. LDB observes the latency of all functions in a running program. It uses a novel, software-only technique called *stack sampling*, where a busy-spinning stack scanner thread polls lightweight metadata recorded in the call stack, shifting tracing costs away from program threads. In addition, LDB uses *event tagging* to record requests, inter-thread synchronization, and context switching. This can be used, for example, to generate per-request timelines and to find the root cause of complex tail latency problems such as lock contention in multi-threaded programs. We evaluate LDB with three datacenter applications, finding latency problems in each. Our results further show that LDB produces actionable insights, has low overhead, and can rapidly analyze recordings, making it feasible to use in production settings.

1 Introduction

Modern datacenter services like search, social networks, and DNN training operate on huge datasets with complex communication patterns and large numbers of servers [17, 28]. Tail latency is a key challenge in this setting because overall performance is often limited by the slowest response [22]. Despite the tremendous effort that goes into optimizing latency-sensitive programs, operators tend to treat high tail latency as inevitable due to the complexity of deployed programs. Therefore, the main method available to operators today is to keep machine utilization low to control for tail latency, wasting both power efficiency and money.

In this paper, our aim is to empower developers to tackle tail latency problems head-on by answering the following question: *Can a debugging tool identify the precise source of tail latency experienced by a request in a server (e.g., the line of code that is responsible)?* This is a significant challenge, as the effort needed to understand tail behaviors is formidable with the tools that exist today. Statistical profilers (e.g., Linux's perf-tool), for example, have only limited utility because their method of periodic sampling captures the average runtime of functions, which may deviate significantly from the tail runtime. Further, they don't account for request semantics, so they cannot differentiate between requests running on the critical path versus the background, making it hard to identify bottlenecks. Instead, developers commonly hand instrument code locations that they *suspect* are problematic, but they can only try a few locations at a time due to instrumentation overhead. Thus, a typical workflow involves multiple iterations of instrumentation location adjustment, deployment, and data collection.

One way to avoid this tedious process would be to use a tool that can instrument all functions simultaneously (e.g., XRay [18]). However, this approach causes significant overhead that can distort an application's behavior. A less invasive option would be to use hardware assistance. For example, Intel recently introduced a hardware extension called Intel Processor Trace (Intel PT) that records every control flow operation (calls, branches, jumps, etc.) to an in-memory log for analysis. NSight recently demonstrated that Intel PT can be used to derive rich tail latency insights, such as a precise timeline of how cycles are spent handling network requests [26].

Unfortunately, Intel PT has drawbacks that make it difficult to use for profiling latency in practice. First, Intel PT is proprietary and requires hardware support, so it is only available on certain platforms. Second, Intel PT generates data at an enormous rate, so it is only feasible to record a few seconds of samples. Finally, Intel PT's compression scheme requires a software decoder that walks a program's object code to reconstruct its control flow. This requires several hours of processing—even for a few seconds of data—prohibiting interactive profiling.

We present LDB, a new latency debugging tool that provides unprecedented visibility into the latency behavior of applications. LDB reports the distribution of the latency of all

```cpp
std::mutex lock;
std::map<int, std::string> db;

void snapshot() {
  std::ofstream out("snapshot.txt");
  std::lock_guard<std::mutex> g(lock);
  for (const auto& kv : db)
    out << kv.first << "," << kv.second << std::endl;
  out.close();
}

void background_thread() {
  while (true) {
    snapshot();
    usleep(10000);
  }
}

void request_handler(int key, std::string& value) {
  std::lock_guard<std::mutex> g(lock);
  db[key] = value;
}

int main() {
  std::thread bg_thread(background_thread);
  for (int i = 0; i < kRounds; i++) {
    int key = std::rand() % dbSize;
    std::string value = generate_random_string();
    request_handler(key, value);
  }
}
```

Figure 1: example.cc: a simple multithreaded program where a foreground thread handles user requests and a background thread snapshots program state every 10 ms.

functions in a process. Furthermore, it allows developers to breakdown the latency faced by a specific request, even when processed by multiple threads, allowing them to zoom in and identify the code responsible for anomalous behavior. LDB provides this information after only seconds of decoding its own traces and without significantly harming the performance of the profiled program, enabling monitoring in production environments. In contrast to Intel PT, LDB is hardware agnostic. In principle, it can be ported to any architecture, and we demonstrate its use on Intel and AMD processors.

The efficiency and portability of LDB stem from a novel, software-only technique, called *stack sampling*. Unlike prior approaches, stack sampling doesn't record per-function timestamps (e.g., the output of the `RDTSC` instruction [18, 35]). Instead, a separate stack scanner thread polls the call stack of every application thread. During each polling cycle, the stack scanner thread performs a backtrace on each call stack to inspect changes to stack frames. Intuitively, a function's stack frame will be resident on the call stack until the function returns, so the higher its execution time, the longer its stack frame will remain resident. LDB exploits this to capture the runtime of all the functions that contribute meaningfully to latency (i.e., those that last longer than its sub-us polling interval).

While stack sampling is based on a simple premise, we had to overcome several challenges to make it work in practice. First, it is not possible for one core to access another core's stack pointer register, so we had to find an alternative way to locate the deepest stack frame. Second, there is not enough information available in stack frames to discern between repeated invocations of the same function so we had to find a way to detect them. Third, the stack scanning thread could race with application threads causing it to observe corrupted stack frames, so we had to develop a mechanism to detect and discard bad samples. Finally, backtracing can cause false sharing with variables on the call stack, negatively impacting application performance, so we needed a way to limit this overhead without sacrificing resolution. We discuss our solution to each of these problems in §3.2.

In addition to efficiency improvements, LDB provides better visibility into latency problems through *event tagging*, recording several types of events with timestamps and event-specific metadata. Examples include the start and end of requests; cross-thread interactions like locks; and the transfer of request ownership among threads. This allows LDB to track the timeline of each request and correlate this information across multiple threads. For example, LDB can identify a slow function running inside a critical section that is protected by a lock, and then tie it back to a request that is blocked in another thread waiting to acquire the same lock. LDB also uses event tagging to track context switching, allowing it to identify delays from the OS scheduler.

We demonstrate the value of LDB by profiling two latency-sensitive applications (Memcached and Lucene) and a best-effort application called Qperf, which is a benchmark for the QUIC transport protocol. We show that LDB can detect complex interactions between threads and identify which functions are responsible for impacting performance (both latency and throughput). Then, we provide an evaluation of the performance overhead of LDB when used to profile these three applications. In particular, we show that the overhead of LDB is less than the overhead of Coz and Xray; and comparable to Intel PT on recent Intel architectures. LDB maintains its low overhead across Intel and AMD architectures. On the other hand, the overhead of Intel PT is considerably higher when used on older Intel architectures.

LDB has some limitations. First, it can't yet capture hardware interrupts and some other types of traps into the kernel, which could contribute to tail latency. Second, LDB requires programs or libraries to be recompiled to support stack sampling, so it cannot trace latency inside unmodified binaries or libraries. Finally, if one chooses not to annotate requests in the source code (typically just a few lines), LDB cannot capture information about request latency, but it can still deliver statistics about the latency of each line of code, which is enough for debugging many tail latency problems.

LDB is available as open-source software at https://inhocho89.github.io/ldb/.

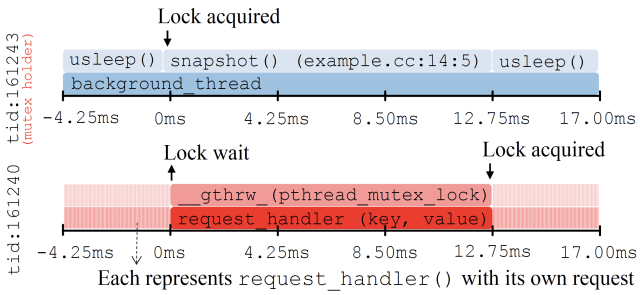


Figure 2: A timeline visualization of the time spent in each function during the longest request. The longest request starts at 0 ms and finishes at 12.75 ms. The thread on the top is the mutex holder, while the thread on the bottom is the request handler, which is blocked waiting to acquire the mutex.

2 Background and Motivation

2.1 Debugging a Tail Latency Problem

Consider the example shown in Figure 1, based on a pattern found in many real programs. A request processing thread and a background thread require synchronized access to the same data. The request processing thread responds by executing `request_handler()`, which inserts items into a `std::map`. Concurrently, the background thread takes a snapshot of the `std::map` every 10 ms. Access to the `std::map` is serialized through a `std::mutex`. As a result, the 99.99th percentile latency of the `request_handler()` function is 10 ms while its median is 244 ns (a 40,000× increase)!

This is a challenging issue to debug because of the rare interaction between the two threads. LDB, however, can easily identify the root cause. It captures everything that happened in the program and can generate a timeline visualization for each request that includes all the involved threads. By plotting the timeline of the longest request (Figure 2), it becomes clear that the snapshot thread (shown on top) delayed `request_handler()` (shown on bottom) by holding the mutex it was trying to acquire. This suggests that tail latency can be improved by optimizing `snapshot()` or reducing the size of its critical section.

However, existing profilers struggle to debug tail latency issues like these. For example, Figure 3 shows the output of perf, one of the widely used performance debugging tools. The majority of time is spent in `generate_random_string()` and other functions under `request_handler()`. `snapshot()` accounts for only 0.6% and was buried under other 13 functions. This result reveals three interesting problems of using perf for tail latency debugging. First, tail behavior is amortized, so it gets buried down under average behaviors. Second, perf is measuring where the CPU cycles go, not how long each function takes, so it is unable to show the time spent on blocking I/O or synchronization. Figure 2 suggests that `snapshot()` runs for over 10 ms and then sleeps for 10 ms, so it should account for at least about 50% time on average. However, much of the time spent on `snapshot()` is spent blocking on I/O, so perf reports only 0.62%. Lastly, perf cannot capture the interplay across threads caused by the mutex.

Function	CPU Time ▽
`generate_random_string`	63.75%
`request_handler`	7.43%
`std::_Rb_tree_increment`	2.82%
...(13 more functions)...	
`snapshot`	0.62%

Figure 3: Perf's output with the example application.

2.2 Intuition and Challenges

We observe that the metadata in x86 stack frames (e.g., the number of stack frames, return instruction pointers, saved based pointers, etc.) remains unchanged as long as a thread is executing a bottlenecked function. LDB takes an approach in which a separate dedicated busy-running thread, called the stack scanning thread, periodically scans these stack frames. It then measures the latency of the function call by examining whether the metadata in the stack frame metadata does not change. If a change is detected in the metadata, it signifies that a function has either returned or that a new function call has been invoked.

The realization of this *stack sampling* idea entails the following challenges:

1. Finding the deepest stack frame. Stack frames form a singly linked list data structure. Starting with the most recent (deepest) stack frame, one can traverse the entire call stack by following the saved base pointers. This traversal is necessary for LDB to ascertain whether the stack frame metadata has been modified or not. The location of the deepest stack frame can be retrieved from the RBP register. However, threads other than the application thread itself cannot access this register, making it challenging for the stack scanning thread to determine where to start traversing the stack frames.

2. Differentiating stack frames for different function calls. When the same function is repeatedly invoked from the same line of code (such as within a loop), the metadata may remain identical across samples of the stack frames. This can lead to an overestimation of the function call latency measurement performed by the stack scanning thread, as it may fail to detect that there were separate invocations to the same function. Therefore, to accurately measure the latency of individual function calls, it is necessary to find a way to differentiate between stack frames from distinct function calls, even if their stack frame metadata appears to be the same.

3. Cache thrashing and false sharing. Stack frames are frequently accessed by the application thread for local variables. If the stack scanning thread accesses stack frames too often, it may lead to performance degradation because of cache thrashing and false sharing. Repeated access to stack frames by the stack scanning thread can cause the data to be continually invalidated from the application thread's cache, harming the performance of the monitored application.

4. Data race for the stack frames. While the stack scanning thread is traversing the stack frame, the stack frame can be concurrently modified by the application thread when a func-

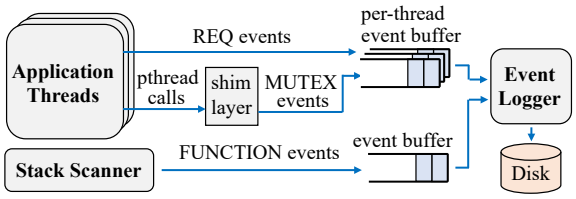


Figure 4: The flow of events that are recorded by LDB.

tion returns or a new function is called. This data race can result in the stack scanning thread collecting incorrect data (e.g., a half updated frame), which leads to inaccurate measurement of function latencies. For precise latency measurement, we need a way to detect and gracefully handle such data races, ensuring the integrity of the data collected by the stack scanning thread.

3 System Design

3.1 Overview

Our objective is to create a lightweight, portable latency profiling tool that can capture fine-grained information about the time spent in each function in a program. Thus, the per-function cost has to be minimal. We achieve this through two key ideas. First, we use a separate busy-polling core to shift away the instrumentation cost that would normally be incurred inside program threads, such as timestamping and recording events to memory. Second, we reduce the trace data generation rate by recording only functions with stack frames that are resident on the call stack for longer than the polling interval. Intuitively, very short functions do not contribute to latency, so it is okay to not spend resources in capturing them.

Building upon these ideas, we propose a new technique, called stack sampling, where a stack scanner thread repeatedly scans the call stacks of application threads. By observing the persistence of stack frames across multiple scans, the stack scanner thread can estimate each function's invocation latency. These invocation latencies can then be integrated with other event sources (e.g., acquiring a mutex, starting to process a request, spawning a thread, etc.) that are tagged with metadata and synchronized timestamps. This enables greater visibility, such as capturing locking interactions across threads.

Event recording. Figure 4 shows how different types of events are tagged and recorded by LDB. LDB has three main components that generate events. First, a *stack scanner*, which runs in a busy-polling thread, scans application threads' call stack and records invocation latencies each time a function returns. Second, a *shim layer* intercepts common threading operations (e.g., `pthread_mutex_lock()`) and records an event before forwarding the operation to its underlying implementation. Finally, application threads can generate events directly when they are annotated by the programmer, such as the start and end of a request. LDB records all events to per-thread shared-memory queues to improve scalability. An *event logger*, running in a separate thread, then gathers the events and stores them to disk for later analysis. Separately, the existing OS performance monitoring subsystem can be

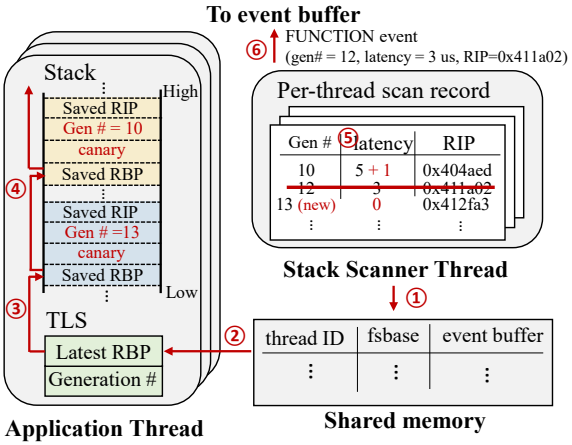


Figure 5: Stack Scanner Thread's Stack Sampling.

used to record scheduling events (not shown in the figure) like context switches and thread migrations [13]. Our design is extensible, and we plan to add additional event sources in the future, such as recording delays caused by interrupts.

3.2 Stack Sampling

Compiler instrumentation. Similar to many profiling tools [18, 21, 30, 33], LDB relies on compiler instrumentation that inserts small, low-overhead changes to the function calling conventions, which enables lightweight latency profiling even without application modification. First, the compiler emits a frame pointer for each stack frame. Normally, most compilers optimize away frame pointers, but they are needed by the stack scanner to backtrace the call stack. While functions can be identified using the return address saved on the stack frame, this value doesn't allow us to differentiate between multiple invocations of the same function. This difference is critical for latency debugging, as we care about the per-invocation latency on each function, not aggregate measures like CPU time. To resolve this problem, LDB uses generation numbers to differentiate different function invocations. If a generation number is different in an otherwise identical stack frame, LDB knows that it was a separate invocation. The compiler appends a generation number to each stack frame. The generation number is a monotonically increasing number, derived by incrementing a word stored in thread-local storage (TLS). Finally, the compiler records the frame pointer of the deepest stack frame (i.e., RBP register value), also placing it in TLS. The use of TLS avoids cache contention between application threads, allowing LDB to scale well across cores.

Sampling the stack. Figure 5 illustrates how the *stack scanner* samples the call stacks of application threads. The stack scanner runs as a separate thread in the same process as the application, allowing it to share its address space. The stack scanner maintains a table of the application threads that are currently running (①). For each application thread, it fetches the frame pointer of the deepest stack frame by reading from its TLS region, (②) and starts scanning the call stack (③).

It traverses all the stack frames up to the main function by following the stack frame pointers (④). While traversing the call stack, it collects each generation number, which is located at a fixed offset from the current frame pointer, along with the return address of the stack frame.

Latency calculations. When the stack scanner collects information from the call stack, it updates its *scan records*, which are a table of metadata for each stack frame (⑤). If the scanner detects a new stack frame, it creates a new scan record and records the current timestamp and generation number. If an existing scan record's stack frame is not found during the new scan, LDB concludes that the function has returned and generates the FUNCTION event with the generation number, latency, and return address (⑥). It then removes the scan record. We now discuss the various enhancements we made to this basic procedure to address validating stack frames, avoiding race conditions, and minimizing probing effects.

Validating stack frames. Another challenge is in identifying valid stack frames. For example, even a program compiled using LDB's compiler may still be linked against a library that does not contain LDB instrumentation. Thus, some stack frames may not have valid generation numbers. To detect this, LDB reserves an eight bytes space in the stack frame, called a canary. The canary contains a known magic value that the stack scanner looks for before parsing the generation number. If it is missing, latency is not reported for that function, but any parent functions that have the canary will still be reported. To avoid pursuing invalid stack frames, LDB stops traversing when the canary in the current stack frame is invalid, or the next base pointer is invalid. Thus, LDB is guaranteed to terminate its stack traversal. Further, it avoids segmentation faults by validating whether a certain memory address is between the start of the call stack (base pointer of the very first stack frame recorded at thread start) and the end of the call stack (latest RBP value in the TLS) before reading it.

Preventing data races. The application thread could race with stack scanning if it calls or returns from a function while the stack scanner thread is traversing the call stack. To avoid collecting invalid samples, the stack scanner uses TLS data as a sequential lock, a form of optimistic concurrency control [6]. Because the frame pointer changes with each function call or return, and the generation number changes with each function call, TLS data can be used to verify that the collected generation numbers are valid. The stack scanner compares TLS data before and after each stack scan, and if they don't match, it discards the collected sample and tries again.

Reducing probing effects. Another potential concern is that reading the stack could impact an application's performance. For example, if an application thread frequently modifies a variable stored on its stack frame, and it lands in the same cache line as a stack frame, this could result in false sharing between the stack scanner and the application.

To prevent this, LDB uses TLS data to detect function calls and returns, and initiates stack sampling only when they occur. This avoids all false sharing during function execution. The stack scanner supports this by polling the most recent generation number stored in each TLS region, and waiting for it to increase before sampling the thread's call stack. The stack scanner then proceeds with the scan, retrying if there was a race condition (which is rare). Once it gets a valid sample, it stops scanning until the next time the generation number increases. The generation number is placed in a dedicated cache line, allowing it to remain in the shared cache state. Therefore, no coherence traffic is generated while it is polled (until it is modified). LDB also supports pausing the stack scanner between probes (e.g., delaying for 1 µs). However, we found that the above technique allows LDB to poll in a tight loop with negligible probing effects and better resolution.

Security. As the stack scanner shares the address space with the application thread, the application may be at threat if the stack scanner is compromised. In this paper, we assume that the LDB compiler and library are not compromised and will undergo a thorough security review.

3.3 Tracing Cross-thread Request Handling

LDB analyzes cross-thread interaction with three types of events: request events, synchronization events, and scheduling events. Each event is timestamped and included in the trace. The time duration between two events (e.g., waiting for and acquiring a lock) along with other functions that happen between the two events, help construct a rich timeline. To minimize the extra latency required for event logging, each event is recorded to a per-thread circular event buffer. Then, the events in the event buffers are polled by the event logger which persists the events to disk.

Request events. For multi-threaded applications, it is hard to figure out which threads are responsible for high tail latency. To enable per-request tracing with a multi-thread environment, LDB provides an API for developers to annotate when a thread starts and finishes handling a request. Using request annotations, LDB constructs the timeline for a specific request showing the interaction between the threads handling the same request and revealing which threads contribute to a long request processing time. In particular, all function invocations in a thread that happen between a REQ_START and a REQ_END are counted towards the timeline of the processing of that request. Request tagging is optional, but the more the application developers tag events including when a request is temporarily placed in a queue (i.e., REQ_BLOCK), the more accurate timeline LDB can construct.

Synchronization events. Contention for shared resources can be a major source of latency. Visibility into synchronization events (i.e., mutex wait, mutex acquire, and mutex release) can play a key role in identifying performance bottlenecks in the presence of cross-thread interactions. Mutex events reveal not only which mutex is contended and how long it delays a request but also which function the mutex holder thread is executing while holding the mutex. For mutex events recording, LDB interposes synchronization library functions (e.g.,

pthread mutex) and generates the corresponding mutex events. We discovered recording every mutex event can introduce extra overhead, especially for mutex-intensive applications. To minimize this overhead, LDB decides whether it should record mutex events outside of the critical section after releasing the mutex. If either mutex wait time or lock time exceeds MUTEX_EVENT_THRESH (1 µs by default), it records the mutex event in the event buffer.

Scheduling events. The operating system scheduler can contribute to request latency through context switching between applications or threads. Revealing the delay caused by context switches can guide the developers to look at operating system configurations, not the application, to improve latency. Unlike other types of events, LDB collects the scheduling events from an external source. In particular, we collect scheduling information with `perf-sched` for Linux. LDB timestamps the events with the same clock source as the external tool to generate a unified timeline.

3.4 Analysis Script

LDB provides an analysis script to generate per-function statistics for collected latency samples. Further, it provides another analysis script that constructs a timeline of specific requests with function names and line numbers if request tags are given. It stitches together the events generated by application threads (i.e., request and synchronization events), the stack scanner, and the OS scheduler (i.e., scheduling events).

Constructing the timeline for a specific request, identified by its request ID, requires stitching together all events that occurred during the processing of that request. Such a timeline can have multiple components, requiring the script to make multiple passes over the data generated by the profiler. First, the script looks through the event log until it observes the REQ_START event with the request ID, indicating the arrival of that request. The script tracks all FUNCTION events generated by the thread processing that request thereafter. Upon reaching a MUTEX_WAIT event, if the thread experiences non-negligible wait time (e.g., longer than 1 µs between the MUTEX_WAIT and MUTEX_LOCK events), the script scans the event log backward to identify the mutex holder thread by searching for a MUTEX_LOCK event with the same mutex. Once the thread holding the mutex is identified, the script logs all FUNCTION events produced by that thread until it releases the mutex. Then, the script continues logging FUNCTION events by the original thread processing the request until it finds a REQ_BLOCK, REQ_END, or REQ_END_ALL event. The output of the script is a log of all events impacting the processing time of the request, each event identified by (event info, thread ID, start time, end time) tuple. Such information can be easily visualized as shown in Figure 6.

4 Implementation

We implemented a prototype of LDB for the x86 architecture and the Linux environment. Our implementation has three components: 1) an extension to LLVM [32], called LLVM-LDB, to instrument stack frames, 2) a stack scanner library to poll the generation numbers and calculate latency values, and 3) an API and bindings to `pthread` library to capture request and synchronization events automatically. Our implementation integrates with the Linux performance monitoring subsystem (`perf-sched`) to track context switches [13]. We also developed scripts to parse and analyze the data recorded by our tool. The core LDB tool is ≈900 lines of C code, the scripts are ≈1,200 lines of Python code, and the changes made to LLVM are ≈250 lines of C++ code. In this section, we describe more implementation-specific details for LDB.

4.1 LLVM-LDB

Reserving TLS. We reserve two 8 B TLS variables (generation number and the latest RBP register value) at a fixed offset from the TLS base address (FS base) with LLVM `ModulePass`. We make sure that LDB TLS variables are inserted into the TBSS section after all the in-application TLS variables are inserted so that LDB TLS variables are located at a fixed offset from the FS base.

Stack frame instrumentation. We modified the sequence of the function prologue and epilogue in the LLVM x86 backend. In the function prologue, we reserve the 16 B space in a stack frame by decrementing the RSP before the RBP is pushed into the stack frame. After the RBP is updated to the current RSP, we fill up the reserved stack space and update the TLS variables. First, we increment the generation number in TLS and copy it into the reserved space. Then, we set up the canary. Finally, now that the stack frame is ready to be scanned, we update the latest RBP in TLS to the current RBP register value so that the stack scanner can start scanning from a newly created stack frame.

In the function epilogue, we revert the instrumented operations in the prologue. First, before tearing down the stack frame in the function epilogue, the compiler first updates the latest RBP in TLS to avoid the current stack frame being scanned while it is being destroyed by copying the saved RBP in the current stack frame—which holds the RBP of the parent stack frame—into the TLS region. After the saved RBP is popped from the stack frame with a standard epilogue sequence, RSP is incremented by 16 to destroy the reserved space for LDB. In total, we add 9 instrumentation instructions (7 in the prologue and 2 in the epilogue) which add less than 1 ns to each function call.

Thread instrumentation. We instrument the main function to initialize LDB using LLVM `ModulePass`. The initialization allocates the shared memory and per-thread event buffer before registering the main thread into the shared memory with its thread ID, FS base address, and event buffer address. Then, LDB launches the stack scanner thread and the logger thread. To initialize a newly launched thread and clean up the state before it exits, we interpose on `pthread_create()`. Before a newly created thread executes its original start routine, LDB allocates the per-thread event buffer and registers

the thread into the shared memory. After the original thread start routine returns, it frees the event buffer and deregisters from the shared memory so that the exited thread is no longer scanned by the stack scanner.

4.2 The LDB API and Parameters

Request tagging API. LDB provides a way to annotate the threads with the following C APIs:

```
void ldb_req_start(uint64_t req_id, void *queue=NULL);
void ldb_req_block(uint64_t req_id, void *queue);
void ldb_req_end(uint64_t req_id);
void ldb_req_end_all();
```

When a thread starts to handle a request, the thread can be annotated with the request ID using `ldb_req_start()`. Optionally, if a request is dequeued from a software queue, the queue address can be specified. Multiple threads can be annotated with the same request ID with parallel processing, and a single thread can be annotated with multiple request IDs for batch processing. When a thread needs to enqueue a partially executed request into the queue the thread can hand off the responsibility of the request with `ldb_req_block()`. It indicates that the current thread is not responsible for the request anymore, but the current thread or another thread will resume processing the request later. If a thread finishes processing a request, it can clear the annotated request ID with `ldb_req_end()`. Alternatively, when a thread needs to clear all the annotated request IDs to the current thread, it can use `ldb_req_end_all()`. We decided to allow the programmer to specify the request ID, so that it can be correlated at the RPC level in coordination with other tools.

5 LDB Use Cases

To demonstrate the broad utility of LDB, we illustrates four use scenarios: visualizing a timeline of a specific request, debugging tail latency, debugging throughput, and studying the latency of specific functions. We evaluate these use scenarios with two latency-sensitive applications and one throughput-oriented application:

1. Memcached is a multithreaded, latency-sensitive, in-memory key-value store. We debug two different workloads: SET and GET. The SET workload exposes mutex and memory-intensive code paths. Each SET request can access a global lock, `slabs_lock`, multiple times to allocate and free the memory and a hash table bucket lock, `items_lock`, to update the hash table. Additionally, when a Memcached memory is saturated, it may need to acquire `lru_lock` to evict stale items. On the other hand, a GET request only needs to acquire the `items_lock` before fetching a value from the hash table.

   We allocated 10GB of memory for Memcached and used 100 million keys, evenly distributing them across requests. The value lengths are uniformly distributed between 4B and 1024B. We use the default hash power, which automatically grows based on the number of key-value pairs inserted into the hash table.

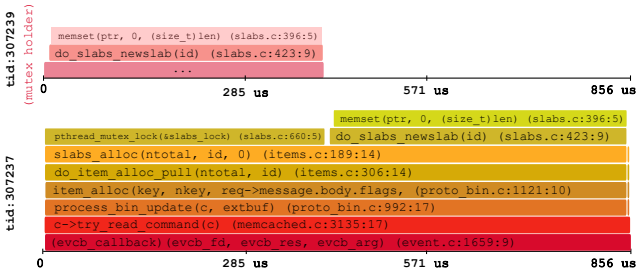


Figure 6: Timeline of the request of the longest request processing time in Memcached SET workload constructed by LDB.

2. Lucene is a multithreaded, latency-sensitive in-memory search engine library [3]. Lucene's processing time is much longer than Memcached, helping us demonstrate the value of LDB under a variety of conditions. We used a dataset of 403,619 COVID-related tweets. Each client generates single-term search queries based on the word distribution in the dataset. For each search request, Lucene first retrieves the list of document IDs from the `Segments` data structure that maps a word to a list of document IDs. Once the list of all relevant document IDs is retrieved, it fetches the pre-computed score (relevance between the document and the search query) for each document and returns the top 100 documents with high scores. All shared data structures are protected by a mutex, so multiple mutexes are acquired to serve each request.
3. Qperf [5] is a performance measurement tool for Quicly, Fastly's implementation of the QUIC protocol [31]. Unlike the other two applications, it measures the highest possible throughput between a server and a client. To achieve the highest throughput, we modified the original Qperf implementation to busy-poll for incoming packets. This application helps showcase the value of LDB when measuring the average per-packet latency for each function, by identifying functions that harm the average throughput of the application. We use Reno as the congestion control algorithm and enable generic segment offload (GSO).

To use LDB on the applications listed above, we compiled the applications with `LLVM-LDB`. In addition, we inserted tagging annotations at each code location where a thread starts to handle a new request (Memcached and Lucene) or a new packet (Qperf). These points were easy to identify, and only required 2–4 LOC changes.

5.1 Reconstructing the Timeline of the Request

When the application tags each request with a unique request ID, LDB can construct a timeline of any tagged request including interactions with other threads, which has been expensive with existing tools. Figure 6 shows an example timeline of a request in the Memcached SET workload. We picked the request with the longest request processing time we observed during the initial slab allocation phase because it provides a simple yet strong example of the value of LDB.

The detailed request timeline of LDB immediately shows what slowed down the processing of the request, including

Function	p50	p99	p9999 ▽
slabs_alloc() ↳ pthread_mutex_lock(&slabs_lock)	< 1	13.57	22.00
do_item_unlink_nolock() ↳ STATS_LOCK()	< 1	6.93	20.51
lru_pull_tail() ↳ pthread_mutex_lock(&lru_locks[])	2.14	17.53	19.62
do_item_link() ↳ STATS_LOCK()	< 1	14.64	19.50
item_unlink_q() ↳ pthread_mutex_lock(&lru_locks[])	2.03	10.56	18.88

Figure 7: Latency statistics of top 5 functions (and its caller) ranked by 99.99th percentile latency in Memcached SET workload. All numbers are in µs.

Function	p50	p99	p9999 ▽
resp_finish() ↳ THR_STATS_LOCK()	< 1	9.04	18.55
transmit() ↳ THR_STATS_LOCK()	< 1	9.66	18.26
do_item_get() ↳ assoc_find()	< 1	10.10	18.25
item_lock() ↳ mutex_lock(&item_locks[])	< 1	10.12	17.19
resp_start() ↳ memset()	1.00	10.28	17.00

Figure 8: Latency statistics of top 5 functions (and its caller) ranked by 99.99th percentile latency in Memcached GET workload. All numbers are in µs.

Function	p50	p99	p9999 ▽
IndexSearcher::search() ↳ Scorer::score()	72.18	2,005	6,232
Norm::bytes() ↳ IndexInput::readBytes()	657.7	1,690	4,899
boost::make_shared() ↳ new()	30.59	61.60	78.61
SegmentReader::docFreq() ↳ TermInfosReader::get()	< 1	57.05	61.59
TopDocsCollector::topDocs() ↳ populateResults()	< 1	20.27	25.09

Figure 9: Latency statistics of top 5 functions (and its caller) ranked by 99.99th percentile latency in Lucene workload. All numbers are in µs.

interactions with another request. When the request processing thread starts to handle the request, it waits for the slabs_lock mutex. LDB does not only tell the waiting time for the slabs_lock but also helps identify the thread holding the mutex and the function it's executing. In this example, the thread holding the mutex executes memset() while holding slabs_lock. After the thread processing the request acquires the lock, the dominant request processing time is spent executing memset() that took 645.7 µs. Such fine-grained tracing helps identify the main culprit which is performing memset() while holding the slabs_lock. This appears naturally in an LDB trace but is nearly impossible to identify using any existing tool without considerable manual work.

5.2 Tail Latency Debugging

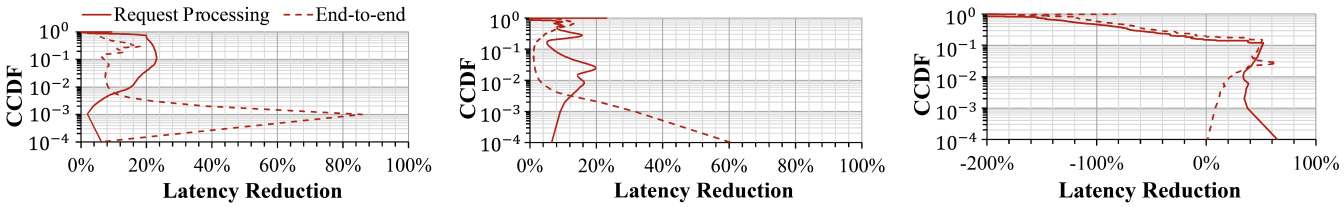
With Memcached and Lucene, we demonstrate that LDB can list functions that contribute to high tail latency, giving an insight as to how to improve their tail latencies. Note that LDB can generate latency statistics per line of code (Figure 7-9, 11) without request annotation.

For Memcached SET workload, Figure 7 lists the top five tail-contributing functions of Memcached ranked by 99.99th percentile latency. All five functions perform locking. Three out of the five functions contend for global locks related to memory management (slabs_lock) and statistics collection (stats_lock). To fix the tail latency from the slabs_lock, one could consider reducing contention by using a per-thread cache [19] and by zeroing memory without holding the lock. The stats_lock, on the other hand, could be fixed by either not using a lock, which would reduce accuracy, or by maintaining per-thread stats. Finally, the other two functions use the per-slab class lock, which is required for updating the LRU timestamp and evicting stale key-value pairs. To reduce the latency, one could fine-tune the chunk size growth factor (-f) based on the value length distribution.

For Memcached GET workload, Figure 8 shows that two of the top five functions are from per-worker thread locks (THR_STATS_LOCK). In Memcached, each network connection is assigned to one of the worker threads, but the requests can be processed by any worker thread. While the worker thread processes the request, it needs to acquire the lock of the worker thread that owns the network connection to update the statistics counters. When there is a small number of connections compared to the number of worker threads, or when the load is skewed to a subset of network connections, the per-worker thread statistics lock can be congested. The solutions mentioned above for stats_lock apply here too. Another two of the top five functions are for the hash table data structure (assoc_find() and per-bucket lock, item_locks). When a hash collision happens in the hash table, assoc_find() iterates over the bucket to find the item with the same key while holding the item_lock. One should consider initializing the Memcached with higher hashpower.

The last one is for memory operation to clear the allocated memory for a response. Considering that a response buffer will be overwritten with response data, one could consider removing the memset() operation, but care must be taken to avoid sending uninitialized data.

For Lucene workload, Figure 9 reports that the top two functions dominate the tail request processing time. Once Lucene receives a search query, it first fetches a list of document IDs by binary searching Segments after reading Segments in IndexInput::readBytes(). Once it has the list of document IDs, it looks up the score (the relevance between the query and the document) for each document and enqueues the document ID with its score into the max heap tree in Scorer::score(). In this case, a tail latency problem arises because the most popular term in the dataset appears in 88,558 documents. Thus, Scorer::score() needs to iterate 88,558 times to look up the score and enqueue it into the max heap tree, which can take 6.2 ms. To reduce this latency, one could consider utilizing an increased level of parallelism [27]. That is, if the length of fetched document ID is too long, the search



(a) Memcached SET (b) Memcached GET (c) Lucene

Figure 10: Latency reduction in request processing and end-to-end after applying patches that fix latency problems identified by LDB.

Function	Avg. Latency ▽
`send_pending()` ↳ `send_dgrams()`	29.63 µs
`allocate_ack_eliciting_frame()` ↳ `do_allocate_frame()`	2.79 µs
`encrypt_packet()` ↳ `ptls_aead__do_encrypt()`	2.41 µs

Figure 11: Top 3 functions (and its caller) ranked by the highest average latency in Qperf workload. The average processing time for 32 packets is 38.11 µs.

Function	CPU Time ▽
`__libc_recvfrom`	7.61%
`send_pending`	4.11%
`quicly_send`	2.56%
... (39 more functions) ...	
`do_allocate_frame`	0.28%
... (10 more functions) ...	
`ptls_aead__do_encrypt`	0.20%
... (152 more functions) ...	
`send_dgrams`	0.04%

Figure 12: Top 5 functions ranked by the highest CPU time in Qperf workload reported by Linux perf.

application could use multiple threads where each thread fetches the score of a subset of document IDs.

The other three functions are less significant. The memory allocation for reading the `Segments` with `new()` takes up to 79 µs, fetching the score of a document with `get()` takes up to 62 µs, and popping the top 100 documents from the max heap tree in `populateResults()` takes up to 25 µs.

Actionable Insights. To demonstrate that LDB provides actionable insights that developers can use to improve the latency behavior of real applications, we patch Memcached and Lucene using the output of LDB. We show both the request processing time, revealing the improvement to just the part of the application that LDB can profile, and the end-to-end processing time, which includes other sources of tail latency like the kernel network stack and the network fabric.

We patched Memcached to (1) preallocate the slabs to avoid memory allocation while serving the request, (2) fine-tune the object size of each slab to avoid contention in slab classes by specifying minimum object size and adjusting chunk size growth factor, and (3) convert global and per-connection stats into per-thread stats. Figure 10 (a) and (b) show the improvement in the request processing time and end-to-end latency after applying the patch. Because multiple responses can be batched before written to the wire, the improvement of end-to-end latencies is larger than the request processing times at some tail percentiles. The patch reduces the 99th percentile request processing time by 15% and 99th percentile end-to-end latency by 8% for SET workload; 99th percentile request processing time by 16% and 99th percentile end-to-end latency by 3% for GET workload.

For Lucene, we patched it to add inter-request parallelism, using four concurrent threads to serve each request. Figure 10 (c) shows the improvement in request processing time and end-to-end latency with the patch applied. The increased parallelism hurts performance for short requests due to the straggler effect, synchronization, and scheduling overhead, but achieves our overall goal of reducing request latency in the tail. This problem has been studied extensively in prior work, which suggests a more sophisticated approach would be to dynamically adjust parallelism based on the number of instantaneous requests in the system and the execution time [27]. The patch reduces 99th percentile request processing time by 34% and 99th percentile end-to-end latency by 13%.

5.3 Debugging Throughput of Qperf

We use LDB to debug the average performance of Qperf, demonstrating its value beyond tail-latency debugging. In particular, we profile the egress path on a Qperf server, focusing on the average per-packet latency, allowing us to determine an upper bound on achievable throughput. We find that each batch of 32 1500-byte packets takes 38.11 µs on average, putting a cap on throughput at around 9.8 Gbps. Note that actual throughput has to be lower because not all batches nor packets are maximum sized. Further, the server performs other functions beyond continuously transmitting data packets (e.g., process and transmit acknowledgments).

We use LDB to identify which functions take the most time on average for transmission handling, revealing throughput bottlenecks. Figure 11 shows the top three functions with the highest average latency. The biggest bottleneck, responsible for 77.7% of the processing time of a batch, is `send_dgrams()` which transmits packets through the kernel's `sendmsg()`, showing that the biggest performance bottleneck lies in the kernel. Other bottlenecks include memory allocation in (`do_allocate_frame()`) and encryption (`encrypt_packet()`). The remaining processing time for a batch of packets can be attributed to a collection of lower-latency functions. Thus, to improve the throughput, one should optimize the network stack (e.g., by using kernel-bypass), memory operations, and cryptographic operations.

To highlight the value of the profile produced by LDB,

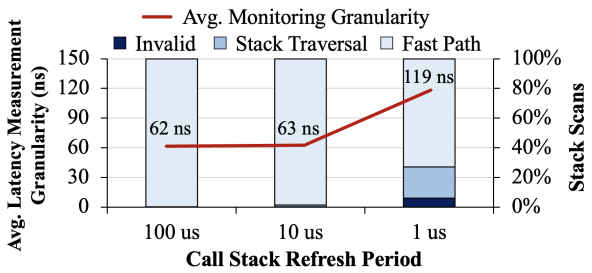


Figure 13: Average latency measurement granularity and the breakdown of stack scanning iterations with different call stack refresh periods in the synthetic application.

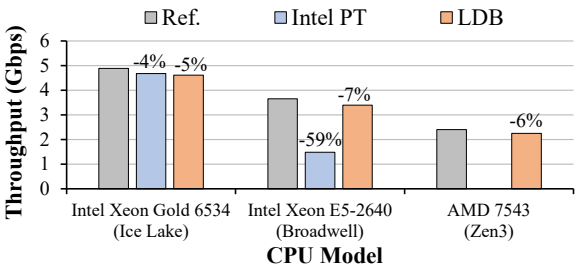


Figure 14: Average throughput of reference (without any profiling), Intel PT, and LDB with Qperf workload with different CPU architectures.

we compare its output to the profile produced by Linux's `perf`. Figure 12 reports the list of function names ranked by highest CPU time by `perf`. It shows that `perf` cannot pinpoint any of `send_dgrams()`, `ptls_adad__do_encrypt()`, or `do_allocate` functions that are responsible for 91% of the packet processing time, reporting that they consume 0.04%, 0.28%, and 0.2% of the CPU time, respectively. In particular, `perf`'s focus on average CPU time provides very coarse grain results, focusing on top-level functions like `quicly_send` which encapsulate all egress path functionality. Furthermore, it doesn't differentiate between functions on the critical path of egress traffic, and those happening periodically off the critical path, and it can't tie kernel delays to functions. Thus, we conclude that LDB can provide superior insights even when average performance is the focus of the debugging process.

6 Performance Evaluation

Our evaluations answer the following key questions:

1. What is LDB's latency measurement granularity?
2. Is LDB more portable than hardware-assisted latency debugging systems?
3. Can LDB limit the overhead it places on applications?
4. Can the trace data from LDB be decoded quickly?
5. How much does each component contribute to overhead?

Testbed. We use two machines with eighteen-core Intel Xeon Gold 6534 3.0GHz CPU (Ice Lake), 64GB RAM, and Mellanox ConnectX-6 200GbE NIC. For the portability experiment(§ 6.2), we compare its performance with Intel Broadwell machines (Intel Xeon E5 2640 v4 2.4GHz CPU, 64GB RAM, and Mellanox ConnectX-4 25GbE NIC) and AMD Zen3 Milan machines (AMD 7543 2.8GHz CPU, 256GB RAM, and Mellanox ConnectX-5 25GbE NIC). The median network RTT between two machines measured with ICMP packets is 30 µs. We use one machine as a server and the other as a client. Memcached and Lucene clients generate the requests following an open-loop Poisson arrival process, and Qperf clients generate a stream of requests for a data packet to measure the network bandwidth with TCP Reno as transport.

Applications. We use a synthetic application described in §6.1 for microbenchmark. To evaluate the performance of LDB, we reuse the workloads used in §5; Memcached SET/GET and Lucene are latency-sensitive workloads, and Qperf is a throughput-oriented workload.

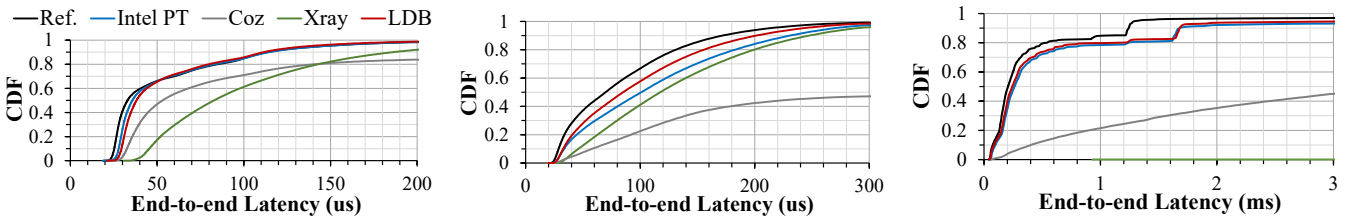
Baseline. We compare LDB to Intel Processor Trace (Intel PT) which backs state-of-the-art latency profilers [2, 4, 26], Coz that profiles the causal relationship between the function speedup and program speedup, and Xray that profiles the application's latency behavior with static timestamping. For Intel PT, we use `perf-intel-pt` provided by Linux to record and decode the Intel PT packets. For a fair comparison, we use a coarse-grained timing packet with `tsc` and decode only function call and return events with command line argument `--call-ret-trace`. We disable return compression (`noretcomp`) for more reliable decoding.

Evaluation Metrics. We report end-to-end latency (for latency-sensitive applications), average throughput (for best-effort application), raw trace size, and decoding time. End-to-end latencies and the average throughput are measured at the clients, and raw trace size and decoding time are measured at the server after the experiment finishes. Raw trace size measures the output size of each system, and decoding time measures the time required to parse the raw output to function-level latencies and to calculate the statistics of the function latencies. Because Intel PT takes too much time to decode, we measure the latency for decoding 1 ms long Intel PT trace. For LDB, we run the experiments for 4 seconds for Memcached and Qperf, and 1 minute for Lucene.

6.1 Microbenchmark

We delve into a detailed analysis of LDB's latency measurement granularity using a synthetic application which repeatedly destroys and reconstructs 20 stack frames through recursion, with a predefined refresh period. We experiment with varying the call stack refresh period from 100 μs to 1 μs and measure the average latency measurement granularity, defined as the average time elapsed between two successive valid stack scans. We further categorize the stack scanning iterations into three groups: invalid scans resulting from sequential lock fails with data races, stack traversals, and fast path iterations where no modification is detected in either the most recent RBP or the generation number in the TLS region.

Figure 13 presents the results. As the call stack refresh period decreases, the application thread interacts with the stack frames more frequently to destroy existing stack frames with function returns and to build new ones with new function calls. This increased frequency leads to cache thrashing and



(a) Memcached SET (b) Memcached GET (c) Lucene

Figure 15: End-to-end latency distribution of reference (without any profiling), Intel PT, and LDB with Memcached SET, GET, and Lucene workload at 20% load.

Workload		Trace Size / s (trace errors / s)	Decoding Time / s
Memcached SET	Intel PT	696.84 MB (2k trace errors)	48.4 min
	LDB	149.38 MB (-79%)	1.7 s
Memcached GET	Intel PT	796.72 MB (5k trace errors)	1.8 hr
	LDB	237.46 MB (-70%)	2.7 s
Lucene	Intel PT	1,066.29 MB (6k trace errors)	3.1 min
	LDB	2.12 MB (-99%)	0.7 s
Qperf	Intel PT	944.03 MB (559k trace errors)	3.7 hr
	LDB	25.4 MB (-97%)	0.8 s

Figure 16: Trace size and decoding time of Intel PT and LDB for four workloads. Trace size and decoding time are normalized by execution time.

data races between the application thread and the stack scanning thread more often, increasing the average granularity in latency measurement with more invalid stack scans. In addition, with more frequent modifications in the stack frames, the stack scanner requires more iterations of full stack frame traversals, which further increases the average latency measurement granularity. In an experiment with a function depth of 20 and 1 μs call stack refresh period, the latency can be measured with the granularity of 119 ns with 6% invalid stack scans, 21% of full scans, and 73% of fast path scans.

When dealing with multiple application threads, the average granularity increases in proportion to the number of application threads being profiled. For finer granularity in latency measurement, multiple stack scanner threads can be used, each profiling a subset of the application threads.

6.2 Portability of LDB

LDB is not designed for a specific platform. In principle, its design can be used on most architectures such as x86, ARM, and RISC-V. However, Intel-PT-based tools are tied to Intel's specific architectures and cannot be ported to other platforms. Our LDB prototype is implemented for x86 architectures and works well on any x86 architectures while Intel PT only works with some Intel processors (later than Broadwell).

To illustrate the portability of LDB, we run the Qperf workload with different x86 CPU models and compare it against the reference (i.e., no latency profiling) and Intel PT. Figure 14 shows the average throughput measured by Qperf on three different CPU architectures. It shows that Intel PT's performance highly depends on the CPU architectures. Even though Intel PT has only a 4% of throughput drop on the recent Ice Lake Intel CPU, it experiences 59% of the throughput drop on an Intel Broadwell CPU, and it cannot be used for AMD processors. On the other hand, LDB has a more consistent overhead of up to 7% thanks to its software-based approach.

6.3 Overheads of LDB

Application performance degradation. To get more confidence in LDB's low overhead, we measure the application performance impact on three latency-sensitive workloads (Memcached SET/GET and Lucene) and compare it to other profiling mechanisms. For the benefit of Intel PT, benchmarks ran on our testbed with Intel Xeon Gold 6534.

Figure 15 shows the end-to-end latency distribution measured at the client when the load is 20% of the system's capacity for Memcached and Lucene. We compare the performance of the applications when no profiling is done (i.e., Ref.) to when LDB, Intel PT, Coz, or XRay is used. For all workloads, Coz has the largest overhead at tail because it intentionally delays all the other threads than the thread being sampled, which makes it impractical to use over live traffic. The overhead of XRay is proportional to the number of function invocations as it statically instruments every function entry / exit to measure the latency. Due to its high overhead, the load exceeds the capacity, leading to extremely high latency with high queueing delay. Intel PT and LDB have comparable overhead across the workloads. LDB increases median(99th percentile) latency by 16%(1%), 22%(10%), and 18%(43%) for Memcached SET, GET, and Lucene workloads while Intel PT increases 9%(2%), 45%(23%), and 27%(64%) in the same setting.

Trace size and decoding time. Figure 16 reports the trace size and decoding time of Intel PT and LDB for the three applications. Intel PT requires high memory / PCIe bandwidth and disk space, especially for applications with more branches and jump instructions. For example, in Qperf, Intel PT outputs 944 MB/s of trace data. In addition, because of the limited memory bandwidth, it drops the event records and results in up to 559 thousand trace errors per second, which makes its visibility limited. To make matters worse, Intel PT takes up to 3.7 hours to decode 1 second of trace data, converting raw branch and jump information into function-level latencies. In contrast, the size of LDB trace is up to 99% smaller than Intel PT, typically requiring less than 250 MB/s, and it only takes a few seconds to decode 1 second of trace data.

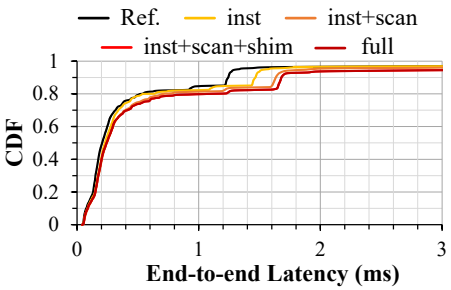


Figure 17: Performance Breakdown of LDB with Lucene workload at 20% load.

6.4 Breakdown of LDB's overhead

We analyzed how much each component contributes to the overhead for Lucene workload with the highest latency distortion under LDB whose median(99th percentile) latency is increased by 35%(69%). We gradually activated four components of LDB: application instrumentation (inst), the stack scanning/logging thread (scan), the shim layer (shim), and Linux scheduling event recording.

Figure 17 shows that instrumentation, the stack scanning, the shim layer, and Linux scheduling event recording are responsible for 11% (23%), 7% (17%), 3% (14%), and 1% (0%) of the median (99th percentile) latency increase, respectively.

7 Related Work

Sampling-based tools. Existing statistical sampling-based tools, such as perf [13] and Coz [21], are unable to analyze tail latency because they sample too slowly. Coz is designed to identify performance bottlenecks by estimating an application's virtual speedups through dynamic experiments that measure thread interactions. By artificially delaying all but one thread, Coz simulates the hypothetical performance improvement of specific functions. Although Coz addresses some of the limitations of perf, it is unsuitable for tail latency debugging because its reliance on statistical sampling causes it to miss tail behaviors. Moreover, when it identifies problematic code lines, it doesn't reveal the underlying reasons for the bottleneck, like LDB can through its request timelines.

Trace-based latency profiling tools. Trace-based tools, like XRay [18] and Intel PT, can capture the precise execution times of every function call but come with their own challenges. XRay, through compile-time instrumentation, records the execution times of individual functions. However, this results in high instrumentation overhead, which limits its use in microsecond-scale RPC applications. Intel PT, a hardware-assisted approach, captures control flow information at every branch. While systems like NSight [26] and MagicTrace [4] leverage Intel PT to debug latency problems, Intel PT's massive data rates, up to 1GB/s, necessitate substantial storage and lengthy decoding times to convert control flow information into latency information, which makes it impractical for real-time debugging. Additionally, Intel PT's overhead varies across generations of Intel CPUs, as shown in §6.2.

Continous profiling. SHIM rapidly collects hardware performance counters and software tags through busy polling [36]. It shares LDB's basic strategy of sampling with a busy-spinning core, but it lacks the ability to measure invocation latency without additional mechanisms, such as our proposed stack sampling techniques. Moreover, one busy-spinning SHIM profiler thread is needed for each hyperthread pair, resulting in high overheads due to competition over shared functional units. LDB, by contrast, can avoid this overhead through a different design that enables one monitoring thread to profile multiple threads running across multiple cores.

Limiting tracing to specific functions. As seen with the case of Xray, timestamp instrumentation at each function's entry and exit entails significant overhead. Thus, some tools limit tracing to a few specific functions at a time. There are various techniques and tools to enable dynamically enabling/disabling timestamp instrumentation: notably, dynamic instruction patching [18, 20, 25], dynamic instrumentation via eBPF [1, 10, 24], and instrumentation via JIT compiler [33]. However, because the scope of functions being profiled is limited, they require multiple iterations with the developer's intervention for latency debugging, and they sacrifice the ability to capture complete timelines. There are efforts to streamline these iterations [7, 29, 30]. AMD offers a suite of profiling tools (e.g., Omnitrace [16] and uProf [15]). Both solutions rely on sampling. Further, Omnitrace offers Coz-like functionality as well as specific function instrumentation. Omnitrace's instrumentation adds 1024 instructions per function compared to LDB's 9 instructions per function.

Distributed latency tracing. Envoy [11], Zipkin [14], Jaeger [12], AWS X-Ray [9], and Apache SkyWalking [8] provide tools to trace a request in a distributed computing environment at an RPC or microservice granularity. These tools may find a service causing high end-to-end latency, but they don't have visibility inside the service. Distributed tracing systems and LDB are complimentary. Problematic services can be found with distributed tracing, while problematic functions in a specific service can be found with LDB.

Mutex bindings. Dynamic data race detectors, like Eraser [34], often use similar mechanisms to interpose on locking functions, but their goal is to instead verify if the application follows a consistent locking protocol.

8 Conclusion

In this paper, we presented LDB, an efficient latency profiling tool with low overhead, high visibility, fast decoding, and portability. It utilizes a key technique, stack sampling, where each function's invocation latency is measured by sampling a unique generation number assigned in the stack frame. With optional request tagging by the developer, LDB can construct the detailed timeline of a request, including cross-thread interactions caused by synchronization, the time spent in functions, and the contribution of the OS scheduler. Our evaluation showed that LDB could profile the latency behavior of three applications and reveal their main performance bottlenecks effectively on multiple platforms with low overhead.

Acknowledgments

We thank our shepherd Yongle Zhang, as well as to the anonymous reviewers for their invaluable feedback. We also thank Kostis Kaffes, John Ousterhout, and David Culler for their constructive feedback; and Cloudlab [23] for providing us with machines of different architectures for portability experiments. This work was funded in part by NSF grants CNS-2104398, CNS-2212098, and CNS-2212099; DARPA FastNICs (HR0011-20-C-0089); VMware; and a Google Research Award.

References

[1] eBPF. https://ebpf.io/.

[2] Fix performance bottlenecks with intel vtune profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[3] Lucene++: c++ port of lucene library. https://github.com/luceneplusplus/LucenePlusPlus.

[4] magic-trace: Diagnosing tricky performance issues easily with intel processor trace. https://blog.janestreet.com/magic-trace/.

[5] qperf: performance measurement tool for QUIC. https://github.com/rbruenig/qperf/.

[6] Sequence counters and sequential locks. https://docs.kernel.org/locking/seqlock.html.

[7] wachy: A new approach to performance debugging. https://rubrikinc.github.io/wachy/.

[8] Apache SkyWalking, 2022. https://skywalking.apache.org/.

[9] AWS X-Ray, 2022. https://aws.amazon.com/xray/.

[10] bpftrace: High-level tracing language for Linux systems, 2022. https://bpftrace.org/.

[11] Envoy Proxy, 2022. https://www.envoyproxy.io/.

[12] Jaeger: open source, end-to-end distributed tracing, 2022. https://www.jaegertracing.io/.

[13] perf: Linux profiling with performance counters, 2022. https://perf.wiki.kernel.org/index.php/Main_Page.

[14] Zipkin, 2022. https://zipkin.io/.

[15] AMD uProf, 2023. https://www.amd.com/en/developer/uprof.html.

[16] Omnitrace: Application profiling, tracing, and analysis, 2023. https://github.com/AMDResearch/omnitrace/.

[17] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[18] D. M. Berris, A. Veitch, N. Heintze, E. Anderson, and N. Wang. Xray: A function call tracing system. Technical report, 2016.

[19] J. Bonwick and J. Adams. Magazines and vmem: Extending the slab allocator to many cpus and arbitrary resources. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference, June 25-30, 2001, Boston, Massachusetts, USA*, pages 15–33. USENIX, 2001.

[20] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *ATC*, 2004.

[21] C. Curtsinger and E. D. Berger. Coz: Finding code that counts with causal profiling. In *SOSP*, 2015.

[22] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 2013.

[23] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, et al. The design and operation of cloudlab. In *ATC*, 2019.

[24] B. Gregg. Linux bpf superpowers. http://www.brendangregg.com/blog/2016-03-05/linux-bpf-superpowers.html, 2016.

[25] B. Gregg and J. Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*. Prentice Hall Professional, 2011.

[26] R. Haecki, R. N. Mysore, L. Suresh, G. Zellweger, B. Gan, T. Merrifield, S. Banerjee, and T. Roscoe. How to diagnose nanosecond network latencies in rich end-host stacks. In *NSDI*, 2022.

[27] M. E. Haque, Y. H. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *ASPLOS*, 2015.

[28] K. M. Hazelwood, S. Bird, D. M. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, pages 620–629. IEEE Computer Society, 2018.

[29] J. Huang, B. Mozafari, and T. F. Wenisch. Statistical analysis of latency through semantic profiling. In *EuroSys*, 2017.

[30] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: performance bug detection in the wild. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 155–170. ACM, 2011.

[31] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *SIGCOMM*, 2017.

[32] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[33] Y. Luo, K. Rodrigues, C. Li, F. Zhang, L. Jiang, B. Xia, D. Lion, and D. Yuan. Hubble: Performance debugging with in-production, just-in-time method tracing on android. In *OSDI*, 2022.

[34] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *TOCS*, 1997.

[35] S. Yang, S. J. Park, and J. Ousterhout. NanoLog: A nanosecond scale logging system. In *ATC*, 2018.

[36] X. Yang, S. M. Blackburn, and K. S. McKinley. Computer performance microscopy with shim. *ISCA*, 2015.

UFO: The Ultimate QoS-Aware CPU Core Management for Virtualized and Oversubscribed Public Clouds

Yajuan Peng*[1,3], Shuang Chen*[2], Yi Zhao[2], and Zhibin Yu[2,3]

[1]*Southern University of Science and Technology, China*
[2]*Shuhai Lab, Huawei Cloud*
[3]*Shenzhen Institutes of Advanced Technology, Chinese Academy of Science*

Abstract

Public clouds typically adopt (1) multi-tenancy to increase server utilization; (2) virtualization to provide isolation between different tenants; (3) oversubscription of resources to further increase resource efficiency. However, prior work all focuses on optimizing one or two elements, and fails to considerately bring QoS-aware multi-tenancy, virtualization and resource oversubscription together.

We find three challenges when the three elements coexist. First, the double scheduling symptoms are 10× worse with latency-critical (LC) workloads which are comprised of numerous sub-millisecond tasks and are significantly different from conventional batch applications. Second, inner-VM resource contention also exists between threads of the same VM when running LC applications, calling for inner-VM core isolation. Third, no application-level performance metrics can be obtained by the host to guide resource management in realistic public clouds.

To address these challenges, we propose a QoS-aware core manager dubbed UFO to specifically support co-location of multiple LC workloads in virtualized and oversubscribed public cloud environments. UFO solves the three above-mentioned challenges, by (1) coordinating the guest and host CPU cores (vCPU-pCPU coordination), and (2) doing fine-grained inner-VM resource isolation, to push core management in realistic public clouds to the extreme. Compared with the state-of-the-art core manager, it saves up to 50% (average of 22%) of physical cores under the same co-location scenario.

1 Introduction

Motivation. Most cloud data centers operate at very low resource utilization[25, 26, 29, 44]. Other than users over-provisioning their resources [19, 26, 46], resource is especially underutilized in public clouds due to two additional reasons. First, most cloud providers, if not all, provide monthly or yearly subscriptions, which are much cheaper than on-demand pricing[1, 3, 15]. Most users therefore almost never release their resources even when their virtual machines (VMs) are completely idle. Second, there is no explicit label such as online/offline, or high-/low-priority workloads for users' VMs. This forms the black-box nature of public clouds: co-located VMs on the same host machine are equally important, similar to the co-location of multiple LC workloads studied in recent years[22, 48, 51]. This leads to generally lower resource utilization of public clouds than private clouds.

Resource isolation leverages software and hardware mechanisms to partition resources between co-located workloads, and to increase the degree of workload co-location without hurting application's quality of service (QoS)[23, 37, 38]. Resource isolation is especially important for interactive LC workloads such as web search and key-value stores[22, 26, 51]. However, when attempting to apply resource isolation in real public clouds, we find that the state-of-the-art QoS-aware resource managers are (1) *sub-optimal* due to lack of consideration of the virtualization layer and resource over-provisioning, and (2) *inapplicable* to public clouds due to lack of labels, inputs, or feedback for dynamic resource management.

Challenge 1. In virtualized environments, there are two layers of resource management: one in the host OS (e.g., scheduling physical CPU cores, pCPUs, for VMs), and another in the guest OS (e.g., scheduling virtual CPU cores, vCPUs, for users' applications). This is a common problem known as double scheduling, and prior work[36, 41, 55, 58, 62] has proposed co-scheduling or guest-host coordination to improve locks, interrupts, synchronization, preemption, load balancing, etc.

However, we find previous studies are ineffective for LC applications that suffer uniquely from double scheduling. Specifically, if the guest OS is oblivious of the allocated resources on the host, the host OS (rather than guest OS) would experience extensive context switches and high scheduling delay, leading to substantial QoS violations. This is intrinsically related to the sub-millisecond nature of tasks in LC applications. In addition, resource contention also happens between thread groups that belong to the same VM. Such inner-VM contention is especially intense under LC applications.

Challenge 2. Prior work relies on application-level input (such as real-time 99th percentile latency) to guide QoS-aware resource management [22, 23, 43, 44]. This makes two assumptions that do not hold for realistic public clouds. The first assumption is that an application has to monitor its own latency. In reality, many applications do not trace their own

*Equal contribution.

latency, and it is impossible to ask users to provide tracing points to probe their user-level QoS satisfaction degree. The second assumption is that the monitored value can be leveraged by the host OS. This requires interfaces for users to communicate with the cloud provider, to pass application-level information down to components outside the user's VM.

Opportunities. Major cloud providers all provide their own guest OS images [2, 4]. With Cent OS Linux reaching its end of life[5], an increasing number of users are migrating to in-house guest OS images. This leads to a great opportunity for cloud providers to explore guest-host coordination in guest/host kernels, to tackle the two challenges above. By transmitting host's core isolation decision to the guest OS and adjusting core management in the guest accordingly, we observe up to 4.3× throughput improvement under QoS requirements for LC workloads. By leveraging guest's scheduling frequency as a VM's performance indicator, the host OS is able to make QoS-aware core allocation decisions without disturbing users' applications inside the VMs. Furthermore, we observe that core isolation between different thread groups of the same VM leads to an additional 20% of improvement under the same amount of hardware.

Our work. In this paper, we propose UFO, a practical QoS-aware resource manager that targets virtualized public cloud environments, and push core isolation to the next level. Leveraging guest-host coordination, it solves the above-mentioned two challenges that prior work fails or misses to address, and aims to accommodate as many VMs as possible without QoS violations. We make the following contributions:

1. We find that the state-of-the-art QoS-aware core managers neglected the virtualization layer, and made unrealistic assumptions of the interactions between users and the cloud platform. Therefore, they are sub-optimal and inapplicable for virtualized public clouds.
2. We present a comprehensive characterization study to showcase how guest-host coordination and inner-VM isolation can greatly improve resource efficiency in virtualized environments in a black-box manner.
3. We devise UFO, a practical QoS-aware core manager for virtualized and oversubscribed public clouds. UFO does not require user-level QoS input, and instead leverages scheduling frequency from the guest OS as QoS indicators; this makes UFO applicable for public clouds. UFO dynamically adjusts core allocations for each VM on the host and the guest OS concurrently; this makes UFO considerate for virtualization overhead, especially when cores are oversubscribed. Furthermore, UFO dynamically manages an emulator pool to avoid core contention between emulator and vCPU threads.
4. We evaluate UFO under constant and dynamic loads, and show that UFO outperforms prior work by up to 50% (average of 22%) in core saving under the same co-location scenario, and up to 60% (average of 27%) higher input load under the same amount of cores, with negligible overhead.

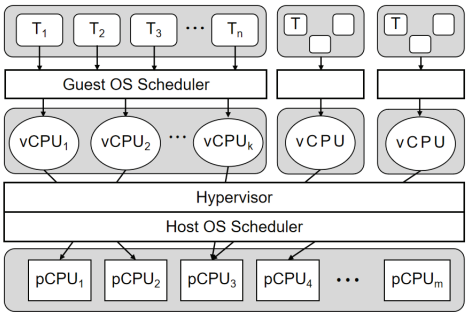


Figure 1: Double scheduling in a typical cloud server. The host OS schedules VMs' vCPU threads to physical cores (pCPUs), while the guest OS of each VM schedules users' application threads to vCPUs.

2 Background and Related Work

2.1 QoS-Aware Resource Management

There are mainly two types of workloads in the public cloud. Latency-critical (LC) workloads have strict quality-of-service (QoS) requirements, usually defined in tail latency. Best-effort (BE) applications are typically throughput-oriented, and are at lower priority than LC applications. QoS-aware resource management refers to techniques that can co-locate various workloads on the same server node [26, 45] while meeting LC workloads' QoS requirement; BE applications can be slowed down or even suspended when needed. Recent work is able to co-locate multiple LC applications on the same node while meeting each LC application's QoS target [22, 43, 48, 51].

However, previous studies have two drawbacks. First, they all rely on application-level performance metrics to guide QoS-aware resource management, and claim this could be achieved by monitoring on the client side, or relying on the application itself to report latency[22, 43]. In public clouds, there is no explicit QoS target defined for LC workloads, and it is impossible to ask users to provide tracing points to probe their user-level QoS satisfaction degree. Instead, cloud providers should find server-side indicators that generally correlate well with QoS, and the cloud providers can legally obtain the indicators without user intervention. A series of work like Arachne [53], Shinjuku [34] and Caladan [28] aim to provide sub-millisecond QoS satisfaction. However, they all rely on deep coordination with users' applications and make *intrusive modifications* to applications or their runtime libraries. Second, the virtualization overhead is significantly overlooked under (most if not) all QoS-aware resource managers.

2.2 Virtualization and Double Scheduling

In public cloud, virtualization is prevalent to ensure user privacy and security. Virtual machines and hosts interact through the hypervisor, and are independent and unaware of each other. Figure 1 shows the current software stack in a typical public cloud. The host OS scheduler schedules the logical CPUs

(vCPUs) of many virtual machines on the physical CPUs (pCPUs) of a server. The users' applications are scheduled by a task-level scheduler (Guest OS scheduler) of VMs, and most operations within the virtual machines are opaque to the hypervisor. Due to the use of multi-core processors and multi-threaded applications, this mutually unaware double-scheduling [57] introduces a semantic gap between hosts and VMs, which leads to problems including lock-holder preemption (LHP) [35, 59], lock-waiter preemption (LWP) [59], blocked-waiter wakeup (BWW) [27], etc. There are two types of solutions to resolve the double scheduling symptom.

2.2.1 Pure-host Solutions

These solutions modify the hypervisor and the host OS to reduce the overhead of virtualization and double scheduling. Adjusting scheduling priority on the host OS is one of the most common approaches, to alleviate LHP and LWP [27, 33], achieve fairness between VMs or load balancing [40, 63], and reduce CPU fragmentation [52, 54]. Advanced hardware supports, such as Intel's Pause Loop Exit (PLE) [41, 56] and AMD's Pause Filter (PF) [47], are also leveraged by the hypervisor and the host OS to detect excessive spinning in the guest OS, to address various preemption problems.

2.2.2 Guest-Host Coordination

These solutions bridge the semantic gap by coordinating the guest and the host OSes, and involves modifications to both the guest and the host OSes.

vCPU ballooning [24, 57] is one of the pioneer works in this line. It leverages CPU hot-(un)plugging to make #vCPUs match #pCPUs, so it completely avoids the double scheduling symptoms. The core allocation is static based on VM priority, making vCPU ballooning fail to adapt to varying load, and fail to handle co-location of LC applications. UFO inherits the same philosophy of vCPU ballooning, to make #vCPUs match #pCPUs. However, UFO targets LC applications, and features in finding the right metric, designing a proper algorithm and an effective controller to support dynamic core adjustment.

eCS [18, 36] annotates critical sections in the guest OS so that the host OS can adjust scheduling priority accordingly. The host states are further transmitted to the guest OS alleviating LWP and BWW problems. PLE-KVM [32] annotates vCPU status in the guest OS, and adjusts vCPU scheduling in the host OS, to mitigate excessive pause-loop-existing (PLE) events and excessive spinning. Similar to vCPU ballooning, eCS and PLE-KVM do not study LC workloads. But since they are open-sourced, we are able to conduct a quantitative comparison with eCS and PLE-KVM in Section 3.6.

3 Characterization

In this section, we characterize three typical LC cloud applications and compare them with BE applications. We illustrate the limitations of current CPU core isolation mechanisms, and show the potential of core management in virtualized and oversubscribed public clouds.

3.1 Methodology

We launch four 8-vCPU VMs on a 16-core host, i.e., 32 vCPUs on 16 pCPUs. This simulates an oversubscription ratio of two.[1] We enable hyper-threading because real public clouds always enable it, so a pCPU is essentially a hyperthread of a physical core. We avoid different VMs sharing the same physical core. Therefore, an 8-vCPU VM can be allocated on 2, 4, 6 or 8 pCPUs. We mainly experiment with three LC applications, Memcached, NGINX, and MySQL. Section 5 includes more details on the evaluated applications, the test bed, and the testing strategy. For simplicity, we run the same workloads in the four VMs such that all the VMs perform homogeneously. This makes core allocation decisions trivial: all the VMs should be allocated with the same number of cores. This allows us to focus on observing the performance differences under various core allocation mechanisms. For each LC application, we experiment with increasing amount of load measured in request-per-second (RPS). Each RPS is tested three times, each lasting for 60s. The geometric mean of the 99th percentile of latency of the four VMs is reported after each test. We then report the average result of the three tests for any given RPS.

3.2 Guest OS Should Coordinate with Host OS

Conventional approaches [22, 44, 51] manage core allocation completely on the host OS. The guest OS is unaware of the host OS; it does not know how many physical cores are actually allocated to the VM. We show that this is insufficient; guest being aware of the host allocations can achieve much higher resource efficiency.

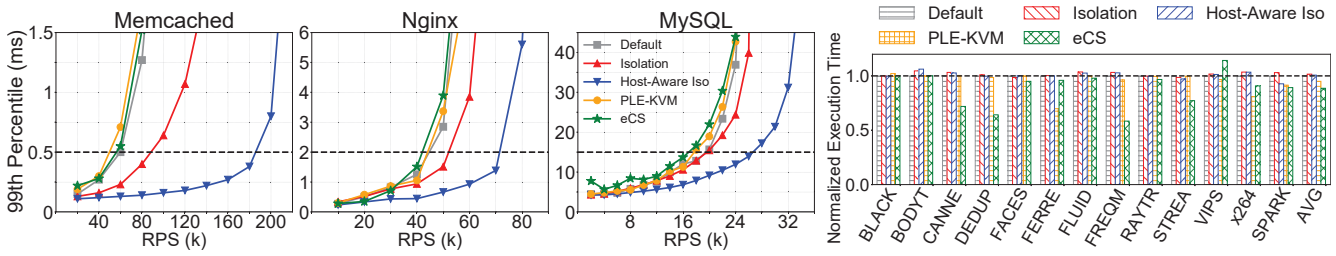
We compare three core allocation mechanisms:

- **Default:** We rely on the scheduling policy of OS to schedule VMs. All the VMs share the same 16 pCPUs, and can be freely scheduled on these pCPUs.
- **Isolation:** We isolate the four VMs, each assigned four pCPUs on the host.
- **Host-Aware Isolation:** On top of *Isolation*, the guest OS is aware that the VM is allocated with only four pCPUs , and schedules jobs to only four vCPUs. We implement this by statically hot-unplugging four vCPUs in the guest OS.

Figure 2 shows how VMs perform when running different workloads under the three mechanisms. We find that under QoS targets (horizontal dotted lines in Figure 2), isolation achieves up to 33% (average of 18%) higher load than default, and host-aware isolation furthers increases the maximum load under QoS by up to 25%-125% than isolation. This is because under host-aware isolation:

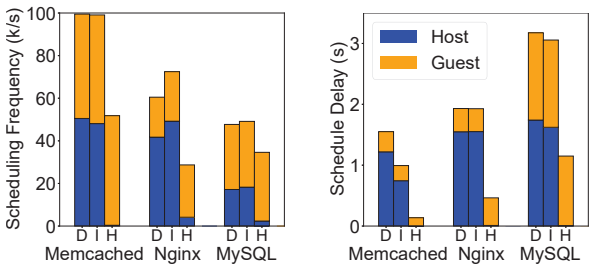
- **Host-side context switches are reduced by 92%.** Figure 3a shows the number of context switches per second (i.e., scheduling frequency) happening in the guest and the host OS. Memcached, NGINX and MySQL run at RPS of

[1] Higher oversubscription ratio usually signals higher resource efficiency, e.g., VMware suggests an oversubscription ratio of at least three [10].



(a) LC **(b) BE**

Figure 2: Performance under five core allocation mechanisms. For LC applications, we show the 99th percentile tail latency with increasing input load (RPS). Horizontal dotted lines represent applications' QoS targets. For BE applications, we show the execution time of each benchmark normalized to that under the *Default* manager. Lower is better.



(a) Scheduling Frequency **(b) Scheduling Delay**

Figure 3: Scheduling frequency and scheduling delay under default (D), isolation (I), and host-aware isolation (H), decomposed into host-side (blue bars) and guest-side (orange bars).

60k, 50k and 20k, respectively, which are the maximum RPS under QoS with the default core manager (see the cross points of gray curves and the horizontal dotted lines in Figure 2a). Because host-aware isolation ensures #vCPUs=#pCPUs, the host OS rarely needs to schedule vCPU threads off a physical core. Note that users' applications are unchanged and the number of application threads does not decrease with fewer vCPUs, so context switches in the guest OS are not reduced. Since a context switch of a vCPU thread on the host involves additional switches of the VMCS structure [55, 60], which is more expensive than a normal context switch of an application thread, we can obtain huge benefits from the reduction of only host-side context switches in virtualized environments.

- **Host-side scheduling delay is reduced by 99%**. Scheduling delay of a process is defined by the time between the process wake-up and its actual running.[2] Figure 3b shows scheduling delay on the host and the guest. As a consequence of reduced context switches, host-side scheduling delay decreases to almost 0 under host-aware isolation. Reduced scheduling delay is critical for LC applications because scheduling delay significantly affects requests' latency. When a request comes in, the application threads inside the guest OS are woken up, leading to schedule events on the guest OS, which may further lead to schedule events on the host. Scheduling delay inside and outside the VM both affect the degree of QoS satisfaction.

[2] We obtain scheduling delay of the guest/host OS by executing `perf sched record` and then `perf sched timehist`, and reporting the sum of scheduling delay of all the processes.

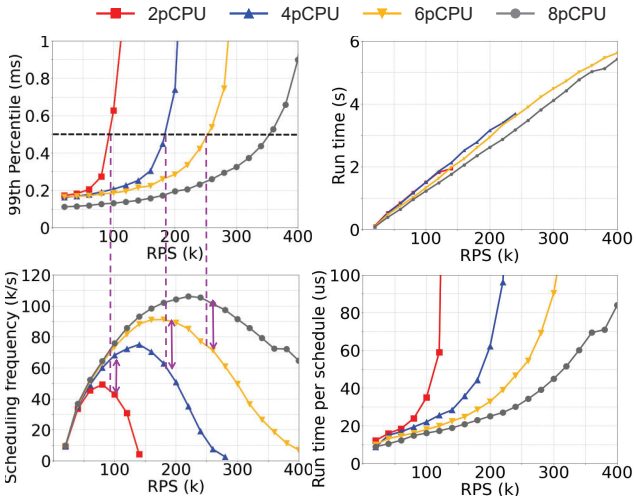


Figure 4: Memcached with increasing input load under various #pCPUs. The top left figure shows Memcached's tail latency, and the horizontal line at $y = 0.5ms$ represents the QoS target. The other three figures show guest-side process runtime, scheduling frequency, and runtime per schedule (division of the previous two metrics).

Appendix A.1 and A.2 include additional analysis on VM exits and caches. In summary, *for LC applications, the guest should be aware of the actual physical core allocations on the host, and dynamically adjust the number of runnable vCPUs.*

3.3 Host OS Should Learn from Guest OS

In public clouds, VMs are black-box to the cloud provider, and application-level performance metrics are hard to obtain. It is therefore difficult to use application-level metrics as indicators to guide resource management under a QoS target in public cloud. This has been an open question in industry for a long time. We successfully solve the open question by identifying a set of great indicators from the guest OS.

- **Guest-side scheduling frequency is a concave function with input load.** As the bottom left subfigure in Figure 4 shows, under a certain #pCPUs, guest-side scheduling fre-

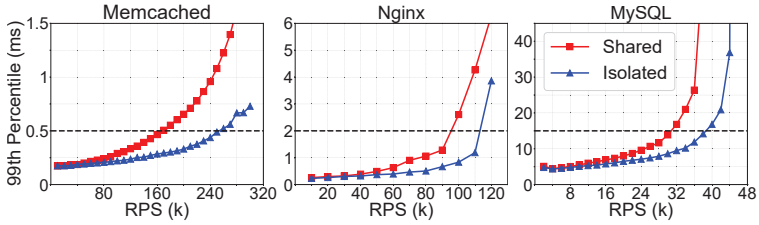


Figure 5: vCPU and emulator threads share 8 pCPUs (Shared), or are isolated with 6 and 2 cores (Isolated, #vCPU is hot-unplugged to 6).

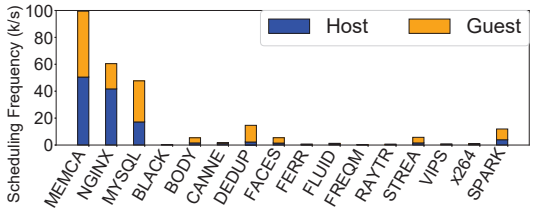
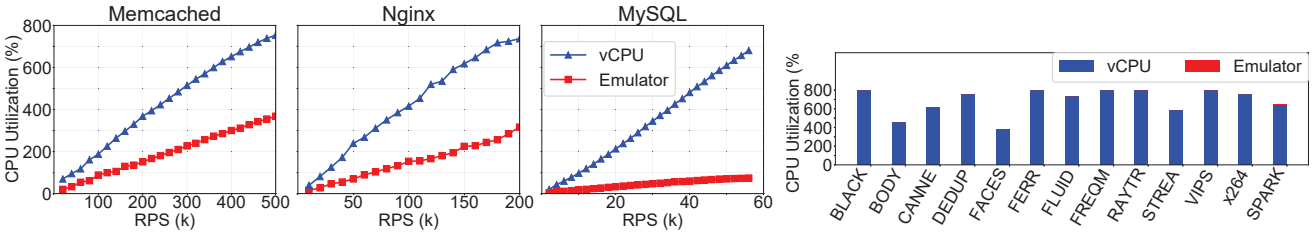


Figure 6: LC applications experience orders of magnitude more scheduling frequency than BE applications.



(a) LC **(b) BE**

Figure 7: CPU utilization of vCPU and emulator threads. BE applications barely use emulator threads.

quency first increases and then decreases along with input load increasing. This is because at low input load (i.e., few requests per second), request inter-arrival time is longer than request processing time, meaning that the application is idle in between requests. As input load increases to a certain extent, new requests come when the processing for old requests has not been finished, and the application threads are active for longer to process consecutive requests. This leads to longer runtime per schedule and fewer context switches. The right two figures in Figure 4 show the total runtime, and the average runtime per schedule, i.e., we divide the total runtime by scheduling count. We see the total time scales linearly with input load. In contrast, the runtime curve per schedule is flat at the beginning, and increases super-linearly after a certain threshold.

- **Guest-side scheduling frequency is a great indicator of application's tail latency.** Comparing the left two figures in Figure 4, we find that the peak point of each curve of scheduling frequency is also the point where application's tail latency starts to increase super linearly with input load. This is because the peak point implies that requests' queuing delay starts to accumulate, which will lead to increased tail latency. Meeting QoS in public clouds essentially means avoiding queuing delay to grow exponentially. This means that we should try staying to the left, or around the peak point of the scheduling frequency curve, and avoid deviating too much to the right of the peak point.
- **Guest-side scheduling frequency can help guide QoS-aware core management decisions on the host OS.** As Figure 4 shows, when *RPS* is less than 100*k*, 2 pCPUs are sufficient to meet QoS. We find that scheduling frequencies under various #pCPUs all overlap. When *RPS* increases to 100k, 4 pCPUs are needed. Guest-side scheduling frequencies under 2 and 4 pCPUs have a 40% difference, while the difference between 4 and 6 pCPUs is only 5%. This motivates us to compare scheduling frequencies between adjacent pCPU counts to obtain the best core allocation.

We show the generality of the findings above to more LC applications in Appendix A.3. It shows that *by comparing guest-side scheduling frequencies between adjacent pCPU counts, we can find the best core management decisions.*

3.4 A Single VM Needs Core Isolation Too

A QEMU-KVM process has two main thread groups, vCPU and emulator threads [31]. Emulator threads are responsible for handling interrupt requests for VM hardware emulation [17]. We find the two thread groups have different core demands, and interfere with each other when sharing cores.

Figure 5 shows how each application in a VM performs when its vCPU and emulator threads (1) share the 8 pCPUs (Shared), or (2) partition the 8 pCPUs into 6 and 2 cores (Isolated), and adopt host-aware isolation in the vCPU core group. We find that:

- Compared with Shared, Isolated achieves 15%-50% higher RPS under the same core count while meeting QoS.
- Emulator threads of LC applications are quite active. Figure 7 shows that CPU utilization of emulator threads is 15%-50% of that of vCPU threads. This is because LC applications are naturally networked applications; requests and responses are all transmitted through network, and part of these network operations are handled by emulator threads. Since request processing time is at the scale of tens of microseconds to a few milliseconds, the network interrupt processing time is a non-negligible part of the end-to-end request latency of LC applications.
- CPU utilization of both vCPU and emulator threads increase linearly with input load.

The observations suggest that *(1) CPU utilization of either vCPU or emulator threads is a great indicator of application's input load, and (2) core allocation of both vCPU and emulator threads should be dynamically adjusted based on input load.*

3.5 How about BE applications?

We have clarified three needs for LC applications: (1) host-aware isolation, (2) monitoring of guest-side scheduling frequency, and (3) isolation between vCPU and emulator threads. However, the three techniques are not necessary for BE applications. In this section, we reveal the fundamental differences between LC and BE applications that lead to the differences in the effectiveness of the three techniques.

We experiment with 12 BE applications from the PARSEC benchmark suite [21] (each benchmark is abbreviated by its first five characters), and 99 queries from TPC-DS [16] deployed in Spark [20]. The spark cluster consists of three 8-vCPU VMs, one master VM and two slave VMs. We report the total execution time of all the 99 queries in Figure 2b, and the average statistics of the three VMs in Figure 6 and 7b.

First, *LC applications are fundamentally different from BE applications in that LC applications are comprised of a number of short requests.* Request latency of LC applications is at sub-millisecond to millisecond granularity (Figure 2a). A thread only runs for tens of microseconds before being scheduled out (Figure 4). On the contrary, BE applications typically consist of fewer and longer tasks. When scheduled, a thread runs for the entire timeslice before being scheduled out. This difference is reflected in context switches that have orders of magnitude difference (Figure 6). Section 3.2 shows that host-aware isolation mainly reduces host-side context switches, which consequently reduces host-side scheduling delay and HLT VM exit handled time. Since BE applications do not suffer from host-side context switches like LC ones, they do not benefit much from host-aware isolation. Figure 2b shows that host-aware isolation reduces execution time by merely 0.7% on average compared with the default core manager. Spark achieves the most reduction (i.e., 7%) because of its relatively higher host-side scheduling frequency (Figure 6).

Second, BE applications have almost no usage of the emulator threads. As shown in Figure 7b, for BE applications, the average CPU utilization of emulator threads is less than 1%. Utilization of each query in Spark is detailed in Appendix A.4. As a result, it is not necessary to separate vCPU and emulator threads for BE applications.

3.6 Comparison with Prior Work

Section 2.2.2 mentioned that prior work on guest-host coordination evaluates only BE workloads. We study the effectiveness of PLE-KVM [32] and eCS [36] in Figure 2, and find that they indeed perform well for BE applications. PLE-KVM and eCS reduce execution time by up to 30% and 42% (average of 5% and 12%), respectively. However, they behave similarly as the default mechanism for LC applications, and significantly under-perform host-aware isolation. This is because PLE-KVM and eCS do not attempt to reduce host-side context-switches or scheduling delay, a severe issue when running LC applications. Also, we find that when prioritizing certain vCPU thread, PLE-KVM and eCS sometimes introduce significant unfairness between vCPUs and VMs.

3.7 Summary

In summary, we make the following key observations:

- Partitioning cores on the host OS is insufficient. The guest OS should be aware of the core allocation decision made by the host, and adjust core management in the guest OS as well such that the number of usable vCPU count is no more than the allocated pCPU count for the VM.
- Partitioning resources between different VMs is insufficient. Threads belonging to the same VM contend for resources as well; vCPU and emulator threads should be isolated within the same VM as well.
- Without application-level performance metrics, the guest OS can still provide useful hints like scheduling frequency and scheduling delay to guide core management on the host (and the guest), meeting application-level QoS targets.
- Prior work considers only BE applications, and fails to unlock the full potential of guest-coordination for LC applications.

4 UFO Design

UFO is a feedback-based controller that dynamically manages core allocation for virtualized public clouds, aiming to accommodate more VMs on a single host without violating applications' QoS. In this section, we describe UFO in detail.

4.1 Design Principles

UFO is designed with three design principles in mind:

- **Prioritize for LC applications:** Our primary goal is to meet QoS for LC applications. In public clouds, there is no user-specified QoS target. So our goal is to avoid LC applications suffering from extensive queuing delay. Since prior works on guest-host coordination all focus on conventional throughput-oriented batch jobs [32, 36, 57], UFO focuses on LC applications. It simply lets all BE applications share the idle pool, and does not explicitly handle core allocation for each BE application.
- **Optimize for virtualized and oversubscribed environments:** Virtualization and oversubscription are ubiquitous in public clouds. This means that double scheduling always exists, and VMs usually do not get as many physical cores as their vCPU size. We tackle the challenges brought by virtualization and oversubscription by guest-host coordination and vCPU-emulator isolation.
- **Focus on core management:** Despite many other shared hardware resources, UFO is currently designed for only

Algorithm 1: UFO's main function.

```
while TRUE do
    monitor for 1s;
    // Adjust the emulator pool first.
    if emuUtil <50% then
        remove cores from the emulator pool as long as
          emuUtil is under 70%;
    else
        if emuUtil >70% then
            assign more cores to the emulator pool to make
              emuUtil under 50%;
            continue;
    // Adjust the vCPU pool for each VM
    for i = 1..N do
        // Record the sample; fit model upon sufficient
           samples; adjust model upon inaccuracy detection
        updateModel(i, pCPU[i], cpuUtil[i], schedFreq[i]);
        // Find the best core count
        p = predict(i, pCPU[i], cpuUtil[i]);
        if p ≠ pCPU[i] then
            // Adjust core allocations on the guest& host
            modify(i, p);
            sleep 3s;
            pCPU[i] = p;
```

core management. This is because other resources such as last-level cache and memory bandwidth do not require special treatment in virtualized and oversubscribed environments, and there are many prior work that specifically targets management of these resources [38, 50, 61, 64].

4.2 Resource Pool

UFO splits all the physical cores on each host into three pools:

- **vCPU Pool** consists of N core groups for N running VMs, and one group is assigned for one VM's vCPU threads. For VM i, UFO dynamically expands or shrinks its pCPU cores ($pCPU[i]$) based on UFO's core predictor (Section 4.5).
- **Emulator Pool** consists of cores allocated for running VMs' emulator threads. All the VMs share the emulator pool. The emulator pool dynamically expands or shrinks based on its CPU utilization (Section 4.3).
- **Idle Pool** consists of all the leftover cores on the host. It provides the source of free cores. An increasingly larger idle pool signals capability to host more VMs, while a shrinking pool signals system overload (Section B.4). The vCPU and emulator pools interact with the idle pool constantly. We introduce how cores flow between pools below. Note that idle VMs (i.e., vCPU utilization is consistently below 1%) are also placed in the Idle Pool. When UFO detects an increase in vCPU utilization, it will move the active VM to the vCPU pool.

4.3 UFO Controller

As shown in Algorithm 1, UFO consists of three stages:

1. **Monitor:** UFO monitors the CPU utilization of the emulator pool on the host OS, and the CPU utilization and scheduling frequency on each guest OS every second.
2. **Adjust the emulator pool:** UFO always makes sure the emulator pool is large enough to satisfy the need of emulator threads. This is achieved by keeping CPU utilization of the emulator pool under 70%. Upon detection of low utilization (i.e., $emuUtil < 50\%$) of the emulator pool, more cores will be moved from the emulator pool to the idle pool. The thresholds can all be adjusted based on practical needs. UFO checks and adjusts the emulator pool before the vCPU pool. This is because the resource demand of emulator threads is only a fraction of the demand of vCPU threads. However, if emulator threads do not get enough cores, vCPU threads will have to be assigned more cores to meet application's performance target (Section 3.4), leading to lower resource efficiency.
3. **Adjust the vCPU pool:** Figure 8 shows the system components for vCPU adjustment in UFO. CPU utilization and scheduling frequency are continuously monitored inside each guest OS, and are fed to the collector in the host OS. The collector in the host OS is responsible for collecting data samples and fitting models upon enough samples ($updateModel()$, Section 4.4, detailed algorithm in Appendix B.1). Then, UFO calculates the best core allocation through the core predictor ($predict()$, Section 4.5, detailed algorithm in Appendix B.2). If differing from the current core allocation, UFO will adjust core allocations through the enforcer components on the host and the guest OS ($modify()$, Section 4.6).

UFO is monitored and scheduled at a fine-grained level of seconds. Refer to Appendix B.3 for more details.

4.4 Modeling of Scheduling Frequency

As discussed in Section 3.3, guest-side scheduling frequency is a great indicator of application's QoS level, and can be used to guide core management decisions. However, it is impractical to obtain the entire curve of scheduling frequency under any input load and any pCPU count. Therefore, UFO leverages VM's CPU utilization as a proxy of application's input load, and builds a model to predict guest-side scheduling frequency under any CPU utilization given a number of (guest-side CPU utilization, guest-side scheduling frequency) pairs.

Figure 10a shows an example of the fitted curves of Nginx's guest-side scheduling delay under eight pCPUs. We find quadratic function fits the bests, represented by $f(u) = Au^2 + Bu + C$ where u is CPU utilization ranging from $[0, 100 * \#pCPU]$. R^2 [6] is a goodness-of-fit measure for predictions. R^2 closer to 1 represents higher accuracy.

The collector in the host OS continuously collects guest-side scheduling frequency and CPU utilization. As each VM

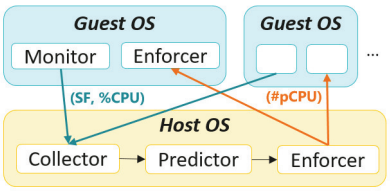
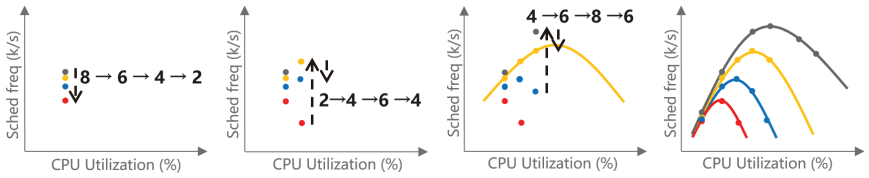
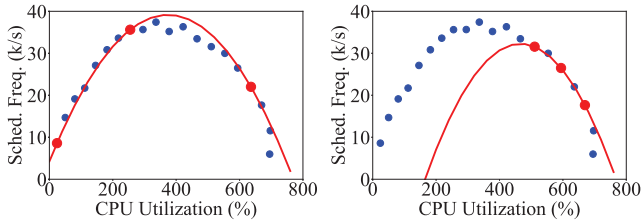


Figure 8: Guest-host coordination in UFO. Monitored events from the guest OSes are fed to the host OS for core prediction. Core adjustment is enforced in the guest and host at the same time.



(a) Initial stage. **(b) Collection stage.** **(c) Fitting stage.** **(d) Full profile.**

Figure 9: Four stages in UFO's vCPU adjustment. (a) and (b) are early stages with few samples, so UFO has to try various core counts. (c) starts model fitting as long as three samples are accumulated for a certain core count. (d) obtains more samples under each core count, fitting more curves and adjusting previous fittings. The best core count can be directly predicted in this stage.



(a) Good samples: $R^2 = 0.962$ **(b) Bad samples:** $R^2 = -0.029$

Figure 10: Fitted curves of Nginx's guest-side scheduling delay under eight pCPUs. All the samples are marked blue, while the samples used for the fitting are marked red. Both fitted curves use three samples. The left figure has samples spanning both low and high CPU utilization, while the right figure is fitted with samples only from high CPU utilization.

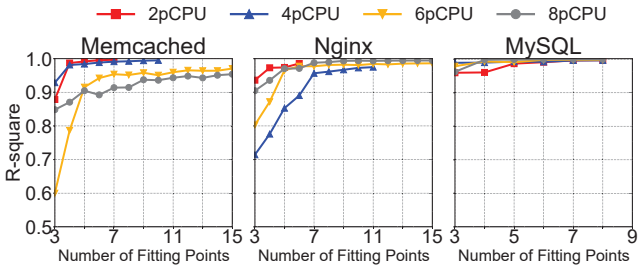


Figure 11: Model accuracy (R^2) of scheduling frequency with increasing number of fitting points. Seven points are enough to reach $R^2 > 0.95$ in most cases.

runs, more samples under various core counts and various CPU utilization are collected. For any given pCPU count, when sufficient samples are collected (e.g., Figure 9c), the curve of scheduling frequency with CPU utilization can be fitted. Ideally, three points determine a quadratic function. However, due to data instability, we find three may not be enough generally. Figure 10 shows a good and a bad case using three fittings points. In Figure 10a, the three samples span both sides of the concave function, and we can successfully fit a model with $R^2 > 0.95$. However, if the three samples all fall into the same side of the curve as shown in Figure 10b, the fitted model can be totally inaccurate. In general, more samples lead to higher accuracy. Practically, we find eight samples to be more than enough for a fitted model to achieve $R^2 > 0.9$, shown in Figure 11. Therefore, UFO maintains the last eight samples under each core count for each VM.

For a given pCPU count, when three samples are recorded, UFO starts fitting the model, though the model may be inaccurate due to skewed samples. The inaccuracy will eventually come to light as the VM continuously runs. Upon detection of inaccuracy (i.e., the difference between the fitted value and the true value is more than 5%), the model will be refitted using the latest and more samples. Storing the latest eight samples and refitting models upon detection of inaccuracy allow UFO to adjust to workload churn inside the VM. For instance, if a VM previously runs Memcached and later runs Nginx, the models will all be updated with the latest samples.

4.5 Core Adjustment

The goal of UFO's core predictor on the host OS is to find the best core allocation for each VM. After the best core count is predicted, the enforcer components on the host and the guest OS will adjust pCPU count for the VM.

As discussed in Section 3.3, c pCPUs is the best core allocation if and only if:

- Scheduling frequency (SF) does not increase much under $c+2$ cores, i.e., $\frac{SF[c+2]-SF[c]}{SF[c+2]} < x$.
- Scheduling frequency drops significantly under $c-2$ cores, i.e., $\frac{SF[c]-SF[c-2]}{SF[c]} > x$.

In this paper, we set the threshold x to 30%. Figure 4 shows why. The vertical lines on the left pictures show the maximum RPS that each pCPU count can sustain while meeting QoS, which are 98k/180k/260k under 2/4/6 pCPUs. The vertical lines run through the figure of guest-side scheduling frequency, such that we can compare scheduling frequency of different #pCPUs under these input loads. We find the difference in scheduling frequency between 2&4/4&6/6&8 pCPUs under 98k/180k/260k RPS is 30%/29%/27% (double-headed arrows in Figure 4). Considering other applications (elaborated in Appendix A.3), we find 30% to be a practically good threshold that universally works for most LC applications.

In reality, UFO will go through four stages in the core predictor, shown in Figure 9. Initially, each VM is given #vCPU pCPUs (8 in the example). Without any model fitting, UFO

has to iteratively try fewer pCPUs, obtain and record SF, check if SF decreases by more than 30%, and finally find the best core count (2 in Figure 9a). When the input load increases in Figure 9b, UFO has to iteratively try increasing #pCPUs until SF increases by less than 30%, and reverts back to last #pCPUs (2->4->6->4 in Figure 9b). To reduce these trials, UFO adopts some heuristics based on CPU utilization when models are not completely fitted, to keep per-core CPU utilization (PCU) roughly within 40%-80%. Suppose current pCPU count is p and current PCU is u. This means that UFO will stop reducing cores if $u * \frac{p}{p-2} > 80\%$, and will stop increasing cores if $u * \frac{p}{p+2} < 40\%$. As the VM runs, more samples are collected, and UFO will start fitting curves when 3 samples are accumulated under a certain pCPU count (Figure 9c), so that UFO can directly infer SF under this pCPU count next time. UFO fits more curves and continuously adjusts the fitting with the latest samples. When the full profile is obtained (Figure 9d), UFO can directly predict the best count without any trial.

4.6 UFO Implementation

Sharing information between guests and the host. Similar to the `kvm_steal_time` implementation [14, 36], when launching a new VM, the guest OS allocates a 32B read-write memory segment that is shared between the guest and the host OS. After the memory allocation, the guest OS sends the memory address of the memory segment to the host OS through a hypercall. The host OS handles the hypercall from each guest OS, recording the memory address.

This segment consists of three 8B integers: two integers are written by the guest-side kernel module, including the latest guest-side CPU utilization and scheduling frequency; another integer is written by the host-side kernel module, denoting the number of pCPUs allocated for this VM.

Guest-side kernel module. A kernel module is installed on each guest OS, recording the guest-side CPU utilization and scheduling frequency to the shared memory every second. It also monitors if #pCPUs in the shared memory changes. If so, it will offline/online some vCPUs in the guest OS.

Host-side kernel module. A kernel module is also installed on the host OS. It periodically does the following tasks: it first reads all the guest-side CPU utilization and scheduling frequency samples, and maintains the latest 8 samples under each pCPU count for each VM. It then fits or adjusts the model, and predicts the pCPU count for each VM. Finally, it writes the prediction result to the shared memory, and adjusts the core allocation of each VM on the host OS accordingly by executing `virsh vcpupin`.

Host-side emulator management. There is another script running continuously on the host OS. This script reads the aggregate CPU utilization of emulator threads of all the VMs every second, and adjusts the emulator pool using `virsh emulatorpin`.

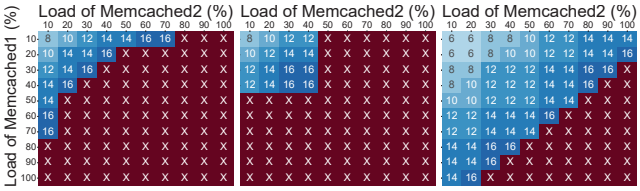


Figure 12: Colocation of 2 Memcached VMs. Each cell represents the minimum required #pCPUs to meet QoS when each Memcached run at the fraction of their *max loads* indicated in the x and y axes. A brown cell with cross mark represents unreachable load combination, meaning that 16 #pCPUs are still insufficient to meat QoS target of both Memcached instances.

5 Experimental Setup

5.1 Hardware Platform

Table 1 shows the specifications of our experimented platform. Both Hyperthreading and turbo boost are enabled to emulate real cloud setups. Since UFO is an intra-node manager, we only experiment with one server node. Clients (i.e., load generators) run on another Intel Xeon machine, with a 10Gbps network link to the server node.

5.2 Applications

We follow testing strategies from prior work [22] to set up our applications, load generators, constant and dynamic load experiments. Table 2 shows the details of our experimented LC applications and the testing strategy. These applications are widely used in industry, and cover different application domains. All the load generators use the default exponential inter-arrival time distributions to simulate a Poisson process, where requests are independent with each other [42]. This means that even under a given constant request-per-second, requests are not generated uniformly. Load still fluctuates at sub-second granularity as shown in Figure 20.

We set the QoS target of each application as the 99th percentile latency of the knee of the latency-with-RPS curve, as marked in Figure 2a and Figure 5, and listed in Table 2 [22]. We define *max load* of each application (also listed in Table 2) as the maximum RPS under the QoS target when 8 vCPUs exclusively run on 8 pCPUs, without emulator interference (emulator threads are allocated on another 8 pCPUs). Note that this means the actual resource need to reach 100% of max load under QoS is more than 8 pCPUs.

We sweep the input load in 10% increments from 10% to 100% of the max load, and record the number of total pCPUs allocated while meeting QoS targets.

5.3 Baselines

We compare UFO with two baselines:

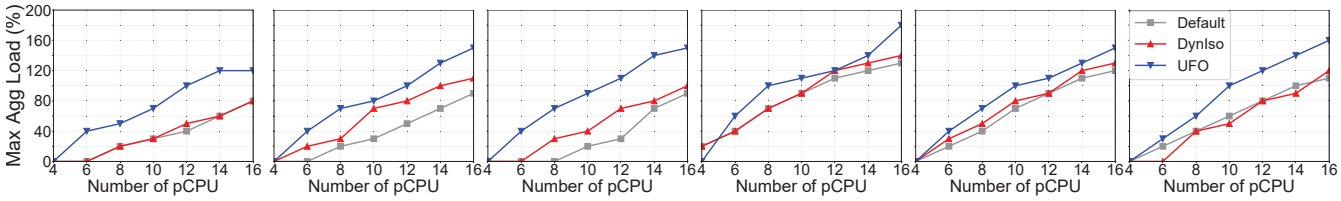
- **Default:** We rely on the OS to schedule VMs. For a given colocation scenario, we increase the number of pCPUs until all the colocated VMs meet QoS target. This is an ideal version of the default manager. Default does not isolate between vCPU and emulator threads.

Table 1: Platform Specification

Model	Intel Xeon Platinum 8378
Microarchitecture	Icelake
Cores/Socket	32 (2 sockets in total)
Threads/Core	2
Hyperthreading	Enabled
Turbo Boost	Enabled
Virtualization Technology	QEMU-KVM
Host and Guest Kernel	5.10
VM Size	8 vCPU, 16 GB memory

Table 2: Latency-critical applications.

Application	**Memcached [11]**	**Nginx [13]**	**MySQL [12]**
Domain	Key-value store	Web server	Database
QoS Target	0.5ms	2ms	15ms
Max Load under QoS	350k	120k	50k
Load Generator	Mutated [7]	wrk2 [9]	sysbench [8]
Dataset	One million <key,value> pairs	10,000 html files of 4KB each	20 tables, each with one million entries
Request Type	100% GET requests	Get file content	OLTP transactions, each with 18 select and 2 update queries



(a) Mem$+Mem$ **(b) Mem$+Nginx** **(c) Mem$+MySQL** **(d) Nginx+Nginx** **(e) Nginx+MySQL** **(f) MySQL+MySQL**

Figure 13: Colocation of 2 VMs. A+B means that VM_1 and VM_2 run application A and B, respectively. The y-axis represents the maximum aggregated load that each core manager is able to sustain while meeting QoS of both VMs under a certain #pCPU.

- **DynIso:** Similar to the Isolation mechanism in Section 3.2, this mechanism does core isolation of vCPU threads between VMs. Since the input load may vary with time, we leverage the feedback-control loop like PARTIES [22] to dynamically decide #pCPUs for each VM, by comparing the application's tail latency with the QoS target. It is an ideal version of the Isolation mechanism. DynIso does not isolate between vCPU and emulator threads.

6 Results and Analysis

In this section, we first evaluate UFO with constant load, and then with dynamic load. We also decompose UFO into vCPU management only and isolation of emulator threads, to show the effectiveness of each component in UFO. Finally, we quantify the overhead of UFO.

6.1 Constant Load

Figure 12 shows colocations of 2 VMs, both running Memcached, under the three core managers. The x and y axes denote the load of Memcached run at the fraction of *max loads*, and the values in the heatmap denote the minimum number of pCPUs required to meet both VMs' QoS targets. Smaller values (or lighter colors) indicate higher resource efficiency. The brown cells with "x" marked indicate that 16 pCPUs are still insufficient to meet QoS Target of both Memcached instances under certain load combinations.We find that:

- DynIso is superior to Default when both VMs run at medium loads, e.g., 40% of max load. This is because DynIso physically separates the two VMs, and is able to eliminate interference at a higher extent.
- Default outperforms DynIso when VMs' loads are imbalanced for two reasons. First, Default manages resources at the granularity of timeslices (i.e., tens of milliseconds). It allows a VM to utilize only a fraction of a pCPU under very low load, and leave the rest of the pCPU to other colocated VMs. Second, DynISO has the restriction to allocate at most #vCPU pCPUs to a VM, while Default could potentially assign more than this amount when any VM is under very high load. For example, when the two Memcached VMs run at 70% and 10% of their max loads, VM2 requires only 2 pCPUs to meet QoS, so the remaining 14 pCPUs cores are all used by VM1, including 8 pCPUs for VM1's vCPU threads, and 4 pCPUs for its emulator threads.
- UFO significantly outperforms Default and DynIso. The number of blue cells is 64 under UFO, meaning that UFO is able to meet QoS of both Memcached instances under 64% of all the load combinations, while Default and DynIso can only achieve 19% and 16%, respectively.

For easier quantitative comparison, for each pCPU count, we record the maximum aggregated load (MAL) that each core manager is able to sustain while meeting QoS of both VMs. Figure 13 plots MAL of six 2-VM co-location mixes (Appendix C.1 shows all the heatmaps). Comparing with DynISO, UFO achieves up to 60% (average of 27%) higher MAL under the same number of pCPUs. On the other hand, under the same input load of each VM, UFO saves up to 50% (average of 22%) cores than DynISO.

In summary, UFO is able to achieve higher MAL under the same pCPU count, and save pCPUs under the same load.

6.2 Dynamic Load

In this section, we evaluate how UFO reacts to various load fluctuation patterns. Figure 14 shows a 3-VM co-location mix, each VM running a different application with a different fluctuation pattern. We assume no prior samples and no previously fitted model in UFO in this experiment. In order to focus

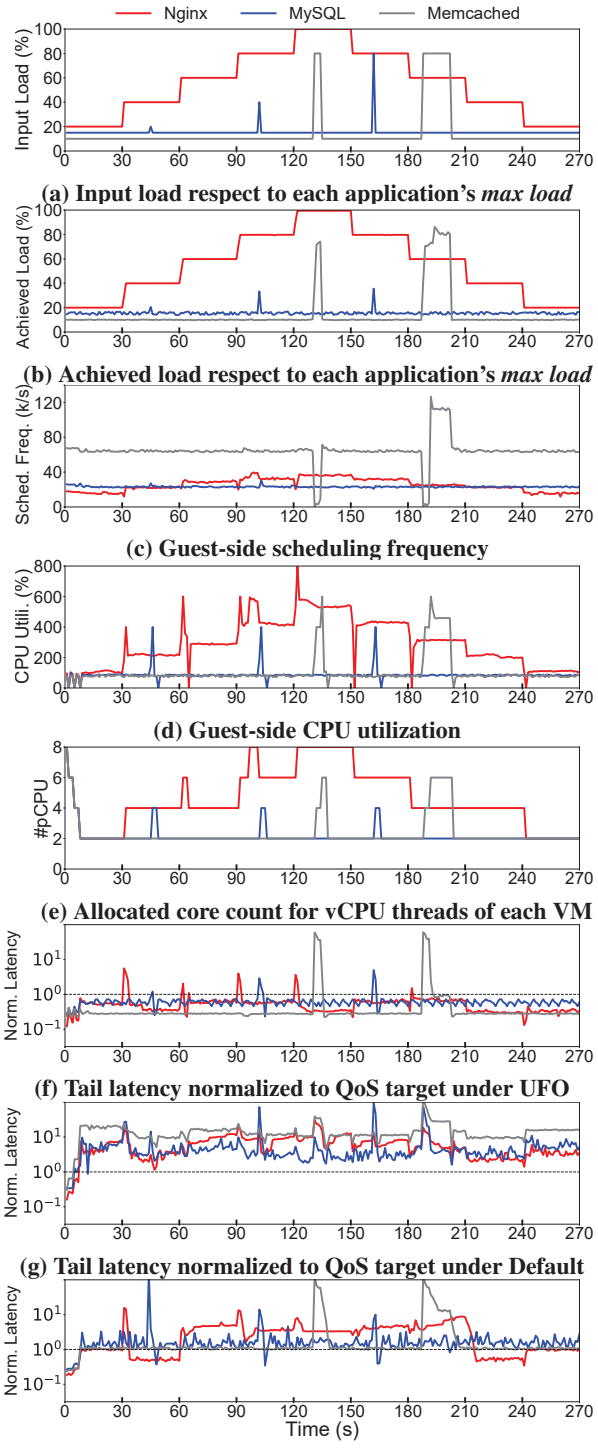


Figure 14: UFO's reactions to fluctuating load patterns. Horizontal dotted lines in Figure 14f, 14g and 14h represent QoS targets.

on evaluating the effect of dynamic core adjustment for vCPU threads among different core managers, we exclude the interference of emulator threads by also applying vCPU-emulator isolation to Default and DynIso, and keep the emulator pool large enough. To compare tail latency, we keep #pCPUs the same on the host OS under all the core managers.

Initially, each VM is given 8 pCPUs. UFO detects low per-core CPU utilization (PCU), and gradually reduces #pCPU. After each adjustment, UFO checks (1) if PCU is higher than 80%, and (2) if guest-side scheduling frequency decreases by more than 30%. Due to the super low load, UFO successfully reduces #pCPU to 6, 4, and 2 at 2s, 6s, and 9s, respectively. QoS is consistently satisfied in this phase. We notice that every time vCPU offlining is performed, CPU utilization of the corresponding VM suddenly drops to 0, then recovers in 1-3s (see the first 10s in Figure 14d).

We also compare UFO with Default and DynIso without guest-host coordination. Default and DynIso are both unable to meet QoS if allocating the same number of cores as UFO.

6.2.1 Diurnal Load Fluctuations

Cloud applications typically have diurnal load fluctuations [30], i.e., load gradually increases during the day, and decreases during the night. Nginx (red lines in Figure 14) mimics such a pattern. The load gradually increases from 20% to 100% in units of 20%, and then drops back to 20%, each step lasting for 30s, and the total duration is 270s.

At 30s, Nginx's load increases and is reflected in an increase in CPU utilization to 184% (PCU of 92%). UFO increases its #pCPUs to 4 and waits for 3s. At 34s, UFO finds that increasing #pCPUs to 6 will lead to PCU lower than 40% (55%*4/6=37%), so it stops increasing #pCPUs further. In this phase, QoS violation only lasts for 3s (Figure 14f) because it takes 1s for UFO to observe load surge and increase #pCPUs, and it takes an additional 2s for system to stabilize and for tail latency to recover. Note that every time vCPU is hot-plugged, CPU utilization bursts and then recovers in 1-3s.

At 60s, UFO detects increase in CPU utilization, and increases #pCPUs to 6. However, after waiting for 3s, UFO finds that guest-side scheduling frequency increases by only 21%, less than the 30% threshold. This means that 6 #pCPUs is too much. Therefore, UFO reverts #pCPUs back to 4 at 64s.

Similarly, UFO increases #pCPUs at 90s and 120s, and successfully resolves the QoS violations of Nginx in 1-3s. At 150s, UFO detects load drop. Since UFO has experienced the same load before, it quickly decreases #pCPUs to the right count without any reversion of #pCPUs.

In summary, UFO is capable of handling diurnal load changes. It reacts to one second after any load change is detected, and performs better as more samples are collected. On the contrary, Default and DynIso consistently violate QoS under the same core count.

6.2.2 Sub-Second Load Bursts

Load bursts refer to a sudden load increase, followed by a load drop. We test two types of bursts that last for varying amounts of time, to test UFO's responsiveness to bursty loads.

We follow testing strategy in Shenango [49] to test sub-second load bursts. The load of MySQL (blue lines in Figure 14a) suddenly changes from the default 15% to 20%,

40% and 80% at 45s, 100s and 160s, respectively. Each burst lasts for only 1s. UFO is able to observe the load burst, and it increases #pCPUs accordingly. However, soon after UFO reacts, the burst has disappeared and application's tail latency recovers by itself. In fact, there is no need for UFO to take reaction. Also, because MySQL is assigned 2 pCPUs initially, if the load burst exceeds the processing capability of 2 pCPUs (i.e., 30% in this case), requests would be dropped, showing in the difference between input load and achieved load in Figure 14a and 14b. Note that latency suffers more under Default (Figure 14g) and DynIso (Figure 14h) despite the reaction time. This shows the effectiveness of UFO's guest-host coordination.

UFO is designed to operate at second-granularity like prior work [22, 44]. Recent studies [28, 34, 49, 53] show the ability to resolve microsecond-scale QoS violations. However, as discussed in Section 2.1, they have to intrude users' applications, and are applicable only in private clouds. Achieving sub-second QoS in public clouds remains an open problem.

6.2.3 Bursts with Increasing Duration

We then increase the burst duration to 5s and 15s. As the grey lines in Figure 14a show, Memcached bursts from 10% to 80% at 130-135s, and 187-202s. Note that this vast degree of load burst is not realistic; loads usually gradually increase or decrease with small spikes [30]. This stress test aims to show the boundary of UFO's responsiveness.

When bursting to 80% of load, Memcached in fact needs 6 pCPUs to meet QoS. Since UFO adjusts 2 pCPUs at a time (with no prior samples and model fitting), it takes two steps to reach the desired core count. At 130s, UFO detects load increase and increases #pCPUs to 4 and 6 at 131s and 134s, respectively, reaching the desired core count in 4s. The second spike lasts longer for 15s. UFO still reaches the desired core count after two steps, and resolves the QoS violation in 4s. This shows that the responsiveness of UFO depends on the number of steps for UFO to adjust. Upon vast load increases, the longer the burst, the better UFO handles.

6.3 Decomposition of UFO

Figure 15 shows the effectiveness of each component in UFO. We experiment with 16-vCPU VMs to show more #pCPU options. UFO-vCPU does not separate emulator and vCPU threads, while UFO-emulator does not online/offline vCPUs to match pCPU count. We find that:

- Compared with the *max load* under Default, UFO-vCPU and UFO-combined are able to achieve an average of 19.8% and 69.8% higher load under QoS, respectively.
- UFO-emulator sometimes even underperforms Default. This is because after reserving some cores for emulator threads, vCPUs are stacked onto fewer pCPUs, causing severe double scheduling symptoms. This is especially detrimental for VMs with high emulator activity. Therefore, it is important to apply vCPU-emulator separation and vCPU-pCPU coordination at the same time.

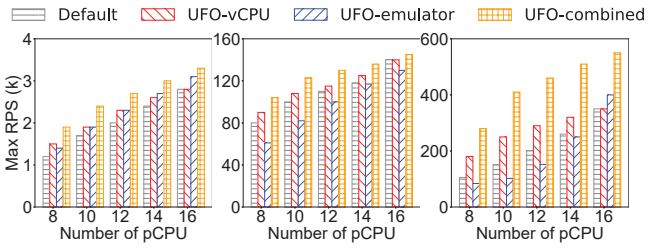


Figure 15: Decomposition of UFO's effectiveness.

6.4 UFO Overhead

UFO mainly involves the following overhead:

- **Guest-side monitoring**: We compare CPU and memory resource utilization, and application's latency with and without guest-side monitoring. The CPU and memory resource consumption is barely observable, and application's tail latency remains the same.
- **Host-side kernel module**: The host OS is responsible for data collection and core prediction. This kernel module takes less than 5% of CPU utilization on average, since it is only active for a few tens of milliseconds every second. Memory consumption scales linearly with the number of VMs and the total number of pCPUs on the host. Assuming a maximum of 50 VMs and 200 pCPUs on the host, memory usage is less than 2MB.
- **Time to online or offline vCPUs in the guest OS**: It takes 20-30ms to online or offline a single vCPU for LC applications. This overhead increases almost linearly (i.e., slightly sublinear) with the number of vCPUs to online or offline (more details in Appendix C.2). Prior work [24] shows that the overhead can be reduced to tens of microseconds per vCPU. We hope similar techniques could be merged to mainline of Linux kernel in the future.
- **Performance impact of onlining/offlining vCPUs in the guest OS**: As shown in Figure 14, onlining vCPUs cause a sudden burst in guest-side CPU utilization, and offlining vCPUs cause a sudden decrease in CPU utilization down to 0. It takes up to 3s for CPU utilization to recover. This explains why we wait for 3s after each vCPU adjustment.

7 Conclusion

We have presented UFO, a core manager for latency-critical applications in virtualized and oversubscribed public clouds. UFO leverages guest-host coordination and inner-VM core isolation, to push core management to the extreme. UFO outperforms state-of-the-art core managers by up to 50% in core saving under the same colocation scenario. It also increases the aggregate system load by up to 60% under the same amount of core resources while still ensuring QoS.

References

[1] Alibaba cloud elastic compute service. https://www.alibabacloud.com/product/ecs.

[2] Alibaba cloud linux. https://alibaba.github.io/cloud-kernel/os.html.

[3] Amazon ec2. https://aws.amazon.com/ec2/.

[4] Amazon linux. https://github.com/amazonlinux/amazon-linux-2023.

[5] Centos end of life date. https://endoflife.date/centos.

[6] Coefficient of determination. https://en.wikipedia.org/wiki/Coefficient_of_determination.

[7] Github page of mutated load generator. https://github.com/scslab/mutated.

[8] Github page of sysbench load generator. https://github.com/akopytov/sysbench.

[9] Github page of wrk2 load generator. https://github.com/sc2682cornell/wrk2.

[10] Guidelines for overcommitting vmware resources.

[11] Memcached official website. https://memcached.org/.

[12] Mysql official website. https://www.mysql.com/.

[13] Nginx official website. https://www.nginx.com/.

[14] Steal time for kvm. https://lwn.net/Articles/449657/.

[15] Tecent cloud virtual machine. https://cloud.tencent.com/product/cvm.

[16] Tpc-ds homepage. https://www.tpc.org/tpcds/.

[17] Using virsh emulatorpin in virtual environments with nfv. https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/10/html/ovs-dpdk_end_to_end_troubleshooting_guide/using_virsh_emulatorpin_in_virtual_environments_with_nfv.

[18] Jeongseob Ahn, Chang Hyun Park, Taekyung Heo, and Jaehyuk Huh. Accelerating critical os services in virtualized systems with flexible micro-sliced cores. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[19] Maryam Amiri and Leyli Mohammad-Khanli. Survey on prediction models of applications for resources provisioning in cloud. *Journal of Network and Computer Applications*, 82:93–113, 2017.

[20] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

[21] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[22] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.

[23] Shuang Chen, Angela Jin, Christina Delimitrou, and José F Martínez. Retail: Opting for learning simplicity to enable qos-aware power management in the cloud. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 155–168. IEEE, 2022.

[24] Luwei Cheng, Jia Rao, and Francis C. M. Lau. Vscale: Automatic and efficient processor scaling for smp virtual machines. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[25] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, 2017.

[26] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[27] Xiaoning Ding, Phillip B. Gibbons, Michael A. Kozuch, and Jianchen Shan. Gleaner: Mitigating the Blocked-Waiter wakeup problem for virtualized multicore applications. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 73–84, Philadelphia, PA, June 2014. USENIX Association.

[28] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[29] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *Proceedings of the International Symposium on Quality of Service*, pages 1–10, 2019.

[30] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629. IEEE, 2018.

[31] SR Jiri Herrmann, Dayle Parker, and Scott Radvan. Red hat enterprise linux 7 virtualization tuning and optimization guide, 2015.

[32] Kenta Ishiguro, Naoki Yasuno, Pierre-Louis Aublin, and Kenji Kono. Mitigating excessive vcpu spinning in vm-agnostic kvm. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 139–152, 2021.

[33] Weiwei Jia, Jianchen Shan, Tsz On Li, Xiaowei Shang, Heming Cui, and Xiaoning Ding. vSMT-IO: Improving I/O performance and efficiency on SMT processors in virtualized clouds. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 449–463. USENIX Association, July 2020.

[34] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for {μsecond-scale} tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.

[35] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Opportunistic spinlocks: Achieving virtual machine scalability in the clouds. *ACM SIGOPS Operating Systems Review*, 50(1):9–16, 2016.

[36] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scaling guest {OS} critical sections with ecs. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 159–172, 2018.

[37] Harshad Kasture, Davide B Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 598–610, 2015.

[38] Harshad Kasture and Daniel Sanchez. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. *ACM SIGPLAN Notices*, 49(4):729–742, 2014.

[39] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[40] Hwanju Kim, Sangwook Kim, Jinkyu Jeong, and Joonwon Lee. Virtual asymmetric multiprocessor for interactive performance of consolidated desktops. *SIGPLAN Not.*, 49(7):29–40, mar 2014.

[41] Hwanju Kim, Sangwook Kim, Jinkyu Jeong, Joonwon Lee, and Seungryoul Maeng. Demand-based coordinated scheduling for smp vms. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 369–380, 2013.

[42] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[43] Yuhang Liu, Xin Deng, Jiapeng Zhou, Mingyu Chen, and Yungang Bao. Ah-q: Quantifying and handling the interference within a datacenter from a system perspective. pages 471–484, 2023.

[44] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

[45] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible colocations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259, 2011.

[46] Paul Marshall, Kate Keahey, and Tim Freeman. Improving utilization of infrastructure clouds. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 205–214. IEEE, 2011.

[47] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. VEE '05, page 13–23, New York, NY, USA, 2005. Association for Computing Machinery.

[48] Rajiv Nishtala, Vinicius Petrucci, Paul Carpenter, and Magnus Sjalander. Twig: Multi-agent task management for colocated latency-critical cloud services. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 167–179. IEEE, 2020.

[49] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.

[50] Jinsu Park, Seongbeom Park, Myeonggyun Han, Jihoon Hyun, and Woongki Baek. Hypart: A hybrid technique for practical memory bandwidth partitioning on commodity servers. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–14, 2018.

[51] Tirthak Patel and Devesh Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 193–206. IEEE, 2020.

[52] Yaqiong Peng, Song Wu, and Hai Jin. Robinhood: Towards efficient work-stealing in virtualized environments. *IEEE Transactions on Parallel and Distributed Systems*, 27(8):2363–2376, 2016.

[53] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne:{Core-Aware} thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, 2018.

[54] Jia Rao and Xiaobo Zhou. Towards fair and efficient smp virtual machine scheduling. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, page 273–286, New York, NY, USA, 2014. Association for Computing Machinery.

[55] Stijn Schildermans, Jianchen Shan, Kris Aerts, Jason Jackrel, and Xiaoning Ding. Virtualization overhead of multithreading in x86 state-of-the-art & remaining challenges. *IEEE Transactions on Parallel and Distributed Systems*, 32(10):2557–2570, 2021.

[56] Jianchen Shan, Xiaoning Ding, and Narain Gehani. Apples: Efficiently handling spin-lock synchronization on virtualized platforms. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):1811–1824, 2017.

[57] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. Schedule processes, not vcpus. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, pages 1–7, 2013.

[58] Orathai Sukwong and Hyong S Kim. Is co-scheduling too expensive for smp vms? In *Proceedings of the sixth conference on Computer systems*, pages 257–272, 2011.

[59] Boris Teabe, Vlad Nitu, Alain Tchana, and Daniel Hagimont. The lock holder and the lock waiter pre-emption problems: Nip them in the bud using informed spinlocks (i-spinlock). In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 286–297, 2017.

[60] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[61] Xiaodong Wang, Shuang Chen, Jeff Setter, and José F Martínez. Swap: Effective fine-grain management of shared last-level caches with minimum hardware support. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 121–132. IEEE, 2017.

[62] Chuliang Weng, Zhigang Wang, Minglu Li, and Xinda Lu. The hybrid scheduling framework for virtual machine systems. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 111–120, 2009.

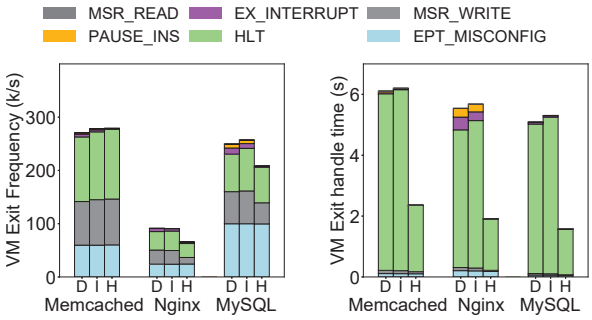
[63] Song Wu, Zhenjiang Xie, Haibao Chen, Sheng Di, Xinyu Zhao, and Hai Jin. Dynamic acceleration of parallel applications in cloud platforms by adaptive time-slice control. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 343–352, 2016.

[64] Ying Zhang, Jian Chen, Xiaowei Jiang, Qiang Liu, Ian M Steiner, Andrew J Herdrich, Kevin Shu, Ripan Das, Long Cui, and Litrin Jiang. Libra: Clearing the cloud through dynamic memory bandwidth management. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 815–826. IEEE, 2021.

Appendix

A Extended Characterization

A.1 Impact on VM Exits under Host-Aware Isolation



(a) #VM Exits **(b) VM Exit Handled Time**

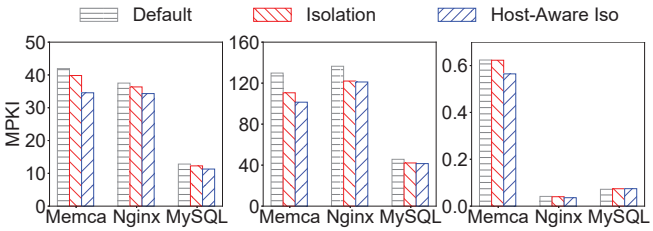
Figure 16: VM exit frequency and VM exit handled time under default (D), isolation (I), and host-aware isolation (H), decomposed by VM exit reason.

VM exits are handled 2x faster on the host under host-aware isolation. Figure 16 shows the number of VM exits and the handling time of VM exits per second under the three core managers, decomposed by exit reason, collected and reported by `perf kvm`. We find that the number of VM exits does not vary much among the three mechanisms, but the handling time is reduced by more than 60%.

For LC applications whose requests come and go, a HLT VM exit is triggered every time when the application becomes idle in between requests, and a VM entry is triggered every time when a new request is received. HLT handling time is the time between a HLT vm exit and the subsequent vm entry. It signals how fast the system reacts to new requests. Under host-aware isolation, running vCPU count never exceeds pCPU count, so that there are always free pCPUs to handle VM entries as quickly as possible. However, under other mechanisms, vCPU count is more than pCPU count, and VM entries cannot be consumed immediately since there may be no idle pCPU.

A.2 Impact on Caches under Host-Aware Isolation

Caches are better utilized under host-aware isolation. Figure 17 shows misses-per-kilo-instructions (MPKI) of L1 instruction cache (L1I), L1 data cache (L1D), and last-level cache (LLC). Compared with Default, Isolation reduces L1D and L1I MPKI by up to 5% and 15% (average of 4.1% and 11%), respectively. This is because under Default, a vCPU can freely move around between 16 pCPUs, while Isolation restricts a vCPU to a fixed set of 4 pCPUs, eliminating cache pollution and context switches with other VMs' vCPUs. This explains why Isolation outperforms Default in Figure 2, despite occasionally higher guest-side scheduling frequency. Host-aware isolation further reduces L1D, L1I and LLC MPKI by up to 12.5%, 7% and 10% (average of 8.5%, 2.7%, and 6%), respectively. This is because our host-aware isolation can reduce the context switches and in turn decrease the times of cache line evictions.



(a) L1 dcache **(b) L1 icache** **(c) Last-level cache**

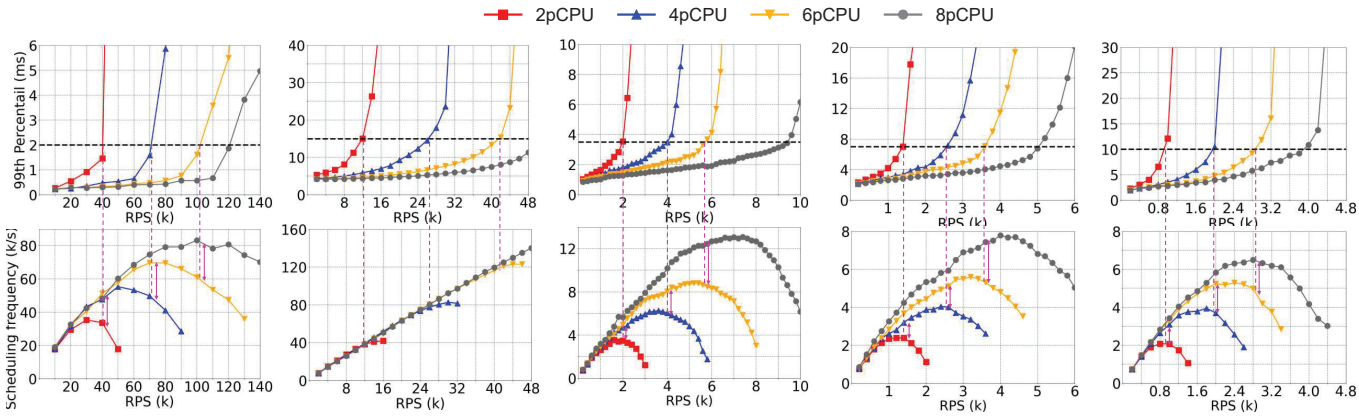
Figure 17: Cache misses-per-kilo-instructions (MPKI) under three core managers.

A.3 Indications of Guest-Side Scheduling Frequency

To show the generality of guest-side scheduling frequency, we experiment with three more LC applications from Tailbench [39], including ImgDNN, Xapian, and Moses. Adding Nginx and MySQL, Figure 18 shows tail latency and guest-side scheduling frequency with increasing RPS under various pCPU count of the five LC applications (Memcached is already shown in Figure 4).

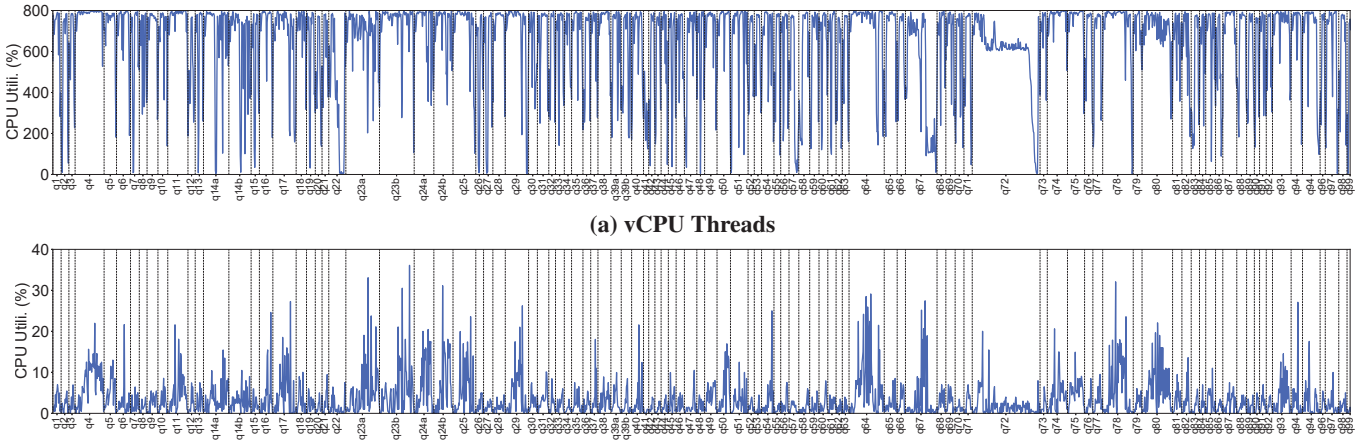
For all the evaluated applications except for MySQL, guest-side scheduling frequency is a concave function with RPS. Given a certain input load, we can compare guest-side scheduling frequency between adjacent pCPUs, and check if the difference is above a certain threshold. 30% universally work for these LC applications. For instance, for Nginx at RPS=80k, scheduling frequency under 8 and 6 pCPUs are very similar. However, when reducing pCPU count to 4, guest-side scheduling frequency is reduced by 43%. Therefore, we have to assign 6 pCPUs for Nginx at RPS=80k.

MySQL is the only exception: scheduling frequency increases with RPS and then reaches a short plateau. We find that MySQL is fundamentally different from other applications in that a single MySQL request involves multiple application threads and thus multiple guest-side context switches. Suppose one MySQL request triggers X context switches and the current RPS is R, there will be $X * R$ context switches per second regardless of how small or large R is. On the contrary, a request of the other five LC applications is only processed by a single application thread. At low load, the application thread will be scheduled in upon request arrival, and scheduled out after request processing. When load increases and there is no free time between consecutive requests, the application thread will not be immediately scheduled out after a single



(a) Nginx (b) MySQL (c) ImgDNN (d) Xapian (e) Moses

Figure 18: Relationship between application's tail latency and guest-side scheduling frequency of five LC applications, apart from Memcached which is already shown in Figure 4. The x-axis represents increasing input load. The horizontal dotted lines represent QoS targets. The vertical purple lines represent the maximum RPS under QoS under each pCPU count. Purple arrows represent the difference in guest-scheduling frequency between adjacent pCPUs.



(a) vCPU Threads

(b) Emulator Threads

Figure 19: CPU utilization of vCPU threads and emulator threads when running the 99 queries in Spark. The x-axis represent time, but is marked by query index to differentiate different queries. Vertical dotted lines also split all the queries.

request, and will keep processing the next request, resulting in reduced context switches at higher load.

This potentially means that we need to set a different threshold for MySQL, e.g., 5%. Fortunately, we find that this is not necessary. As introduced in Section 4.5, UFO also adopts heuristics based on CPU utilization to guide core adjustment. This is primarily to reduce trials when models are not fitted for most applications. However, it also sets a bottom line for core adjustment: when 30% is too relaxed for some applications, UFO will not keep reducing pCPUs which may cause QoS violations. UFO will ensure that per-core CPU utilization is under 80%.

There are some alternative methods to handle cases like MySQL. UFO does not adopt these methods currently, but we leave more options for comprehensiveness. The first method is to reduce the threshold from 30% to 5%. While this is conservative for the other five applications, it is a simple and universal approach, and we can achieve lower tail latency (at the cost of more resources). For instance, as shown in Figure 18a, if the threshold is set to 30%, Nginx at RPS=40k would request only 2 pCPUs to reach a tail latency of 2ms. If setting the threshold to 5%, Nginx at RPS=40k would be assigned 4 pCPUs, reaching a tail latency of 0.5ms. The second approach is to differentiate the two types of LC applications, and set different thresholds for each category. As shown in Figure 18, when the full profile of guest-side scheduling frequency is obtained, it is very easy to distinguish MySQL-like applications from other applications since scheduling frequency does not drop at higher load. We set the threshold to 5% once identifying these applications, and keep the threshold of 30% for all other LC applications.

Table 3: Symbols in UFO

N	Number of virtual machines
$vCPU[i]$	Requested vCPU count of VM_i
$pCPU[i]$	Allocated pCPU count of VM_i
$emuUtil$	Current CPU utilization of the emulator pool
$cpuUtil[i]$	Current CPU utilization of VM_i
$schedFreq[i]$	Current scheduling frequency of VM_i
$SF[i][c]$	Recorded scheduling frequency of VM_i under c pCPUs A dictionary of (u, f) pairs, where u and f are CPU utilization and scheduling frequency, respectively.
$Model[i][c]$	Fitted model of scheduling frequency of VM_i under c pCPUs using $SF[i][c]$, a quadratic function denoted by (A_{ic}, B_{ic}, C_{ic}) Default value is $NULL$ if not fitted yet
$Pred[i][c]$	Max CPU utilization that can be sustained for VM_i under c pCPUs, predicted using $Model[i][c]$ and $Model[i][c+2]$ $Pred[i][0] = 0$. If $c > 0$, default is -1 if not predicted yet.

A.4 CPU Utilization of Spark SQL

Figure 19 shows CPU utilization of vCPU and emulator threads when running each query in Spark. vCPU threads may run up to 800% of CPU utilization (8-vCPU VMs), while emulator threads use at most 40% of CPU utilization. The relative low usage of emulator threads makes isolation of emulator threads ineffective for Spark.

B Extended UFO Design

In this section, we show the detailed algorithms in the UFO design. The algorithms cover all the four stages in UFO's vCPU adjustment, shown in Figure 9. Table 3 includes all the symbols used in the algorithms.

B.1 Update Modeling of Scheduling Frequency

Algorithm 2 shows how and when the modeling of scheduling frequency is updated. For VM_i, when it just gets launched on the host, no samples have been recorded (i.e., $len(SF[i][c]) = 0$ for any c — the number of cores), and no models have been fitted yet (i.e., $Model[i][c] = NULL$ is for any c).

As the VM runs, an increasing number of samples at various CPU utilization and pCPUs are recorded, and we maintain the last eight samples in $SF[i][c]$ for each c. For a given pCPU count, when three samples are recorded, we will start fitting the model, though the model may be inaccurate due to skewed samples. The inaccuracy will eventually come to light as the VM runs and more samples are collected. Upon detection of inaccuracy (i.e., the difference between the fitted value and the true value is more than 5%), the model will be refitted using the latest and more samples. Storing the latest eight samples and refitting models upon detection of inaccuracy allow UFO to adjust to workload churn inside the VM. For instance, if a VM previously runs Memcached and later runs Nginx, SF and $Model$ will all shortly be updated with the latest samples.

Algorithm 2: $updateModel(i,c,u,f)$: Update samples and the fitted model of VM_i under c pCPUs. VM_i's current CPU utilization and scheduling frequency are u and f, respectively.

```
if u is not in SF[i][c] then
    // The first time for VM_i to reach util of u under c cores
    add (u, f) to SF[i][c];
    if len(SF[i][c]) > 8 then
        // Too many samples. Maintain the latest 8 samples for modeling
        remove the eldest element in SF[i][c] ;
if Model[i][c] == NULL then
    if len(SF[i][c]) == 3 then
        // First time to collect enough samples for model fitting
        Fit Model[i][c];
        if Model[i][c+2] ≠ NULL then
            // Record the maximum CPU utilization that c pCPUS can sustain under QoS
            Pred[i][c] = max_{getSF(i,c,u)/getSF(i,c+2,u) ≥ 0.7} u;
else
    if |f − SF[i][c][u]|/f > 5% or
      |f − (A_ic * u^2 + B_ic * u + C_ic))|/f > 5% then
        // The current recorded value or the fitted value is inaccurate
        Fit Model[i][c];
        if Model[i][c+2] ≠ NULL then
            // Update the max CPU utilization
            Pred[i][c] = max_{getSF(i,c,u)/getSF(i,c+2,u) ≥ 0.7} u;
return;
```

When models of consecutive pCPUs are fitted, i.e., both $Model[i][c]$ and $Model[i][c+2]$ are obtained, we start calculating $Pred[i][c]$. We clarify the reasoning behind $Pred[i][c]$ in Appendix B.2.

B.2 Core Predictor

The goal of UFO's core predictor $predict(i,c,u)$ is to output the best core allocation for VM_i, given its current pCPU count c and current CPU utilization u. Note that its scheduling frequency is monitored and fed to the module of $updateModel()$ (see Algorithm 1), so we do not need to explicitly pass it as an argument in the core predictor.

As discussed in Section 3.3, we can leverage guest-side scheduling frequency to guide core allocation decisions on the host OS. Comparing the relationship between scheduling frequency and application's tail latency, we find two conditions to help determine the best core allocation.

Suppose VM_i current runs at CPU utilization u with scheduling frequency f under c pCPUs, c is the best core allocation if and only if:

Algorithm 3: $predict(i,c,u)$: Predict core allocation for VM_i whose current pCPU count is c and CPU utilization is u.

```
for p == 2, 4 .. vCPU[i] do
    // Check if we can directly predict based on u
    if Pred[i][p−2] == −1 then
        continue;
    if Pred[i][p−2] < u and Pred[i][p] ≥ u then
        return p;
p = c;
if p+2 ≤ vCPU[i] and getSF(i,p,u)/getSF(i,p+2,u) < 0.7 then
    // c pCPUs is not enough, check more pCPUs
    while p+2 ≤ vCPU[i] and getSF(i,p,u)/getSF(i,p+2,u) < 0.7 do
        p=p+2;
else
    // c pCPUs is already enough, check fewer pCPUs
    while p−2 > 0 and getSF(i,p−2,u)/getSF(i,p,u) > 0.7) do
        p=p-2;
return p;
```

Algorithm 4: $getSF(i,c,u)$: Get scheduling frequency of VM_i under c pCPUs and u CPU utilization.

```
if u in SF[i][c] then
    // A previously recorded pair
    return SF[i][c][u];
if Model[i][c] ≠ NULL then
    // A previously fitted model
    return A_ic * u^2 + B_ic * u + C_ic;
// Adjust pCPUs to collect scheduling frequency
modify(i, c);
sleep for 3s;
updateModel(i, c, cpuUtil[i], schedFreq[i]);
return schedFreq[i];
```

- $c+2$ is too much, i.e., scheduling frequency does not increase much with more cores.
- $c-2$ is not enough, i.e., scheduling frequency drops significantly with fewer cores.

UFO checks the two conditions above by comparing f with scheduling frequency f' under adjacent pCPU counts, and checks if the difference between f and f' is within a certain threshold.

B.3 Monitoring and Reaction Frequency

As discussed in Section 4.3, UFO operates at second granularity, monitoring every second, and waiting for three seconds after each core adjustment. This is because:

- As shown in Figure 20, monitoring at finer granularity causes unstable results. As introduced in Section 5.2, even under constant load, there are many load and latency fluctuations due to the exponential request inter-arrival distribution. UFO is designed for black-box public clouds and does not assume any application-level knowledge like prior fine-grained approaches [28, 49]. It aims to meet QoS in the long run (i.e., at least over tens of seconds like Heracles [44] and PARTIES [22]).

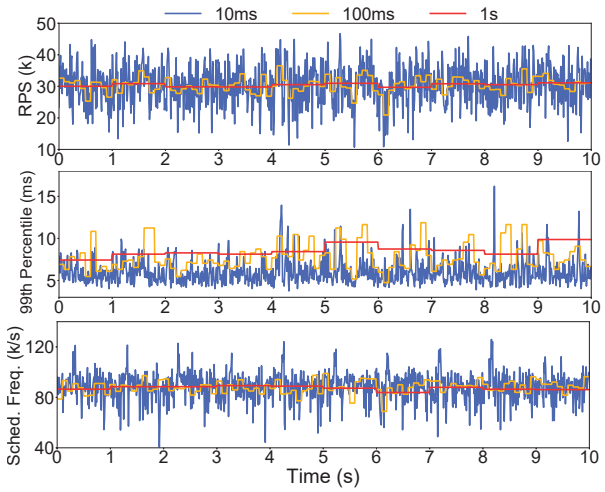


Figure 20: Data stability with increasing monitoring interval when MySQL runs at RPS of 30000. Despite constant RPS, request inter-arrival time follows an exponential distribution, causing significant load and latency fluctuations at sub-second granularity.

- We find that after vCPU onlining/offlining in the guest OS, it will be up to 3s for guest-side CPU utilization to stabilize (more discussion in Section 6.4). We evaluate how this affects UFO's responsiveness to bursty loads in Section 6.2.

B.4 Interaction with the Cluster Manager

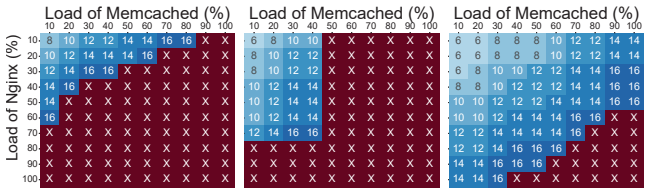
UFO is an intra-node core management, and should interact with the high-level cluster manager based on the size of the idle pool. There are three cases of the idle pool size:

- If the idle pool consistently has more than a handful of idle cores (e.g., $\#idle\ cores > 2$), this signals system underloading, and UFO will signal the cluster manager to schedule more tasks to the node to make use of the idle pool.
- If the idle pool is constantly small (e.g., $0 \leq \#idle\ cores \leq 2$), this means there is *just* the right amount of system load, and the cluster manager should stop scheduling more tasks to the node.
- If the idle pool consistently fails to supply the vCPU or the emulator pool with more needed cores, UFO will signal the cluster manager to migrate some tasks to other nodes.

C Extended Evaluation

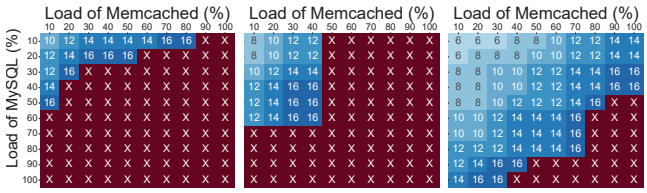
C.1 Heatmaps of 2-VM Colocation Mixes

We show heatmaps of all the 2-VM colocation mixes in Figure 21, 22, 23, 24 and 25. Heatmaps of colocation of Memcached and Memcached is already shown in Figure 12.



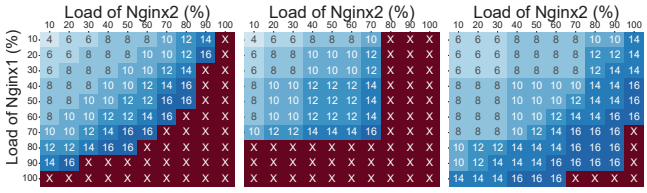
(a) Default **(b) DynIso** **(c) UFO**

Figure 21: Colocation of Memcached and Nginx.



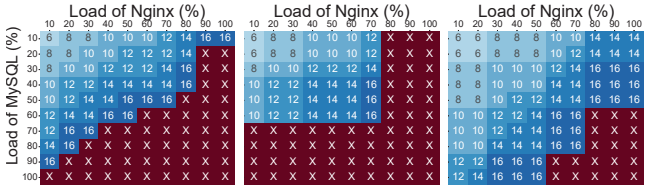
(a) Default **(b) DynIso** **(c) UFO**

Figure 22: Colocation of Memcached and MySQL.



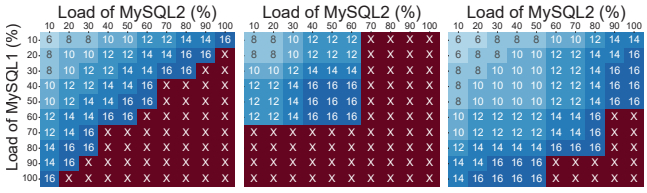
(a) Default **(b) DynIso** **(c) UFO**

Figure 23: Colocation of Nginx and Nginx.



(a) Default **(b) DynIso** **(c) UFO**

Figure 24: Colocation of Nginx and MySQL.



(a) Default **(b) DynIso** **(c) UFO**

Figure 25: Colocation of MySQL and MySQL.

C.2 UFO Overhead

Figure 26 shows the time required for onlining and offlining increasing number of vCPUs in the guest OS under various workloads. For each experiment, we consecutively online and offline vCPUs for 1000 times, obtain the total overhead, and report the average overhead per adjustment. For LC workloads, the overhead is about 20-30ms per vCPU. Overhead is higher under BE workloads. This is because BE applications tend to operate at very high CPU utilization (Figure 7). When offlining a certain vCPU, application threads running on the vCPU will all be migrated to other vCPUs. Therefore, higher CPU utilization usually results in higher overhead.

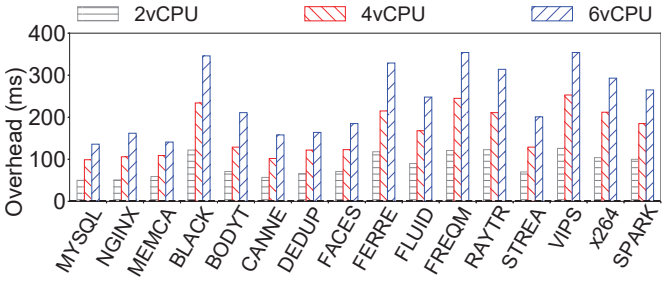


Figure 26: Average time spent on onlining or offlining 2, 4 and 6 vCPUs under various workloads. LC applications operate at their *max load*.

Automatic Parallelization of Software Network Functions

Francisco Pereira Fernando M. V. Ramos Luis Pedrosa
INESC-ID, Instituto Superior Técnico, University of Lisbon

Abstract

Software network functions (NFs) trade-off flexibility and ease of deployment for an increased challenge of performance. The traditional way to increase NF performance is by distributing traffic to multiple CPU cores, but this poses a significant challenge: *how to parallelize an NF without breaking its semantics?* We propose Maestro, a tool that analyzes a sequential implementation of an NF and automatically generates an enhanced parallel version that carefully configures the NIC's Receive Side Scaling mechanism to distribute traffic across cores, while preserving semantics. When possible, Maestro orchestrates a shared-nothing architecture, with each core operating independently without shared memory coordination, maximizing performance. Otherwise, Maestro choreographs a fine-grained read-write locking mechanism that optimizes operation for typical Internet traffic. We parallelized 8 software NFs and show that they generally scale-up linearly until bottlenecked by PCIe when using small packets or by 100 Gbps line-rate with typical Internet traffic. Maestro further outperforms modern hardware-based transactional memory mechanisms, even for challenging parallel-unfriendly workloads.

1 Introduction

With the transition of Network Functions (or NFs) from custom, fixed-function devices to software running on commodity hardware came a well known performance challenge. As line-rates kept increasing, the networking community kept proposing new tools, techniques, and architectural enhancements to overcome individual bottlenecks. User-mode frameworks, like DPDK [38], bypass the kernel, avoiding costly context switches; DDIO [41] places incoming packets directly in the CPU cache as they arrive; and NICs implement Receive Side Scaling (RSS) [72] to consistently distribute traffic across multiple CPU cores using a configurable hash-function. Despite this wealth of tools, the challenge of developing performant software at these time scales is considerable, typically requiring parallelization [30] and, with it, a deep knowledge of low-level architectural details such as cache-friendly allocation, cache-coherence-aware coordination, and a deep understanding of the RSS hashing mechanism.

Although parallelization is paramount to achieving high performance, ensuring equivalence between parallel and sequential implementations is hard [22, 35, 50, 63, 67]. Thus, we argue that *developers need not shoulder the burden of fine-grained parallelization themselves*. Much like how developers typically do not write entire code-bases in assembly language, allowing a compiler to analyze their code, extract its functionality, and build an assembly implementation that is equivalent in semantics, we argue that the fine-scaled parallelization of NFs should follow a similar approach. Developers should implement sequential versions of their NFs, benefiting from the inherent simplicity of testing, debugging, and updating such systems, and when deploying to production they can "compile" the NF to obtain its parallelized version.

There are two key insights supporting the solution for this challenge. Due to the increasingly pervasive use of NF frameworks amenable to symbolic execution [1, 3, 10, 13, 17, 37, 45, 52, 68, 76], the first key insight is that this technique can be used to not only analyze the NF and infer how it maintains state, but also automatically generate modified versions of it. The second key insight is that by knowing how the NF maintains its state, we can configure the RSS mechanism to send packets accessing the same state to the same core, aiming to minimize inter-core coordination in a parallel implementation, thus maximizing performance.

With these key insights in mind, we propose **Maestro**, a tool that automatically analyzes a software NF and generates a new implementation that distributes the workload across multiple cores while preserving the semantics of the sequential implementation. This analysis builds a comprehensive symbolic model of how the NF stores and accesses state, and how that state is structured around flows. Flows (also called *flowspace* [50] and *scope* [22] by prior work) describe related packets—identified through packet header fields—that the NF logically tracks as an isolated unit. A firewall, for example, often tracks TCP/UDP flows, identified by the packet 5-tuple (source and destination IPs and ports and the IP protocol number), whereas a traffic monitor may identify flows by destination IP alone. As NFs typically store state on a per-flow basis [50, 69], Maestro learns how flows are defined in the NF by extracting the constraints that define how packets access state. We then use a solver to find an RSS configuration that distributes traffic across multiple CPU cores, in such a way as to minimize costly inter-core coordination. Our tool then automatically generates a new implementation of the NF that parallelizes its operation accordingly.

When possible, Maestro generates an implementation based on a *shared-nothing architecture*, wherein RSS is configured to forward packets of the same flow to the same CPU core, completely eliminating any inter-core coordination. When the NF is not compatible with such a model, Maestro can still generate a parallel implementation where cores share state but accesses to that state are coordinated by a read-write locking mechanism that, while not as performant

as a shared-nothing architecture, can still perform well under typical (Zipfian) Internet traffic.

Maestro draws inspiration from prior work in NF analysis [50] and verification [76, 77], as well as the wisdom of a wide body of research on NF performance [25, 30, 45, 60]. We also use the lessons learned by many before us that address the challenges of *manually* parallelizing NFs, including NUMA considerations [29], configuring RSS for symmetric flow handling [74], and rebalancing load with skew [8].

Maestro handles DPDK NFs which store state using the Vigor API [76]. For these NFs to be amenable to ESE, they are implemented under some constraints, which we describe in §5. These limitations, however, pertain only to NFs given as input to Maestro, and not to the generated parallel solutions.

We evaluate the performance of Maestro by parallelizing 8 DPDK NFs. Our experimental evaluation shows that NFs that can be parallelized using the shared-nothing architecture scale linearly with the number of cores used until bottlenecked by PCIe when using small packets or by 100 Gbps line-rate with typical Internet traffic [12]. The remaining NFs that require read-write locks to maintain their semantics vary their performance with the workload. High-churn traffic–where most packets establish a new flow–requires more writing to shared state, degrading performance. Fortunately, the majority of packets in typical Internet traffic belong to a minority of flows [12], requiring less state writing and allowing more concurrency. Under this read-heavy traffic, Maestro's lock-based parallel NFs perform comparably to a shared-nothing model. Notably, when Maestro had to resort to locking, equivalent versions of the NFs that use hardware transactional memory [54] (TM) to preserve semantics (via the Restricted Transactional Memory interface [42]) were unable to outperform our optimized locks, as we show in §6.4. We also show that NFs automatically parallelized by Maestro rival in performance with ones manually parallelized using VPP [7].

In §2, we describe the inherent challenge of parallelizing NFs, to better motivate our work. We subsequently present the main contributions of our work, describing the Maestro architecture in §3 and several key optimizations in §4. In §5 we discuss Maestro's inherent limitations. In §6, we evaluate Maestro and the performance of the parallel NFs it generates. Finally, we describe related work in §7 and conclude with final thoughts in §8.

2 Why Parallelization is Hard

Ideally, one would parallelize an NF by spinning up individual instances per core, each running independently, and using the NIC to evenly distributing traffic among them. NFs, however, typically store state that persists across packets. Sharing this state among cores requires coordinating access to it, but minimizing this coordination is crucial to achieving high performance. Parallel implementations that require no state sharing among their instances (and therefore no synchronization) are called *shared-nothing*. Implementing a shared-nothing implementation of a stateful NF requires carefully configuring the NIC to distribute traffic to each core in a way that aligns with how state is structured in the NF. With such a mechanism, state is *sharded* across cores and packets accessing the same state always find themselves on the same core.

The NIC can perform this traffic distribution in hardware using the Receive-Side Scaling (RSS) mechanism [72]. This mechanism hashes packet headers using a user-defined set of fields and a hash key. The computed hash is subsequently used to direct traffic to different queues which can deliver the packets to different cores. To send, for example, packets of the same TCP flow to the same core, one would configure RSS to hash the source and destination IP addresses, and TCP/UDP ports, and the IP protocol number (*i.e.* the 5-tuple), ensuring that any two packets with the same 5-tuple will have the same hash and will end up on the same core.

This leads us to the traditional method for building parallel shared-nothing NFs: first, developers shard state in the NF, building a full understanding of how state is accessed under all circumstances. They then use this sharding solution to construct an RSS configuration that distributes traffic accordingly. This approach, however, poses three big challenges:

1. Finding the right sharding solutions is hard. Though some NFs simply shard on the 5-tuple, many others require a more careful approach. One common use case involves symmetrical access to state based on the 5-tuple so that incoming traffic—that has the source and destination swapped—access the same state as outgoing traffic [74]. Other NFs require a more coarse-grained partitioning: some policers and traffic monitors only use the destination addresses to index state, connection limiters may only use source addresses, and network address translators (NATs) will typically shard on the WAN's server address and port (as all the other addresses and ports are translated). Simply sharding on the 5-tuple here would require expensive coordination (*e.g.* locks), as cores are unable to act independently.

Arriving at sharding solutions is harder than generically using locks each time state is accessed. The developer needs intricate knowledge of the NF's semantics and internals, particularly around how state is kept and manipulated. This thought process must not only take place upon initial implementation, but also as the NF code evolves over time. Augmenting a firewall with a connection limiter feature renders the previously configured 5-tuple sharding obsolete, requiring a complete rethink of how it should be sharded.

2. Finding the right RSS configuration is hard. Even if we take the sharding solution for granted, configuring RSS accordingly is difficult. For trivial cases, this is just a matter of selecting the right fields to hash but more complex scenarios can require carefully crafting the RSS key. Such an approach was used in [74] to handle symmetrical TCP/UDP flows, but manually tracking the sharding constraints and finding internal symmetries in the hash key that pair with those constraints quickly becomes unmanageable. For NFs with other sharding

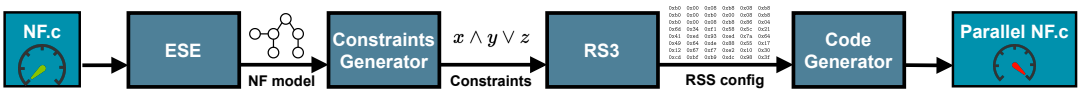


Figure 1: Maestro's architecture

requirements, the problem becomes even harder. Not all sets of fields are supported by NICs [39, 40], requiring a specific RSS key that cancels out some bits to circumvents this limitation. One might even require symmetry between different interfaces (when incoming and outgoing traffic use different NICs), which requires a separate but interrelated configuration and key for each NIC. More complex NFs can shard state in ways that do not neatly fit into any common case, requiring a custom formulation which, as before, may need to be completely rethought from scratch should the NF change over time. Some cases are outright infeasible, due to inherent NIC limitations, at which point a well-placed warning could help guide developers towards better solutions.

3. Writing performant parallel code is hard. Even if a developer correctly shards the NF and properly configures RSS to achieve a valid shared-nothing solution, they can still be leaving performance on the table. Though shared-nothing goes a long way towards ensuring good performance, many more minute details play a further role in parallel code. Packet buffers and state must now be cache-aligned to avoid false cache-line sharing. Memory allocation must be NUMA-aware to avoid slower remote accesses across the QPI bus. Even exogenous factors like traffic skew must now be considered [8] to fully realize the potential of a parallel implementation.

Getting any of these issues wrong can stand in the way of performance, correctness, or both, but are ultimately amenable to automation. Our tool—Maestro—tackles the first challenge by analyzing how the NF keeps its state and finding the constraints that packets that need to be sent to the same core must satisfy. It further tackles the second challenge by formulating an SMT problem and using a solver to find the right RSS keys that satisfy the sharding requirements. Finally, Maestro addresses the third challenge by automatically generating a parallel implementation that is semantically equivalent to its sequential counterpart. The generated code fully handles NIC initialization and RSS configuration, cache-alignment, load-balancing, and NUMA considerations. Even when a shared-nothing approach is not possible, Maestro can still help by generating an optimized lock-based parallel implementation that uses carefully crafted read-write locks to minimize inter-core coordination with typical Internet power-law traffic.

3 Maestro Architecture

Maestro uses symbolic analysis to extract information on how the NF maintains state, and with it infer possible dependencies between parallel instances. This analysis is crucial to achieve synchronization-free parallelization that shards state by carefully splitting traffic among cores. How this careful orchestration of packets can be used to avoid synchronization among parallel instances is better explained via an example.

3.1 Parallelizing a firewall

Consider a firewall NF connecting a LAN and a WAN that only forwards packets from the WAN that correspond to flows started in the LAN. To keep track of ongoing flows, it stores flow information in a map. Packets from the WAN lookup flow information symmetrically relative to packets from the LAN, naturally swapping source and destination fields.

Note that not all packets need access to all entries in the map: only the ones belonging to the packet's flow. As such, in a parallel execution, making sure that *packets of the same flow are sent to the same core*, conjoined with the fact that packets of the same core are processed sequentially, allows us to parallelize this firewall without any synchronization between its instances—a *shared-nothing* architecture.

This orchestration of packets from the same flow to the same core requires a specific RSS configuration. Not only must we send LAN packets of the same flow to the same core, but also their (symmetric) WAN responses. A configuration partially fulfilling these requirements was already found by Woo and Park [74][1]. By adapting their configuration to the firewalls' needs, we ensure that every packet that needs access to the same memory region is sent to the same core.

3.2 Generalizing NF parallelization

The above parallelization process is well tailored for our firewall, but different NFs keep state in different ways, and thus require different sharding solutions. Moreover, when access to specific state precludes flow-sharding, synchronization is necessary to maintain semantics.

Maestro deals with this parallelization process automatically by using the architecture shown in Figure 1. Maestro starts by analyzing the NF using Exhaustive Symbolic Execution (ESE) [18, 45, 76] to retrieve a sound and complete model of its behavior. Then, it hands the model over to a three stage pipeline: (1) the Constraints Generator, which uses this model to analyze how the NF keeps its state and arrive at a sharding solution; then (2) the RSS configuration generator stage—for which we built a library called RS3—that uses a solver to find an RSS configuration that steers packets following the sharding rules found by the previous stage to the same core; and finally (3) the Code Generator, that generates a parallel implementation that configures the RSS accordingly and adds additional synchronization mechanisms if needed.

[1] Woo and Park's solution considers only a single RSS configuration, whereas our firewall deals with two ports (LAN and WAN), each requiring independent configurations. Although their findings are transposable to this scenario, it still requires expertise from the developers.

3.3 Extracting the NF's model

Maestro uses ESE to extract the complete NF's model. This allows us to not only analyze how the NF maintains its state, but also generate modified versions of its implementation.

The extracted model is an execution tree containing all the possible code execution paths a packet can trigger. Each node on this graph is either conditional (representing a branch condition), a stateful operation (representing a call to a stateful data structure, *e.g.* a map or a vector), or packet operation (*e.g.* forwarding, dropping, etc.). Both the packet and stateful data are traced as symbols, and every node contains a list of constraints on these symbols that can be given to a solver to query their possible values under any code path.

3.4 Finding the sharding solution

The NF model is passed to the Constraints Generator, which is tasked with finding a sharding solution that allows shared-nothing parallelization. The idea is to find the constraints that hold true between packets that access the same state, *i.e.* packets that must be processed on the same core. This is intrinsically tied to how the NF maintains state. For example, in a map for two operations to access the same state they must use the same *key*. By symbolically tracking how such keys are derived from packets, we reason about the constraints on packets that access common state.

Building a stateful report. The Constraints Generator starts by analyzing the NF's model and builds a stateful report (SR) of all the performed stateful operations. Each SR entry specifies the operation's name (*e.g.* `map_put`), object instance, and other relevant arguments (*e.g.* the key used), and all the possible constraints on both the received packet and other stateful data when the operation was performed (*e.g.* `map_put` was called when a UDP packet arrived from interface 0).

Filtering entries. After building the SR, the Constraints Generator removes all entries related to read-only objects (*e.g.* routing tables that are filled on start-up and never updated). Such read-only accesses to shared state do not require coordination among cores and need not be reasoned about. Should all accesses be read-only, the SR will be left empty and Maestro asks the Code Generator to generate a parallel implementation that uses RSS with the sole purpose of load-balancing traffic among cores (we explain the RSS mechanism in §3.5).

Analyzing the entries. The use of any data structure can potentially preclude a shared-nothing approach, and therefore we need to infer the conditions under which it is safe to perform stateful operations concurrently for each of them (or if no such conditions exist). We present the analysis for one of the most predominant data structures: the map [2, 50, 50, 76].

The map stores data indexed by a key. This data can be accessed via the function `map_get`, and modified with `map_put`. Two map calls access the same memory region if and only if they are given the same key. For a shared-nothing approach, packets that trigger map calls to the same instance using the same key need to be steered to the same core. This alone is,

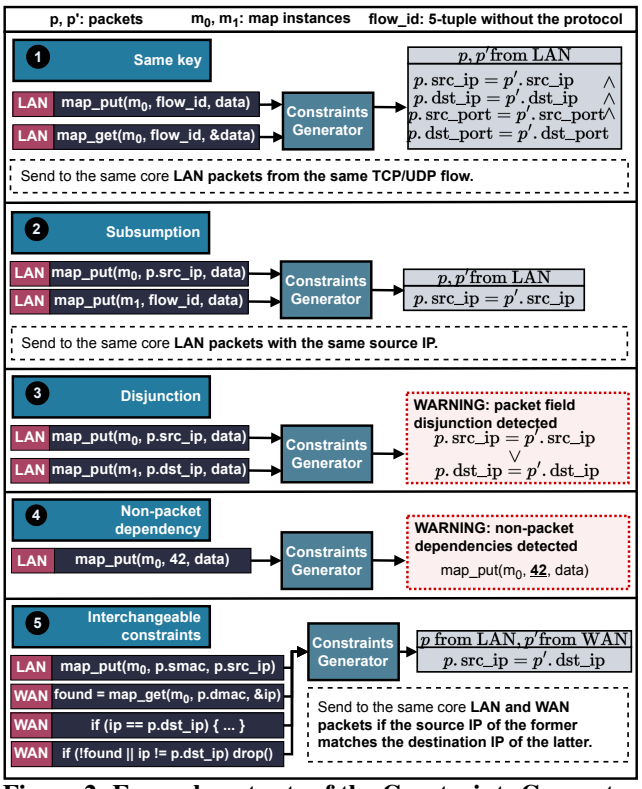


Figure 2: Example outputs of the Constraints Generator.

however, insufficient: we need to not only take into consideration any RSS limitations, but also reason about the use of multiple different map instances (or other data structures), each independently tied to the previous requirement. With this in mind, we designed a set of rules to guide Maestro towards finding correct shared-nothing sharding solutions:

- **R1** *Key equality.* The most obvious case is when two packets access the same map instance using the same key. In this case, the Constraints Generator builds the constraint from the formulas for the keys (❶ in Figure 2).
- **R2** *Subsumption.* If a map instance is accessed using a subset of the packet fields used to access a second instance, then the subset takes precedence over its larger counterpart. That is, the coarser-grained requirement wins over the finer-grained one. This is exemplified in scenario ❷ in Figure 2: sending packets with the same source address to the same core will also guarantee that packets with the same 5-tuple are also sent to the same core. More generally, we can always use a subset of the required packet fields. As we will see later, this rule can act further in concert with others to resolve incompatibilities.
- **R3** *Disjoint dependencies.* Accesses using disjoint sets of packet fields are problematic. An NF that keeps a pair of independent counters, one for source addresses and another for destination addresses, requires packets with the same source address *or* the same destination address to be sent to the same core. Due to limitations in the RSS mechanism, this is not possible: configuring it with

both the source and destination fields will guarantee that packets with the same source *and* destination will be sent to the same core. Maestro warns the user and provides the fundamental reason why the shared-nothing approach cannot be applied (❸ in Figure 2).

R4 *Incompatible dependencies.* RSS uses packet fields to steer packets to cores. This means that using keys containing (1) incompatible RSS packet fields or (2) no packet fields at all will completely block our attempt at correctly steering packets to cores. This is the case, for example, of NFs which index data with constant keys, as exemplified in case ❹ of Figure 2. Again, in this case, Maestro provides feedback to the user as to why the shared-nothing approach is unfeasible[2].

R5 *Interchangeable constraints.* We define a pair of constraints as *interchangeable* if they trigger the same NF behavior. This allows us to completely replace constraints matching rules R3 or R4 with others that, if interchangeable, do not prohibit shared-nothing parallelization.
Example ❺ of Figure 2 showcases this scenario. In this example, the packet is dropped when we fail to find the MAC address entry on the map, or whenever the incoming IP does not match with the stored address. Although the NF stores source addresses using an RSS-incompatible dependency on our NIC [39] (source MAC), the Constraints Generator finds that the NF's behavior is *exactly the same* whether we shard on the MAC address or the destination address. In this case, these constraints are interchangeable, which allows Maestro to shard on either of them. Because the former uses an incompatible RSS field, the Constraints Generator opts for using the latter one for sharding.
By sharding with the IP address, changing solely this field can cause the packet to be sent to a different core. Although it may still find a matching entry of its MAC address on the map, it will find a different IP address stored on that same entry, and hence the packet will be dropped. Both not finding the MAC entry and mismatching the IP value result in the same behavior from the NF.

These rules allow Maestro to correctly find sharding solutions for a wide range of NFs (as we show in §6). Note that only R1 is specific to data structures that use a key to index state (*e.g.* maps, vectors, sketches). R2, R3, R4, and R5 are otherwise data structure agnostic, and Maestro employs them to all entries, regardless of their specific data structure. Though much of this analysis focuses on maps, it can be used as building blocks for others. Moreover, we need only reason about these details *once* per data-structure (or, at most, each time a breaking change is made). Once data-structure developers encode such properties into Maestro, NF developers can freely use these stateful data structures to build their NFs.

[2]Maestro behaves in a similar manner when finding global counters updated by every packet, as it bars it from implementing a shared-nothing parallel solution.

Name	Description
map	Stores integers indexed by arbitrary data.
vector	Stores arbitrary data indexed by integers.
dchain	Time-aware integer allocator.
sketch	Count-min sketch [21].

Table 1: Stateful constructors currently supported by Maestro.

Table 1 shows the stateful constructors currently supported by Maestro.

Even when Maestro fails to find a shared-nothing solution, it still provides the developer the fundamental reason why (*e.g.* constant keys or non-packet dependencies). When met with this result, the developer is faced with a decision: either use this feedback to tweak the NF implementation so that it becomes amenable to shared-nothing parallelism, or request a lock-based implementation from Maestro.

Generating the constraints. The next step in the Maestro pipeline is to generate the actual constraints, *i.e.*, the conditions that, if satisfied by a pair of packets, dictate that they must be sent to the same core. Towards this end, Maestro iterates over each pair of report entries of the same state instances, creating SMT formulas stating that both keys must be equal, and joining them all together with logical *OR*s.

Finally, we note that RSS must be independently configured on each interface. As such, the constraints generated by Maestro are interface-specific, reasoning about pairs of packets which may arrive from separate interfaces. Case ❺ from Figure 2 exemplifies this. It requires LAN packets to be sent to the same core as packets from the WAN if the source address of the former equals the destination address of the latter.

Figure 3 shows the constraints found by the Constraint Generator when analyzing our firewall example. It finds that LAN packets with the same addresses and ports must be sent to the same core, and similarly for WAN packets. It also finds that WAN and LAN packets must be sent to the same core if they have the same, but swapped, sources and destinations.

3.5 Finding the right RSS configuration

The previous stage tackled the challenge of finding a shared-nothing sharding solution, producing constraints between packets that when true require the packets to be processed on the same core. We now focus on materializing this sharding solution by automatically finding RSS configurations that satisfy these constraints.

RSS is a hardware mechanism in the NIC that steers packets to core-specific queues. Once configured with an RSS key and a set of packet fields, it extracts from incoming packets the values of those fields and feeds them to a toeplitz-based hash-function [56]. This function, depicted in Figure 4, works by continuously left rotating the key k while iterating through the selected packet fields bits d. The running 32-bit hash value is *XOR*'ed with the current 32 least significant bits of the key whenever the current bit d_i is 1. The resulting hash is used to

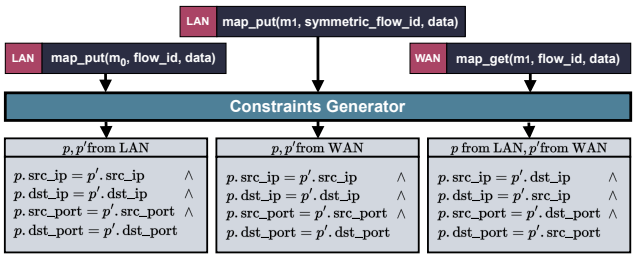


Figure 3: From the firewall's SR to its sharding constraints.

index an indirection table containing queue identifiers, and the packet is inserted in the corresponding queue.

Two packets with the same hash will be sent to the same core. Given the configurability of the RSS hashing function, we use it to ensure that packets that need to be processed on the same core will have the same hash. For simple constraints we can arrive at a satisfying RSS configuration solely by correctly choosing the packet field set (*e.g.*, hashing only source and destination IPs and ports when requiring TCP packets with the same 5-tuple to be sent to the same core). However, what if (1) the NF requires a subset of packet fields that can only be used as a group in the RSS mechanism (*e.g.*, a traffic monitor that shards solely the destination IP), (2) it requires complex constraints between packets (*e.g.*, a Hierarchical Heavy Hitter sharding on multiple subnets of the source IP and/or source ports), or (3) there are constraints between packets arriving in different interfaces (which is the case for many NFs requiring both LAN and WAN interfaces, as in NATs, Firewalls, Connection Limiters, *etc.*)?

To address these scenarios in a generalized way, we built RS3, a C library capable of taking constraints as inputs and outputting RSS configurations that satisfy them. It uses the Z3 solver [23] to find suitable configurations by encoding the problem in a logical format. Maestro uses RS3 to generate RSS configurations that satisfy the constraints given by the Constraints Generator module.

Building the statement. The query given to the solver needs to encode the following problem: *given set of constraints, find RSS keys that generate the same hash for every pair of packets that satisfy them*. To build this statement, we need to encode both the hash function and the constraints into an SMT format.

Let k be a 52 byte[3] RSS key, d and d' hash inputs for each of the packets (whose sizes depend on the extracted packet fields, *e.g.* 12 bytes for source and destination IPs and ports), and $h(k,d)$ the 32 bit hash. Also, let $|k| \geq |d|+|h|$, $H(k,k',d,d')$ be true iff $h(k,d) = h(k',d')$, and $C(d,d')$ be the constraint between d and d' provided by the constraint generator.

Hash function. As shown in Figure 4, $H(k,k',d,d')$ can be represented as:

$$\bigwedge_{b=0}^{|h|-1}\left[\bigoplus_{x=0}^{|d|}(d[x]\wedge k[x+b]) = \bigoplus_{y=0}^{|d'|}(d'[y]\wedge k'[y+b])\right] \quad (1)$$

[3] Value for the Intel E810 100G NIC [39], but trivially adjustable in RS3.

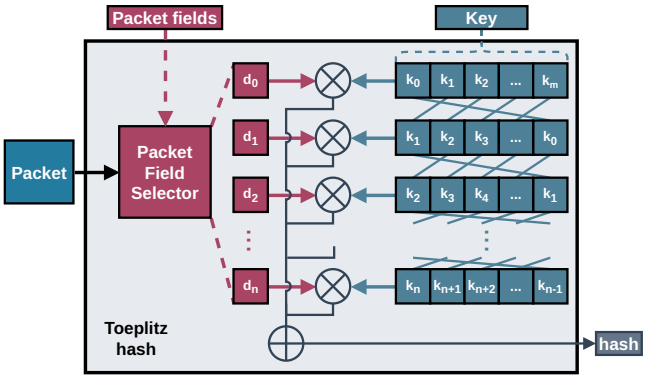


Figure 4: Toeplitz-based hash function.

Note that although the size of the key is lower bounded, it should not have any influence on the feasibility of finding a suitable hash configuration. Only a subset of its bits are used on the hash function, and therefore constrained by our requirements, with the other bits being free to take any value.

Base statement. Initially, let us encode the following query: *find a single key k such that, given any two hash inputs d and d′ that obey the constraints C, their corresponding hashes will always be equal.* That is:

$$\forall_{d,d'} \cdot k \neq 0 \wedge \left[(C(d,d') \wedge d \neq d') \rightarrow H(k,k,d,d')\right] \quad (2)$$

Having the key be 0 would always output 0 valued hashes, steering all packets to a single core, so we prevent the key from taking that value.

Compatibility with multiple keys. Each interface can have its RSS mechanism individually configured. With that in mind, let $C_{ij}(d,d')$ be the constraint between a pair of packets coming from ports i and j, configured with the keys k_i and k_j respectively. Note that $C_{ij} = C_{ji}$, therefore it is enough to consider, for example, all the constraints $C_{ij:\{j\leq i\}}$. For Equation (2) to be multi-key aware, we simply conjunct the constraints across all i and j, allowing the solver to manage each key combination problem as a specific statement that must be true. That is, for n ports:

$$\forall_{d,d'} \cdot \bigwedge_{i=1}^{n}\bigwedge_{j=1}^{i}\left[(C_{ij}(d,d') \wedge d \neq d') \rightarrow H(k_i,k_j,d,d')\right] \quad (3)$$

Compatibility with varying sets of RSS packet fields. Just as different ports may need distinct RSS keys, we may also need to configure RSS to use different sets of packet fields depending on the interface. One way to address this would be to consider hash inputs $d_0,\ldots,d_{n-1}$ for n interfaces. This, however, highly increases the complexity of the query, making it harder for the solver to find a solution[4]. Another way to look at it would be to extend the hash inputs to include the union of both field-sets and to deal with any unused bits. To make the statement in Equation (3) consider constraints between packets arriving at different ports with different RSS packet field options, we again add more clauses to our large

[4] For n interfaces, and thus considering $d_0,d'_0,\ldots,d_{n-1},d'_{n-1}$, with 96 bit hash inputs we would have to deal with $2\times 96\times n$ free bits.

conjunction, now considering all relevant RSS field sets, all while extracting for each one the required least significant bits of d and d' accordingly.

When given the constraints of our firewall, RS3 outputs two RSS keys, one for each NIC interface. The symmetry between the keys resembles the findings in [74], but generalized to two interfaces, rather than just one.

3.6 Code Generator

This stage takes the generated RSS configuration, as well as the NF's model, and outputs a parallel implementation of the original NF. Because the model is a sound and complete representation of the original NF, it can be used to generate an implementation identical in functionality to the original one. More importantly, it can be modified to employ shared-nothing parallelism by (1) configuring RSS, (2) allocating the state independently for each core, (3) making sure that each stateful call uses the data structures' instances of that particular core, and (4) launching the NF in multiple cores. Appendix A.1 contains adapted code excerpts from our Firewall example, showing both the sequential implementation used as input to Maestro and the final generated parallel shared-nothing implementation.

Parallel implementation with locking mechanisms. When Maestro rules out a shared-nothing solution, it can fall back to generating parallel implementations that use locking mechanisms. In this scenario, it configures RSS with both a random key and all the available RSS-compatible packet fields, as now all cores share the same state.

Maestro also needs to carefully coordinate access to shared data using read/write locks. As such, we distinguish read-packets from write-packets: the former trigger only stateful read operations, and the latter trigger at least one write. To efficiently handle this scenario, we created a custom, highly optimized read/write lock implementation that entirely avoids cache-line sharing when acquiring read locks. We do this with a series of per-core, cache-aligned, atomic spin-locks that indicate whether the core has permission to proceed. Acquiring a read lock requires just locking the current core's lock. To perform a write, however, one must lock all core-specific locks (in order, to avoid deadlocks). With this in place, we speculatively process all packets as read-only until they attempt to perform a write operation, at which point we stop processing, release the local lock, acquire all core-specific locks, and restart processing the packet from the beginning.

The performance toll is minimized when an NF is subjected to read-heavy workloads (see §6.4), as read-only packets need only acquire a core-specific cache-aligned lock, and have no need to atomically write to any shared variable, or write to shared data. As all write-packets start out as read-packets before backtracking, starvation is not an issue.

4 Implementation challenges

Finding good RSS keys. The first set of keys found by the solver is often not ideal. If, for example, the solver finds a key with all but the first bit set to zero, the hash, though semantically valid, will only ever be `0x0` or `0x80000000`. This leads to packets being sent to only two cores.

The solution employed by RS3 involves setting the value 1 to as many bits as possible in the keys, so long as they still satisfy the given statement. This is known as a Partial MAXSAT problem [19]. We give the solver a statement that its corresponding solutions should always satisfy—Equation (3), hard constraints—and also a set of clauses that they should try to satisfy—soft constraints. The soft constraints correspond to a chain of logical *ANDs* setting each key bit to 1. There is no need for maximizing the number of satisfied soft constraints. Most of the times, a randomly selected set of bits with the value 1 is enough to avoid corner case problems like the one mentioned above. As such, Maestro uses a slightly modified version of the diagnosis-based approach introduced by Fu and Malik [33]. It begins by seeding the key with random bits. Then, if the combined hard and soft constraints are not satisfiable, we get the UNSAT core from the solver and randomly discard a subset of these soft constraints, repeating as necessary until either a key is found or no further soft constraints are left, indicating that no such key exists. Due to the randomized nature of this algorithm, we use multiple parallel solvers to independently find keys until one is found with an acceptable workload distribution.

NUMA considerations. In a NUMA environment, each possible combination of NIC, memory, and CPU pinning influences throughput. Our machines (see §6) have 100 Gbps NICs with 2 interfaces, thus both interfaces are pinned to the same NUMA node. Under these circumstances, pinning the packet buffers to the same NUMA node as the NIC is optimal [29].

Another important consideration is that the dominant contention factor in parallel packet processing applications is the cache, specifically for Intel Data Direct I/O (DDIO) resources [25, 55]. Using DDIO, the packets coming from the NIC are directly placed in the last level cache (LLC) of the NUMA node. Contention happens when the number of concurrent packets exceeds the available reserved space for I/O in the LLC, at which point packets evict each other and performance suffers. Maestro allocates packet buffers close to the NIC, but keeps state local to each core's NUMA node. Deciding where to run each thread is, however, a deployment challenge, not an implementation one, and therefore out of scope for Maestro. Nevertheless, our experience has taught us a simple rule of thumb: if the LLC is large enough to hold all packet buffers at line-rate, then we should pin both the CPU and memory to the same NUMA node as the NIC. If, however, the LLC is too small, resulting in contention—as occurs with older processors—then it's better to distribute cores evenly across NUMA nodes, thus increasing the total available LLC. Though we have seen scenarios where using multiple NUMA nodes was best, in our testbed the LLC proved sufficiently large to justify using a single NUMA node, and all our experiments in this paper follow this guideline.

Traffic skew. The expression "mice and elephants" is typically used to describe packet flow distributions on the Internet [12, 36, 53]. These follow a Zipfian distribution, where a large fraction of packets relate to but a few flows, and the remaining ones share a small slice of traffic.

While traffic with a uniform distribution leads to packets being uniformly distributed to cores, traffic following a Zipfian distribution can overload a subset of cores, causing *skew*. This performance difference is shown in Figure 5, which demonstrates how the parallel firewall throughput varies with the traffic distribution. The Zipfian traffic was generated with parameters from [60], which were found by analyzing a real-world traffic sample from a University network in [12]. This generated traffic has 50k packets and 1k flows, 48 of which responsible for 80% of the traffic. RSS was configured with five different random keys and the error bars represent the min/max performance. Performance is influenced by both the RSS key and the indirection table, as more hash collisions cause more packets being sent to the same core. Under uniform traffic, the indirection table's entries are expected to be equally accessed, and thus uniformily filling it leads to evenly spreading packets across cores. With Zipfian traffic, however, the higher density of certain flows leads to more accesses to some entries, overloading some cores. Note that when using a single core we see better performance under Zipfian traffic due to an increased cache hit-rate when accessing state [60], though the effect is less prominent when more cores are used.

RSS++ [8] fixes the distribution problem imposed by Zipfian traffic by dynamically adjusting the indirection table according to the traffic. It balances the indirection table by swapping entries associated with overloaded cores for ones associated with underloaded ones. It also provides us with mechanisms for state migration across cores which avoid both blocking and packet reordering. We implemented static versions of these mechanisms in Maestro, but their dynamic versions could be used to handle changes in skew over time.

State sharding. When applying shared-nothing parallelization, Maestro not only allocates each data structure instance on each core, but further adjusts each data-structure's capacity, keeping approximately constant the total amount of memory used for all cores by reducing the per-core amount.

This raises an interesting question about the semantics of filling up state in a shared-nothing parallel version of an NF, which slightly differs from the sequential or lock-based parallel versions. As each core now has a reduced capacity, it is possible to exhaust the capacity of one core despite there being spare room in others. Ultimately, when a core becomes "full", it will behave in the same way locally as the sequential NF would globally (*e.g.* by dropping packets from new flows). As the RSS++ mechanism redistributes flows across cores to counteract traffic skew, this also affects state distribution, making it harder to exhaust any one core.

This state sharding has the desirable side-effect of optimizing the NF's cache utilization. If each core has a smaller working-set, more of it will fit in the local L1+L2 data caches. This provides an extra performance advantage to the shared-nothing approach on top of that of parallelization on its own.

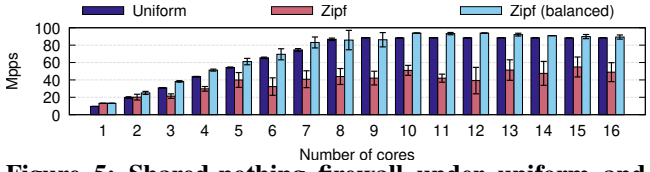


Figure 5: Shared-nothing firewall under uniform and Zipfian traffic, with and without balanced tables.

Lock-based rejuvenation. When following a read-write lock-based parallelization approach, flow rejuvenation can be a challenge. As simply reading state requires updating the flow entry aging data, a naive implementation would require a write lock for all packets, with dire consequences for performance. Maestro circumvents this issue by implementing an optimized rejuvenation algorithm that operates locally in each core for most cases. We first modify the data-structures to hold multiple cache-aligned copies of the entry aging data, one per core. Each core then manages state aging locally for each entry, allowing the age of the entries to deviate from core to core as packets from the same flow arrive at different cores at different times. When eventually one core believes it should expire an entry, only then does it acquire a write lock. At this point, the core inspects the aging data for that entry on all cores. If the flow indeed expired on all cores, it is cleared out globally. If, however, another core is found where the entry has not yet expired, the local timestamp is re-synced with the newest one. Ultimately, if packets from the same flow regularly hit all cores, no write-locks are ever needed.

Implementation. Maestro uses the KLEE symbolic execution engine, extending it with 14,859 lines of C++ code. We also implemented RS3 in 3,964 lines of C code, independently from Maestro [5].

5 Assumptions and limitations

NF limitations. To allow ESE, NFs must fit within some limitations, much like the ones enumerated in [44]: i) there must be a clean separation between stateful and stateless operations, a constraint put in practice by only allowing state to persist within a set of well-defined data structures; ii) loops must be statically bounded; and iii) no pointer arithmetic is allowed outside the data-structures. These constraints are already enforced for safety reasons in commonly used packet processing framework like eBPF[6] [3], a widely used framework in both academia and industry[7] [1, 10, 13, 17, 37, 52, 68].

RSS limitations. For Maestro to consider other hash function besides the standard toeplitz-based one, they would have to be formulated as an SMT problem and added to RS3. This requires having their implementation openly disclosed.

[5] Our code is openly available at [4].

[6] NFs developed in eBPF store their state in kernel-maintained maps [2].

[7] There is an ongoing effort in adapting Maestro to also accept eBPF NFs as input (an effort already set in motion by PIX [44]).

In practice, a more limiting factor is packet field selection: shared-nothing approaches can only be applied if state is sharded using RSS-compatible packet fields. DPDK's API [43] reference includes all possible field combinations that RSS can use (*e.g.* IPv4/IPv6 TCP/UDP flow tuples), but each NIC only implements a subset of them [39, 40].

Attacking state sharding. We mentioned earlier that it would be possible to "fill-up" a single core with fewer flows in a shared-nothing parallel NF than would otherwise be needed in the sequential or lock-based parallel versions. This could potentially be used as a DoS attack vector, reducing the cost for an attacker to block new flows from being admitted. RSS++ flow redistribution addresses this for well-behaved traffic, but an attacker can subvert this by specifically using flows that induce exact RSS hash collisions. Colliding flows end up on the same entry within the RSS indirection table and thus cannot be split apart.

Though out-of-scope for this paper, Maestro provides some defense from such attacks due to the randomization used to generate RSS keys. Even assuming the attacker has access to the NF source code and understands how it can be sharded across cores, different random RSS keys that comply with the sharding constraints will still distribute different flows in a different way. Without access to the actual key generated in RS3, the attacker would have a harder time reverse-engineering a set of co-located flows, mitigating their ability to induce the kind of persistent skew needed in a successful attack.

6 Evaluation

In this section, we evaluate Maestro and the three different types of parallel implementations it can generate: (1) *shared-nothing*, (2) *lock-based*, and (3) parallel solutions using *hardware transactional memory* [54] via the Intel's Restricted Transactional Memory interface [42]. We aim to answer four questions: (i) how long does it take Maestro to parallelize NFs? (ii) how well does the performance of these parallel implementations scale with the number of cores? (iii) what are the impacts on performance of the various parallelization strategies that Maestro can use? and (iv) how do Maestro's automatic parallel implementations fare against highly-optimized manually parallelized versions?

6.1 Target NFs and Microbenchmarks

To evaluate Maestro we analyzed 8 NFs—a simple forwarder (NOP), a policer, a bridge, a firewall (FW), a port scan detector (PSD), a NAT, a load-balancer (LB), and a connection limiter (CL). These are open-source NFs, most are non-trivial in complexity, and all have been used by a body of previous work [44, 45, 76][8]. In this section, we present a brief description of each, and show how Maestro parallelizes them[9]. For each NF, we measured how much time Maestro took to generate a parallel implementation (shared-nothing when possible, lock-based otherwise), summarizing the results in Figure 6.

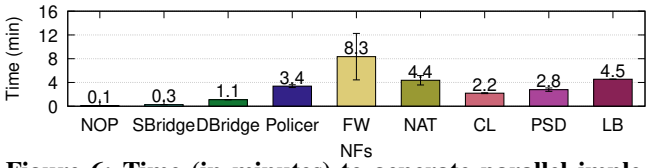


Figure 6: Time (in minutes) to generate parallel implementations for each NF (averaged over 10 runs).

NOP. This is a simple forwarding no-operation NF, *i.e.* a stateless NF that simply forwards all packets that arrive from one interface to the other. Maestro finds that this NF has no state, and provides no constraints between packets arriving at the same core. RSS is thus configured with all available packets fields and a random key on both ports.

Policer. This NF aims to limit each user's download rate, identifying users by their IPv4 address. When Maestro analyzes this NF, it finds that state is indexed by the destination IP address, implying that packets with the same destination address must be sent to the same core. Because this constraint uses the destination IP address, the chosen RSS packet field options must contain this field. Although DPDK allows RSS packet field options containing only IP addresses, our NICs do not support this option. Maestro thus chooses a packet field option that includes IP addresses and TCP/UDP ports. This increases the complexity of the constraints on the key, increasing the generation time in Figure 6.

Bridge. A bridge associates MAC addresses with interfaces, and redirects packets accordingly. In a typical MAC learning bridge, the association between source MAC addresses and input interface is learned dynamically. When analyzing this NF, Maestro detects that state is indexed by a MAC address, which is a field not supported by RSS on our NIC. As such, Maestro warns the user that it cannot generate a shared-nothing implementation, opting for read/write locks instead.

By modifying the NF to disable dynamic MAC learning, leaving only statically configured MAC-Port bindings, the NF becomes more amenable to parallelization (as all state is read-only), albeit with reduced functionality. This further illustrates the ability of Maestro to inform developers and help guide the development process by pointing out relevant trade-offs between functionality and performance. With this in mind, we created two versions of this NF: the standard bridge with dynamic MAC learning (DBridge) and a static one with fixed bindings (SBridge). When analyzing SBridge, Maestro encounters only read-only data structures, requiring no specific constraints on the RSS configuration. As with NOP, Maestro generates a random RSS key and uses all the available packet fields on all ports.

FW. This is the same firewall we have been using as a running example throughout the paper (§3.1). It indexes state with typical flow information on the LAN (source and destination addresses and ports), and symmetrically on the WAN. Maestro generates a shared-nothing implementation

[8] As mentioned in §5, the requirement that NFs be amenable to ESE can prevent Maestro from analyzing many existing codebases.

[9] Every automatically generated parallel solution can be found on [4].

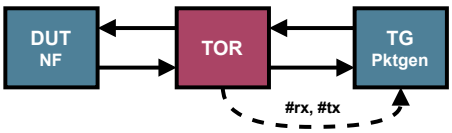


Figure 7: Testbed for our experiments.

that shards state by the flow information, sending WAN packets corresponding to symmetric LAN sessions to the same core as these (as shown in Figure 3).

PSD. A Port Scan Detector (PSD) counts how many distinct destination TCP/UDP ports each host (source IP) has touched within a given time frame. Above a threshold, connections to new ports are blocked, preventing port scans. Maestro analyzes the PSD and finds that it uses only the source IP to access one map, but also the source IP and destination port to access another. As such, the constraints for accessing the first map subsume those of the second (R2) and Maestro finds an RSS key that shards based only on source IPs.

NAT. A NAT translates addresses between a LAN and a WAN, allowing multiple clients in the LAN to share a single public IP in the WAN [70]. It keeps track of flows initiated in the LAN, but to aid with translation it associates a unique external port with each flow. Reply packets from the WAN are checked to see if their address and port match those on record before subsequently translating the destination address and port to match those of the client.

Maestro notices that the NAT associates flows with external ports using a map, fitting case R4 in §3.4. However, it also finds an additional constraint fitting case R5: packets from the WAN are only translated if they target the hosts that started the session in the first place. This constraint allows for sharding based on the external server's IP address and port.

Much like its sequential implementation, the parallelized NAT enforces unique ports inside each core. It does not, however, enforce this uniqueness *across* cores, a feature that does not break semantic equivalence. Whereas on the sequential implementation the allocated ports were used to distinguish between sessions, now the sharding solution allows for packets sent to different cores (hence pertaining to different external servers) to have the same allocated ports.

CL. A Connection Limiter (CL) aims to limit how many connections any single client (source IP) can make to any single server (destination IP) over a wider time frame (*e.g.* several days). Given the longer time frames involved, this NF uses a memory-efficient count-min sketch [21] to estimate the connection count from each client to each server. For new connections, the source and destination IPs are used to index the sketch, indexing a configurable number of entries based on different hashes (5 by default in our case). If all entries surpass the connection limit, the packet is dropped, preventing the new connection. Otherwise, each entry is incremented.

As with the PSD, Maestro finds two different access patterns: the 5-tuple indexes a connection tracking map, while the source and destination IPs index the sketch. Again, the latter constraint subsumes the former and Maestro shards based on source and destination IPs.

LB. LB is a Maglev-like load balancer [27]. Its main goal is to distribute traffic coming from the WAN to a series of identical servers on the LAN. LB registers new servers when it receives their packets coming from the LAN, and matches packets coming from the WAN with previously registered servers, keeping track of flows to ensure the same server handles packets from the same flow.

In order to maintain semantic equivalency between a shared-nothing parallel implementation and a sequential implementation, packets that find an available server in the sequential implementation must also find it available in the other. This ultimately means that all cores would need to have all backends registered in their local state. That said, packets coming in from the LAN in such a parallel implementation would only be able to be registered in a single core, preventing packets that arrive at other cores from seeing it. With this limitation in mind, it becomes impossible for multiple cores to hold an identical set of backend servers without coordination, thus preventing the use of a shared-nothing model. The Maestro analysis detects this issue when analyzing the LB SR. Lacking a better alternative, Maestro issues a warning and opts for a read/write lock based approach.

6.2 Performance Benchmarking Methodology

To benchmark the NFs, we use a standard testbed topology [16], connecting a traffic generator (TG) and a device under test (DUT), as shown in Figure 7. Both devices connect through a top-of-rack (TOR) switch from which we collect packet counters at the end of each experiment. Both TG and DUT are equipped with dual socket Intel Xeon Gold 6226R @ 2.90GHz, 96 GB of DRAM, and Intel E810 100 Gbps NICs. Turbo Boost, Hyper-Threading, and power saving features were disabled, as recommended by DPDK.

To measure throughput, the TG replays a given traffic sample (a PCAP file) in a loop at a given rate via the outbound cable for 10s per experiment. The DUT receives this traffic, processes it, and sends it back via the return cable, allowing the TG to measure latency. We further use the TOR to infer loss at the DUT, and—through comparison with the TG report—to also detect when packets were lost within the TG as well. We use DPDK-Pktgen [49] on the TG to find the maximum rate with less than 0.1% loss. We exclude and repeat sporadic experiment runs where loss within the TG—as opposed to the DUT—limited the results. When studying scalability, we repeatedly reevaluate the NF, while varying the number of cores it may use. We perform 10 measurements per experiment for statistical relevance and show error bars with min/max values. Our experiments properly handle NUMA considerations and indirection table rebalancing (§4).

6.3 Picking the Workload

In this section we analyze how different workloads impact performance, and ultimately establish the right workload con-

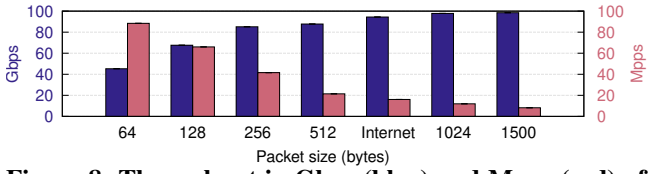


Figure 8: Throughput in Gbps (blue) and Mpps (red) of the parallel NOP running on 16 cores for different packet sizes.

figuration to evaluate all Maestro's parallelization solutions.

Packet size. To measure the impact of packet size on the performance of NFs, we ran the NOP on all cores and generated traffic with fixed-sized packets (40k uniformly distributed flows), varying the size on each iteration. The results (Figure 8) show that typical Internet traffic [12] and large packets easily achieve line-rate (100G), but that smaller packets struggle to keep up, reaching only ~45Gbps with 64B packets—even with such a trivial NF. Prior work [6, 57, 65] has pointed out that this bottleneck comes from PCIe 3.0 x16 and cannot be overcome without improved hardware. Unless stated otherwise, further experiments in this paper use 64B packets. As we measure more complex NFs that limit throughput below the 90Mpps shown in Figure 8, the bottleneck shifts from PCIe to the CPU, illustrating the NF's intrinsic performance.

Churn. The performance of parallel NFs can vary significantly for read or write workloads. In networking terms, this typically relates to *churn*, or the rate at which new flows are added and expired. This is particularly important for lock and TM based implementations, where creating new flows can lead to costly aborted transactions or exclusive write locks.

We start by studying these churn effects on performance by focusing on the read/write lock-based parallel firewall, and comparing it to its shared-nothing counterpart. To conduct churn experiments, ideally one would generate traffic live that changes flows periodically in an online manner. We found it challenging to generate such traffic programmatically at line-rate so we followed an alternative solution: generating PCAPs with different levels of *relative churn*—measured in flows/Gbit. As Pktgen varies the replay rate of the PCAP to probe the NF, the resulting *absolute churn*—measured in flows/minute or fpm—changes in tandem. This guarantees that our experiments converge to an equilibrium where the highest rate is found for the given churn. Once we find this rate, we can multiply the PCAP's relative churn with the experimental rate to compute the absolute churn.

With this in mind, we built PCAPs which (i) were small enough to fit in memory; (ii) changed enough flows to produce the desired relative churn; (iii) evenly spread these changes throughout the traffic; and (iv) were cyclic (*i.e.* the flows that expire at the start of the PCAP are created at the end). We then replay these files in a loop for 10s as in all other experiments.

Figure 9 shows how the FW—parallelized with different approaches—scales under varying amounts of churn. As absolute churn is computed based on the achieved rate, note that it too has error bars. Under low or no churn, the lock-based FW

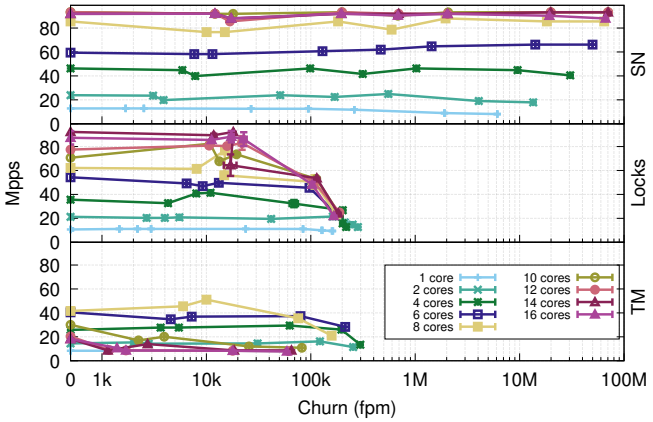


Figure 9: Churn study of the shared-nothing (top), lock-based (middle), and TM (bottom) parallel firewall.

scales well until bottlenecked by PCIe. At a churn of ~100k fpm we start observing the collapse of performance as the use of more cores just wastes more cycles busy-waiting under exclusive write locks. Under heavy churn, performance is abysmal as all cores end up contending for write locks. Note that the churn limit of an NF depends on the size of packets—Figure 9 uses 64B packets but for Internet traffic [12] the lock-based FW handles churn up to 400k fpm.

The results also show just how badly the FW parallelized with transactional memory handles churn. Although a useful tool in other domains, it proves ineffective when dealing with networked applications under churn.

The shared-nothing approach, unlike the lock-based one, suffers almost no performance variation with churn up to at least ~100M fpm, a great advantage over the lock-based implementation. Benson *et al.* [12] tell us to expect up to 6M fpm in typical data-center traffic—within the ability of our shared-nothing FW, but not the lock-based one. University networks—typically with less than 15k fpm—could easily be handled even by our lock-based FW.

We focus the rest of this evaluation on studies without churn, giving the lock and TM based approaches the benefit of the doubt and illustrating their *best-case* performance.

6.4 Performance benchmarks

With parallel versions of each of the above 8 NFs generated, we now evaluate their performance and scalability. By default, Maestro generates a shared-nothing implementation when possible, falling back to read/write locks otherwise. This choice can, however, be overriden, and Maestro can specifically generate parallel implementations using read/write locks and TM for any of the NFs, upon request.

Parallelization technologies. We now study the performance and scalability of each NF while being parallelized for each of the three approaches. As per §6.3, the workload used is composed of uniformly-distributed, read-heavy, small packets[10]. Figure 10 shows throughput as a function of the

[10]Experimental results using Zipfian traffic are shown in Appendix A.2

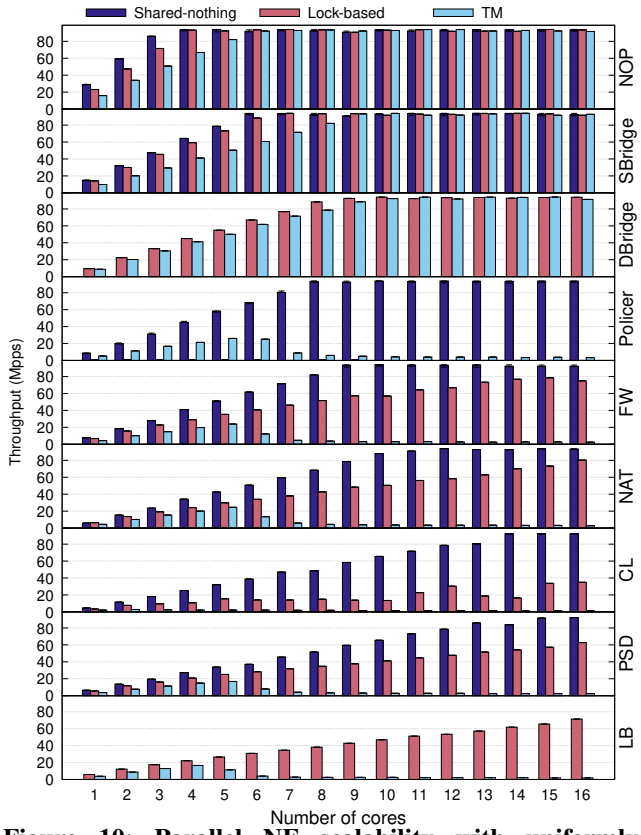


Figure 10: Parallel NF scalability with uniformly-distributed, read-heavy, small packets, using a shared-nothing approach when possible, read/write locks, and TM. Maestro cannot do a shared-nothing DBridge or LB.

number of cores. Our raw performance is comparable to measurements from other recent works [30], but we focus our attention on *scalability*. Though most NFs top out their performance before using all 16 cores due to bottlenecks in the PCIe bus or the memory controller, the takeaway here is the relative performance of the different approaches.

For all NFs where a shared-nothing approach was feasible, this option scales linearly until bottlenecked by the PCIe bus and then plateaus—an ideal outcome. The lock-based implementations—though slower than their shared-nothing counterparts when available—still scale fairly well but do not always reach the PCIe bottleneck with 16 cores[11]. The Policer shows what happens to these locks when writes are inevitable: as every packet must update the token bucket state, every packet requires an exclusive write lock, and performance suffers catastrophically. Fortunately, this NF can be sharded by IP address, so is amenable to the shared-nothing approach.

The benefits of state sharding (§4) become clear when we compare the shared-nothing approaches with the lock-based ones for the more state intensive NFs, *i.e.* the FW, NAT, CL, and PSD. When each core holds less state due to sharding, more of it fits in the core-local L1+L2 cache. In a shared-nothing approach where cores work independently on different working-sets this leads to an added performance improvement due to better caching, in addition to the benefits of parallelization. As a result, performance for few (< 4) cores can be worse than linear scalability would predict and using many cores can have an added boost in comparison. Running these experiments with a workload of only 256 flows—which fits entirely in L1 cache—nullifies this effect.

A surprising takeaway is that TM does not work well with the kinds of workloads found in more complex NFs, even in the absence of churn. For simpler NFs it performs quite well, scaling linearly with the number of cores, though still operating more slowly than both shared-nothing and lock-based alternatives. In these cases TM eventually catches up with the other approaches, albeit needing more cores to do so. However, for more complex NFs TM performs abysmally, as the likelihood of a transaction aborting increases.

Ultimately, the clear winner is the shared-nothing approach, with the best backup option consistently being our read/write locks. The PSD—our most CPU intensive NF which stands to gain the most from parallelization—performs 19× better with 16 cores than a single-core version, due to the *compound effects* of parallelization and improved cache efficiency.

Maestro does not deeply affect latency. We subjected all NFs to a 1Gbps uniform background traffic of 64B packets and collected 1000 latency probes within 10 seconds. We detected no noticeable differences on the average and tail latency values between the sequential NFs and their respective parallel implementations, regardless of the adopted parallelization strategy. Pktgen measured an average of $12 \pm 2\mu s$ for CL and $11 \pm 1\mu s$ for the remaining NFs.

VPP comparison. Finally, we compare Maestro with the Vector Packet Processing framework (VPP) [7, 31], which extends the concept of batch processing to the entire packet processing pipeline with the purpose of increasing performance by minimizing instruction cache misses. VPP follows a converse approach to Maestro: packets are processed in batches in a shared-memory parallel environment where packets can end-up on any core without regard to flows or locality. Developers must then adapt the way they implement the NF to those assumptions. This approach can require more expertise and development effort, but once NFs are built in this way the framework handles many of the low-level details.

To compare the performance of a Maestro parallelized NF with an expertly developed one for VPP, we pitch our NAT against the VPP `nat44-ei` with the DPDK plugin. These two NFs are the most similar we found between the VPP distribution and our corpus. We further removed a number of features from the VPP NAT to bring their implementations even closer together[12].

Figure 11 shows the performance comparison between

[11] Eventually, all lock-based NFs except for the Policer and CL can reach the PCIe bottleneck using extra cores from the remote NUMA node.

[12] We removed statistical counters, disabled IPv4 checksum checking, completely removed the IPv4 reassembly feature, and finally replaced the IPv4 lookup with static forwarding.

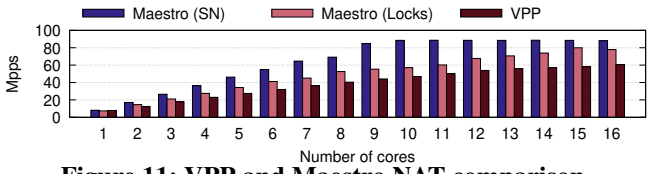


Figure 11: VPP and Maestro NAT comparison.

the parallel Maestro NAT (shared-nothing and lock-based) and `nat44-ei`, all under uniformly distributed 64B packets. Though all approaches scale well, Maestro's shared-nothing decisively outperforms VPP, reaching the PCIe bottleneck with 10 cores. This is due to the shared-memory design that VPP follows. A fairer comparison would be between VPP and the lock-based Maestro NAT, as both use shared-memory. Here both scale more slowly, never fully reaching the PCIe bottleneck up to 16 cores. Maestro slightly outperforms VPP. Further investigation with the `perf` [62] tool showed us that although the Maestro lock-based NAT and the VPP one perform very similar numbers of memory reads and writes per packet, the Maestro NAT more frequently finds the data on L1 cache (Maestro's 55% vs. VPP's 46%) and has to access RAM less frequently (Maestro's 3% vs. VPP's 4%). The key takeaway though, is that Maestro's *automatically* parallelized NFs perform competitively with expertly developed, manually parallelized NFs, without as much of a hassle.

7 Related Work

Fast packet processing. To address the performance challenges associated with software NFs, new packet I/O frameworks were proposed [3, 15, 24, 64]. To achieve high packet processing rates these solutions explore several types of optimizations including zero-copy, kernel bypass, I/O batching, and multi-queue support [9]. VPP [7] even expands batching to the whole packet processing pipeline in order to reduce instruction cache misses. Most implementations of network functions today [28, 71, 75, 77], including those from Maestro, rely on Intel DPDK [38], a kernel-bypass packet processing framework that provides a set of software libraries and drivers for fast packet processing, providing multi-core and NUMA-aware functionalities.

NF acceleration. PacketMill [30] accelerates NFs by carefully managing packet metadata and performing code-optimizations across the whole network stack. Another approach to improve the performance of a software NF is to leverage the platform hardware. Previous work [26, 46, 59, 71, 73] has explored multi-core CPU architectures, showing the significant improvements they can achieve on an NF's performance, but also the challenges involved. Papadogiannaki *et al.* [59], for instance, explored the advantages of a shared-nothing model over a lock based implementation. The goal of Maestro is to offer the advantages of parallelization to NFs, for free. Although their work focused on the most efficient utilization of available resources, we use their shared-nothing model as guidance for automated generation of parallel network functions. These solutions are *manual*, requiring extensive expertise and painstaking effort from the developer.

De Carli *et al.* [22] proposed a concurrency model for software IDSes that uses program analysis to infer the NF's flow semantics, feeding that information to a software scheduler that steers packets to shared-nothing threads. Though the concepts share similarities, Maestro's approach differs from theirs by (1) considering a wider class of NFs more generally, rather than IDSes in particular; (2) using ESE to extract fine-grained state access patterns, as opposed to their less granular program-slicing approach; and (3) handling packet steering entirely in hardware by generating RSS configurations for NICs, avoiding the bottleneck of the software scheduler and allowing Maestro parallelized NFs to scale better.

Flow steering. Although some NICs support rich flow-steering configurable features [39, 58], these are orthogonal to RSS and do not replace it. Using them to assure semantic equivalence on a shared-nothing implementation may require frequently adding/deleting a large amount of rules (specially under high churn), which can heavily affect performance [47].

NF verification and synthesis. In recent years, verification techniques have started to be applied to network functions. Some of the most relevant work includes verification of network properties [48, 51], configurations [11, 32], and NFs [76]. More recently, the research community has started exploring synthesis approaches for SDN-based control [20], data plane programs [34, 61, 78], and BGP configurations [14, 66]. Our work fits into this line, by analyzing sequential NFs to automatically generate accelerated versions.

8 Conclusions

In this paper we presented Maestro, a tool to automatically parallelize sequential network functions. Maestro judiciously configures the NIC's RSS mechanism to distribute traffic across cores, while preserving semantics, resorting to locking mechanisms only when necessary. Maestro significantly improved performance for all the NFs we analyzed—scaling-up performance linearly until hitting fundamental bottlenecks in PCIe, the memory controller, or line-rate—while reducing developer effort to the push of a button.

Acknowledgments

We are grateful to our shepherd, Tom Barbette, and the anonymous NSDI'24 reviewers. We thank Hugo Sadok for his comments on earlier drafts of the paper. We also thank Paolo Romano and Daniel Castro for their help on handling the TM approaches. This work was supported by the European Union (ACES project, 101093126), INESC-ID (via UIDB/50021/2020), and the SALAD-Nets CMU-Portugal/FCT project (2022.15622.CMU). Francisco Pereira was supported by the FCT scholarship PRT/BD/152195/2021.

References

[1] Cilium project, 2023. https://cilium.io/.

[2] eBPF maps, 2023. https://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf_maps.html.

[3] Express Data Path, 2023. https://en.wikipedia.org/wiki/Express_Data_Path.

[4] Maestro source code, 2023. https://github.com/snaplab-dpss/maestro/tree/nsdi24.

[5] Maestro's test suit, 2023. https://github.com/snaplab-dpss/maestro-eval/tree/nsdi24.

[6] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. Understanding host interconnect congestion. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, HotNets '22, page 198–204, New York, NY, USA, 2022. Association for Computing Machinery.

[7] David Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. High-Speed Software Data Plane via Vectorized Packet Processing. *IEEE Communications Magazine*, 56(12):97–103, 2018.

[8] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. RSS++: Load and State-Aware Receive Side Scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast Userspace Packet Processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, page 5–16, USA, 2015. IEEE Computer Society.

[10] David Beckett, Jaco Joubert, and Simon Horman. ACM SIGCOMM 2018 Morning Tutorial on Host Dataplane Acceleration (HDA). *New York, NY, USA*, 2018. https://conferences.sigcomm.org/sigcomm/2018/tutorial-hda.html.

[11] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 155–168, New York, NY, USA, 2017. Association for Computing Machinery.

[12] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, page 267–280, New York, NY, USA, 2010. Association for Computing Machinery.

[13] Gilberto Bertin. XDP in practice: integrating XDP into our DDoS mitigation pipeline. In *Technical Conference on Linux Networking, Netdev*, volume 2, pages 1–5. The NetDev Society, 2017.

[14] Rudiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. Config2Spec: Mining network specifications from network configurations. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 969–984, Santa Clara, CA, February 2020. USENIX Association.

[15] Nicola Bonelli, Stefano Giordano, and Gregorio Procissi. Network Traffic Processing With PFQ. *IEEE Journal on Selected Areas in Communications*, 34(6):1819–1833, 2016.

[16] S. Bradner and J. McQuaid. Benchmarking Methodology for Network Interconnect Devices. RFC 2544, RFC Editor, 03 1999. https://tools.ietf.org/rfc/rfc2544.txt.

[17] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient software packet processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990. USENIX Association, November 2020.

[18] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[19] Byungki Cha, Kazuo Iwama, Yahiko Kambayashi, and Shuichi Miyazaki. Local search algorithms for partial maxsat. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, page 263–268. AAAI Press, 1997.

[20] Haoxian Chen, Anduo Wang, and Boon Thau Loo. Towards example-guided network synthesis. In *Proceedings of the 2nd Asia-Pacific Workshop on Networking*, APNet '18, page 65–71, New York, NY, USA, 2018. Association for Computing Machinery.

[21] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[22] Lorenzo De Carli, Robin Sommer, and Somesh Jha. Beyond pattern matching: A concurrency model for stateful deep packet inspection. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 1378–1390, New York, NY, USA, 2014. Association for Computing Machinery.

[23] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[24] Luca Deri. Improving Passive Packet Capture : Beyond Device Polling. *Proceedings of SANE*, 2004:85, 2004.

[25] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 11, USA, 2012. USENIX Association.

[26] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 15–28, New York, NY, USA, 2009. Association for Computing Machinery.

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

[28] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, page 275–287, New York, NY, USA, 2015. Association for Computing Machinery.

[29] Paul Emmerich, Maximilian Pudelko, Simon Bauer, and Georg Carle. User Space Network Drivers. In *Proceedings of the Applied Networking Research Workshop*, ANRW '18, page 91–93, New York, NY, USA, 2018. Association for Computing Machinery.

[30] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. PacketMill: Toward per-Core 100-Gbps Networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 1–17, New York, NY, USA, 2021. Association for Computing Machinery.

[31] FD.io. Vector Packet Processing - One Terabit Software Router on Intel Xeon Scalable Processor Family Server. Technical report, 2017. https://fd.io/docs/whitepapers/FDioVPPwhitepaperJuly2017.pdf.

[32] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A General Approach to Network Configuration Analysis. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 469–483, USA, 2015. USENIX Association.

[33] Zhaohui Fu and Sharad Malik. On Solving the Partial MAX-SAT Problem. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 252–265, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[34] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch Code Generation Using Program Synthesis. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 44–61, New York, NY, USA, 2020. Association for Computing Machinery.

[35] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 163–174, New York, NY, USA, 2014. Association for Computing Machinery.

[36] Liang Guo and I. Matta. The war between mice and elephants. In *Proceedings Ninth International Conference on Network Protocols. ICNP 2001*, pages 180–188, 2001.

[37] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and*

Technologies, CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery.

[38] Intel. Data Plane Development Kit, 2010. https://www.dpdk.org.

[39] Intel. *Intel® Ethernet Controller E810 Datasheet*, 10 2022. Version 2.4. https://www.intel.com/content/www/us/en/content-details/613875/intel-ethernet-controller-e810-datasheet.html.

[40] Intel. *Intel® Ethernet Controller X710/XXV710/XL710 Datasheet*, 6 2022. Version 4.1. https://www.intel.com/content/www/us/en/content-details/332464/intel-ethernet-controller-x710-xxv710-xl710-datasheet.html.

[41] Intel. Intel data-direct I/O technology, 2023. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html.

[42] Intel. Restricted Transactional Memory Overview, 2023. https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/restricted-transactional-memory-overview.html.

[43] Intel. RSS compatible packet fields on the DPDK RSS API, 2023. https://github.com/DPDK/dpdk/blob/4fceceed5b5e9fbf04acffd66239c79d81e79260/lib/ethdev/rte_ethdev.h#L572.

[44] Rishabh Iyer, Katerina Argyraki, and George Candea. Performance Interfaces for Network Functions. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 567–584, Renton, WA, April 2022. USENIX Association.

[45] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. Performance Contracts for Software Network Functions. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 517–530, USA, 2019. USENIX Association.

[46] Muhammad Jamshed, Jihyung Lee, Sangwoo Moon, Deokjin Kim, Sungryoul Lee, and Kyoungsoo Park. Kargus: A Highly-scalable Software-based Intrusion Detection System Categories and Subject Descriptors. *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 317–328, 2012.

[47] Georgios P. Katsikas, Tom Barbette, Marco Chiesa, Dejan Kostić, and Gerald Q. Maguire. What You Need to Know About (Smart) Network Interface Cards. In Oliver Hohlfeld, Andra Lutu, and Dave Levin, editors, *Passive and Active Measurement*, pages 319–336, Cham, 2021. Springer International Publishing.

[48] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 9, USA, 2012. USENIX Association.

[49] Keith Wiles. Pktgen - Traffic Generator powered by DPDK, 2023. https://github.com/pktgen/Pktgen-DPDK.

[50] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 239–253, USA, 2016. USENIX Association.

[51] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying Network-Wide Invariants in Real Time. volume 42, page 467–472, New York, NY, USA, sep 2012. Association for Computing Machinery.

[52] Marios Kogias, Rishabh Iyer, and Edouard Bugnion. Bypassing the Load Balancer without Regrets. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 193–207, New York, NY, USA, 2020. Association for Computing Machinery.

[53] Kun-chan Lan and John Heidemann. A Measurement Study of Correlations of Internet Flow Characteristics. *Comput. Netw.*, 50(1):46–62, jan 2006.

[54] James R Larus and Ravi Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 1(1):1–226, 2007.

[55] Antonis Manousis, Rahul Anand Sharma, Vyas Sekar, and Justine Sherry. Contention-Aware Performance Prediction For Virtualized Network Functions. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 270–282, New York, NY, USA, 2020. Association for Computing Machinery.

[56] Microsoft Inc. RSS Hashing Functions, 2023. https://learn.microsoft.com/en-us/windows-hardware/drivers/network/rss-hashing-functions.

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

of the ACM Special Interest Group on Data Communication, SIGCOMM '18, page 327–341, New York, NY, USA, 2018. Association for Computing Machinery.

[58] NVIDIA. Mellanox ConnectX-5 Ethernet Adapter Card, 2020. https://network.nvidia.com/files/doc-2020/pb-connectx-5-en-card.pdf.

[59] Eva Papadogiannaki, Lazaros Koromilas, Giorgos Vasiliadis, and Sotiris Ioannidis. Efficient Software Packet Processing on Heterogeneous and Asymmetric Hardware Architectures. *IEEE/ACM Transactions on Networking*, 25(3):1593–1606, 2017.

[60] Luis Pedrosa, Rishabh Iyer, Arseniy Zaostrovnykh, Jonas Fietz, and Katerina Argyraki. Automated Synthesis of Adversarial Workloads for Network Functions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 372–385, New York, NY, USA, 2018. Association for Computing Machinery.

[61] Francisco Pereira, Gonçalo Matos, Hugo Sadok, Daehyeok Kim, Ruben Martins, Justine Sherry, Fernando M. V. Ramos, and Luis Pedrosa. Automatic Generation of Network Function Accelerators Using Component-Based Synthesis. In *Proceedings of the Symposium on SDN Research*, SOSR '22, page 89–97, New York, NY, USA, 2022. Association for Computing Machinery.

[62] Perf. `perf`: Linux profiling with performance counters, 2023. https://perf.wiki.kernel.org.

[63] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, page 227–240, USA, 2013. USENIX Association.

[64] Luigi Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 9, USA, 2012. USENIX Association.

[65] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. Ensō: A Streaming Interface for NIC-Application Communication. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 1005–1025, Boston, MA, July 2023. USENIX Association.

[66] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. Snowcap: Synthesizing Network-Wide Configuration Updates. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 33–49, New York, NY, USA, 2021. Association for Computing Machinery.

[67] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. Rollback-recovery for middleboxes. volume 45, page 227–240, New York, NY, USA, aug 2015. Association for Computing Machinery.

[68] Nikita Shirokov and Ranjeeth Dasineni. Open-sourcing Katran, a scalable network load balancer, 2018. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/.

[69] Vishal Shrivastav. Stateful Multi-Pipelined Programmable Switches. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 663–676, New York, NY, USA, 2022. Association for Computing Machinery.

[70] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022, RFC Editor, 01 2001. https://www.rfc-editor.org/rfc/rfc3022.

[71] Martino Trevisan, Alessandro Finamore, Marco Mellia, Maurizio Munafo, and Dario Rossi. Traffic Analysis with Off-the-Shelf Hardware: Challenges and Lessons Learned. *Comm. Mag.*, 55(3):163–169, mar 2017.

[72] Amy Viviano. Introduction to Receive Side Scaling, 2023. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling.

[73] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 87–98, New York, NY, USA, 2013. Association for Computing Machinery.

[74] Shinae Woo and Kyoungsoo Park. Scalable TCP session monitoring with symmetric receive-side scaling. *KAIST, Daejeon, Korea, Tech. Rep*, 144, 2012.

[75] Xiaoban Wu, Peilong Li, Yongyi Ran, and Yan Luo. Network measurement for 100 GbE network links using multicore processors. *Future Generation Computer Systems*, 79:180–189, 2018.

[76] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. Verifying Software Network Functions with No Verification Expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19,

page 275–290, New York, NY, USA, 2019. Association for Computing Machinery.

[77] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. A Formally Verified NAT. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 141–154, New York, NY, USA, 2017. Association for Computing Machinery.

[78] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. Gallium: Automated Software Middlebox Offloading to Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 283–295, New York, NY, USA, 2020. Association for Computing Machinery.

A Appendix

A.1 Code excerpts from Maestro

We present here the pseudo-code of the firewall NF used throughout the paper, both its sequential and parallel shared-nothing implementations. These serve to provide a sense of what the Maestro pipeline both accepts as input (Figure 12) and automatically generates as output (Figure 13). As such, we reiterate that these are *not* complete examples, but only pseudo-code, as they were shortened and simplified for clarity purposes. The complete solutions can be found on our public GitHub repository [4].

Notice the symmetry of the RSS hashes (lines 7 to 25 in Figure 13), as it is what ultimately enables its shared-nothing approach. As explained in §6.1, this symmetry allows packets coming from the WAN to be sent to the same core as their corresponding symmetric packets from the LAN.

A.2 Macrobenchmarks with Zipfian traffic

While in Figure 10 we show how throughput varies for different parallelization techniques under uniform traffic, here we repeat the experiment with Zipfian traffic instead [12] (we describe this Zipfian traffic in §4). We balanced the indirection table for each implementation to better handle the skew, as described in §4. The results are shown in Figure 14.

```
01: struct Map *flows;
02:
03: #define LAN 0
04: #define WAN 1
05:
06: // Run once.
07: int init() {
08:     map_init(flows, 65536);
09: }
10:
11: // Run by each packet.
12: void process(int port, pkt_t pkt) {
13:   if (port != WAN) {
14:     flow_t flow = {
15:       pkt.src_ip, pkt.dst_ip,
16:       pkt.src_port, pkt.dst_port
17:     };
18:
19:     map_put(flows, flow, port);
20:     forward(WAN);
21:   } else {
22:     flow_t inv_flow = {
23:       pkt.dst_ip, pkt.src_ip,
24:       pkt.dst_port, pkt.src_port
25:     };
26:
27:     int out_port;
28:     bool found = map_get(
29:       flows, inv_flow, &out_port);
30:
31:     if (!found) {
32:         drop();
33:     } else {
34:         forward(out_port);
35:     }
36:   }
37: }
```

Figure 12: Pseudo-code of the sequential firewall used as an example throughout the paper.

```
01: // One map for each thread.
02: struct Map** flows;
03:
04: #define LAN 0
05: #define WAN 1
06:
07: uint8_t RSS_HASH_PORT_0[52] = {
08:   0xa1, 0x24, 0x00, 0x15, 0x00, 0x14, 0xa1, 0x24,
09:   0xa1, 0x24, 0x00, 0x14, 0xa1, 0x24, 0x00, 0x15,
10:   0xa7, 0xfa, 0x11, 0x22, 0x6f, 0xd3, 0xf0, 0x42,
11:   0x1b, 0x6c, 0xeb, 0x14, 0x62, 0x02, 0xa3, 0x44,
12:   0x24, 0x90, 0xf8, 0x1c, 0x43, 0x99, 0xe7, 0xaf,
13:   0x80, 0x73, 0x15, 0xfe, 0x29, 0x5a, 0x73, 0xd0,
14:   0x55, 0x85, 0xf2, 0xc4
15: };
16:
17: uint8_t RSS_HASH_PORT_1[52] = {
18:   0x00, 0x14, 0xa1, 0x24, 0xa1, 0x24, 0x00, 0x15,
19:   0x00, 0x14, 0xa1, 0x24, 0x00, 0x14, 0xa1, 0x24,
20:   0x6a, 0xe3, 0xac, 0x86, 0x3e, 0xcb, 0x7e, 0x73,
21:   0x83, 0x15, 0xcb, 0x75, 0xc4, 0x73, 0x2c, 0xda,
22:   0xdb, 0x05, 0x31, 0x46, 0xdb, 0xd4, 0x76, 0x5a,
23:   0xa8, 0x20, 0x9d, 0x0a, 0x44, 0x7a, 0xc6, 0xae,
24:   0x5d, 0x72, 0x34, 0x9c
25: };
26:
27: // Run by each worker thread.
28: int init() {
29:   unsigned core_id = rte_lcore_id();
30:
31:   if (core_id == rte_get_main_lcore()) {
32:     rss_configure(
33:       LAN, RSS_HASH_PORT_0, IP_TCP | IP_UDP);
34:     rss_configure(
35:       WAN, RSS_HASH_PORT_1, IP_TCP | IP_UDP);
36:   }
37:
38:   map_init(flows[core_id], 65536);
39: }
40:
41: // Run by each packet on a specific worker thread.
42: void process(int port, pkt_t pkt) {
43:   unsigned core_id = rte_lcore_id();
44:
45:   if (port != WAN) {
46:     flow_t flow = {
47:       pkt.src_ip, pkt.dst_ip,
48:       pkt.src_port, pkt.dst_port
49:     };
50:
51:     map_put(flows[core_id], flow, port);
52:     forward(WAN);
53:   } else {
54:     flow_t inv_flow = {
55:       pkt.dst_ip, pkt.src_ip,
56:       pkt.dst_port, pkt.src_port
57:     };
58:
59:     int out_port;
60:     bool found = map_get(
61:       flows[core_id], inv_flow, &out_port);
62:
63:     if (!found) {
64:       drop();
65:     } else {
66:       forward(out_port);
67:     }
68:   }
69: }
```

Figure 13: Pseudo-code of the firewall, but now parallelized by Maestro with a shared-nothing architecture (and described in §6.1).

The key takeaways are the same as in Figure 10: when available the shared-nothing approach is always preferred; the lock-based solutions frequently do not scale as well as their

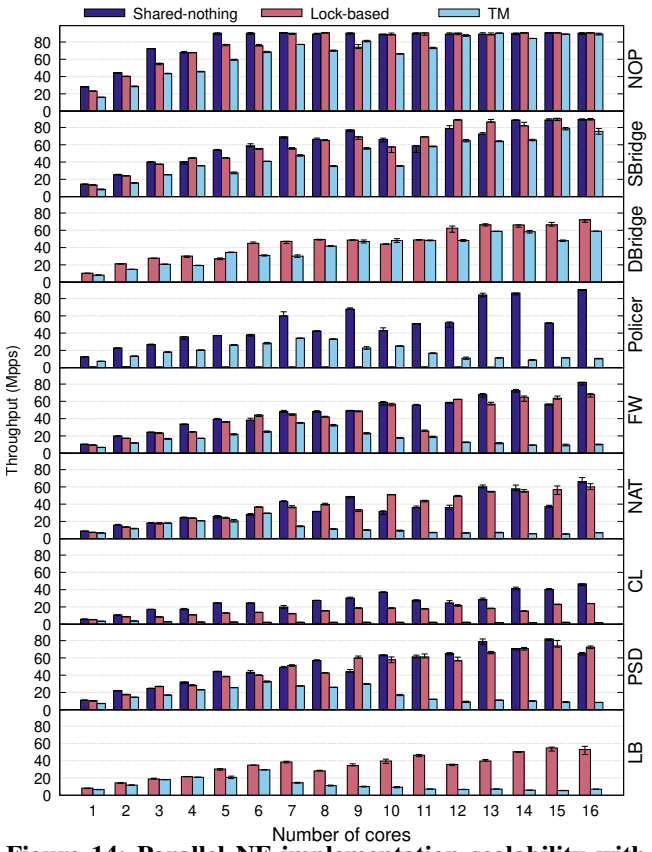


Figure 14: Parallel NF implementation scalability with Zipfian, read-heavy, small packet traffic, using a shared-nothing approach when possible, read/write locks, and TM.

shared-nothing alternatives and suffer in more state-intensive NFs; and TM-based approaches perform unreliably.

We do, however, find differences between these results and their counterparts under uniform traffic. Although under uniform traffic it is rather clear that throughput scales up with the number of cores when using the shared-nothing approach, with Zipfian traffic this scaling is not always consistently monotonic. This is to be expected, as the efficacy of balancing load across cores may not consistently improve when more cores are added. Indeed, when many cores are used, a single elephant flow can bottleneck a single core, limiting the maximum throughput we will be able to achieve in our experimental setup. This is particularly limiting for computationally and state intensive NFs—such as the the Connection Limiter—which are unable to perform as well with Zipfian traffic as they do with uniform. These results nevertheless confirm that Maestro generated NFs almost always perform as well with Zipfian traffic as they do with uniform.

A.3 Reproducibility

We make Maestro's code publicly available in [4]. In that repository, one can find not only the source code for the entire pipeline, but also the complete set of NFs we mention on this paper, along with their corresponding parallel solutions found by Maestro and described in §6.1.

We also make available our test suit in [5]. It contains all the required scripts to generate Figures 5, 8 to 11 and 14. They were tested on 2 machines with dual socket Intel Xeon Gold 6226R @ 2.90GHz, 96 GB of DRAM, and e810 Intel NICs [39], and running Ubuntu 22.04.

AutoSketch: Automatic Sketch-Oriented Compiler for Query-driven Network Telemetry

Haifeng Sun[1] *Qun Huang*[1] *Jinbo Sun*[2] *Wei Wang*[3] *Jiaheng Li*[1]
Fuliang Li[3] *Yungang Bao*[2] *Xin Yao*[4] *Gong Zhang*[4]
[1]*National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University*
[2]*Institute of Computing Technology, CAS* [3]*Northeastern University, China* [4]*Huawei Theory Department*

Abstract

Recent network telemetry witnesses tremendous progress in two directions: query-driven telemetry that targets expressiveness as the primary goal, and sketch-based algorithms that address resource-accuracy trade-offs. In this paper, we propose AutoSketch that aims to integrate the advantages of both classes. In a nutshell, AutoSketch automatically compiles high-level operators into sketch instances that can be readily deployed with low resource usage and incur limited accuracy drop. However, there remains a gap between the expressiveness of high-level operators and the underlying realization of sketch algorithms. AutoSketch bridges this gap in three aspects. First, AutoSketch extends its interface derived from existing query-driven telemetry such that users can specify the desired telemetry accuracy. The specified accuracy intent will be utilized to guide the compiling procedure. Second, AutoSketch leverages various techniques, such as syntax analysis and performance estimation, to construct efficient sketch instances. Finally, AutoSketch automatically searches for the most suitable parameter configurations that fulfill the accuracy intent with minimum resource usage. Our experiments demonstrate that AutoSketch can achieve high expressiveness, high accuracy, and low resource usage compared to state-of-the-art telemetry solutions.

1 Introduction

The continuous growth of modern data centers inspires tremendous progress on network telemetry, which forms the basis of various network management tasks, such as anomaly detection [32, 43, 99], performance analysis [74, 78, 92, 100], and fault diagnosis [5, 13, 14, 33, 44, 93, 106]. Recent advances in network telemetry can be categorized into two classes: *query-driven telemetry* and *telemetry algorithms*. Query-driven telemetry [30, 66, 67, 97] considers *expressiveness* as the primary goal. They provide extensive operators for users to build various telemetry tasks, and then automatically deploy the operators into switches. Therefore, users do not need to concern with the implementation details in switch hardware. However, the resource-efficient realization of operators is not well addressed in existing query-driven telemetry solutions. In particular, many operators maintain flow-level *states*. As the number of interested flows increases, maintaining the states easily exhausts switch memory [30] or network bandwidth [67].

The other class is telemetry algorithms. Their common idea is to leverage approximate techniques (e.g., sampling [73, 76, 77], top-k counting [6, 79], and sketch [1, 34, 36, 54, 55, 94, 103]) that sacrifice a small portion of accuracy to relax resource requirements. Even though recent studies show that such approximate techniques can achieve both resource efficiency and bounded errors, it remains non-trivial to utilize them. First, each approximate technique is designed for several specific telemetry applications and fails to support all tasks. Second, an approximate technique needs careful parameter configuration to simultaneously reach high accuracy and resource efficiency. Thus, for each telemetry task, users need to select the appropriate algorithms, carefully realize them, and tune parameters in switches, which needs domain knowledge on the programming models and hardware restrictions.

In this paper, we present AutoSketch that integrates the advantages of both query-driven telemetry and sketch algorithms. Its goal is to fully exploit the resource efficiency of telemetry algorithms while hiding the complicated details of implementing and configuring them in network devices. This is extremely significant because (i) users lack the domain knowledge of underlying telemetry algorithms and architecture; and (ii) the hardware resources of existing network devices are limited. On the one hand, AutoSketch follows recent operator-based telemetry languages. It defines extensive interfaces for users to build telemetry applications with both *built-in* operators and *user-defined* operators, which achieves high expressiveness. On the other hand, AutoSketch automatically converts these operators to sketch algorithms, which achieves high accuracy and low resource usage.

However, it is challenging to bridge the gap between the high-level operators and the sketch algorithms. First, sketch

*Qun Huang is the corresponding author.

algorithms incur errors due to their approximation nature (although the errors are bounded). Users should be capable of perceiving and controlling the incurred errors for telemetry applications. Second, user-defined operators complicate the conversion to sketch. In particular, many user-defined operators maintain numerous states, which are hard to realize as sketch. For some operators, there is even no ready-made sketch algorithm to realize them. Finally, sketch algorithms expose various parameters to configure, which needs to take underlying hardware into account.

AutoSketch addresses the challenges in three aspects:

- First, AutoSketch extends its operator-based interface to allow users to specify the desired accuracy as an *accuracy intent* (§2). Then, AutoSketch employs a *compiler* that automatically generates sketch-based solutions that fulfill the intent (§3). The compiler hides the details on switch hardware and algorithm details. To this end, administrators can control the extent of approximation without concerns with underlying realization.
- Second, AutoSketch constructs efficient sketch instances for the stateful operators (§4). For built-in operators, AutoSketch realizes them by classical sketch algorithms. For user-defined operators, AutoSketch decomposes each of them into several simple operators with the aid of syntax analysis. Then, AutoSketch designs *sketch-like structures* to instantiate the decomposed operators. Each sketch-like structure follows the same idea of sketch that maps traffic into different cells, but employs different methods to aggregate cells (depending on the user-defined functions). The sketch-like structures provide theoretical guarantees to estimate the impact of hash conflicts like classical sketch.
- Finally, AutoSketch formulates the parameter tuning into an optimization problem, which reveals the hardness of parameter tuning (§5). We propose a searching algorithm to find the *most suitable* configuration that fulfills the accuracy intent and incurs minimum resource usage. The algorithm employs various optimization techniques and eliminates the user burdens of tuning sketch parameters.

We build a prototype of AutoSketch that targets PISA [10] switches and compare AutoSketch with existing query-driven telemetry systems and sketch-based telemetry algorithms. We show that AutoSketch can implement telemetry applications in less than 20 lines of code without extra efforts to tune configurations. Under 200 K active flows per 100 ms, AutoSketch still reaches 99% recall, 97% precision, and 2.8% errors with only 0.84 MB switch memory and 84 KB/s bandwidth on average for 11 applications. We release our source code at **https://github.com/N2-Sys/AutoSketch**.

2 AutoSketch Interface

AutoSketch follows recent telemetry languages (e.g., Sonata [30] and Marple [67])that provide operator-based interface. It abstracts network traffic as a stream of *tuples*, each of which consists of a *key* and several *fields*. An operator performs specific computations on each input tuple and then generates some output tuples if needed. A telemetry application is defined as an operator graph in which vertices are operators. A (directed) edge indicates that the output tuples of the upstream operator are consumed by the downstream operator.

Operator	Description
filter(*bool_expr*)	Check the boolean expression *bool_expr* for each tuple and preserve tuples satisfying conditions.
map(*fields*, [*expr*])	Transform each input tuple into an output tuple consisting of *fields*, whose values are set by *expr*.
distinct(*fields*)	Categorize input tuples based on *fields*, and preserve one tuple for each category (i.e., delete duplicated tuples with the same field values).
reduce(*fields*, *val*)	Categorize tuples according to *fields* and sum up *val* for each category.
zip(*s*, *field*)	Merge the input stream with tuples of another stream *s* containing the same field values in *fields*.
groupby(*states*, *udf*)	Invoke user-defined function *udf* to update *states*.

Table 1: Operators of AutoSketch Interface

Operators. Table 1 summarizes the operators supported by AutoSketch. Currently, AutoSketch provides five *built-in* operators: `filter`, `distinct`, `map`, `reduce`, and `zip`. Among them, `distinct` and `reduce` are *stateful*, i.e., maintaining internal states for tuple processing. Here, an operator state refers to a key-value pair associating with tuples that have the same key. To achieve more expressiveness, AutoSketch further provides a `groupby` operator that allows users to customize the processing logic in two aspects. First, a `groupby` operator can maintain user-defined *states*. Second, each `groupby` operator associates with a *user-defined function* (`UDF`) to define the processing logics on the tuples and states. AutoSketch supports three types of statements in a `UDF`: (1) assignment statements, (2) arithmetic statements, and (3) conditional statements.

Accuracy intents. AutoSketch realizes stateful operators (i.e., `distinct`, `reduce`, and `groupby` with states) using sketch techniques, which inevitably incurs errors. To characterize the errors, AutoSketch provides users with *accuracy intent* interface, which can specify user-desired accuracy. In the context of operator-based processing, an accuracy intent consists of four accuracy metrics:

- *Recall*: the ratio of true output tuples to all true tuples that should be output.
- *Precision*: the ratio of true output tuples to all output tuples.
- *Average relative error (ARE)*: the mean relative error of all output tuples.
- *Confidence*: the probability that all other accuracy metrics (e.g., recall, precision) are met.

Users can selectively use the accuracy metrics for a telemetry application. For a metric, users can specify the minimal/maximal value or the desired range for it (e.g., `ARE_max` = 1%, `recall` = 95% ± 5%). If a metric is not specified, AutoSketch considers that there is no requirement on it. Note that the intents are set for the entire application instead of individual operators. Thus, users do not need to concern with how to set

```
1  def nonmt(tcp.seq):
2    if maxseq < tcp.seq or maxseq == 0:
3      maxseq = tcp.seq
4    else:
5      nm_count += 1
6
7  nm[precision_min=99%,ARE_max=1%,confidence=95%]=
8    PacketStream()
9    .filter(ipv4.protocol==TCP)
10   .groupby({5tuple:(maxseq, nm_count)}, nonmt)
```

Application 1: TCP non-monotonic detection

per-operator errors to form the application-level accuracy. AutoSketch automatically handles the application-level intents by configuring operator-level parameters.

Example. Application 1 presents an example that monitors non-monotonic sequence numbers of TCP flows. The application first selects TCP packets by a `filter` operator (line 9). Then it uses the `groupby` operator to define a `UDF`, namely `nonmt` (line 10). The `groupby` operator maintains two states. The `maxseq` state tracks the maximal sequence number of each flow (line 3). The `nm_count` state counts the number of packets whose sequence numbers are lower than `maxseq` for each flow (line 5). The intent requires that with a probability of 95%, the monitored flows achieve 99% precision, and per-flow ARE is below 1% (line 7).

Generality. The operators in AutoSketch come from recent query-driven telemetry systems [30, 67, 105]. They cover a rich set of telemetry applications according to these studies. We do not claim the novelty of this design. However, to the best of our knowledge, we are the first to define accuracy intent interface that (1) empowers automatic control operator errors introduced by sketch and (2) hides the underlying implementation and configuration details of sketch. Some prior works (e.g., BlinkDB [3]) also define accuracy intent but they control sampling rate of input traffic instead of operator behaviors. On the other hand, some efforts [19, 45, 105] also attempt to apply sketch algorithms to query-driven telemetry, but lack the accuracy intent interface, leaving the error configuration to the users.

Limitations. The programming model has three limitations due to the resource constraints of commodity devices. First, AutoSketch cannot support complicated statements, such as loops and floating-point arithmetic, because commodity switch ASICs only allow limited recirculation and such statements are too expensive [26]. Second, the states in an operator must have limited dependency: users cannot define two states whose processing depends on each other (details in §4.1). Finally, the accuracy intent should not consume more resources than hardware capacity. When a telemetry application compromises any of these restrictions, AutoSketch throws an error as existing data-plane compilers [70]. We will support more compiling modes in future work, such as suggesting accuracy intents and returning multiple configurations within accuracy ranges for users to select from.

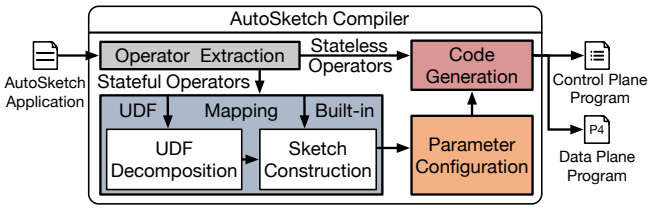


Figure 1: AutoSketch Compiler.

3 AutoSketch Compiler

AutoSketch designs a general compiler in order to bridge the gap between our interface and the underlying hardware. The compiler takes a telemetry application composed of several operators and an accuracy intent as input. It converts the application to a P4 program that runs in the data plane to record per-packet information, and a control plane program that retrieves telemetry results and reports the results to users.

Figure 1 depicts the compiler internals. For an input application, the compiler first extracts its operators. Then it maps the stateful operators into sketch instances. For built-in stateful operators, the compiler directly constructs corresponding sketch instances. For `groupby` operators with complicated `UDFs`, the compiler first decomposes each `UDF` into several simple ones, and then performs the construction. After sketch mapping, AutoSketch configures parameters for the sketch instances to satisfy the accuracy intent. Finally, AutoSketch assembles output programs by integrating the converted sketch instances with the stateless operators.

Although sketch mapping and parameter configuration have been studied in prior works (§3.1 and §3.2), it is non-trivial to combine them because we need to fulfill the application-level accuracy intent. Specifically, when an application is mapped to numerous combinations of different sketch algorithms, it remains an open issue to configure operator-level sketch parameters to satisfy application-level accuracy intent [58].

3.1 Mapping for Stateful Operators

AutoSketch constructs a sketch instance for each stateful operator instead of using a single universal sketch for the entire application. The reason is that existing universal sketch algorithms are also composed of multiple basic structures [36, 50, 55], whose overall resource usage is nontrivial and hard to be optimized. In contrast, per-operator sketch mapping allows more fine-grained sketch selection and tuning.

Specifically, the compiler employs different strategies for built-in operators and `groupby` operators. For built-in stateful operators, since their functionalities are common, the compiler realizes them with well-known sketch algorithms, e.g., Count-Min Sketch (CM) [20] and Bloom Filter (BF) [8].

For `groupby` operators, AutoSketch follows the idea of sketch to construct sketch-like structures. As shown in Figure 2, a sketch-like structure comprises multiple rows of cells, where each cell maintains the state values of the `groupby` operator. To process a tuple, we hash its key to one cell in

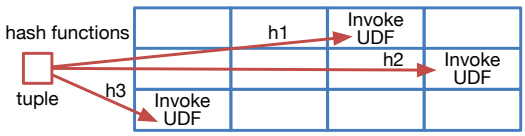


Figure 2: Sketch-like structure in AutoSketch.

each row, and then invoke the UDF to update the states in the hashed cells. When querying for the key, we mitigate the errors caused by hash conflicts. Specifically, we use the same hash functions to select cells, and aggregate the cells based on the linearity or monotonicity of the state. We use the aggregated results to estimate the state value. We also provide theoretical analysis for the sketch-like structures. Thus, compared to existing sketch algorithms, our sketch-like structures support more types of states and user-defined processing logics, while preserving bounded telemetry errors. Note that we choose two-dimensional (2D) matrices for the sketch-like structures because their parameters are simpler and fewer than that of higher-dimensional matrices, which incur more switch resources to handle the additional dimensions.

Challenges. The mapping is non-trivial for both built-in and groupby operators. For each type of built-in operator, there could be multiple possible sketch algorithms that can realize its functionalities. (e.g., reduce operator can be mapped to Count-Min Sketch [20] or Count Sketch [12].) The algorithms achieve different levels of resource-accuracy trade-offs. We need to choose suitable algorithms to meet application accuracy intents. For groupby operators, the challenge comes from the complexity of UDFs. If we map a complicated UDF to a single sketch instance entirely, every cell includes all operator states. In this case, the incurred computational resources are overwhelming. In particular, a UDF may contain multiple state updates. However, existing switch ASICs only support limited per-cell operations. For example, PISA switches only allow one read-and-modify operation for each memory address and a small number of conditional judgments (i.e., the boolean expression in the if statement) [89].

Solutions (§4). For built-in operators, AutoSketch estimates the efficiency of all combinations of candidate sketch algorithms. Then it chooses the one incurring minimum resource usage while reaching the desired accuracy. For UDFs, AutoSketch decomposes every complicated UDF via abstract syntax trees, a common technique for syntax analysis. Each decomposed UDF contains exactly one state. Thus, AutoSketch can construct sketch instances for decomposed UDFs with limited resource usage, and tune the parameters for each state individually. Although Newton [105] also maps operators into sketches, it does not allow UDFs. Note that AutoSketch is not binding to several specific sketches or sketch-like structures. It is expected for AutoSketch to support newly proposed sketches [87, 101], by (i) adding new sketch candidates for existing stateful operators or (ii) introducing new operators (e.g., top operator). Note that the actual errors are varying depending on specific sketches used for operators.

3.2 Parameter Configuration

AutoSketch automatically tunes the parameters of the constructed sketch instances. The parameter configuration targets both high accuracy and resource efficiency. That is, AutoSketch aims to use minimum resources for achieving a specified accuracy intent.

Challenges. Although existing sketch algorithms provide theoretical analysis to characterize the relationship between parameters and accuracy, it remains challenging to tune parameters for AutoSketch in two aspects. First, as in recent studies [21, 36, 57], the theoretical analysis of existing sketch algorithms typically addresses worst-case scenarios, thereby providing limited guidelines for parameter tuning. Second, a telemetry application usually comprises multiple sketch instances. Existing works on merging sketch [2, 48] only apply to multiple sketch instances of the same type. Thus, it remains challenging to integrate the errors of multiple heterogeneous sketch instances to form the overall accuracy of an application, because the errors are quite different across sketch algorithms. For example, some algorithms produce false positives [8], while others introduce a relative error for each flow [20]. It is hard to characterize the errors in a uniform method.

Solutions (§5). The difficulty of integrating the errors of individual sketch instances motivates us to employ a benchmark-based parameter tuning. Specifically, AutoSketch formulates the parameter configuration into an optimization problem. Then it searches all possible configurations to find out the most efficient configuration. For each configuration, AutoSketch estimates its overall accuracy via benchmark experiments. AutoSketch proposes an efficient searching algorithm with several techniques to reduce the searching overhead. Note that our optimization tunes parameters of each operator to satisfy application-level accuracy intent, while existing configuration methods only focus on planning for placement and refinement [30] or configuration for single data structure [31]. Thus, their optimization objectives are all different.

Discussion. AutoSketch adopts static configuration instead of dynamic adjusting [63, 64] for two reasons. First, dynamic approaches require *precisely* estimating the resulting accuracy after adjusting. This is infeasible since existing theoretical analysis typically works for worst scenarios [21, 36, 57]. Second, recent studies on dynamic reconfiguration are not adopted by commodity switches yet [60, 102]. Thus, we leave this issue in future work.

4 Mapping to Sketch

4.1 Operator Decomposition

AutoSketch exploits Abstract Syntax Tree (AST) to decompose a UDF. An AST is a tree that characterizes the syntax structure of a specific function. AutoSketch utilizes AST to perform decomposition in three steps as follows.

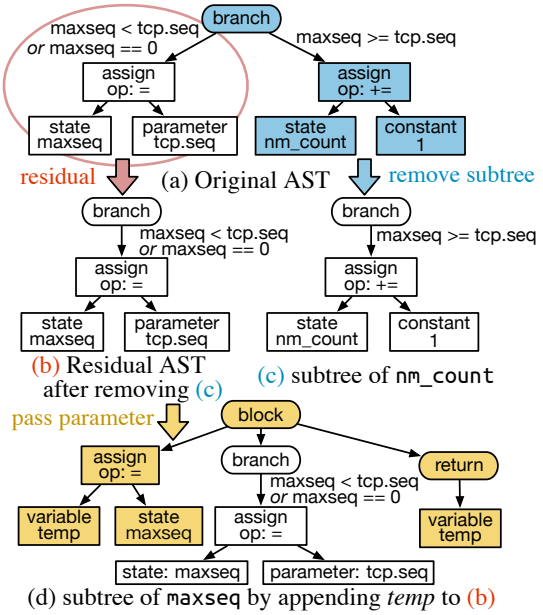


Figure 3: Example of nonmt (Application 1) decomposition.

Step 1: build AST. We first build an AST for the UDF. In an AST, a leaf node represents a state, a parameter, a constant, or a temporary variable. Non-leaf nodes correspond to statements that manipulate their child nodes. Recall that there are three types of statements: assignment statements, arithmetic statements, and conditional statements (§2). For an *assignment* or *arithmetic* node, its children refer to the operands. A *branch* node corresponds to a conditional statement. Its edges to the child nodes are associated with the judgment expressions. The first child refers to the If branch, while the second child indicates the Else branch (if any). For better organization, our AST further introduces a *block* node, which indicates that its children are executed sequentially. Figure 3(a) shows the AST of UDF nonmt in Application 1. The AST contains a branch node as the root. The two children are the assignment nodes for maxseq and nm_count derived from the If and Else branches, respectively. Due to space limitations, we omit the decomposition for maxseq < tcp.seq or maxseq == 0 but the procedure is similar.

Step 2: partition AST. Next, AutoSketch partitions the AST into several small ASTs, such that there is exactly one state being modified in each partitioned AST. Algorithm 1 illustrates the partitioning of an AST. Our basic idea is to repeatedly remove subtrees containing state modifications. Specifically, we enumerate each state and travel the AST (line 3). For each assignment node that modifies the state (lines 5-6), we build a subtree st comprising nodes from the root to this assignment node including its child nodes (line 7). Then, we remove nodes in the subtree from the original AST. The removal travels from the root to the assignment node (line 9). If a node does not appear in other root-to-leaf paths (line 10), we remove this node and all its descendant nodes (lines 11-12). Otherwise, for a node appearing in other root-to-leaf paths,

Algorithm 1 AST Partition

```
Input : The original AST of UDF
 1: function PARTITIONAST(ast)
 2:     partitioned_asts = []
 3:     for each state s maintained by UDF do
 4:         ST = []
 5:         for each assignment node u in ast do
 6:             if u modifies s then
 7:                 Build a subtree st from root to u (with u's children)
 8:                 Add st to ST
 9:                 for each node v from root to u do
10:                     if v does not appear in other root-to-leaf paths then
11:                         Remove v and its descendants from ast
12:                         break
13:         Construct a new_ast by merging the subtrees in ST
14:         Add new_ast to partitioned_asts
15:     return partitioned_asts
```

we keep the node in the AST, such that the connectivity of other paths is preserved. Note that there could be multiple statements modifying a state. Thus, we can partition multiple subtrees from the original AST for one state. In this case, we form a larger AST for this state by merging the duplicate nodes in the multiple subtrees (line 14).

Figure 3(b) and (c) illustrate the partitioning procedure for the AST of nonmt. We start with partitioning modifications on state nm_count. We identify the subtree modifying nm_count as shown in Figure 3(c) (also marked in blue in Figure 3(a)). In the residual AST in Figure 3(b), there is only one state maxseq, and thereby we have no need to remove subtree.

Step 3: pass parameters. Finally, AutoSketch passes state values among the partitioned ASTs because a partitioned AST may need to read state values from another AST. Since there exist multiple versions of the state values, we only pass the needed versions with temporary variables. In Figure 3, updating state nm_count depends on the value of state maxseq. Thus, we extend the AST of maxseq in Figure 3(b) to a new AST in Figure 3(d). Since the value of maxseq before updating is actually needed, we add an assignment statement that saves the value before updating in a variable *temp*. The variable is returned and passed to AST of nm_count after all statements are executed. We put more examples in Appendix.

Discussion. AutoSketch restricts the dependency among states in a groupby operator. For two states A and B, if updating A needs to access B, we say that A depends on B. In this case, the decomposition result must be unique: the stage that deploys partitioned AST of B must be before that of A. Thus, AutoSketch does not allow *mutual dependency*. By mutual dependency, we mean that the two states depend on each other. When mutual dependency occurs, AutoSketch cannot determine which state should be placed on the preceding stage. Then, it throws an error during compilation time.

4.2 Sketch Construction for UDFs

AutoSketch constructs a sketch-like structure for each decomposed UDF. As shown in §3.1, it maps each key into one cell

Algorithm 2 Update procedure of sketch-like structure

```
Input : The packet tuple t
Variables: Sketch-like structures C = map()
Variables: Bloom filter bf, Keyarray = [], l = 0
Variables: Incflags = map(), Decflags = map()        ▷ Initial value true
 1: function UPDATE(t)
 2:     Extract key k from t
 3:     if k ∉ bf then
 4:         Insert k to bf
 5:         if Keyarray is not full then
 6:             Keyarray[l] = k
 7:             l += 1
 8:         else
 9:             Send k to controller
10:     for each UDF in decomposed UDFs do
11:         SL = C[UDF]                  ▷ SL is the sketch-like structure of UDF
12:         for i = 1 to d do
13:             j = h_i(k)
14:             old_value = value of SL[i][j]
15:             Invoke UDF(SL[i][j], t)
16:             new_value = value of SL[i][j]
17:             if old_value > new_value then
18:                 Incflags[UDF]=false          ▷ non-increasing
19:             else if old_value < new_value then
20:                 Decflags[UDF]=false          ▷ non-decreasing
```

with hash functions. Then, AutoSketch updates the mapped cells with the UDF, or queries per-key information by aggregating the mapped cells. We elaborate on the detailed realizations of the update and query procedures as follows.

Update procedure. Recall that a state in a groupby operator is a collection of key-value pairs. Thus, in addition to updating *values* in the sketch-like structures, it also needs to store the keys. One possible solution is to store the keys within each sketch-like structure. However, this incurs overwhelming memory overheads, because there are duplicate keys in the multiple decomposed UDFs. Thus, AutoSketch separates the keys outside the sketch-like structures.

Algorithm 2 details the update procedure for the multiple decomposed UDFs. To separate keys from sketch values, it maintains an array Keyarray to store keys, a counter l indicating the length of Keyarray, and a Bloom Filter bf to check the existence of keys. Upon the arrival of a tuple, we extract its key k and check the occurrence of the key with bf (lines 2-3). If the key never appears, we insert the key into the bf and append the new key to the end of Keyarray (lines 4-7). If Keyarray is full, we send k to controllers (line 9). Then we update the sketch-like structures. Specifically, for each UDF, we hash the key to locate the cells containing the state of the key (line 13). We invoke the function to update the state (line 15). We maintain Decflags and Incflags for each UDF. The two variables indicate whether state values are monotonically decreasing or increasing, respectively (lines 17-20). The two variables will assist the query procedure in the next.

Query procedure. The query procedure is performed in the control plane. The controller periodically collects data plane information, including the sketch-like structures, the Keyarray, and the two variables indicating the monotonicity of the UDF. Then, the controller enumerates the keys in

Algorithm 3 Query procedure of sketch-like structure

```
Input: The key k, The monotonicity flag Decflag, Incflag
1: function QUERY(k, Decflag, Incflag)
2:     rets = []
3:     for i = 1 to d do
4:         SL = C[UDF]              ▷ SL is the sketch-like structure of UDF
5:         j = h_i(k)
6:         r = SL[i][j]
7:         Add value of r to rets
8:     if Decflag then                    ▷ Monotonically decreasing
9:         return the maximal value in rets
10:    else if Incflag then               ▷ Monotonically increasing
11:        return the minimum value in rets
12:    else                               ▷ No monotonicity
13:        return the median value in rets
```

Keyarray to query their state values in the sketch-like structures (recall that a sketch-like structure corresponds to one state in the original UDF). Specifically, AutoSketch hashes a key to obtain the mapped cells in a sketch-like structure. Then it aggregates the cells to estimate the state value.

AutoSketch mitigates the errors caused by hash conflicts by leveraging the monotonicity of a UDF.

- Monotonically increasing: AutoSketch takes the minimum value among the mapped cells as the estimate, since all cells overestimate the true value of a key.
- Monotonically decreasing: AutoSketch takes the maximum value among the cells, similarly.
- No monotonicity: AutoSketch uses the median of the cells. The rationale comes from the Chernoff Bound [61]. Specifically, for each flow, the occurrences of overestimates and underestimates from the multiple cells are expected to be the same. Thus, using the median yields an accurate result.

Algorithm 3 summarizes the query procedure. The function takes key k and the monotonicity variables of the UDF as input. It hashes k into $SL[i][j]$ in the i-th row with $j = h_i(x)$ and records the value of state $SL[i][j]$ in rets (lines 2-7). Then it uses the monotonicity flags Decflag and Incflag to decide which value in rets to return (lines 8-13). We put the theoretical analysis of sketch-like structures in Appendix C.

Control plane overheads. Although the decomposition increases the number of stateful operators, it does not amplify the query overheads in the control plane. The reason is that the controller only needs to query the operators that form the eventual results. For example, in many applications that follow "Count-Distinct with Threshold" [18], only the last operator is of interest. Since the actually queried operators are more lightweight and much traffic is pruned in previous operators, we do not introduce additional control plane overheads.

4.3 Sketch Selection for Built-in Operators

AutoSketch supports multiple types of sketch algorithms to instantiate built-in operators, so as to adapt to diverse applications and accuracy intents. We leverage a *sampling* technique to select the most efficient algorithms for a given application and accuracy intent. Specifically, we evaluate different com-

binations of sketch algorithms, and select the combination that uses minimum resource usage to fulfill the accuracy. The evaluation uses synthetic workloads to perform benchmark analysis (see §5), so that we can estimate the actual resource-accuracy trade-off of the sketch algorithms. For each built-in operator, we generate several configurations for each potential algorithm that can realize this operator. We not only calculate the theoretical configuration suggested in the literature, but also randomly sample some other configurations.

Note that the sampling only addresses a limited number of configurations, which is acceptable for algorithm selection. Thus, we still need comprehensive configurations in §5. We put more optimizations for sketch mapping in Appendix D.

5 Searching for Configurations

5.1 Problem Formulation

AutoSketch aims to find out a configuration that satisfies input accuracy intent but incurs minimum resource usage. We state the problem as follows (the complete problem formulation is in Appendix E).

Application configuration. We define an application configuration c as a vector of $3n$ variables, $c = (d_1, w_1, s_1, d_2, w_2, s_2, ..., d_n, w_n, s_n)$, given that n is the number of mapped sketch instances. Each variable in the vector is a parameter that needs to be configured. For the i-th sketch instance, the two parameters d_i and w_i indicate the depth and width of the 2D sketch structure, respectively. s_i is the number of stages occupied by the i-th instance, which limits the maximal available resources for each sketch. Here, we omit the size of each element, because it is a constant determined while sketch-based mapping.

Constraints. A configuration needs to satisfy two types of constraints. The first type is about accuracy. It requires that the accuracy of the telemetry application should reach the user-specified accuracy intent. The other type of constraint ensures that the resource usage (stage, ALU, and memory) of a configuration should not exceed the available ones provided by PISA switches. Further, since ALUs and memory are associated with each stage, we also require that a configuration does not compromise per-stage resource restrictions.

Objective. The objective of parameter tuning is to find the configuration that satisfies all the constraints (including accuracy constraints and resource constraints) while incurring minimal resource usage. Currently, AutoSketch considers two types of resources: *register memory* and *stateful ALU*. The resource usage of any configuration c is quantified as a score: $SC(c) = \alpha n_{ALU} + \beta n_{mem}$, where n_{ALU} refers to the number of needed ALUs for the configuration c and n_{mem} refers to the used register memory. Here, α and β are two user-specified parameters to weight ALUs and memory. Note that the number of ALUs also relates to in-switch processing delay because it determines how many match-action tables each packet needs to go through. Thus, if we aim to minimize the processing delay instead of resource usage, we can simply set $\beta = 0$.

Algorithm 4 Searching Algorithm for Configuration

```
Input  : Telemetry application T composed of n mapped sketch instances
Input  : Accuracy intent EB specified by the user
Global variables : S = [], done = [], candidate = []
function SEARCH(T, EB)
    candidate = LHS_INIT(n)
    while candidate not empty do
        Take a configuration c from candidate
        Evaluate T by c
        Add c to done
        if c satisfys EB then
            Add c to S
        neighbors = CALCNEIGHBOR(c)
        for x in neighbors do
            if EXAMINE(x) then
                add x to candidate
    if S is not empty then
        return the configuration whose SC is minimal in S
    else                                   ▷ no configuration satisfies EB
        return the configuration whose accuracy is highest in done
```

Hardness. It remains challenging to adopt classical optimization techniques to solve it. As summarized in §3.2, (1) existing theoretical analysis of sketches considers the worst case; (2) it is challenging to integrate the errors of individual sketch instances into application-level accuracy intent. Thus, we cannot rely on theoretical analysis to estimate actual accuracy for practical workloads or configure parameters. On the other hand, some studies [49, 51, 98] advocate learning the configuration parameters by deep reinforcement learning [62]. However, as telemetry applications vary, it is unacceptable to train a model for each application separately.

5.2 Benchmark-based Searching

AutoSketch performs benchmark analysis to evaluate configuration accuracy and selects configurations accordingly. Specifically, AutoSketch employs several representative workloads. For each configuration, AutoSketch evaluates its accuracy with the workloads. If the resulting accuracy meets the accuracy intent in all workloads, we consider that this configuration can fulfill the intent in practice. Currently, AutoSketch provides several synthetic workloads for the benchmark. The synthetic workloads follow typical distributions of network traffic (e.g., power-law) and are generated with different parameters (e.g., different skewness for power-law distributions) to mimic various scenarios. In addition, we randomly inject additional network traffic in some intervals during the searching to simulate traffic bursts. Users can select the workloads based on their needs or use their own trace. Here, the programs required for the benchmark (including the decomposed `UDF`) are automatically generated by AutoSketch.

Obviously, the benchmark searching is time-consuming because the number of possible configurations is huge. Consider an example in which a sketch instance runs in a switch with four ALUs and 1 MB memory. Even in this simple ex-

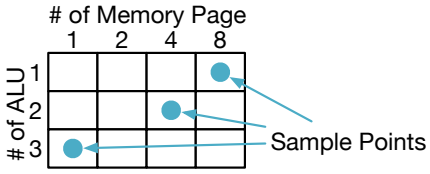


Figure 4: Latin Hypercube Sampling for 2D-space.

ample, the number of possible configurations is above 5 K (the detailed calculation is in Appendix F). Taking an application containing two sketch instances, the time usage of benchmark evaluation for one configuration takes 1-2 minutes under a 4 Gb workload. AutoSketch optimizes the searching procedure in the following paragraphs.

Searching procedure (Algorithm 4). The searching starts with invoking LHS_INIT (see below) to initialize a list of candidate configurations (line 2). Then AutoSketch iteratively pops one configuration, and performs benchmark experiments to evaluate its accuracy (lines 3-6). If this candidate fulfills the intent, it is saved in a set S (lines 7-8). For each candidate, AutoSketch also generates its neighboring configurations that incur similar resource overheads (line 9). Among these neighboring configurations, AutoSketch prunes useless ones and adds the residual to the candidate list (lines 10-12). After all the candidates are evaluated, if there exist configurations fulfilling the intent, AutoSketch returns the one with minimum resource usage (lines 13-14). Otherwise, AutoSketch returns the configuration with the highest accuracy (lines 15-16).

LHS-based configuration initialization. AutoSketch constructs initial configurations based on Latin Hypercube Sampling (LHS) [56]. LHS is a statistical method to generate representative sampled configurations in multi-dimensional spaces. It requires that the generated samples have distinct sampled values in every dimension. Figure 4 shows an example of 2D-space. In this case, LHS generates three sample points. Each row has exactly one sample point. Each column has at most one point. Note that the second column has no sample point because adding any point in this column causes conflicts with other points. In this way, LHS guarantees that the sampled points are evenly distributed in the parameter space. We leave the complete algorithm in Appendix G.

Hardware-aware configuration generation. After evaluating a candidate c, AutoSketch generates the neighboring configurations of c. Function CALCNEIGHBOR leverages the hardware features to reduce the number of configurations. First, existing switches allocate memory at the granularity of 16 KB pages. Thus, we require that the total size of a configuration aligns to the page size. Second, the range of a hash value is a power of two. Thus, we only consider the configurations in which per-row cell count is also a power of two. Specifically, the generation depends on whether c meets the accuracy intent. If c satisfies the intent, AutoSketch generates new configurations with fewer resources, by either decreasing the number of hash functions by one or halving the number of memory pages. Otherwise, if c does not fulfill the intent, AutoSketch either adds one hash function or doubles memory pages to generate configurations with more resources. We leave the pseudo code of CALCNEIGHBOR in Appendix H.

Configuration pruning. AutoSketch applies several rules to prune configurations. AutoSketch first examines unnecessary configurations, including (1) configurations that have been evaluated, (2) configurations whose resource usage exceeds a limit, and (3) configurations that consume more resources than a configuration already satisfying the intent.

In addition, AutoSketch stops when sufficiently good configurations are found. Specifically, if the initial configuration satisfies the intent, AutoSketch stops decreasing resources when a configuration fails to meet the intent. Otherwise, if the initial configuration does not satisfy the intent, AutoSketch stops when a satisfying configuration is found. Therefore, AutoSketch guarantees that it can always find a feasible solution that meets the accuracy intent if such configurations exist. Otherwise, AutoSketch throws an error.

6 Implementation

Interface. We leverage the syntax sugar of Python to implement our interface as a domain-specific language embedded in Python. Specifically, we define a *PacketStream* class and realize all operators as its member functions. For each function, both the input and output have the type PacketStream. This inherently realizes the chaining syntax as shown in §2. We plan to integrate OmniWindow [83] with our interface to support general window mechanisms in our future work.

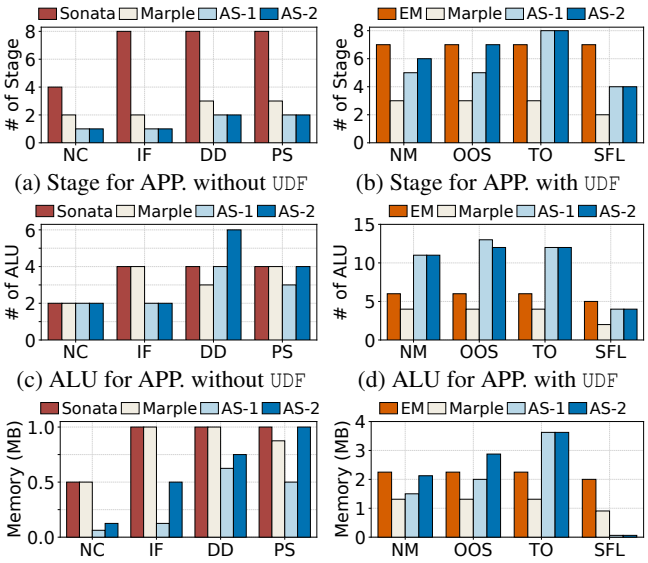
Compiler. We provide two sketch algorithms for each stateful built-in operator. Specifically, a `reduce` operator can be mapped to a Count-Min Sketch [20] or a Count Sketch [12], while we use Bloom Filter [8] and Counting Bloom Filter [23] for `distinct`. We leverage multi-threading techniques to parallelize the time-consuming benchmark experiments. Currently, AutoSketch supports two types of PISA-based target backends, namely Tofino [89] and BMv2 [9]. Our implementation realizes the idea of this paper (i.e., constructing and optimizing sketch instances). Other program optimization (e.g., Chipmunk [26], BitSense [22]) is out of our scope.

Runtime. We realize a control plane runtime and a data plane runtime to deploy AutoSketch. The data plane runtime installs the output P4 programs of AutoSketch into switches and sends sketch structures to the controller. The control plane runtime invokes the query procedures and reports results to users.

Multiple applications and distributed deployment. To deploy multiple applications over distributed devices, we merge the DAGs of each application into a large application and abstract the switch along the packet transmission path into a big pipeline [16, 25]. We leave the searching for this large application on the big pipeline abstraction with different accuracy intents as our future work.

Telemetry Application	Lines of Code			
	Sonata	Marple	AS	AS-P4
Used in existing query-driven telemetry (see §7.2)				
New TCP Conns.(NC) [97]	6	8	7	192
TCP Incomplete Flows (IF) [97]	12	16	15	237
Port Scan (PS) [40]	6	11	6	285
DDoS (DD) [96]	9	14	6	277
TCP timeouts (TO) [67] *	-	6	7	567
TCP non-monotonic (NM) [67] *	-	7	9	618
TCP out-of-sequence (OOS) [67] *	-	6	7	618
TCP SYN Flood (SF) [97] *	-	17	20	635
Used in existing sketch-based telemetry (see §7.3)				
Superspreader (SP) [96]	6	11	6	271
Cardinality (CD) [34]	3	8	4	227
Heavy Hitter (HH) [20]	5	10	5	271

Table 2: (Exp#1) LoC of telemetry applications (* indicates that the application contains `UDF`).



(a) Stage for APP. without `UDF` (b) Stage for APP. with `UDF`
(c) ALU for APP. without `UDF` (d) ALU for APP. with `UDF`
(e) Memory for APP. without `UDF` (f) Memory for APP. with `UDF`

Figure 5: (Exp#2) Switch resource usage against query-driven telemetry.

7 Evaluation

We conduct experiments that compare AutoSketch with state-of-the-art telemetry solutions in various aspects. We summarize our findings for AutoSketch: (1) It implements 11 telemetry applications in <20 lines of code without concerns with parameter configurations (Exp#1 and Exp#5); (2) It satisfies the accuracy intent of at least 95% precision, 97% recall, and at most 2.8% ARE for all the applications with only less than 6 ALUs, 1.06 MB memory and 115 KB/s bandwidth on average (Exp#2, Exp#3 and Exp#4); (3) It consumes around 50% switch resource usage to achieve the intent compared to most classical sketch algorithms (Exp#6 and Exp#7); (4) It achieves better effectiveness than query planning of Sonata and scale to large-volume traffic (Exp#8); (5) It efficiently finds out configurations with high accuracy (Exp#9 and Exp#10 in Appendix I). We put the complete results in Appendix J.

7.1 Setup

Telemetry Applications. Our evaluation considers 11 representative telemetry applications in Table 2. The applications are commonly used to evaluate telemetry solutions, including both query-driven telemetry [30, 67, 105] and sketch-based telemetry. Among these applications, TO, NM, OOS and SFL contain `UDF`s. We configure the applications as described in the literature. We put their realizations in Appendix K.

Workloads. We use CAIDA traces from 2016 to 2019. For stress testing, we simultaneously run PktGen [72] on eight servers to replay the traffic at 40x speed: every 100 ms interval in our result corresponds to 4 s traffic of the origin trace. Thus, each interval contains around 200 K active flows and 6 Gb traffic (60 Gbps). Further, we inject additional 20 K flows at 10% intervals to evaluate AutoSketch under traffic bursts.

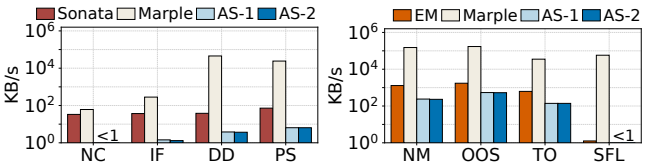
Accuracy metrics. For all applications except for CD, we measure both recall and precision. For TO, NM, and OOS that measure per-flow packet count of interest, we additionally evaluate their ARE. For CD, we measure the relative error (RE): $\frac{n-\hat{n}}{n}$, where n and $\hat{n}$ are the true and estimated number of flows, respectively.

Resource metrics. We consider three types of switch resources: (1) *on-chip memory* used to maintain the states of stateful operators; (2) stateful ALU used to calculate hash functions and update state values; and (3) stage in the PISA switch associated with its own ALUs and memory. Note that we do not report the results on PHV because the packet tuples passing across stages mainly include the parsed fields in packet headers, whose overheads are limited.

Accuracy intents. We use two accuracy intents for AutoSketch: one aims to achieve 95% recall, 95% precision, and 3% ARE, while the other requires 99% recall, 99% precision, and 1% ARE. Both intents set the confidence as 95%. Therefore, we report the 95-th percentiles among the intervals, with the minimum and maximum values as the error bars. For each application, we apply the two intents to generate their P4 programs, denoted by AS-1 and AS-2, respectively.

7.2 Compare with Query-driven Telemetry

Methodology. We compare AutoSketch with two query-driven telemetry systems, Sonata [30] and Marple [67]. For Sonata, we configure each stateful operator with 2^{16} counters and utilize the refinement plan $* \rightarrow 8 \rightarrow 32$ as in its open-source prototype. For Marple, we implement its key-value cache and evict old keys to handle hash collisions. We configure 2^{16} cache slots as in the paper [67]. For applications with `groupby` operators, we also consider a naive approach, namely *entire mapping* (EM), that converts each `UDF` operator entirely into a sketch-like structure (see §3.1). Since EM does not resolve hash collisions, we configure the structure with 256 K cells to handle the >200 K flows per interval.



(a) Applications without UDF. (b) Applications with UDF.

Figure 6: (Exp#3) Bandwidth usage against query-driven telemetry.

(Exp#1) Expressiveness. We first compare the expressiveness of AutoSketch with Sonata and Marple, both of which regard expressiveness as the "first-class citizens" in their designs. Table 2 shows that AutoSketch needs similar lines of code (LoC) to implement the eleven telemetry applications. On the other hand, it needs hundreds of lines if we directly use P4. Further, AutoSketch allows expressing accuracy intents, while the Sonata and Marple do not have such features.

(Exp#2) Resource usage. This experiment evaluates the resource usage of AutoSketch in switches. We deploy each solution atop a Barefoot Wedge 100B-32X (Tofino) PISA switch [89], and then measure the incurred resource usage. We consider eight telemetry applications in this experiment. Recall that Sonata does not support UDFs, while EM only applies to UDFs. Thus, we present the results of the eight applications separately, according to the existence of UDFs.

Figure 5 presents the results. For applications without UDFs, Sonata occupies much more stages than others due to its refinement mechanism which reduces the traffic sent to the stream processor. AutoSketch consumes less or the same amount of ALUs and memory as Sonata and Marple except for DD under AS-2, but achieves lower bandwidth overhead (Exp#3) and higher accuracy than Sonata (Exp#4). For example, the memory overheads of Sonata and Marple for NC and IF are 8 times higher than AS-1. For applications with UDFs, EM incurs six more stages, nine more ALUs, and 1.6 MB more memory than AS-1 over the total resources of four applications and suffers from serious hash conflicts and incurs low accuracy (see Exp#4). However, AutoSketch consumes two to fine more stages, seven to nine more ALUs and 15%-175% more memory than Marple. The reason is that the user-defined states in UDFs are complicated, thereby AutoSketch enlarges the resource usage to mitigate hash conflicts. However, Marple has to deal with hash conflicts by sending excessive traffic to the control plane (see Exp#3).

(Exp#3) Bandwidth usage. Figure 6 presents the traffic sent to the control plane of each approach. Although the current bandwidth of the control plane is in the magnitude of 100 Gbps, it must hold numerous data plane devices. Thus, it is critical to save control plane bandwidth [15, 30, 46, 67]. AutoSketch incurs 115 KB/s bandwidth for all the applications, while Sonata, EM and Marple consume 45.1 KB/s, 913.2 KB/s, and 485.2 MB/s, respectively. Here, Sonata reduces the bandwidth usage at the cost of higher switch resources (see Exp#2). The reasons why EM and Marple incur higher bandwidth consumption than AutoSketch are as follows. For Marple, it needs to evict conflicting packets to the control plane. For EM, it suffers from severe hash conflicts, so a large number of normal packets are wrongly identified as anomalies and then sent to the control plane.

(Exp#4) Accuracy. This experiment compares the accuracy of AutoSketch with Marple, Sonata, and EM. Figures 7(a) and 7(b) present the results of the applications without UDFs. Marple achieves zero error for all applications because they completely handle hash conflicts. However, it incurs excessive resource overheads and bandwidth consumption (see Exp#2 and Exp#3). Sonata well ensures the precision of all applications but fails at the recall metric. The reason is that Sonata decreases the counting value during the refinement process and many abnormal flows are not wrongly identified as normals. AutoSketch achieves high accuracy. In particular, it satisfies our two accuracy intents for all applications.

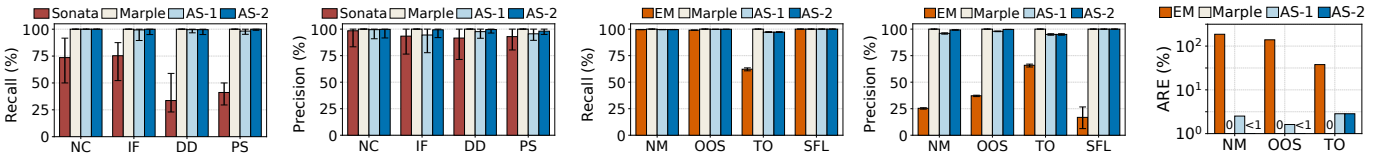
Figures 7(c) to 7(e) show the results on the applications with UDFs. Marple achieves zero error for the four applications. Except for TO, all AutoSketch applications meet both intents. Here, TO also achieves 95% precision and 97% recall, which is close to the intent. However, the recall of TO for EM is only 62% and the average precision for EM is even much lower (only 36%). EM also incurs high ARE: the ARE for NM even exceeds 186%. The reason is that some states are extremely sensitive to hash conflicts. When binding all states in one sketch-like structure, hash conflicts will seriously compromise their accuracy.

7.3 Compare with Sketch-based Telemetry

Methodology. We compare AutoSketch with existing sketch algorithms. In Table 2, we select three applications that have been extensively studied by existing sketch algorithms. We consider two general-purpose sketch-based solutions FlowRadar (FR) [50] and OpenSketch (OS) [96] that can support all three applications. FR proposes a general sketch structure to support various applications. For OS, we build the algorithms in P4 as suggested by the original paper [96].

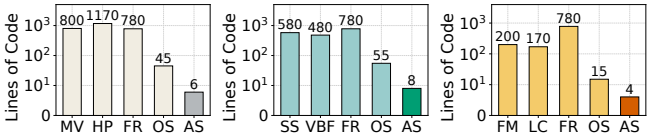
In addition to FR and OS, we also select two state-of-the-art sketch algorithms that are specifically designed for each application: in heavy hitter detection (HH), we compare AutoSketch with MV-Sketch (MV) [87] and HashPipe (HP) [79]; in superspreader detection (SP), we compare AutoSketch with SpreadSketch (SS) [86] and Vector Bloom Filter (VBF) [52]; in cardinality estimation, we compare AutoSketch with FM-Sketch (FM) [24] and Linear Counting (LC) [91].

We realize these algorithms based on their published designs except MV and SS that provide open-source implementation. We also apply AS-1 and AS-2 for AutoSketch and the sketch algorithms. We follow the guidelines in the literature to configure these sketches. Here, we present the results of AS-1. The results of AS-2 are in Appendix J. We do not present the bandwidth usage of sketch-based solutions because they only



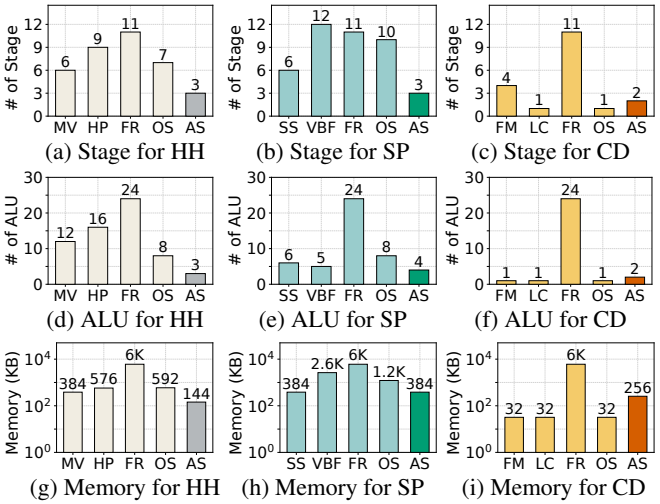
(a) Recall for APP. w/o `UDF` (b) Precision for APP. w/o `UDF` (c) Recall for APP. w/ `UDF` (d) Precision for APP. w/ `UDF` (e) ARE for APP. w/ `UDF`

Figure 7: (Exp#4) Accuracy against query-driven telemetry.



(a) Heavy Hitter (b) Superspreader (c) Cardinality

Figure 8: (Exp#5) Expressiveness against sketch-based telemetry.



(a) Stage for HH (b) Stage for SP (c) Stage for CD

(d) ALU for HH (e) ALU for SP (f) ALU for CD

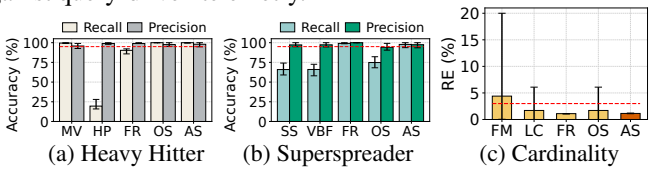
(g) Memory for HH (h) Memory for SP (i) Memory for CD

Figure 9: (Exp#6) Switch resource usage against sketch-based telemetry.

transfer the sketch structures to the controller. Thus, we can infer the bandwidth from their memory usage in Exp#6.

(Exp#5) Expressiveness. We first compare the expressiveness of AutoSketch with the sketch algorithms. Figure 8 shows that AutoSketch only needs less than ten LoC to express each telemetry application. However, existing sketch algorithms require hundreds of lines of P4 code. Note that OS provides one library that includes some pre-built sketches. Therefore, OS can express one telemetry application in dozens of configuration LoC. Further, AutoSketch automatically generates corresponding P4 programs with parameter configurations, while sketch algorithms need many efforts to tune parameters.

(Exp#6) Resource usage. We compare the resource usage of AutoSketch with the sketch algorithms. Figures 9(a) to 9(i) show that AutoSketch only consumes 50% switch resources compared to most existing sketch algorithms except for LC and FM (OS utilizes LC to realize cardinality). The reason is that existing sketch algorithms typically consider worst-case scenarios, while AutoSketch selects the most suitable configuration via benchmarking-based searching. FM and LC use very limited resources because they only support the simple



(a) Heavy Hitter (b) Superspreader (c) Cardinality

Figure 10: (Exp#7) Accuracy against sketch-based telemetry.

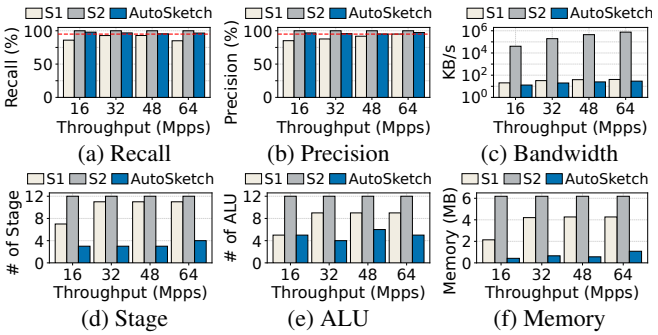
cardinality estimation at the cost of occasional unacceptable error (Exp#7). FR incurs much higher resource usage since it has complicated structures to support general telemetry tasks. HP and VBF take the multi-level data structure that needs more stages than other solutions. OS incurs more resources than AutoSketch to detect heavy hitters and super spreaders, since it adopts reversible sketch [75] to maintain flowkey.

(Exp#7) Accuracy. We compare the accuracy of AutoSketch with the sketch algorithms (the red dotted line in the figure indicates the accuracy intent). For the heavy hitter detection (Figure 10(a)), all solutions can reach 95% precision. However, the recall of HP and FR is only 20% and 90%, respectively. The reason is that their desired resources exceed the hardware capacity. MV achieves similar accuracy as AutoSketch, but needs more resources (see Exp#6). For the superspreader detection (Figure 10(b)), SS, VBF, and OS reach high precision but cannot fulfill the recall goal, since they are sensitive to hash conflicts. For the cardinality estimation (Figure 10(c)), FM cannot satisfy the intent and its RE suffers from terribly high error. AutoSketch meets the accuracy intent in all the telemetry applications for two reasons. First, our parameter configuration avoids worst-case parameters in traditional theoretical analysis. Second, our randomly injecting additional traffic adapts to traffic bursts.

7.4 Micro Benchmarks

(Exp#8) Parameter tuning. We compare our benchmark-based parameter tuning with the query planning of Sonata. We consider two versions of Sonata, denoted by S1 and S2, respectively. S1 is the default setup in Sonata's open-source prototype [69] that does not deal with hash collisions. S2 resolves hash conflicts by mapping each flow to d counters and evicting a flow to the remote stream processor if hash collisions occur (c.f. §3.1 of [30]). We follow the guidelines in [30] to make the partition and refinement decision via the ILP. We stress test all systems with different packet rates.

We present the DDoS detection under AS-1 due to space limitations. Figure 11 shows the results. Even though S1 carefully tunes the data structure size and the refinement plan ($* \rightarrow 16 \rightarrow 32$), it still fails to satisfy AS-1 (Figure 11(a) and



(a) Recall (b) Precision (c) Bandwidth
(d) Stage (e) ALU (f) Memory
Figure 11: (Exp#8) Parameter tuning.

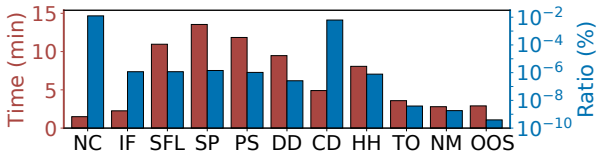


Figure 12: (Exp#9) Searching cost.

11(b)) because it does not resolve hash conflicts. In contrast, S2 achieves zero error at the cost of higher switch resource and bandwidth overhead. Compared to S1 and S2, AutoSketch satisfies the accuracy intent with lower switch resource and bandwidth overhead. The reason is that although Sonata efficiently determines offloaded operators, it lacks the optimization for individual operators. On the opposite, AutoSketch targets parameter tuning for each operator. The two approaches are orthogonal to each other, which can be combined to realize both benefits.

(Exp#9) Searching costs. We evaluate the cost of the benchmark-based configuration searching algorithm. Figure 12 presents the searching time (red bar) and the ratio between the number of configurations actually evaluated and the number of all possible configurations (blue bar). As in §5.2, the number of possible configurations is extraordinarily huge. Our algorithm greatly prunes the number of evaluated configurations by three to nine magnitudes. In particular, our searching algorithm leverages hardware features to prune configurations (see §5), which decrease the candidate configurations by at most six magnitudes. Other techniques further reduce by two magnitudes. All the applications converge a configuration close to the best solution (see Exp#10 in Appendix I) within 15 minutes using a single core, with an average time of 6.5 minutes. It is acceptable because the searching is performed once before an application is deployed.

8 Related Works

Query-driven telemetry. Existing studies on query-driven telemetry can be categorized into two classes. The first class is based on text-based expression [4, 66, 95, 97]. Specifically, PathQuery [66] focuses on the path-based traffic monitoring. NetQRE [97] combines application-level aggregation operations with regular-expression-like pattern matching. dShark [95] leverages the json-based telemetry language to declare fields and then invoke callback functions for further processing. The second class provisions a collection of operators for users to form telemetry applications [19, 29, 30, 38, 45, 47, 67, 105]. AutoSketch follows the line of operator-based telemetry with two enhancements: the accuracy intent in the language, and the automatic compiler to generate sketch-based programs.

Telemetry algorithms. There are three classes of telemetry algorithms. Sampling algorithms [11,41,73,76,77,82] consume limited resources but suffer from high errors. Counter-based approaches [6, 79] only address top-k flows, thereby failing to support generic applications. Sketch algorithms maintain compact data structures that allow multiple flows to share counters [1, 28, 35, 50, 54, 55, 94, 96, 101, 103]. With proper parameters, they achieve both high accuracy and resource efficiency. However, they suffer from heavy implementation and configuration burdens. AutoSketch hides such complexity with its programmability.

Telemetry architectures. Host-based telemetry systems [27, 42,65,84] suffer from limited network visibility. Switch-based systems [7, 17, 39, 59, 68, 71, 80, 81, 88, 90, 104] process traffic with high-speed ASICs but suffer from the limited memory. Recent studies often combine different network entities. SwitchPointer [85] utilizes the switch memory as a directory that points to the telemetry data on end-hosts. Sonata [30] abstracts the switch and end-host into a big streaming processor. Marple [67] treats the switch memory as the cache of the key-value store on end-hosts. MOZART [53] coordinates the switches and end-hosts to select and monitor traffic. OmniMon [37] integrates the capabilities of end-hosts, switches, and the controller. AutoSketch is orthogonal to these works.

9 Conclusion

This paper proposes AutoSketch, a network telemetry solution that bridges the gap between query-driven telemetry and sketch-based telemetry algorithms. AutoSketch extends the existing operator-based interface to allow users to specify the accuracy intent. It designs a compiler to translate the stateful operators in telemetry applications into sketch instances in the data plane. AutoSketch automatically configures parameters to fulfill the accuracy intent. Experiments demonstrate that AutoSketch achieves high expressiveness and high accuracy with limited hardware resources.

Acknowledgements

We thank our shepherd, Alan Zaoxing Liu, and the anonymous reviewers for their valuable comments. The work was supported in part by National Key R&D Program of China (2019YFB1802600), Joint Funds of the National Natural Science Foundation of China (U20A20179), National Natural Science Foundation of China (62172007).

References

[1] Anup Agarwal, Zaoxing Liu, and Srini Seshan. HeteroSketch: Coordinating network-wide monitoring in heterogeneous and dynamic networks. In *Proc. of USENIX NSDI*, 2022.

[2] Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff Phillips, Zhewei Wei, and Ke Yi. Mergeable summaries. In *Proc. of ACM PODS*, 2012.

[3] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proc. of ACM EuroSys*, 2013.

[4] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. Snap: Stateful network-wide abstractions for packet processing. In *Proc. of ACM SIGCOMM*, 2016.

[5] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the blame game out of data centers operations with netpoirot. In *Proc. of ACM SIGCOMM*, 2016.

[6] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo C Luizelli, and Erez Waisbard. Constant Time Updates in Hierarchical Heavy Hitters. In *Proc. of ACM SIGCOMM*, 2017.

[7] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. In *Proc. of ACM SIGCOMM*, 2020.

[8] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of ACM*, 13, 1970.

[9] BMv2 PISA switch. https://github.com/p4lang/behavioral-model.

[10] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proc. of ACM SIGCOMM*, 2013.

[11] Marco Canini, Damien Fay, David J. Miller, Andrew W. Moore, and Raffaele Bolla. Per Flow Packet Sampling for High-Speed Network Monitoring. In *Proc. of COMSNETS*, 2009.

[12] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312:3–15, 2004.

[13] Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. One primitive to diagnose them all: Architectural support for internet diagnostics. In *Proc. of EuroSys*, 2017.

[14] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proc. of ACM SIGCOMM*, 2016.

[15] Xiang Chen, Qun Huang, Peiqiao Wang, Hongyan Liu, Yuxin Chen, Dong Zhang, Haifeng Zhou, and Chunming Wu. Mtp: Avoiding control plane overload with measurement task placement. In *Proc. of IEEE INFOCOM*, 2021.

[16] Xiang Chen, Hongyan Liu, Qun Huang, Peiqiao Wang, Dong Zhang, Haifeng Zhou, and Chunming Wu. Speed: Resource-efficient and high-performance deployment for data plane programs. In *Proc. of IEEE ICNP*, 2020.

[17] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *Proc. of CoNEXT*, 2019.

[18] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proc. of ACM SIGCOMM*, 2020.

[19] Xin Cheng, Zhiliang Wang, Shize Zhang, Xin He, and Jiahai Yang. Intstream: An intent-driven streaming network telemetry framework. In *Proc. of IEEE CNSM*, 2021.

[20] Graham Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms*, pages 58–75, 2005.

[21] Xenofontas Dimitropoulos, Paul Hurley, and Andreas Kind. Probabilistic lossy counting: an efficient algorithm for finding heavy hitters. *Proc. of ACM SIGCOMM CCR*, 38, 2008.

[22] Rui Ding, Shibo Yang, Xiang Chen, and Qun Huang. Bitsense: Universal and nearly zero-error optimization for sketch counters with compressive sensing. In *Proc. of ACM SIGCOMM*, 2023.

[23] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. on Networking*, 8:281–293, 2000.

[24] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31:182–209, 1985.

[25] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proc. of ACM SIGCOMM*, 2020.

[26] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In *Proc. of ACM SIGCOMM*, 2020.

[27] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. SIMON: A simple and scalable method for sensing, inference and measurement in data center networks. In *Proc. of USENIX NSDI*, 2019.

[28] Junzhi Gong, Tong Yang, Haowei Zhang, Hao Li, Steve Uhlig, Shigang Chen, Lorna Uden, and Xiaoming Li. HeavyKeeper: An accurate algorithm for finding top-k elephant flows. In *Proc. of USENIX ATC*, 2018.

[29] Arpit Gupta, Rüdiger Birkner, Marco Canini, Nick Feamster, Chris Mac-Stoker, and Walter Willinger. Network monitoring as a streaming analytics problem. In *Proc. of ACM HotNets*, 2016.

[30] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-Driven Streaming Network Telemetry. In *Proc. of ACM SIGCOMM*, 2018.

[31] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. Modular switch programming under resource constraints. In *Proc. of USENIX NSDI*, 2022.

[32] Chi-Yao Hong, Matthew Caesar, Nick Duffield, and Jia Wang. Tiresias: Online anomaly detection for hierarchical operational network data. In *Proc. of IEEE ICDCS*, 2012.

[33] Ningning Hu, Li (Erran) Li, Zhuoqing Morley Mao, Peter Steenkiste, and Jia Wang. Locating internet bottlenecks: Algorithms, measurements, and implications. In *Proc. of ACM SIGCOMM*, 2004.

[34] Qun Huang, Xin Jin, Patrick P C Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. SketchVisor: Robust Network Measurement for Software Packet Processing. In *Proc. of ACM SIGCOMM*, 2017.

[35] Qun Huang and Patrick P. C. Lee. A Hybrid Local and Distributed Sketching Design for Accurate and Scalable Heavy Key Detection in Network Data Streams. *Computer Networks*, 91:298–315, 2015.

[36] Qun Huang, Patrick P. C. Lee, and Yungang Bao. SketchLearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference. In *Proc. of ACM SIGCOMM*, 2018.

[37] Qun Huang, Haifeng Sun, Patrick P. C. Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proc. of ACM SIGCOMM*, 2020.

[38] Arthur S. Jacobs, Ricardo J. Pfitscher, Rafael H. Ribeiro, Ronaldo A. Ferreira, Lisandro Z. Granville, Walter Willinger, and Sanjay G. Rao. Hey, lumi! using natural language for Intent-Based network management. In *Proc. of USENIX ATC*, 2021.

[39] Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. Burstradar: Practical real-time microburst monitoring for datacenter networks. In *Proc. of ACM ApSys*, 2018.

[40] Jaeyeon Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *Proc. of IEEE Symposium on Security and Privacy*, 2004.

[41] Srikanth Kandula and Ratul Mahajan. Sampling Biases in Network Path Measurements and What To Do About It. In *Proc. of ACM IMC*, 2009.

[42] Khandelwal, Anurag and Agarwal, Rachit and Stoica, Ion. Confluo: Distributed Monitoring and Diagnosis Stack for High-Speed Networks. In *Proc. of USENIX NSDI*, 2019.

[43] Daniel Kopp, Matthias Wichtlhuber, Ingmar Poese, Jair Santanna, Oliver Hohlfeld, and Christoph Dietzel. Ddos hide & seek: On the effectiveness of a booter services takedown. In *Proc. of ACM IMC*, 2019.

[44] Jan Kučera, Ran Ben Basat, Mário Kuka, Gianni Antichi, Minlan Yu, and Michael Mitzenmacher. Detecting routing loops in the data plane. In *Proc. of ACM CoNEXT*, 2020.

[45] Paolo Laffranchini, Luis Rodrigues, Marco Canini, and Balachander Krishnamurthy. Measurements as first-class artifacts. In *Proc. of IEEE INFOCOM*, 2019.

[46] Jonatan Langlet, Ran Ben-Basat, Sivaramakrishnan Ramanathan, Gabriele Oliaro, Michael Mitzenmacher, Minlan Yu, and Gianni Antichi. Zero-cpu collection with direct telemetry access. In *Proc. of ACM HotNets*, 2021.

[47] Christopher Leet, Robert Soulé, Yang Richard Yang, and Ying Zhang. Flow algebra: Towards an efficient, unifying framework for network management tasks. In *Proc. of IEEE INFOCOM*, 2021.

[48] Jakub Lemiesz. On the algebra of data sketches. *Proc. VLDB Endowment*, 14:1655–1667, 2021.

[49] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proc. of VLDB Endow.*, 2019.

[50] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A Better NetFlow for Data Centers. In *Proc. of USENIX NSDI*, 2016.

[51] Zhao Li, Haifeng Sun, Zheng Xiong, Qun Huang, Zehong Hu, Ding Li, Shasha Ruan, Hai Hong, Jie Gui, Jintao He, Zebin Xu, and Yang Fang. Noah: Reinforcement-learning-based rate limiter for microservices in large-scale e-commerce services. *IEEE Transactions on Neural Networks and Learning Systems*, 34(9):5403–5417, 2023.

[52] W. Liu, W. Qu, J. Gong, and K. Li. Detection of superpoints using a vector bloom filter. *IEEE Trans. on Information Forensics and Security*, 11:514–527, 2016.

[53] Xuemei Liu, Meral Shirazipour, Minlan Yu, and Ying Zhang. MOZART: Temporal Coordination of Measurement. In *Proc. of ACM SOSR*, 2016.

[54] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and General Sketch-Based Monitoring in Software Switches. In *Proc. of ACM SIGCOMM*, 2019.

[55] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proc. of ACM SIGCOMM*, 2016.

[56] Beckman RJ McKay MD and Conover WJ. A comparison of the three methods for selecting values of input variable in the analysis of output from a computer code. *Technometrics;(United States)*, 1979.

[57] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Proc. of ICDT*, 2005.

[58] Ruijie Miao, Fenghao Dong, Yikai Zhao, Yiming Zhao, Yuhan Wu, Kaicheng Yang, Tong Yang, and Bin Cui. Sketchconf: A framework for automatic sketch configuration. In *Proc. of IEEE ICDE*, 2023.

[59] Chris Misa, Ramakrishnan Durairajan, Reza Rejaie, and Walter Willinger. Revisiting network telemetry in coin: A case for runtime programmability. *IEEE Network*, 35:14–20, 2021.

[60] Chris Misa, Walt O'Connor, Ramakrishnan Durairajan, Reza Rejaie, and Walter Willinger. Dynamic scheduling of approximate telemetry queries. In *Proc. of USENIX NSDI*, 2022.

[61] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[62] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 2015.

[63] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *Proc. of ACM SIGCOMM*, 2014.

[64] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. SCREAM: Sketch Resource Allocation for Software-defined Measurement. In *Proc. of ACM CoNEXT*, 2015.

[65] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In *Proc. of ACM SIGCOMM*, 2016.

[66] Srinivas Narayana, Mina Tashmasbi Arashloo, Jennifer Rexford, and David Walker. Compiling Path Queries. In *Proc. of USENIX NSDI*, 2016.

[67] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *Proc. of ACM SIGCOMM*, 2017.

[68] Inband network telemetry. http://p4.org/wp-content/uploads/fixed/INT/INT-current-spec.pdf.

[69] Sonata open-source repository. https://github.com/Sonata-Princeton/SONATA-DEV.

[70] p4c: P4 Compiler. https://github.com/p4lang/p4c.

[71] Mengying Pan, Robert MacDavid, Shir Landau-Feibish, and Jennifer Rexford. Memory-efficient membership encoding in switches. In *Proc. of ACM SOSR*, 2020.

[72] PktGen. https://pktgen-dpdk.readthedocs.io.

[73] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. Planck: Millisecond-scale Monitoring and Control for Commodity Networks. In *Proc. of ACM SIGCOMM*, 2014.

[74] Arjun Roy, Deepak Bansal, David Brumley, Harish Kumar Chandrappa, Parag Sharma, Rishabh Tewari, Behnaz Arzani, and Alex C. Snoeren. Cloud datacenter sdn monitoring: Experiences and challenges. In *Proc. of ACM IMC*, 2018.

[75] Robert Schweller, Zhichun Li, Yan Chen, Yan Gao, Ashish Gupta, Yin Zhang, Peter Dinda, Ming Yang Kao, and Gokhan Memik. Reversible Sketches: Enabling Monitoring and Analysis over High-Speed Data Streams. *Proc. of IEEE/ACM Trans. on Networking*, pages 1059–1072, 2007.

[76] Vyas Sekar, Michael K. Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G. Andersen. cSAMP: A System for Network-Wide Flow Monitoring. In *Proc. of USENIX NSDI*, 2008.

[77] Vyas Sekar, Michael K Reiter, and Hui Zhang. Revisiting the Case for a Minimalist Approach for Network Flow Monitoring. In *Proc. of ACM IMC*, 2010.

[78] Rachee Singh, Sharad Agarwal, Matt Calder, and Paramvir Bahl. Cost-effective cloud edge traffic engineering with cascara. In *Proc. of USENIX NSDI*, 2021.

[79] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-Hitter Detection Entirely in the Data Plane. In *Proc. of ACM SOSR*, 2017.

[80] John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. Turboflow: Information rich flow record generation on commodity switches. In *Proc. of ACM EuroSys*, 2018.

[81] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *Proc. of USENIX ATC*, 2018.

[82] Junho Suh, Ted Taekyoung Kwon, Colin Dixon, Wes Felter, and John Carter. OpenSample: A Low-Latency, Sampling-Based Measurement Platform for Commodity SDN. In *Proc. of ICDCS*, 2014.

[83] Haifeng Sun, Jiaheng Li, Jintao He, Jie Gui, and Qun Huang. Omniwindow: A general and efficient window mechanism framework for network telemetry. In *Proc. of ACM SIGCOMM*, 2023.

[84] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with pathdump. In *Proc. of USENIX OSDI*, 2016.

[85] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed Network Monitoring and Debugging with SwitchPointer. In *Proc. of USENIX NSDI*, 2018.

[86] Lu Tang, Qun Huang, and Patrick P. C. Lee. Spreadsketch: Toward invertible and network-wide detection of superspreaders. In *Proc. of IEEE INFOCOM*, 2020.

[87] Lu Tang, Qun Huang, and Patrick PC Lee. Mv-sketch: A fast and compact invertible sketch for heavy flow detection in network data streams. In *Proc. of IEEE INFOCOM*, 2019.

[88] Ross Teixeira, Rob Harrison, Arpit Gupta, and Jennifer Rexford. Packetscope: Monitoring the packet lifecycle inside a switch. In *Proc. of ACM SOSR*, 2020.

[89] Tofino. https://www.barefootnetworks.com/products/brief-tofino/.

[90] Weitao Wang, Xinyu Crystal Wu, Praveen Tammana, Ang Chen, and TS Eugene Ng. Closed-loop network performance monitoring and diagnosis with SpiderMon. In *Proc. of USENIX NSDI*, 2022.

[91] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. A linear-time probabilistic counting algorithm for database applications. *Proc. of ACM TODS*, 15:208–229, 1990.

[92] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. Zeno: Diagnosing performance problems with temporal provenance. In *Proc. of USENIX NSDI*, 2019.

[93] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing missing events in distributed systems with negative provenance. In *Proc. of ACM SIGCOMM*, 2014.

[94] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic Sketch: Adaptive and Fast Network-wide Measurements. In *Proc. of ACM SIGCOMM*, 2018.

[95] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. dShark: A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces. In *Proc. of USENIX NSDI*, 2019.

[96] Minlan Yu, Lavanya Jose, and Rui Miao. Software Defined Traffic Measurement with OpenSketch. In *Proc. of USENIX NSDI*, 2013.

[97] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. Quantitative Network Monitoring with NetQRE. In *Proc. of ACM SIGCOMM*, 2017.

[98] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proc. of ACM SIGMOD*, 2019.

[99] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. *Proc. of NDSS*, 2020.

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

[101] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: High-performance sketch-based measurement over arbitrary partial key query. In *Proc. of ACM SIGCOMM*, 2021.

[102] Hao Zheng, Chen Tian, Tong Yang, Huiping Lin, Chang Liu, Zhaochen Zhang, Wanchun Dou, and Guihai Chen. Flymon: Enabling on-the-fly task reconfiguration for network measurement. In *Proc. of ACM SIGCOMM*, 2022.

[103] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *Proc. of ACM SIGMOD*, 2018.

[104] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. Flow event telemetry on programmable data plane. In *Proc. of ACM SIGCOMM*, 2020.

[105] Yu Zhou, Dai Zhang, Kai Gao, Chen Sun, Jiamin Cao, Yangyang Wang, Mingwei Xu, and Jianping Wu. Newton: Intent-driven network traffic monitoring. In *Proc. of CoNEXT*, 2020.

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

Appendix A Decomposed TCP non-monotonic

```
def nonmt_cf(tcp.seq):
  temp = maxseq
  if maxseq < tcp.seq:
    maxseq = tcp.seq
  return temp

def nonmt_uf(tcp.seq, maxseq):
  if maxseq >= tcp.seq:
    nm_count += 1

nm[precision_min=99%,ARE_max=1%,confidence=95%]=
  PacketStream()
  .filter(ipv4.protocol==TCP)
  .groupby({5tuple: maxseq}, nonmt_cf)
  .groupby({5tuple: nm_count}, nonmt_uf)
```

Application 2: Decomposed TCP non-monotonic

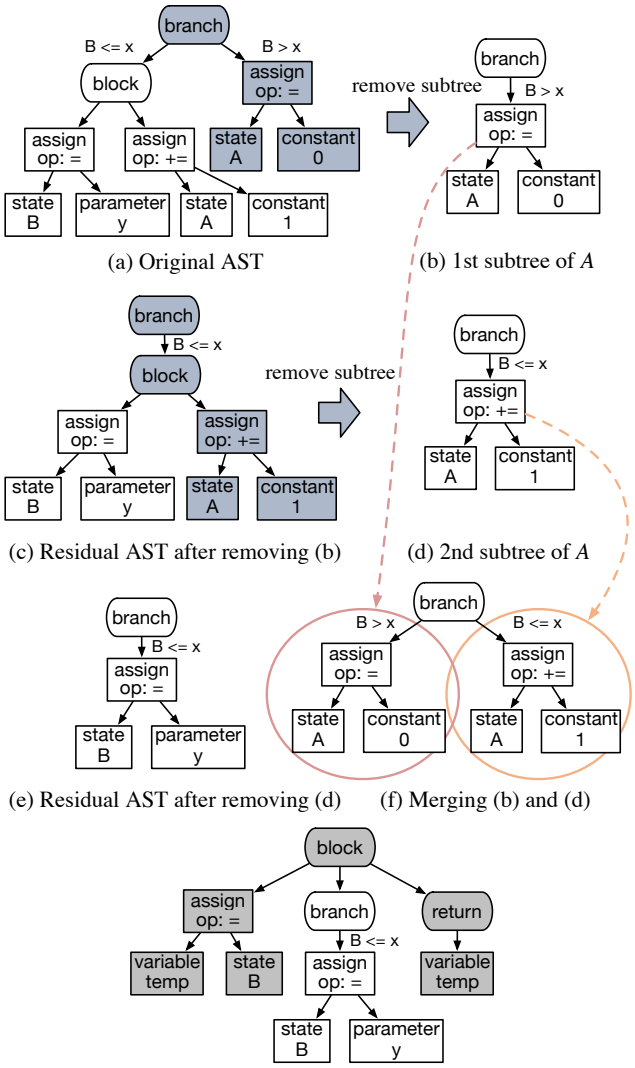
Appendix B Complex Decomposition Example

Figure 13 illustrates the partitioning procedure with a complex example AST. The original AST in Figure 13(a) modifies two states: state *A* and state *B*. We start with partitioning modifications on state *A*. We identify a first subtree modifying *A* as shown in Figure 13(b) (also marked in gray in Figure 13(a)). Note that the root node appears in multiple paths. Thus, the root remains in the residual AST in Figure 13(c), while the assignment node and its children are removed. Since there is still a subtree modifying *A* (marked in gray in Figure 13(c)), we continue to remove it. Figure 13(d) and Figure 13(e) present the second removed subtree and the residual AST. In Figure 13(e), there is only one state *B*, and thereby we do not remove subtrees further. Since there are two subtrees modifying state *A* (in Figure 13(b) and Figure 13(d), respectively), Figure 13(f) integrates them by merging the duplicated root. In Figure 13, updating state *A* depends on the value of state *B*. Thus, we extend the AST of *B* in Figure 13(e) to a new AST in Figure 13(g). Since the value of *B* before updating is actually needed, we add an assignment statement that saves the value before updating in a variable *temp*. The variable is returned and passed to AST of *A* after all statements are executed.

Appendix C Theoretical analysis of Sketch-like structure

We analyze theoretical error bounds according to the behavior of states in sketch-like structures. One type is summable state (i.e., counter), whose sketch-like structure is equivalent to classical CM [20]. The other type is non-summable state (e.g., ingress port number, TCP sequence number).

For non-summable states, we analyze the probability that a flow maintains inaccurate states. Specifically, a flow is accurate if at least one of its hashed cells has no conflicts. Consider



(a) Original AST (b) 1st subtree of A

(c) Residual AST after removing (b) (d) 2nd subtree of A

(e) Residual AST after removing (d) (f) Merging (b) and (d)

(g) Appending temporary variable to (e)

Figure 13: Example of `UDF` decomposition.

N flows in a sketch-like structure with w rows and d columns. We assume that hash functions are perfectly random and independent. The probability that a flow does not collide with others in one row is $(1-\frac{1}{d})^{N-1}$. For w rows, the probability that the flow has at least one row without conflict P is:

$$P = 1-(1-(1-\frac{1}{d})^{N-1})^w \approx 1-(1-e^{-\frac{d}{N}})^w. \quad (1)$$

The users can utilize the above theoretical analysis to estimate the accuracy of the search configuration.

We can further narrow the error bounds according to the monotonicity-based query method of our sketch-like structure. Take `nonmt_cf` in Application 2 as an example, its sketch-like structure maintains the maximal sequence number of each flow, which is monotonically increasing and thus return the minimum value as the query result. In fact, not the conflicts of all the flows will affect query results. We can first sort the packets with the largest sequence number of each flow in ascending order. Considering the ith largest flow, only $N-i$ flows that collide with it will affect the query result, referred as *interference flows*. Here, Equation 1 is updated to $P = 1-(1-(1-\frac{1}{d})^{i-1})^w$. For our sketch-like structure, the query results remain correct as long as one row does not conflict with the interference flows. Therefore, the expectation of the number of flows for which the query results are accurate is

$$E = \sum_{i=1}^{N} 1-(1-(1-\frac{1}{d})^{i-1})^w = N - \sum_{i=1}^{N-1}(1-(1-\frac{1}{d})^{i})^w \quad (2)$$

This is similar to the sketch-like structure of monotonically decreasing. For the case without monotonicity, all the flows can be interference flows.

Appendix D Optimization for sketch mapping

We optimize the sketch instances with several techniques. First, we allow sketch instances to share hash functions: we compute several hash values for each key, and use them to locate cells in multiple sketch instances. This reduces computational units in switches, particularly the decomposition in §4.1 produces numerous sketch instances. Second, we clear the state values of terminated TCP connections to mitigate hash conflicts. Third, since some applications are interested in partial flows, we only record the keys addressed by the applications instead of all flow keys.

Appendix E Formulation of Benchmark-based parameter tuning

Goal	$min(SC(c))$
Constraints	
C_0	$T(c)$ satisfies EB
C_1	$\sum_{i=0}^{n-1} s_i \le S - s_{stateless}$
C_2	$s_i \ge 1, i \in [1,2,...,n]$
C_3	$d_i \in [1,2,...,s_iA], i \in [1,2,...,n]$
C_4	$\frac{w_i b_i}{8} \le M, i \in [1,2,...,n]$
C_5	$\frac{d_i w_i b_i}{8} \le s_i M, i \in [1,2,...,n]$

Table 3: Formulation for searching.

Table 3 formulates the optimization problem of our benchmark-based parameter tuning. First, the accuracy of the telemetry application $T(c)$ should reach the user-specified accuracy intent EB (C_0). Second, the resource usage is bounded by the hardware capability. C_1 indicates the total stage resources that can be used to deploy sketch instances. Here, S refers to the number of stages in match-action pipeline and $S_{stateless}$ indicates the number of stages used for stateless operators, which can be determined before sketch-based mapping. C_2 indicates that each sketch instance should use at least one

stage. C_3 constraints the maximal number of stateful ALUs cannot exceed sA, where A is the number of stateful ALUs per stage. The register memory size allocated for one stateful ALU cannot exceed the size of register memory (in bytes) per stage M (C_4). Meanwhile, the total register memory of one sketch instance should be less than sM (C_5).

Appendix F The number of Configurations

Consider an example in which we deploy a sketch with a two-dimensional matrix structure in a switch with a single stage. The stage contains four ALUs and 1 MB memory. Assume that each cell occupies fours bytes. Since each hash function needs one ALU, we can support at most four rows. When there is only one single row, we have at most $2^{20}/4 = 262144$ cells in the row, resulting 2550 configurations. Similarly, when the number of rows ranges from two to four, the number of configurations is 1270, 840, and 630, respectively. Therefore, there are 5290 possible configurations in total.

Appendix G LHS-based initialization for searching

Algorithm 5 LHS-based initialization

Input : The number of mapped sketch instances n
1: **function** LHS_INIT(n)
2: m = min(ALU_{max}, $Page_{max}$)
3: Initialize m empty application configurations $c_1, c_2, ..., c_m$
4: **for** i = 0 to n **do**
5: `alus` = [1, 2, ..., ALU_{max}]
6: `pages` = [1, 2, 4, ..., $Page_{max}$]
7: **for** j = 0 to m **do**
8: d = random sample in `alus`
9: Remove d from `alus`
10: w = random sample in `pages`
11: Remove w from `pages`
12: $c_j[2i] = d$; $c_j[2i+1] = w$
13: **return** the list of $c_1, c_2, ..., c_m$

Algorithm 5 details the LHS-based initialization. It takes the number of mapped sketch instances n as input and outputs a list of initial configurations sampled by LHS. The algorithm first initializes m empty configurations (line 3). Here, m depends on the minimal value between the available maximal stateful ALUs and memory pages (line 2), in that LHS has to ensure the parameter values of all samples in each dimension are different. As mentioned in §5.1, the application configuration is a vector of $2n$ variables ($c = (d_1, w_1, d_2, w_2, ..., d_n, w_n)$). Therefore, the algorithm fills in the d and w dimensions of all samples one by one in the order of the sketch instance (lines 4-12). For each sketch instance, the algorithm prepares two lists (`alus` and `pages`) of available resource values (lines 5,6). Then, c_1 to c_m iteratively pops the values in `alus` and `pages` randomly to fill in $c_j[2i]$ and $c_j[2i+1]$ (lines 7-12). In this way, the configurations meet the requirement of LHS.

Algorithm 6 Pseudo Code of Function CALCNEIGHBOR

1: **function** CALCNEIGHBOR(c)
2: `neighbors` = []
3: **for** each parameter p **in** c **do**
4: c' = copy of c
5: **if** c satisfys EB **then** ▷ decrease resource
6: **if** p is a row parameter **then**
7: $p' = p - 1$
8: **else**
9: $p' = p/2$
10: **else** ▷ increase resource
11: **if** p is a row parameter **then**
12: $p' = p + 1$
13: **else**
14: $p' = p * 2$
15: Replace p **in** c' by p'
16: Add c' to `neighbors`
17: **return** `neighbors`

Appendix H Hardware-aware configuration generation

Algorithm6 details the hardware-aware configuration generation process. It generates neighbors for a given configuration by changing one parameter in the configuration at a time (line 3). If the given configuration satisfies the accuracy intent, the algorithm calculates its neighbors by decreasing the resource usage (lines 5-9). Otherwise, it calculates its neighbors by increasing the resource usage (lines 10-14). The algorithm considers hardware characteristics. Specifically, when changing the number of rows, it uses 1 as the step size, and when changing the number of columns, it ensures that the number of columns is a power of 2.

Appendix I Additional Experiment

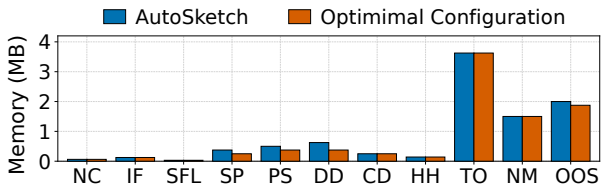


Figure 14: (Exp#10) Memory efficiency of searching results.

(Exp#10) Efficiency of searching results. We compare the resource usage between the configuration searched by AutoSketch and the optimal configuration found by brute-force searching. Figure 14 shows that the memory consumption of AutoSketch is slightly more than the optimal configuration: even the largest difference (DD) is only 0.25 MB. In additional, the searched results consume the same number of stages and at most one more ALU than the optimal ones (not shown in the figure). Such additional costs are acceptable.

Appendix J Complete Experiment Results

Table 4 and Table 5 list the complete experiment results of Exp#2 to Exp#4 in §7.2, including SSH Brute Force detection. Table 6 summarizes the complete switch resources and application accuracies of sketch-based solutions under two accuracy intents (AS-1 and AS-2).

Telemetry Applications	**Resources**																			
	Memory (MB)					**Stage**					**ALU**					**Bandwidth (MB/s)**				
	Marple | Sonata | EM | AutoSketch-1 | AutoSketch-2																			
New TCP Conns.	0.5	0.5	-	0.06	0.13	2	4	-	1	1	2	2	-	2	2	0.06	0.033	-	0.001	0.001
TCP Incomplete Flows	1.0	1.0	-	0.13	0.5	2	8	-	1	1	4	4	-	2	2	0.276	0.036	-	0.001	0.001
DDoS	1.0	1.0	-	0.63	0.75	3	8	-	2	2	3	4	-	4	6	44.24	0.037	-	0.0037	0.0036
Port Scan	0.88	1.0	-	0.5	1.0	3	8	-	2	2	4	4	-	3	4	23.56	0.07	-	0.0063	0.0063
SSH Brute Force	1.25	1.0	-	0.03	0.03	3	8	-	2	2	5	4	-	2	2	0.001	0.001	-	0.001	0.001
TCP non-monotonic	1.31	-	2.25	1.5	2.13	3	-	7	5	6	4	-	6	11	11	147.6	-	1.265	0.233	0.226
TCP out-of-sequence	1.31	-	2.25	2	2.88	3	-	7	5	7	4	-	6	13	12	167.3	-	1.692	0.526	0.517
TCP timeouts	1.31	-	2.25	3.63	3.63	3	-	7	8	8	4	-	6	12	12	34.4	-	0.609	0.136	0.136
SYN Flood	0.91	-	2.0	0.06	0.06	2	-	7	4	4	2	-	5	4	4	56.35	-	0.001	0.001	0.001
Total	10.47	4.5	8.75	8.54	11.11	24	36	28	30	33	32	18	23	53	55	473.8	0.177	3.57	0.909	0.893
Average	1.16	0.9	2.19	0.95	1.23	2.67	7.2	7	3.33	3.67	3.56	3.6	5.75	5.89	6.11	52.64	0.035	0.89	0.101	0.099

Table 4: Complete resource overheads of solutions in Exp#2 and Exp#3.

Telemetry Applications	**Accuracy**														
	Precision					**Recall**					**ARE**				
	Marple | Sonata | EM | AutoSketch-1 | AutoSketch-2														
New TCP Conns.	1.0	0.98	-	0.995	0.998	1.0	0.74	-	1.0	1.0	-	-	-	-	-
TCP Incomplete Flows	1.0	0.93	-	0.95	0.994	1.0	0.75	-	0.993	1.0	-	-	-	-	-
DDoS	1.0	0.92	-	0.976	0.996	1.0	0.34	-	0.996	0.996	-	-	-	-	-
Port Scan	1.0	0.93	-	0.956	0.981	1.0	0.41	-	0.982	1.0	-	-	-	-	-
SSH Brute Force	1.0	1.0	-	1.0	1.0	1.0	1.0	-	1.0	1.0	-	-	-	-	-
TCP non-monotonic	1.0	-	0.25	0.96	0.992	1.0	-	0.996	0.996	0.996	0.0	-	1.86	0.025	0.006
TCP out-of-sequence	1.0	-	0.37	0.98	0.99	1.0	-	0.99	0.998	0.999	0.0	-	1.39	0.016	0.005
TCP timeouts	1.0	-	0.66	0.95	0.95	1.0	-	0.62	0.972	0.972	0.0	-	0.38	0.028	0.028
SYN Flood	1.0	-	0.17	1.0	1.0	1.0	-	1.0	1.0	1.0	-	-	-	-	-

Table 5: Complete accuracy of solutions in Exp#4.

Solutions	**Accuracy Intent 1**						**Accuracy Intent 2**					
	Memory (MB)	**Stage**	**ALU**	**Precision**	**Recall**	**RE**	**Memory (MB)**	**Stage**	**ALU**	**Precision**	**Recall**	**RE**
HashPipe	0.563	9	16	0.99	0.195	-	0.63	12	18	0.989	0.464	-
MV-Sketch	0.375	6	12	0.958	0.995	-	0.63	8	20	0.989	1.0	-
FlowRadar-HH	5.91	11	24	0.989	0.903	-	5.91	11	24	0.989	0.903	-
OpenSketch-HH	0.578	7	8	0.976	1.0	-	0.95	8	8	0.986	1.0	-
AutoSketch-HH	0.14	3	3	0.98	0.999	-	0.27	3	3	0.997	1.0	-
SpreadSketch	0.375	6	6	0.972	0.659	-	0.63	7	10	0.99	0.695	-
Vector Bloom Filter	2.58	12	5	0.973	0.66	-	2.58	12	5	0.973	0.66	-
FlowRadar-SS	5.91	11	24	1.0	0.991	-	5.91	11	24	1.0	0.991	-
OpenSketch-SS	1.2	10	8	0.941	0.748	-	1.75	10	8	0.976	0.728	-
AutoSketch-SS	0.375	3	4	0.975	0.978	-	1.0	3	6	0.983	1.0	-
FM-Sketch	0.031	4	1	-	-	0.044	0.046	4	1	-	-	0.006
Linear Counting	0.031	1	1	-	-	0.017	0.031	1	1	-	-	0.004
FlowRadar-CD	5.91	11	24	-	-	0.011	5.91	11	24	-	-	0.011
OpenSketch-CD	0.031	1	1	-	-	0.017	0.031	1	1	-	-	0.004
AutoSketch-CD	0.25	2	2	-	-	0.011	0.19	2	3	-	-	0.009

Table 6: Complete experiment results of solutions in Exp#6 and Exp#7.

Appendix K Telemetry Applications

We show the function of each application in Table 7 and how to realize these twelve telemetry applications.

#	Application	Description
1	New TCP Connections	Count the number of newly opened TCP connections exceeds threshold.
2	TCP Incomplete Flows	Count the number of incomplete TCP connections exceeds threshold.
3	Port Scan	Detect the end-hosts that send traffic over more than threshold destination ports.
4	DDoS	Detect the end-hosts that receive traffic from more than threshold unique sources.
5	TCP timeouts	Count the number of timeouts for each TCP connection, by checking for packet inter-arrival times around 300 ms (retransmission timer).
6	TCP non-monotonic	Count the number of packets per connection with sequence numbers lower than the maximum so far.
7	TCP out-of-sequence	Count the number of packets per connection arriving with a sequence number that is non-consecutive with the last packet.
8	Superspreader	Detect the end-hosts that contact more than threshold unique destinations.
9	Cardinality	Count the number of flows in the network.
10	Heavy Hitter	Identify large flows that consume more than threshold during a time interval.
11	SYN Flood	Detect the end-hosts whose half-open TCP connections exceeds threshold.

Table 7: Description of telemetry applications.

```
n_syn[precision_min=95%,recall_min=95%,confidence
    =95%] = PacketStream()
    .filter(ipv4.protocol==TCP)
    .filter(tcp.flags==SYN)
    .map((ipv4.dstIP, count), count=1)
    .reduce((ipv4.dstIP), val=count)
    .filter(count>=Thld)
    .distinct((ipv4.dstIP))
```

Application 1: New TCP Connections

```
n_fin = PacketStream()
    .filter(ipv4.protocol==TCP)
    .filter(tcp.flags==FIN)
    .map((ipv4.srcIP, fin_cnt), fin_cnt=1)
    .reduce((ipv4.srcIP), val=fin_cnt)

diff[precision_min=95%,recall_min=95%,confidence
    =95%] = PacketStream()
    .filter(ipv4.protocol==TCP)
    .filter(tcp.flags==SYN)
    .map((ipv4.dstIP, syn_cnt), syn_cnt=1)
    .reduce((ipv4.dstIP), val=syn_cnt)
    .zip(stream=n_fin, (ipv4.srcIP))
    .map((ipv4.dstIP, diff), diff=syn_cnt-fin_cnt)
    .filter(diff>=Thld)
    .distinct((ipv4.dstIP))
```

Application 2: TCP Incomplete Flows

```
port_scan[precision_min=95%,recall_min=95%,
    confidence=95%] = PacketStream()
    .distinct((ipv4.srcIP, tcp.dport))
    .map((ipv4.srcIP, count), count=1)
    .reduce((ipv4.srcIP), val=count)
    .filter(count>=Thld)
    .distinct((ipv4.srcIP))
```

Application 3: Port Scan

```
ddos[precision_min=95%,recall_min=95%,confidence
    =95%] = PacketStream()
    .distinct((ipv4.dstIP, ipv4.srcIP))
    .map((ipv4.dstIP, count), count=1)
    .reduce((ipv4.dstIP), val=count)
    .filter(count>=Thld)
    .distinct((ipv4.dstIP))
```

Application 4: DDoS

```
def to(tin):
    if tin - last_ts > Thld:
        to_count += 1
    last_ts = tin
timeout[precision_min=95%,recall_min=95%,ARE_max
    =3%,confidence=95%] = PacketStream()
    .filter(ipv4.protocol==TCP)
    .groupby({5tuple:(last_ts, to_count)}, to))
```

Application 5: TCP timeouts

```
def nonmt(tcp.seq):
    if maxseq < tcp.seq:
        maxseq = tcp.seq
    else:
        nm_count += 1
nm[precision_min=95%,recall_min=95%,ARE_max=3%,
    confidence=95%] = PacketStream()
    .filter(ipv4.protocol==TCP)
    .groupby({5tuple:(maxseq, nm_count)}, nonmt)
```

Application 6: TCP non-monotonic detection

```
def oos(tcp.seq, payload_len):
    if lastseq != tcp.seq:
        oos_count += 1
    lastseq = tcp.seq + payload_len
tcp_oos[precision_min=95%,recall_min=95%,ARE_max
    =3%,confidence=95%] = PacketStream()
    .filter(ipv4.protocol==TCP)
    .groupby({5tuple:(lastseq, oos_count)}, oos)
```

Application 7: TCP out-of-sequence detection

```
super_spreader[precision_min=95%,recall_min=95%,
    confidence=95%] = PacketStream()
    .distinct((ipv4.dstIP, ipv4.srcIP))
    .map((ipv4.srcIP, count), count=1)
    .reduce((ipv4.srcIP), val=count)
    .filter(count>=Thld)
    .distinct((ipv4.srcIP))
```

Application 8: Superspreader

```
cardinality[ARE_max=3%,confidence=95%] =
    PacketStream()
    .distinct((5tuple))
    .map((5tuple, count), count=1)
    .reduce((), val=count)
```

Application 9: Cardinality

```
heavyhitter[precision_min=95%,recall_min=95%,
    confidence=95%]  = PacketStream()
    .map((5tuple, count),count=ipv4.totalLen)
    .reduce((5tuple), val=count)
    .filter(count>=Thld)
    .distinct((5tuple))
```

Application 10: Heavy Hitter

```
def remap_key(tcp.flag):
    if tcp.flag == SYNACK:
        nkey = ipv4.srcIP
    else:
        nkey = ipv4.dstIP
    return nkey
def sf(tcp.flag, tcp.seq, tcp.ack):
    if tcp.flag == SYNACK:
        nextseq = tcp.seq + 1
        cnt += 1
    else if nextseq == tcp.ack
        cnt -= 1
    return cnt
syn_flood[precision_min=95%,recall_min=95%,
    confidence=95%] = PacketStream()
    .filter(ipv4.protocol==TCP)
    .filter(tcp.flags contains ACK)
    .groupby(null, remap_key)
    .groupby({nkey:(nextseq, cnt)}, sf)
    .filter(cnt > Thld)
    .distinct((nkey))
```

Application 11: SYN Flood

Leo: Online ML-based Traffic Classification at Multi-Terabit Line Rate

Syed Usman Jafri *Purdue University*
Sanjay Rao *Purdue University*
Vishal Shrivastav *Purdue University*
Mohit Tawarmalani *Purdue University*

Abstract

Online traffic classification enables critical applications such as network intrusion detection and prevention, providing Quality-of-Service, and real-time IoT analytics. However, with increasing network speeds, it has become extremely challenging to analyze and classify traffic online. In this paper, we present Leo, a system for online traffic classification at multi-terabit line rates. At its core, Leo implements an online machine learning (ML) model for traffic classification, namely the decision tree, in the network switch's data plane. Leo's design is fast (can classify packets at switch's line rate), scalable (can automatically select a resource-efficient design for the class of decision tree models a user wants to support), and runtime programmable (the model can be updated on-the-fly without switch downtime), while achieving high model accuracy. We implement Leo on top of Intel Tofino switches. Our evaluations show that Leo is able to classify traffic at line rate with nominal latency overhead, can scale to model sizes more than twice as large as state-of-the-art data plane ML classification systems, while achieving classification accuracy on-par with an offline traffic classifier.

1 Introduction

In recent years, there have been increasing calls for a "self-driving" network [10–12, 22] , i.e., a network driven by real-time analytics performed on data at line-rate guided by programmatic control. Self-driving networks can help in tasks such as anomaly detection (e.g., identifying and isolating malicious traffic), and performance monitoring and repair (e.g., identifying flows that see sub-optimal traffic, and rerouting them). A key component of such self-driving networks is the need to run machine learning (ML) inferencing algorithms. For example, network security applications rely on ML algorithms for classifying applications and devices (e.g., IoT device) [24], and detecting anomalous patterns [15, 19].

Traditionally, ML inference algorithms today are run off the network path. For instance, Intrusion Detection Systems (IDS) typically require exporting network data to an off-path IDS device, which runs necessary ML models and flags anomalies. On the one hand, the bandwidth [27] needed to export data from routers is significant. On the other hand, reaction times to take action to resolve security issues is slow and of the order of minutes. Other systems such as Intrusion Prevention require individual packets to be sent off-path and only packets deemed safe are then forwarded. Doing so can incur significant latencies for all packets, which limits the wide-spread adoption of such intrusion prevention devices.

The recent advent of programmable switches offers the unique opportunity of running ML inference algorithms directly on the data plane [30]. For instance, a lighter weight ML model run on a switch could let the vast majority of traffic pass through the switch without delays, with only a smaller portion of the traffic routed to the control plane, where the traffic could be inspected with more sophisticated traffic models. While programmable switches offer promise, there are several challenges as well. First, switches are significantly resource constrained in terms of computation (ALU resources), memory, and pipeline stages. Second, while ML models change frequently, switches offer limited runtime programmability, with runtime updates often requiring a switch reboot.

In this paper, we take a key step towards running ML inference algorithms in the data plane. We focus on decision trees given they are widely used in traffic classification, are interpretable [9], and since they only require features (e.g., comparisons, conditions) that are already available on existing programmable switches. Further, although there is ongoing research on new hardware support for switches [25, 29, 31], our focus is on realizing decision trees on existing switches for ease of deployability considerations.

We make the following **contributions:**

• We analyze initial proposals [6,14,28,30,32], notably [6,30] to support decision trees on programmable switch pipelines. The majority of these approaches [6, 14, 28] follow the natural dependencies of a decision tree, implying they are bottlenecked by the number of stages in a switch for larger trees. While [30, 32] decouples the number of switch stages from tree dependencies, we show that its memory requirements are prohibitive whether SRAM or TCAM is used (§3).

• We present Leo[1], a system that provisions a programmable switch pipeline to support an entire class of trees (e.g., all trees with depth $\leq D$ and at most L leaves) in a runtime programmable manner. Leo exploits the fact that although the decision tree itself could be large, any individual packet only encounters a small subset of tree nodes. Further, Leo selectively flattens portions of a decision tree to judiciously trade off memory and ALU requirements on the one hand, and number of switch stages on the other. Leo achieves these goals using a sub-tree multiplexing mechanism which allows multiple sub-trees in a decision tree to be multiplexed using a layer of ALUs whose features and constraints can be programmed at runtime. This simultaneously allows programmability across trees, and reduces resource requirements for a given tree.

[1] Source code available at: https://github.com/Purdue-ISL/Leo

• We present worst-case bounds on resource requirements (e.g., table sizes) with Leo, to ensure it can sufficiently provision for all trees in a class. We show that unlike IIsy [30, 32] Leo achieves attractive worst-case bounds on memory requirements, while requiring significantly fewer stages than [6, 14, 28].
• Our evaluations on Intel Tofino [3] switches show (i) Leo support much larger models than existing approaches –e.g., it supports complete trees of depth 10 with SRAM, while existing approaches are limited to depth 5 trees; with TCAMs, Leo supports trees of depth 22 with up to 1024 leaves; (ii) on real IDS datasets, Leo achieves classification accuracies comparable to a control plane solution, and significantly better than accuracies achieved with prior work [6, 14, 32] and (iii) evaluations on a real programmable switch testbed show that classification latencies with Leo are 500X lower than performing classification in the router control plane. Overall, the results show Leo is a viable approach to support packet classification in the data plane.

2 Background and Motivation

ML for traffic classification. Consider Intrusion Detection and Prevention systems (IDS and IPS) systems, which match network flows to patterns to identify potential anomalies. IPS systems inspect individual data packets synchronously and take necessary preventive action on malicious traffic (e.g., block traffic). In contrast, IDS systems operate asynchronously over traffic exported from the data plane. While many IDS/IPS systems rely on rules based on packet payload, the increasing trend towards encrypted traffic has spurred interest in behavioral systems that do not rely on packet payloads. Instead these systems only rely on models of network flow statistics (e.g., packet size distributions, inter-arrival times etc.). Behavioral models may also serve as an initial coarse filter that flags potentially malicious traffic to a rule-based systems that performs more detailed inspection. Although we use intrusion detection and prevention as our primary motivating example, many other use cases such as IoT device identification, application classification, buffer sharing in switches, and QoE inference [1, 15, 19, 24] benefit from ML classification models.

Why traffic classification in data plane? With increasing network speeds, the amount of data that needs to be analyzed per unit time has also increased. For example, a single state-of-the-art switch can receive multi-terabits of data per second. Unfortunately, implementing traffic classification in the control plane or remote servers incurs high response latency, which might be unacceptable for synchronous systems such as IPS that require packet processing on the critical path. Our measurements on a real router testbed show that while it only takes few hundreds of nanoseconds to process a packet in the switch's data plane, it could take hundreds of microseconds to simply send a packet to the switch's local control plane and back. Further, the bandwidth of the data path between a switch's local control and the data plane is typically a few 10s of Gbps (implemented using PCIe bus). However, the data plane of state-of-the-art switches runs at multi-Tbps [21]. Thus, even for an asynchronous system such as an IDS, it is not possible to analyze and classify each packet going through the network using the control plane. The alternative is to heavily sample packets, or report digests over longer epochs, which can reduce accuracies and responsiveness. Using a remote server for analysis and classification suffers from the same limitations, namely high latency for classification and server bandwidth acting as bottleneck. These factors motivate us to explore running traffic classification in the data plane.

Programmable switches. Network data plane support for ML inference is facilitated by the recent emergence of programmable switches [5, 20, 31] which allow the architect to deploy programs that are executed on each data packet. Many programmable switches today follow the Protocol Independent Switch Architecture (PISA) data plane model and comprise a parser, processing pipeline, and deparser, which are each programmable [5]. The processing is done by a pipeline of a fixed number of *stages* [7, 8] that execute on every clock cycle. Each stage consists of:
• *Match tables,* which specify the packet header fields to match against, and the corresponding action (e.g., rewrite a packet header field). Match tables may be supported using (i) *SRAMs* which only support exact matches; or (ii) *TCAMs* which support wild card matches (e.g, all packets with source port 80, and arbitrary destination port). TCAMs are more flexible, but they are power hungry and more expensive.
• *Registers and ALUs.* Registers store small amounts of state that persists across packets (e.g., to implement packet counters). Stateful ALUs involve registers and allow computations across packets. Stateless ALUs only take other packet headers as inputs. Computation is limited to fields within a packet.

In addition, each switch consists of a Packet Header Vector (PHV) which contains packet header fields, and metadata used to communicate intermediate results across stages.

3 Design goals and prior work limitations

In this paper, we focus on decision trees for ML classification given that they are widely used, and are easy to interpret unlike "black-box" approaches such as neural networks. Further, the simplicity of a decision tree model (e.g., unlike neural networks, decision trees do not require complex ALU operations such as multiplication) makes them amenable for implementation on existing high-speed data plane architectures. In the rest of this section, we discuss our design goals in mapping decision trees to programmable switch data planes, and why prior attempts [6, 14, 28, 30, 32] fall short. Our **goals** are:

Support a class of models in a runtime programmable fashion. ML policy changes over time as new data is available and models are retrained. Unfortunately, existing switches offer limited runtime programmability (i.e., ability to make changes without rebooting a switch). For instance, while the

	Runtime Prog	Not limited by tree dependency	Implementable in ASIC switch	Low ALU usage	Low memory usage
Infocom [28]	✗	✗	✓	✗	✓
pForest [6]	✓	✗	✗	✓	✓
SwitchTree [14]	✓	✗	✗	✓	✓
IIsy [30, 32]	✓	✓	✓	✓	✗
Leo	✓	✓	✓	✓	✓

Table 1: Comparing Leo to prior work. §3 presents analysis to show the high memory requirements of IIsy, and we analyze Leo's resource requirements in §6.

entries of match tables can be modified at runtime by the control plane, the inputs of an ALU cannot. Ideally, any approach must support an entire class of decision trees without requiring a router reboot (e.g., all trees with a particular depth).

Support sufficiently large models for high accuracy. The accuracy of classification depends on factors such as tree depth and the number of leaves. Figure 1 shows for a real IDS dataset (details in §7) that deeper trees achieve higher accuracy despite using the same number of decision nodes.

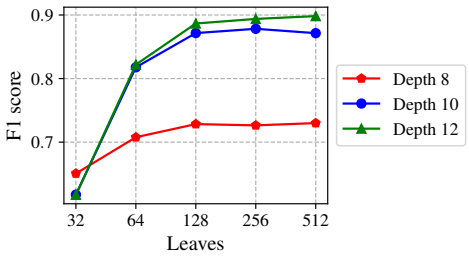
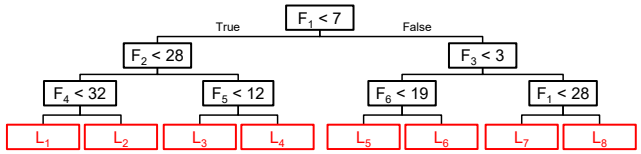


Figure 1: Impact of increasing depth for the same number of leaves using CICIDS-2017 dataset.

Resource efficient for all ML models in a class. The resources required in programmable switches must be acceptable in terms of resources such as ALUs, SRAM, TCAM, and the number of stages. Further, since we are provisioning for a class of ML models (e.g., all decision trees with a particular depth), the mapping must be such that the **worst-case** bounds for mapping for all models within the class is acceptable in terms of resource requirements. The worst-case bounds must scale acceptably, e.g., with the number of leaf nodes in a decision tree, or with tree depth.

Prior work and limitations. Unfortunately, existing approaches [6, 14, 28, 30, 32] to support decision trees in the data plane fall short of the above goals. First, the majority of these works [6, 14, 28] follow a natural strategy, which involves mapping the hierarchical structure of decision trees to the programmable switch pipeline. This requires at least one (and possibly more) stages per tree level. Consequently, tree depth is bottlenecked by the number of pipeline stages. The results are further exacerbated given our analysis above which indicates that sparse and deeper trees are more important to improve classification accuracy. Besides this central limitation, some of these works are not runtime programmable (e.g., [28] uses simple conditionals, and even a minor change would require rebooting the switch to deploy recompiled code), and have high ALU usage (e.g., [28] uses an ALU per decision tree node), while others [6, 14] have only been implemented on software BMv2 switches.



(a)

F1	Code 1
1	0
2	0
⋮	⋮
6	0
7	1
8	1
⋮	⋮
27	1
28	2
29	2
⋮	⋮
32	2

Similar features table for $F_2 \ldots F_6$ with 2 codes each

F6	Code 6
1	0
2	0
⋮	⋮
18	0
19	1
20	1
⋮	⋮
32	1

TCAM Code 1, Code 2, …, Code 6	SRAM # of entries	Leaf
00*0**	2^3	L_1
00*1**	2^3	L_2
01**0*	2^3	L_3
01**1*	2^3	L_4
1*0**0 2*0**0	2^4	L_5
1*0**1 2*0**1	2^4	L_6
1*1***	2^4	L_7
2*1***	2^4	L_8

(b)

Figure 2: Illustrating IIsy's approach for an example decision tree with 6 features each with a maximum value of 32. Feature F1 appears twice and hence its values are mapped to three code words, while other features are mapped to two code words. The final table has entries for all code word combinations. The rules are shown for a TCAM, but the number of entries needed with an SRAM is also shown.

While IIsy [30, 32] addresses many of these shortcomings, we analyze it extensively below, and show that its memory requirements can be prohibitive, whether SRAM or TCAM is used. Table 1 summarizes existing schemes. Leo addresses these limitations with a design that is runtime programmable, not constrained by tree dependencies, has acceptable memory and ALU usage, and implemented on a hardware switch. In the rest of the section, we show IIsy has unacceptably high memory requirements.

Analyzing memory requirements with IIsy. IIsy [30, 32] seeks to break intrinsic tree dependencies by (i) a table per feature which maps feature values into a smaller set of code words. The mapping is such that the final classification result is the same for all values of a feature that share the same code word, no matter what the values of other features are. A final classification table looks at every possible combination of code words across features, and maps them to a classification result. We present two results to show that IIsy's memory requirements grow exponentially with the number of features, whether SRAM or TCAM is used.

Proposition 1 *Consider the requirement that* `IIsy` *must support all trees with a depth D or lower, involving any subset of N pre-determined features with each feature taking values in the range* $[0,K]$ *in a runtime programmable fashion. Then, the total SRAM that must be provisioned to achieve this goal grows exponentially with N.*

Proof sketch. To derive conservative bounds on the size of the combination table, consider a complete decision tree of depth D where each feature appears in the same number of decision tree nodes. Let $I = 2^D - 1$ denote the number of internal nodes. The total number of decision nodes that involve each feature is $\frac{I}{N}$, requiring $\frac{I}{N}+1$ codewords per feature. Since the combination table includes combinations of all possible codewords associated with each feature, the total size is $(\frac{I}{N}+1)^N$ which is exponential in N. Further, each feature table requires K entries since each table explicitly enumerates all values of every feature. Thus, NK total entries is needed across the tables. In practice, some savings is possible owing to default rules (see §A.1). Accounting for this, we obtain the following conservative bound on the the total number of SRAM entries, where the first term is the requirement for feature tables, and the second term the requirement for the combination table.

$$N*(K-\lceil\frac{K}{2^D}\rceil)+\frac{2^D-1}{2^D}*(\frac{2^D-1}{N}+1)^N \qquad (1)$$

We next show `IIsy`'s memory requirements grow exponentially even with TCAMs, focusing on the combination table since this is most crucial to the analysis.

Proposition 2 *There exist a family of decision trees with* $O(N^2+NK)$ *leaves which require at least* $O(\lg^{N-1}(K-1))$ *TCAM rules with* `IIsy`*, where N is the number of features, and each feature has values that could range from* $1\ldots K$

Proof sketch. The proof is based on a family of decision trees shown in Figure 15 in the Appendix for the general case with N features $(F_1\ldots F_N)$ with each feature having values ranging from $1\ldots K$. To provide more intuition, Figure 3 shows a geometric representation for the special case with $N=3$, and $K=4$, where the highlighted cubes correspond to the leaves of the corresponding decision tree.

The intuition behind the tree construction is as follows. First, the decision tree nodes has leaves for each value of a feature F_j when all other features are at their maximum value K. This forces `IIsy` to use a distinct code word for each value of every feature. Next, the decision tree has leaf nodes which correspond to regions where some of the features can take multiple possible values. These nodes will require a large number of code word combinations with `IIsy` since it is forced to use a distinct code word for each feature value. For example, in Figure 3, `IIsy` requires $K=4$ code words for each feature (owing to the decision tree nodes shown by the small cubes). While the inner cube (A_2) which captures the region where all features are $< K$ corresponds to a single decision tree node, it would requires $(K-1)^3$ combinations with `IIsy`. Likewise, each of the regions $A_{1,2}$, $A_{1,1}$, and B_1 (where exactly one feature is K) correspond to single decision tree nodes but would each require $(K-1)^2$ different code word combinations. Even if a default rule were used for the inner cube, $3*(K-1)^2$ code word combinations must still be explicitly covered, which would require at least $3m^2$ TCAM entries where $m=\lceil\log(K-1)\rceil$. We defer the details of the proof for the general case to §A.2 along with models for estimating the size of the feature and combination tables that must be provisioned for a TCAM implementation.

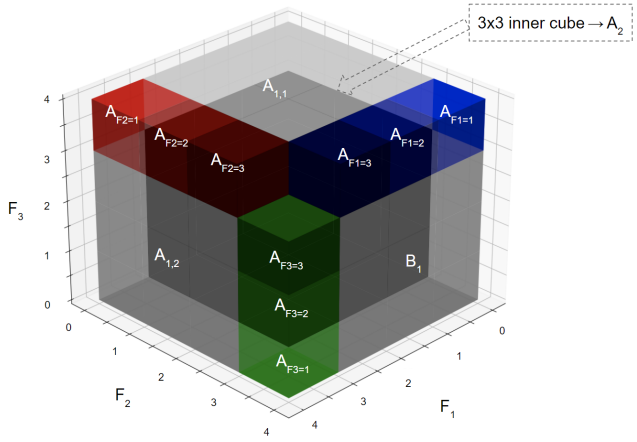


Figure 3: Visualizing general decision tree with three features.

Implications. Using Equation 1, a complete tree of Depth 6 using 9 16-bit features requires over 10^8 SRAM entries with `IIsy`. Using the analysis in §A.2. we can construct a tree with only 200 leaves that needs over 10^6 TCAM entries.

4 Leo Design

In this section, we present the design of `Leo`, a resource-efficient and runtime programmable system that enables data plane traffic classification on PISA switch pipelines.

4.1 Abstraction

Having a well-defined abstraction for a decision tree model is crucial to provide the right balance between flexibly switching between different decision trees at runtime and worst-case resource and performance overhead. The two extreme abstractions of supporting a *specific* decision tree and supporting *all* possible decision trees compromises either on flexibility or on resource constraint respectively. To that end, `Leo` provides the following abstraction for a decision tree model: ***to support any tree from a "class" of decision trees specified by the three tuple (D, L, F) where D is the maximum tree depth, L is the maximum number of leaves, and F is the set of features that the tree nodes can use, subject to the switch resource constraints while allowing for switching between trees of the same class at runtime without switch downtime.***

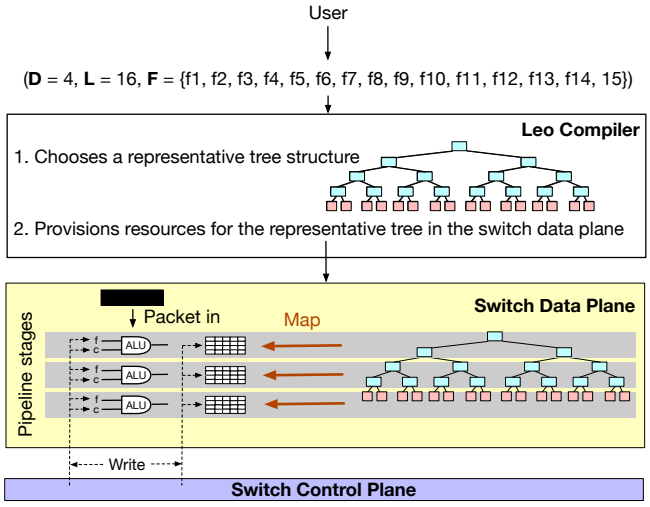


Figure 4: Workflow of Leo. Given a user specified (D,L,F) class, Leo chooses a representative decision tree for that class and maps it to the switch pipeline using multiplexed ALUs and SRAM/TCAM tables (Figure 9). The table inputs are configured at runtime by the switch control plane to implement different decision trees in the class (D,L,F) (Figure 16).

The user can specify a decision tree model to map to the switch pipeline using the tuple (D,L,F). Leo's compiler will try to find a mapping that can express *any* tree in the given class, subject to the switch resource constraints. However, the compiler will return a failed mapping if one or more trees in the given class could not be mapped within the resource constraints of the switch pipeline. If the mapping is successful, the user can choose any tree from the given class to run on the switch, and can switch between trees within the same class at runtime. Note that at any given time, the switch runs exactly one tree from the given class. The above abstraction is general enough to also express other common abstractions for a decision tree model. For example, $(D,2^D,F)$ can express all decision trees with depth D or smaller.

4.2 Design Overview

The workflow for Leo's design is shown in Figure 4.

Representative decision tree. Given a (D,L,F) specification, Leo provisions for a *single* decision tree structure at compile time that can be multiplexed at runtime to implement any tree in the class. We refer to this tree as a *representative decision tree*. A natural choice for a representative decision tree structure would be a complete tree of depth D (and 2^D leaves), as any tree structure with depth $\leq D$ would simply be a sub-tree of the complete tree. For example, in Figure 4, for the $(D=4,L=16,F)$ class, Leo's compiler chooses a complete decision tree of depth 4 as the representative tree structure. However, for classes where $L < 2^D$, this approach over-provisions the resources. In §6, we discuss how Leo reduces the resources to provision by also considering the maximum number of leaves L within the (D,L,F) class.

Mapping Representative Tree to Switch. Next, the compiler efficiently maps the tree structure to the switch pipeline. In doing so, it addresses two key challenges.

Challenge 1. The ability to update the features and constraints of the representative decision tree nodes at runtime to implement different trees in a (D,L,F) class.

Challenge 2. The ability to support decision tree classes with large D (depth) and L (leaves), subject to the resource constraints of the switch.

Leo solves these challenges with two novel techniques it introduces called *decision tree node multiplexing*(§4.3) and *sub-tree flattening and multiplexing* (§4.4)

Runtime reconfiguration. Finally, at runtime, the switch control plane configures the feature and constraint values in each node of the representative decision tree structure to implement a specific decision tree in the class (D,L,F) (§4.5).

4.3 Decision Tree Node Multiplexing

An internal node of a decision tree is implemented using an ALU whose inputs are a feature value $f \in F$ and a constraint c. The output is the result of a comparison (e.g., $f < c$). The set of features F are defined by the user as part of Leo's abstraction (D,L,F), and are stored either in the packet header vector (for stateless features, e.g., `TCP SYN flag`) or inside the switch registers (for stateful features, e.g., `packet count`). The constraints are stored in the switch memory and are populated (and updated at runtime) by the control plane.

To support different decision trees in a given (D,L,F) class, Leo allows that *any* internal node in the decision tree can take *any* of the F features and *any* constraint value as inputs for the conditional statements. To achieve this, Leo implements an internal node of the representative decision tree using a **multiplexed ALU**, an abstraction of which is shown in Figure 5. Conceptually, this comprises a (i) *feature multiplexer,* that can select any of the features at runtime from the set F; (ii) a *constraint multiplexer,* that can select any of the stored constraints; and (iii) an ALU that operates on the selected feature and constraint. We discuss concrete implementation in §5.

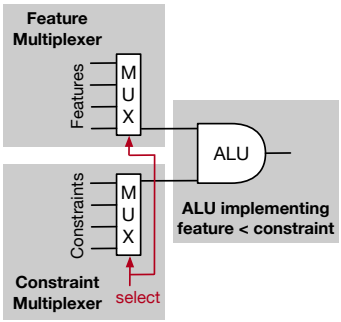


Figure 5: An abstract multiplexed ALU.

Supporting all conditional statements. At compile time, each node in the representative decision tree implements a statement of the form $f < c$. However, a decision tree can have other conditionals (e.g., $f > c$ or $f \leq c$). Unfortunately,

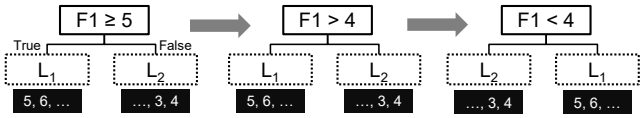


Figure 6: Transforming a $\geq$ node to an equivalent $<$ node.

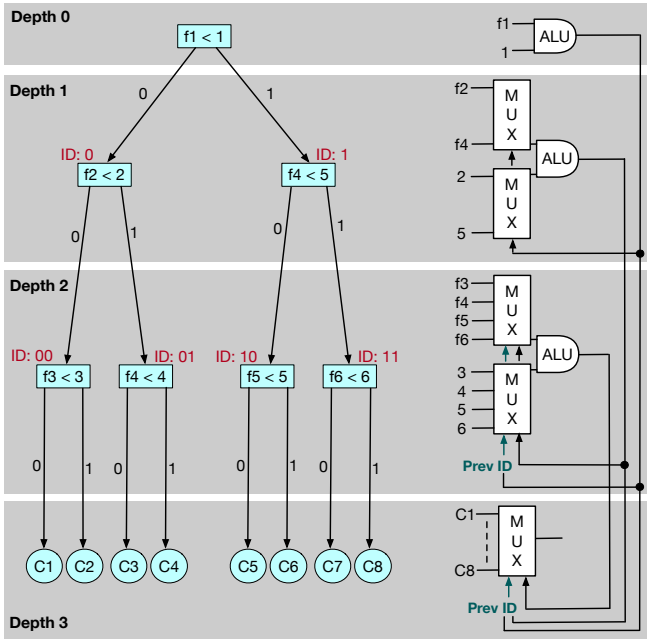


Figure 7: Mapping a decision tree to the switch pipeline using only one multiplexed ALU per tree depth.

the ALU operations in PISA pipelines cannot be changed at runtime. Instead, Leo transforms a more general decision tree into an equivalent version with only $f < c$ conditions offline (Figure 6). Specifically, (i) An $x \leq C$ condition is converted into $x < C+1$ while an $x \geq C$ is converted into $x > C-1$. (ii) $x > C$ is transformed into $x < C$ by simply swapping the left and right sub-trees of the node. Both these transformations maintain the behavior of the original conditions.

4.4 Sub-Tree Flattening and Multiplexing

A naive mapping of a representative decision tree structure to the switch pipeline would require a multiplexed ALU for each tree node implementing the conditional decision at that node. This can be as high as $2^D - 1$ ALUs to support a complete representative decision tree of depth D. However, we observe that at runtime a given packet would access *exactly one* tree node at a given tree depth. The tree node that the packet accesses at depth d would be decided by the path the packet takes through the tree, which, in turn, can be decided by the combination of the **{ID, decision output}** of the decision tree node accessed at depth $d-1$. For example, in Figure 7, if the ID of the node accessed at depth 1 was 0 and the decision output of that node was 1, then the node that will be accessed

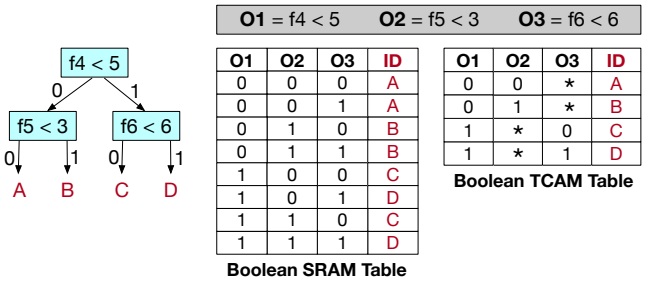


Figure 8: Encoding a decision sub-tree using a boolean table. The boolean table could be stored either in SRAM or TCAM.

at depth 2 will have ID 01. To implement this (Figure 7), Leo only needs to provision one multiplexed ALU per tree depth d. The combination of the node output and node ID at depth $d-1$ (shown as **Prev ID** in Figure 7) is used as the select value for the MUX at depth d to configure the ALU with the correct decision tree node parameters (feature and constraint). Hence, for a D depth tree, only D ALUs are needed.

While the above design requires significantly fewer ALUs compared to the naive design, unfortunately, it does not scale to trees with larger depths. This is because we require one ALU for each tree depth, and the ALU at a given depth could be configured only once we have the outputs of the ALUs from the lower tree depths. This inherent dependency means that ALU at each depth must be mapped to a different switch pipeline stage, thus requiring $D+1$ pipeline stages to implement a tree of depth D (Figure 7). In programmable switches, the number of pipeline stages can be as few as $\sim$10. To overcome this bottleneck, Leo uses two key insights.

First, Leo generalizes the unit of mapping from a single decision tree node as above to a sub-tree of size k nodes. Further, Leo notes that a sub-tree of size k within a decision tree can be represented using a **boolean table** by encoding all possible combinations of the outputs of the decision nodes within the sub-tree. This is illustrated in Figure 8. Hence, assuming sufficient resources per switch pipeline stage, one could implement an entire sub-tree of size k by calculating the output of each decision node in the sub-tree in parallel (using k multiplexed ALUs), and next matching those outputs against the boolean table(s) to figure the output of the entire decision sub-tree. We call this **sub-tree flattening**.

Figure 9 illustrates the idea where the representative tree is partitioned into layers, each layer comprised of similar sub-trees. Using similar insights as earlier, we observe that for any layer, exactly one sub-tree is executed at runtime as decided by the path the packet takes through the tree. For example, in Figure 9, exactly one of the four sub-trees (of size $k=3$) in layer 2 will be executed. Leo generalizes the idea of node multiplexing to **sub-tree multiplexing**, by provisioning for exactly one sub-tree per layer. Given a sub-tree multiplexing unit of size k nodes (which is always a complete binary tree

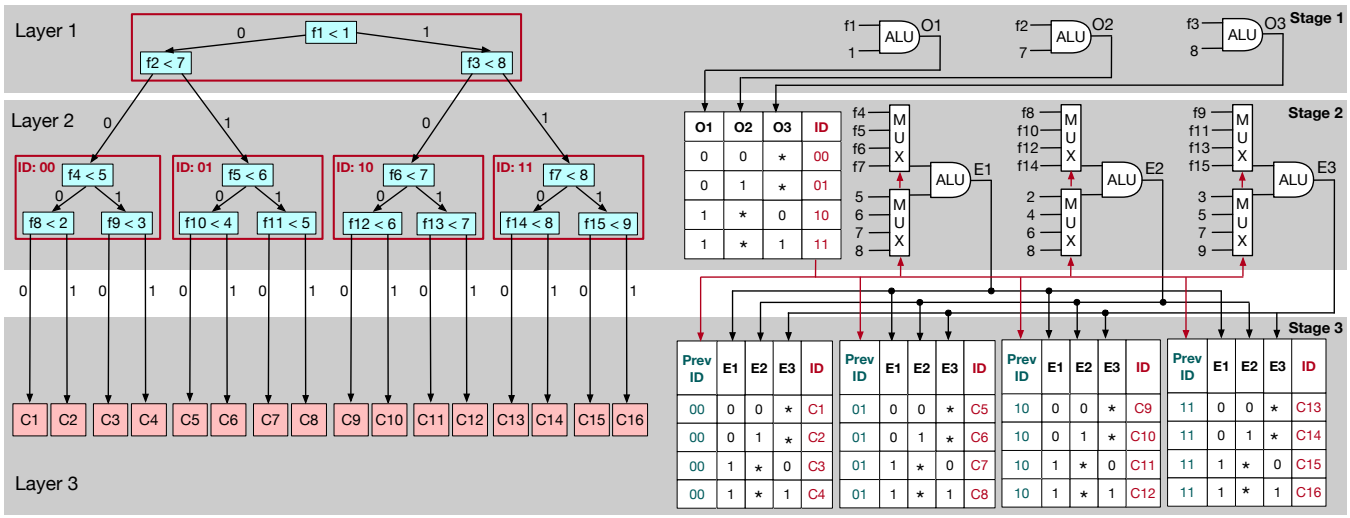


Figure 9: Illustrating sub-tree flattening and multiplexing (with sub-tree size $k = 3$) for a representative (complete) decision tree of depth $D = 4$ from Figure 4. As mentioned in §4.4, it requires $\lceil D/\lceil log_2(k)\rceil\rceil + 1 = 3$ switch pipeline stages to map the tree.

in `Leo`) and n such blocks of sub-trees at layer i, `Leo` only provisions for one sub-tree of size k (i.e., k multiplexed ALUs). In Figure 9, there are $n = 4$ blocks of sub-trees of size $k = 3$ in layer $i = 2$. Hence, `Leo` only provisions $k = 3$ multiplexed ALUs for $i = 2$. While the figure shows all layers having $k = 3$, `Leo` allows for different layers to be associated with different sized sub-trees (and hence k may vary across layers).

Finally, similar to Figure 7, to decide which sub-tree (or leaf node) at layer i to choose, `Leo` requires the ID and the outputs of the sub-tree at layer $i-1$. The combination of the **{ID, decision outputs}** from layer $i-1$ is fed to the feature and constraint multiplexers at layer i to choose the right feature and constraint value for each multiplexed ALU. Note that in Figure 9, we don't need the ID of the node at layer 1 to configure the multiplexers at layer 2; only the decision outputs of the sub-tree at layer 1 (shown as **O1, O2, O3** in the figure) are needed. This is because there is only one sub-tree at layer 1. However, at layer 2, we have 4 sub-trees, and hence in Figure 9, the leaf node at layer 3 is selected using both the ID of the sub-tree accessed at layer 2 (shown as **Prev ID** in the figure) plus the decision outputs from that sub-tree.

Using complete sub-trees of size $k > 1$ nodes as the unit of flattening in every layer, `Leo` only requires $\lceil D/\lceil log_2(k)\rceil\rceil + 1$ pipeline stages to implement a complete decision tree of depth D, as illustrated in Figure 9.

4.5 Runtime Programmability

The inputs to the building blocks of a representative decision tree in `Leo`, namely the feature and constraint multiplexers and the boolean table, can be configured and updated at runtime via the switch control plane without any switch downtime. Thus, by changing the inputs to these building blocks appropriately at runtime, one could implement any decision tree within a given (D, L, F) class. (Figure 16 in the Appendix presents an example of a different tree mapped to the same representative structure in Figure 9).

Handling transient state during runtime tree updates. One key issue with using the control plane for runtime tree updates is that it can take several clock cycles to update all the tree nodes while transitioning from one decision tree to another. Hence, packets arriving during the update might encounter an inconsistent tree state. To handle this, `Leo` maintains two copies of the representative decision tree. At any given time, exactly one of those trees is marked as active, and all incoming packets are directed through the active tree. In order to transition to a new decision tree, the control plane configures the nodes of the inactive decision tree, and once the new tree is configured, it is marked as active, while the previously active tree is marked inactive. Switching the status of a tree from active to inactive and vice-versa is an atomic operation, as it only requires writing to a single register entry storing the tree ID of the active tree. Thus, future incoming packets smoothly transition to the new decision tree without encountering any inconsistent state. The approach doubles the switch resources required to support a representative decision tree, but we show in §7 that the costs are acceptable.

5 `Leo` Implementation

We present details of our implementation in PISA ASICs.

Implementing Multiplexed ALUs. Figure 9 shows the feature and constraint multiplexers, and the Boolean SRAM/TCAM table as distinct components. We however implemented all of them using the same Match/Action table (MAT), with the comparison performed using the stateless ALU in the action field. For concreteness, layer 2 of Figure 9 is implemented as 3 MATs, one per Multiplexed ALU, which respectively pro-

duce the results $E1$, $E2$, and $E3$. The MAT which outputs $E1$ has $F+2$ possible actions. F of the actions (one per feature) involve (i) executing a condition of the form $E1 = f < c$, where f is an appropriate feature in the packet's header or metadata, and c is the relevant constraint loaded in the table memory as an action parameter; and (ii) assigning the ID field for the next layer based on another action parameter. The other two actions include a (i) `NoOp`, which simply disables the multiplexed ALU when a tree of smaller depth is installed; and (ii) `SetLeaf`, which terminates a path in the tree by writing the class label to the packet header. The MATs that compute $E2$ and $E3$ are similar, except that they do not compute the ID field (it suffices one MAT in each layer computes the ID field for the next layer).

Ideally, the operation $E1 = f < c$ is realizable in the stateless ALU. However, we were unable to compare a memory-read parameter (i.e. the constraint) with a header/metadata (i.e. a feature) in a single processing stage. We instead implemented the comparison as $E1 = f - c$ in a single stage. The condition is met if E1 is negative which involves checking that the Most Significant Bit (MSB) of $E1$ is 1. We achieve this with TCAMs using wildcard matching (of the form $1***$) for $E1$ in the next layer Boolean Table, realizing each layer in a single switch stage. With SRAMs, we used an extra stage to extract the MSB of E1, needing two stages per layer.

Choice of k. While larger k can reduce the number of stages, we typically used $k = 3$. Increasing k beyond 3 is constrained by the limit that PISA switches place on the number of header fields within each container group. Two header fields can be used together in an ALU operation only if they are in the same group (e.g., in Figure 9, the different features, and the outputs O and E of each layer are part of the same group). We reuse output headers with $k = 1$. When $k > 1$, neighboring layers must use different output headers as multiple MATs in a layer rely on the output headers of the previous layer. Hence, we reuse across alternate layers (odd layers use inputs E and outputs O, with the opposite for even layers), and our implementation requires $|F| + 2k$ header fields where $|F|$ is the number of features. An increase in k reduces the number of features possible. A potential trade-off that we defer for future investigation is to implement each layer in two stages: (i) a first that determines an ID; and (ii) a second that uses the ID for the multiplexed ALUs. This would reduce the number of header fields in each container group to $|F| + k$ and allow more features, at the expense of more stages per layer.

Handling stateful features. Stateful features requires allocating per-flow memory. We augment the `Leo` abstraction to include a target on the number of flows M that are to be supported. `Leo` maps stateful features to 8-bit or 16-bit registers based on the feature. Features may be discretized to fit the desired budget – e.g., packet lengths are divided by 64 (using bit shifts) which allows lengths ranging from 64 bytes to 16 KB to be represented in 64 byte units using a 8-bit budget. We discuss how other features are discretized in §A.3.3).

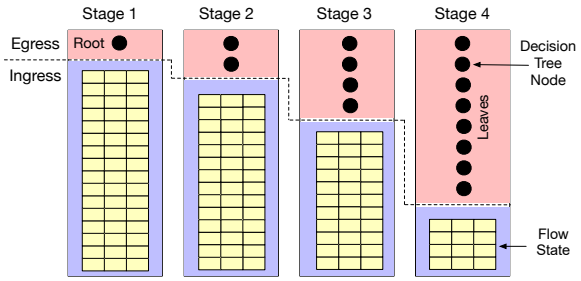


Figure 10: Illustration of how `Leo` maximally utilizes the resources in each physical stage by mapping the feature tables to the ingress pipeline and decision tree to the egress pipeline.

We store stateful features in separate SRAM register tables indexed by flow IDs. The flow ID is calculated as a hash over its 5-tuple. In PISA switches, each packet first traverses through an ingress pipeline followed by an egress pipeline. Each physical pipeline stage serves as both an ingress and an egress stage, with its resources shared between the ingress and egress. In `Leo`, we map the tables containing flow state to the ingress pipeline and the decision tree to the egress pipeline (Figure 10) to maximally utilize pipeline resources. This is motivated by the fact that when mapping the decision tree, many stages consume minimal memory resource (especially the first few stages corresponding to the top tree levels with fewer internal nodes). Thus `Leo` could use the remaining resources in those physical stages to store flow state in the ingress. Alternatively, if we had mapped both the flow state and the decision tree to the ingress pipeline, then the inherent dependency that the flow state must be accessed before the decision tree logic would mean that `Leo` could not use the resources left in each stage after mapping the decision tree nodes. This optimization is useful in scaling `Leo` to a large number of flows with stateful features (§7).

6 `Leo` Analysis

We next discuss how to provision a representative tree in `Leo` that can support all trees in the (D, L, F) class. This requires getting an upper bound on the requirements for every resource across *all possible trees* in that class.

The number of rules in each `Leo` table depends on the number of tree states that are multiplexed. For example, in Figure 9, each MUX in layer 2 (resp. layer 3) would multiplex across 4 (resp. 16) possible states. Let k_i denote the subtree multiplexing unit size in layer i (generalizing the above discussion which assumed $k_i = k$ for all layers). Then, each execution layer produces $k_i + 1$ possible outcomes for each of the states multiplexed by the layer. Thus, the maximum number of states multiplexed in layer i (which we denote by R_i) is at most $R_{i-1} * (k_i + 1)$. Given a (D, L, F) specification, the total number of leaves, and hence the number of internal nodes at each tree level does not exceed L. Thus, the number of states multiplexed by any layer is at most L, reducing the

entries to be provisioned. We now have:

$$R_i = \begin{cases} 1 & \text{if } i \leq 1 \\ min(L, R_{i-1} * (k_i + 1)) & \text{otherwise} \end{cases}$$

If TCAM is used, the number of entries needed in each MUX in layer i is simply R_i, the number of multiplexed states. If an SRAM is used, the number of entries is $R_{i-1} * 2^{k_{i-1}}$ as it depends on the multiplexed states in the previous layer, and the possible combinations of the k_{i-1} outputs of layer $i-1$. Since k values are typically small, this is just a small constant factor of the TCAM rules.

Total memory requirement. When $k_i = k$ in each layer, and we provision for complete trees, the total entries to provision for all tables across all layers (there are k tables per layer) is $(k+1)I+1$ with TCAM and $2^k I+1$ with SRAM. Here, I is the number of internal nodes = 2^D-1. When $k=3$ as in Figure 9, the total entries are $4I+3$ with TCAM and $8I+3$ with SRAM. In contrast, Equation 1 shows for IIsy, the SRAM entries goes as $(I/N+1)^N$, where N is the number of features, while TCAM entries are also exponential in N when I is polynomial. Further, when given a limit L on the number of leaves, and for $k=3$, a comfortable upper bound in memory requirements is $1+DL/2$ with TCAM, and $1+4DL$ with SRAM, with the more general expression being $1+\frac{DL}{log(K+1)}$ with TCAM, and $1+\frac{2^k DL}{log(k+1)}$ with SRAM.

7 Evaluation

Our evaluations address several questions: (i) What classes of decision trees can be supported by Leo, and how does this compare to other approaches in terms of tree size, the number of stages and rules? (§7.1). (ii) What classification accuracies are achievable on real traffic data-sets using decision trees that Leo can support? (§7.2). (iii) How does the decision tree accuracy vary with the number of flows and stateful features? (§7.3). (iv) What are the benefits of implementing decision trees in the data plane? (§7.4).

To answer these questions, we implement Leo, and other schemes using P4 [4], and analyze resource requirements on a Tofino switch. We also evaluate Leo with a control plane solution in a real tested involving a Tofino switch. We evaluate accuracies on publicly available intrusion detection datasets.

7.1 Leo vs. Other Data plane Tree Schemes

Schemes compared. We compare Leo with tree-based approaches [6, 14, 28] and IIsy [30, 32]. Unfortunately, as discussed in §3, existing tree-based approaches are either not runtime programmable [28], or have only been implemented on software switches. For instance, we experimented with the publicly available code of SwitchTree [14] (an extension of pForest [6]) and found it does not compile on the TNA [3] hardware switch. Instead, we modify Leo to mimic these approaches. Our implementation contains several optimizations not present in [6, 14] and is an optimistic bound on the performance of these approaches. Unlike Leo, none of the prior works guarantee correct decision tree execution during a transient period when a tree update is in progress. Hence, we refer to them as pFor/SwTree-NT and IIsy-NT. We also implement a variant of Leo where we disable the mechanisms to ensure correct transient performance, which we call Leo-NT.

Methodology and metrics. We compare schemes with respect to their ability to support all trees within a given class (D, L, F) that constrains the maximum depth D, maximum number of leaves L, and feature set F. We vary the class specifications across our experiments. For any given class, and for all schemes, we must estimate the memory to be provisioned at compile time for each table used in the scheme. This is needed since the goal is to support all trees within a class in a runtime programmable manner and since table sizes cannot be dynamically changed. This requires estimating the worst-case requirement across all decision trees within a class for each table with every scheme. For Leo and pFor/SwTree, we obtain these requirements using the analysis in §6. IIsy does not discuss how to assign sizes to its tables. Thus to guarantee programmability within each class, we utilize the analysis that we presented in §3 (and detailed in §A.1 and §A.2).

Our main metrics include (i) whether a given specification can be realized on an actual hardware switch (we focus on the Tofino Native Architecture (TNA) [3] used in Intel's Tofino line of switch ASICs); (ii) when the specification can be realized, the total number of SRAM or TCAM entries and the number of stages that the scheme requires. We implement all schemes using $P4_{16}$ targeting TNA and consider a specification met on successful compilation. All P4 code was compiled using version 9.11.1 of the Intel Barefoot SDK.

We focus on the resource requirements of the decision tree alone, and explore per-flow state requirements in §7.3. For Leo and pFor/SwTree-NT, the number of features is limited by the number of headers within each container group (§5). pFor/SwTree-NT can support a maximum of 15 features, while Leo can support upto 10 features for $k=3$, and 15 features for $k=1$. With Leo and pFor/SwTree-NT, the memory requirements for the decision tree logic does not grow with the number of features (§6). In contrast, the memory requirements of IIsy-NT grows exponentially with the number of features (§3). In the rest of this section, we compare resource requirements by keeping the number of features fixed with all schemes. §A.3.1 analyzes the maximum tree depth that can be supported as the number of features change. Finally, our classification accuracy experiments (§7.2) explore different combinations of features and depth for each scheme, and report the best operating point for that scheme.

Results with SRAM. Figure 11(a) presents results for classes of the form $(D, 2^D, F)$, which indicates that all trees of depth D or lower must be supported. We varied D while keeping the number of features fixed at 10. All schemes were constrained

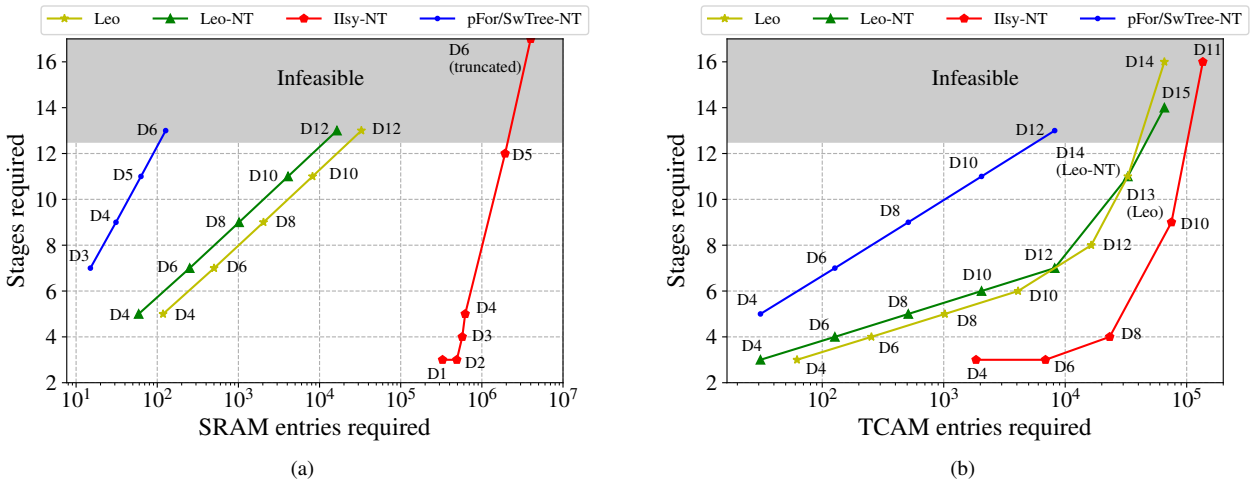


Figure 11: Table entries required to support a programmable class of complete decision trees using SRAM and TCAM. Designs needing more than 12 stages are infeasible on a Tofino switch [8]. Note the logarithm scale on x-axis.

to only using SRAM. First, `pFor/SwTree-NT` can only meet the specification upto $D = 5$, and it is unable to support depth 6 trees. This is because the approach is constrained by the number of switch stages. Second, `Leo` is able to support trees twice as deep as those supported by `pFor/SwTree-NT`, and can support all depth 10 trees. Further it requires half as many stages for the same depth. While the total memory requirement is higher, it is modest in an absolute sense, and this is an acceptable trade-off. Third, the memory requirements with `IIsy-NT` [30, 32] are several orders of magnitude more than `Leo`. Supporting all depth 5 trees requires over 1.9M entries, which is 10000× more than `Leo` (which requires 182 entries). `IIsy` is unable to support trees greater than depth 5 owing to the memory requirement. Finally, although `Leo` uses twice as much SRAM than `Leo-NT`, the table sizes are still small, and there is no impact on the classes of trees supported.

Results with TCAM. Figure 11(b) presents similar results as above but now implementing all tables in all schemes with TCAM. First, all schemes can support larger trees with TCAM, but `Leo` still outperforms – `pFor/SwTree` and `IIsy` can support depth 10 trees, but `Leo` can support any depth 13 tree. Second, for the same specification, `Leo` requires half as many stages as `pFor/SwTree`, and an order of magnitude fewer TCAM entries (for depth 10, the requirement with `IIsy` is 18× that of `Leo`). Third, the higher memory requirements of `Leo` limit it to depth 13 trees (versus depth 14 for `Leo-NT`), owing to both lower TCAM memory available per stage, and the larger memory requirements at higher tree depth. However, `Leo` can still support larger trees with a bound on the number of leaves with negligible impact on classification accuracy as we will see later. Finally, `Leo` and `pFor/SwTree-NT` require fewer stages for the same depth with TCAM relative to SRAM. This is because of our optimizations with TCAMs to avoid an extra computation stage for each layer (§5).

Exploiting bounds on the number of leaves. While the earlier results indicate `Leo` (resp. `Leo-NT`) can support all complete trees up to depth 13 (resp. 14), we next explore specifications that bound both the number of leaves and tree depth. This is beneficial given our analysis in Figure 1, which shows that accuracy greatly improves with tree depth but is less sensitive to the number of leaves for a given depth. Figure 13 shows the maximum number of leaves that can be supported for different tree depths, and indicates that `Leo` (and `Leo-NT`) can support deeper trees given a constraint on the number of leaves. For instance, `Leo` can support depth 22 trees given a specification that the tree has at most 1024 leaves. This is because `Leo` provisions fewer TCAM entries in each layer exploiting knowledge of the leaf constraint. The maximum number of leaves reduces with tree depth because for larger depths there are fewer available stages (and hence TCAM table space). In contrast, `IIsy`, and `pFor/SwTree` can support at most depth 10 trees as discussed earlier. While `Leo` allows fewer leaves than `Leo-NT` (see Appendix for more discussion), the number of leaves is still high and there is practically no impact on classification accuracy (§7.2).

7.2 Classification Accuracy on Real Datasets

Intrusion detection datasets. We used two publicly available and widely used intrusion detection datasets: (i) CICIDS-2017 (`CICIDS`): The dataset [19] consists of multiple attack classes along with a single benign class. Since some attack classes have very little data, we merge the 9 least populated attack classes into a single class resulting in a total of 7 classes. The dataset consists of 78 flow and packet level features. However, not all of these can be deployed on the hardware (e.g. means and percentiles that require division, etc.). Thus, we only select 42 switch-compatible features for our evaluation; and (ii) UNSW-NB15 (`NB15`): This dataset [15] also consists

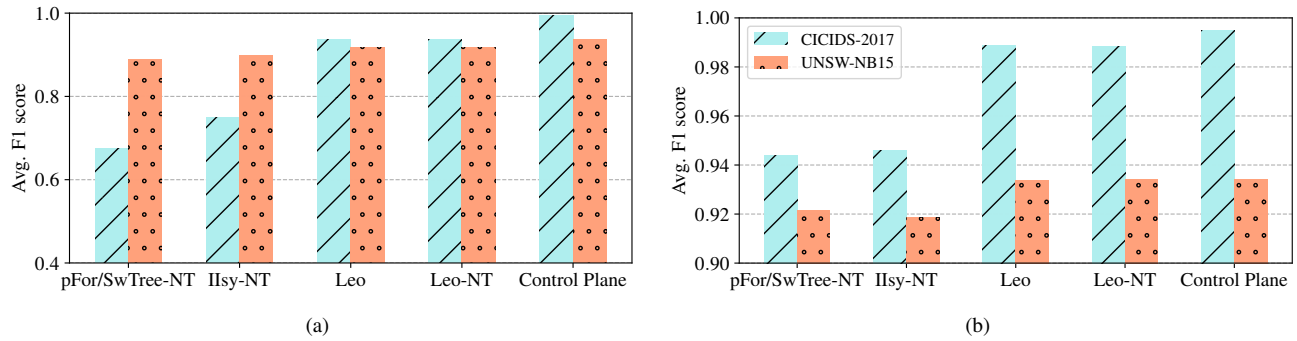


Figure 12: Classification accuracy of the best performing, hardware-supported tree class for each scheme when implemented using (a) SRAM and (b) TCAM. We assume all switch stages are available for the decision tree logic, and do not account for memory requirements of stateful flow features for all schemes (we explore impact of flow state in §7.3.) The control plane implementation is independent of SRAM and TCAM, and hence the accuracy is the same across figures (a) and (b) – Note the y-axis has different scale in figures (a) and (b).

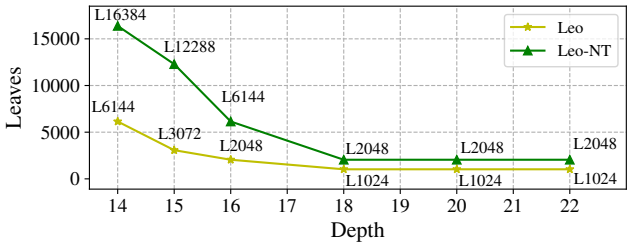


Figure 13: `Leo` can support tree classes beyond depth 13 with a bound on the number of leaves.

of real user traffic interspersed with generated attacks. We use this dataset to classify traffic as malicious or benign. It contains 49 flow and packet level features, out of which we use 22 switch-compatible features.

Training. We use Python3's scikitlearn library to train the decision trees. The CART algorithm was used along with the Shannon entropy loss function. We enabled class weighting to alleviate issues due to imbalance. To find the best set of features for an experiment, we use the Mean Decrease in Impurity (MDI) score and eliminate one least-scored feature recursively until we arrive at the target number of features for an experiment. We experimented with Permutation Importance score and found no difference in accuracy compared to MDI. 75% of the dataset was used for training, and 25% for testing. All results are reported on the test set.

Metrics. We report the average of the per class F1 scores which ensures the accuracy for all classes including those with relatively low samples is considered. Given the highly imbalanced nature of the dataset (most samples are benign), alternative metrics that weigh by the number of samples provide a highly optimistic view of accuracy for all schemes. For each scheme, we evaluated a range of (D, L, F) tree classes that are supported on the switch, we then selected the best performing class for each scheme and present them in Figure 12. Our comparisons include an idealized scheme (Control Plane) which gives an upper bound on the accuracy of the decision tree model for each dataset (by allowing large trees that use all dataset features).

Results. For `CICIDS` and SRAM, `Leo` outperforms both `IIsy-NT` and `pFor/SwTree-NT` achieving an average F1 score of 0.94 which is much higher than the 0.75 of `IIsy-NT` and 0.67 of `pFor/SwTree-NT`. With TCAM, `IIsy-NT` and `pFor/SwTree-NT` support depth 10 trees and thus improve to a 0.94 F1 score, while `Leo` outperforms both by managing a 0.98, due to its ability to support a depth 22 tree.

For the `NB15` dataset, using SRAM `Leo`, outperforms both `pFor/SwTree-NT` and `IIsy-NT` by achieving an average F1 score of 0.92 (versus 0.88). Using TCAM, `Leo` continues to outperform the others while managing to achieve identical accuracy as the control plane. Notice, that all schemes perform better with the `NB15` dataset because of the smaller number of classes. The Appendix presents a breakdown of the F1 score per class for all schemes. We also show results for the `NB15` dataset with multiple classes. Our results show similar trends – `Leo` outperforms `IIsy-NT` and `pFor/SwTree-NT` for each class while performing close to the control plane solution.

7.3 Number of Concurrent Flows Supported

Stateful per-flow features must fit into the memory budget of the switch (§5). Supporting more flows reduces the bits per flow (and which features can be supported), thereby impacting classification accuracy. We explore these issues next.

Methodology and metrics. We experimentally determine the maximum switch memory that can be configured as registers. Given a target M on the number of flows, we determine the budget for per-flow state, and then find different configurations of stateful features that can meet the budget. E.g., a 24-bit budget can be configured as 16-bit * 1 + 8-bit * 1 or as

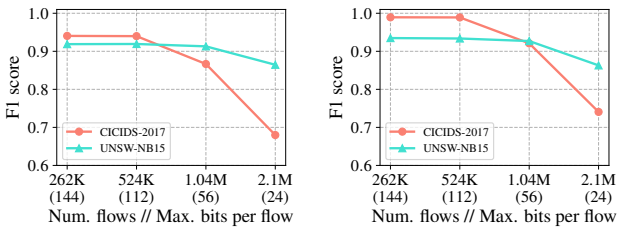


Figure 14: Impact of per-flow state on Leo classification accuracy using (a) SRAM and (b) TCAM.

16-bit * 0 + 8-bit * 3. For each valid configuration, we rank features by their MDI feature importance, and greedily select the top features that fit the configuration. We compute the F1 score achieved for each valid configuration and report the F1 score of the best performing configuration.

Results. Figure 14 summarizes how classification accuracy varies with the number of flows for SRAM and TCAM respectively. The X-Axis corresponds to the number of flows to be supported, and the corresponding budget on per-flow state (e.g., supporting 1.04 M flows imposes a budget of 56 bits per flow). The leftmost points match accuracy levels shown in Figure 12. Leo can support 1.04M concurrent flows with modest degradation, with average F1 score decreasing from 0.94 to 0.86 for SRAM, and decreasing from 0.98 to 0.92 for TCAM and CICIDS. For NB15, F1 scores decrease from 0.92 to 0.91 for SRAM, and are practically unchanged for TCAM. This is because we use the dataset for binary classification of traffic as benign or malicious, an easier classification problem. Table 2 in the Appendix summarizes the features used when supporting 1.04M flows for all 4 configurations.

7.4 Leo vs. Control Plane ML

To evaluate the benefits of implementing decision trees in the data plane versus control plane, we deploy Leo on an Intel Tofino [3] EdgeCore Wedge-100BF-32x [17] switch. We train decision trees of varying depths using the CICIDS dataset allowing the control plane to utilize the complete set of 78 features while restricting Leo to 10. We use tcpreplay to replay packets from select flows from the dataset to ensure similar characteristics of the original traffic are maintained. Besides classification, the switch also implements L2 forwarding. We capture timestamps at the switch ingress and egress to accurately measure the processing delay.

The control plane experiment involves forwarding the packet to the switch CPU for inference with no other functionality in the data plane (besides IPv4 match-based forwarding) while the data plane experiment uses Leo.

We find (see Figure 23 in the Appendix) that Leo is on average 500× faster than the control plane implementation of the same decision tree class. Leo takes about 500 nanoseconds to completely apply a 1024-leaf tree for inference to every packet. This re-asserts the point that a control plane inference scheme is not capable of per-packet classification at multi-terabit line rates. On the other hand, Leo can not only do per-packet classification, but also achieve accuracy comparable to an idealized control plane scheme for real datasets (§7.2).

8 Related Work

Beyond [6, 14, 28, 32]. N3IC [23] implements a binary neural network in the NICs with the goal of accelerating traffic analysis at the edge rather than per-packet in-network traffic analysis. Taurus [25] extends PISA pipelines with a new hardware module implementing a map-reduce abstraction to run neural networks. Unlike Taurus, Leo focuses on efficiently mapping ML inference to existing PISA pipelines, thus enabling data plane ML inference on commercially available hardware. Given training data, a recent work, Homonculus [26], trains an ML model that achieves a given accuracy while meeting the constraints of a switch. Neither Taurus nor Homonculus support runtime programmability. In contrast, our focus is on runtime programmability, and provisioning sufficient resources so *all models in a given class* can be supported. Recent work [29] explores support for runtime programmability for a general class of packet processing programs. In contrast, we only focus on providing runtime programmability for ML inference programs, without hardware changes.

ATP [13] and SwitchML [18] do in-network aggregation in the data plane to accelerate ML training. In contrast, Leo focuses only on ML inference in the data plane, and assumes that ML training is done offline using standard techniques. Finally, there have also been recent works that use a hybrid of control and data plane for traffic analysis. FastFE [2] and DAD [16] both implement ML inference in the control plane and extract feature values in the data plane. As shown in §7.4, such an approach cannot do per-packet traffic analysis.

9 Conclusion

We have presented Leo, a system for online decision tree classification in the data plane. Given a specification of a class of trees, Leo supports any tree in that class at runtime. Leo reduces resource requirements for a given tree through subtree multiplexing, and supports decision trees to be changed at runtime by allowing features and constraints to be reprogrammed. Leo support trees with 2× the depth of prior data plane solutions, resulting in much better classification accuracies comparable to a control plane solution. Evaluations on a real programmable switch testbed show classification latencies with Leo are 500× lower than a control plane approach. Overall, the results show Leo is a viable approach to support packet classification in the data plane.

Acknowledgments. This work was supported in part by NSF CAREER 2239829, NSF Awards 2331111 and 1837023, and research awards from Google and Cisco (23089533).

References

[1] Vamsi Addanki, Maciej Pacut, and Stefan Schmid. Credence: Augmenting datacenter switch buffer sharing with ml predictions. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024.

[2] Jiasong Bai, Menghao Zhang, Guanyu Li, Chang Liu, Mingwei Xu, and Hongxin Hu. FastFE: Accelerating ML-Based Traffic Analysis with Programmable Switches. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure (SPIN)*, 2020.

[3] Intel Barefoot Networks. P4-16 Intel Tofino Native Architecture. https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf. Accessed: 05/04/2023.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, jul 2014.

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, aug 2013.

[6] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. pForest: In-Network Inference with Random Forests. https://arxiv.org/abs/1909.05680, 2019.

[7] The P4 Language Consortium. P4-16 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.2.2.pdf. Accessed: 05/04/2023.

[8] Vladimir Gurevich and Andy Fingerhut. P4-16 Programming for Intel® Tofino™ using Intel P4 Studio™. Open Network Foundation. https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Vladimir-Gurevich-Slides.pdf. Accessed: 05/04/2023.

[9] Arthur S. Jacobs, Roman Beltiukov, Walter Willinger, Ronaldo A. Ferreira, Arpit Gupta, and Lisandro Z. Granville. AI/ML for Network Security: The Emperor has no Clothes. CCS '22, page 1537–1551, New York, NY, USA, 2022. Association for Computing Machinery.

[10] Arthur Selle Jacobs, Ricardo José Pfitscher, Ronaldo Alves Ferreira, and Lisandro Zambenedetti Granville. Refining Network Intents for Self-Driving Networks. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, SelfDN 2018, page 15–21, New York, NY, USA, 2018. Association for Computing Machinery.

[11] Patrick Kalmbach, Johannes Zerwas, Péter Babarczi, Andreas Blenk, Wolfgang Kellerer, and Stefan Schmid. Empowering Self-Driving Networks. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, SelfDN 2018, page 8–14, New York, NY, USA, 2018. Association for Computing Machinery.

[12] Wolfgang Kellerer, Patrick Kalmbach, Andreas Blenk, Arsany Basta, Martin Reisslein, and Stefan Schmid. Adaptable and Data-Driven Softwarized Networks: Review, Opportunities, and Challenges. *Proceedings of the IEEE*, 107(4):711–731, 2019.

[13] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761. USENIX Association, April 2021.

[14] Jong-Hyouk Lee and Kamal Singh. SwitchTree: In-network Computing and Traffic Analyses with Random Forests. *Neural Computing and Applications*, 11 2020.

[15] Nour Moustafa and Jill Slay. UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). In *2015 Military Communications and Information Systems Conference (MilCIS)*, pages 1–6, 2015.

[16] Francesco Musumeci, Valentina Ionata, Francesco Paolucci, Filippo Cugini, and Massimo Tornatore. Machine-learning-assisted DDoS attack detection with P4 language. In *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, 2020.

[17] EdgeCore Networks. EdgeCore Wedge 100BF-32X. https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=181&id=335. Accessed: 05/04/2023.

[18] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[19] Iman Sharafaldin, Arash Habibi Lashkari, and Ali Ghorbani. Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. pages 108–116, 01 2018.

[20] Vishal Shrivastav. Programmable multi-dimensional table filters for line rate network functions. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, 2022.

[21] Vishal Shrivastav. Stateful multi-pipelined programmable switches. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, 2022.

[22] Vishal Shrivastav, Dimitrios Koutsonikolas, and Saurabh Bagchi. Ramps: Next generation platform for real time and resilient iot analytics using mmwave and programmable switches. In *Proceedings of the Fifth Workshop on Distributed Infrastructures for Deep Learning (DIDL) 2021*, DIDL '21, 2022.

[23] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. Re-architecting Traffic Analysis with Neural Network Interface Cards. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.

[24] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. Classifying IoT Devices in Smart Environments Using Network Traffic Characteristics. *IEEE Transactions on Mobile Computing*, 18(8):1745–1759, 2019.

[25] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. Taurus: A Data Plane Architecture for per-Packet ML. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 1099–1114, New York, NY, USA, 2022. Association for Computing Machinery.

[26] Tushar Swamy, Annus Zulfiqar, Luigi Nardi, Muhammad Shahbaz, and Kunle Olukotun. Homunculus: Auto-Generating Efficient Data-Plane ML Pipelines for Datacenter Networks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 329–342, New York, NY, USA, 2023. Association for Computing Machinery.

[27] Shivam Trivedi, Lauren Featherstun, Nathan DeMien, Callum Gunlach, Sagar Narayan, Jacob Sharp, Brian Werts, Lipu Wu, Carolyn Ellis, Lev Gorenstein, Erik Gough, Alex Younts, and Xiao Zhu. PULSAR: Deploying Network Monitoring and Intrusion Detection for the Science DMZ. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, PEARC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Bruno Missi Xavier, Rafael Silva Guimarães, Giovanni Comarela, and Magnos Martinello. Programmable Switches for in-Networking Classification. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, 2021.

[29] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. Runtime Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 651–665, Renton, WA, April 2022. USENIX Association.

[30] Zhaoqi Xiong and Noa Zilberman. Do Switches Dream of Machine Learning? Toward In-Network Classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 25–33, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Yifan Yuan, Omar Alama, Jiawei Fei, Jacob Nelson, Dan R. K. Ports, Amedeo Sapio, Marco Canini, and Nam Sung Kim. Unlocking the Power of Inline Floating-Point Operations on Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 683–700, Renton, WA, April 2022. USENIX Association.

[32] Changgang Zheng, Zhaoqi Xiong, Thanh T Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. IIsy: Practical In-Network Classification. https://arxiv.org/abs/2205.08243, 2022.

A Appendix

A.1 `IIsy` with SRAM.

We present a detailed explanation of Equation 1 elaborating on how default rules are handled. We obtain conservative lower bounds on the size of feature tables and combination tables. To obtain a lower bound on requirements, it suffices to construct a single tree in the class of trees that are to be supported that needs a given amount of memory. Since the combination table, and each of the feature tables are provisioned independently, we present below separate example trees that trigger a minimum memory requirement for each of the tables.

Feature Table. Consider a decision tree of depth D based on only one of the N features, which could take values in the range $[0, K]$. For an SRAM, `IIsy` must explicitly enumerate every value, and map the value to a code word which requires a table of size K. Some savings could be obtained with a

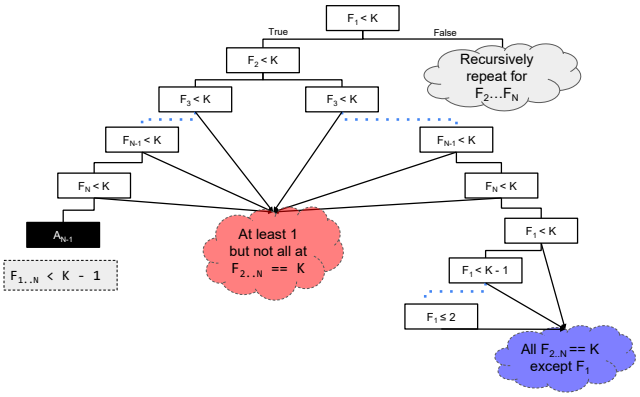


Figure 15: A decision tree that requires exponential TCAM entries with IIsy.

default rule since the values in the largest interval need not be explicitly enumerated. A complete tree with depth D has $I = 2^D - 1$ internal nodes, and partitions the possible feature values into $I+1$ intervals. Consider a tree where all intervals have the same length $\lceil \frac{K}{I+1} \rceil$. The number of SRAM rules required in the feature table for this tree is $K - \lceil \frac{K}{I+1} \rceil$ indicating at least so much memory is required.

Combination Table. To derive conservative bounds on the size of the combination table, consider a complete decision tree of depth D where each feature appears in the same number of decision tree nodes. Let $I = 2^D - 1$ denote the number of internal nodes. The total number of decision nodes that involve each feature is $\frac{I}{N}$, requiring $\frac{I}{N} + 1$ codewords per feature. Since the combination table includes combinations of all possible codewords associated with each feature, the total size is $(\frac{I}{N} + 1)^N$ which is exponential in N. A *default* rule could take away entries corresponding to one leaf. Since there are 2^D leaves in a depth D tree, taking away combinations corresponding to one of the leaves will still require at least a fraction $\frac{2^D-1}{2^D}$ of the remaining combinations. Thus, the total number of entries in the combination table is at least:

$$\frac{2^D - 1}{2^D} * (\frac{2^D - 1}{N} + 1)^N \quad (2)$$

Equation 1 follows by combining the two terms.

A.2 IIsy with TCAM

A.2.1 Proof sketch of Proposition 2

We complete the discussion of the proof sketch for the general case. The proof is based on a family of decision trees shown in Figure 15 for the general case with N features $(F_1 \ldots F_N)$ with each feature having values ranging from $1 \ldots K$.

As discussed earlier, the intuition behind the tree construction is as follows. First, the decision tree nodes has leaves for each value of a feature F_j when all other features are at their maximum value K. This forces IIsy to use a distinct code word for each value of every feature. Next, the decision tree has leaf nodes which correspond to regions where some of the features can take multiple possible values. These nodes will require a large number of code word combinations with IIsy since it is forced to use a distinct code word for each feature value.

In more detail, consider the left sub-tree of the root (feature $F_1 < K$). We have three cases based on other features:

(i) All other features are K (Blue sub-cloud). Here, a different action is chosen depending on the value of F_1, forcing IIsy to pick a different codeword for each value of $F1$.

(ii) All other features are $< K$ (left most node A_{N-1}). Although a single decision tree node, IIsy is forced to represent this using $(K-1)^N$ distinct combinations because each value of each feature is associated with a different codeword.

(iii) All other nodes which together capture the case that at least one, but not all of $F_2, F_3 \ldots F_N$ are K. An action distinct from the above is chosen for these nodes. Consider one such node where $F_N = K$ and all other features are $< K$. Encoding this alone will require $(K-1)^{N-1}$ combinations.

While a default rule can be used to cover the combinations in either Case (ii) or Case (iii), both cases cannot be covered. Thus, IIsy requires at least $(K-1)^{N-1}$ distinct combinations. With a TCAM, this would require at least m^{N-1} rules, where $m = log(K-1)$, which again grows exponentially with N.

It remains to estimate the total number of leaves in the tree. The left sub-tree has $K + 2(N-2)$ leaf nodes (the three cases above are associated with $K-1$ leaves, 1 leaf, and $N-2$ + $N-2$ leaves respectively), The right sub-tree expands in a recursive fashion with the same structure on $(N-1)$ features [i.e., the features $F_2 \ldots F_N$]. Setting and solving a recurrence, the total number of leaf nodes is $N^2 + N * (K-3) + 2$. This leads to the proposition.

A.2.2 Estimating TCAM memory requirements

We present models to obtain conservative lower bounds on the size of feature tables and combination tables when TCAM is used with IIsy. To obtain a lower bound on requirements, it suffices to construct a single tree in the class of trees that are to be supported that needs a given amount of memory. Since the combination table, and each of the feature tables are provisioned independently, we present below separate example trees that trigger a minimum memory requirement for each of the tables.

Estimating feature table size. Consider again a decision tree where all nodes correspond to the same feature which takes values in the range $[0, K]$. With a TCAM, the feature table need not explicitly enumerate all possible feature values. For instance, an interval $[1, 15]$ can be represented with 4 TCAM entries. In general, any interval requires at most $log(K)$ entries, and there are $I+1$ intervals, where I is the number of internal tree nodes. While $(I+1) * log(K)$ serves as an upper bound on the entries needed by the feature table, this is not a conservative lower bound since not all intervals

may need $log(K)$ entries (e.g., the interval $[0,15]$ requires 1 TCAM rule, while $[1,15]$ requires 4). We have devised a simple algorithmic procedure, which partitions the space $[0,K]$ into $I+1$ intervals so the total number of TCAM rules to encode all intervals is maximized. The algorithm works by iteratively partitioning each interval into two such that the total rules across the two intervals is maximized. The iterations proceed until the number of internal nodes matches the desired target or when no further splits are possible. While we do not have a simple closed form expression for this estimate, we denote the total feature table rules required using this procedure as $TCAMFTRules(K,I)$. This is a minimum size `IIsy` needs to provision.

For example, for the numeric range $[0,31]$ using 4 internal decision nodes, the recursive split is as follows:

$$[0,31]$$
$$[0,0],[1,31]$$
$$[0,0],[1,16],[17,31]$$
$$[0,0],[1,8],[9,16],[17,31]$$
$$[0,0],[1,8],[9,16],[17,24],[24,31]$$

This example would allocate $1+4+4+4+3=16$ TCAM rules per feature table.

Combination table rules estimation. We use the number of table rules that `IIsy` requires for the special tree construction (Figure 15) as a conservative lower bound on the size of the combination table. Below, we analyze the rules required for this tree.

Let T_N denote the total number of combinations that `IIsy` must handle in its combination table for this tree. Then, we set up the following recursion. Let $K1$ denote $K-1$. Then,

$$T_N = \begin{cases} \sum_{n=2}^{N}(K1)^n + \sum_{n=2}^{N-1}(K1)^2 + K1 + T_{N-1} & N > 1 \\ K1 & N = 1 \end{cases} \quad (3)$$

The first term models leaves including (i) the leftmost A_{N-1} node where all features are $< K$; and the right child of $F_n < K$ nodes in the pink cloud. These nodes correspond to the case the first n features are $\leq K-1$, and the feature $n+1$ is K. The second term models leaves which are left sub-children of nodes of the form $F_n < K$ in the pink cloud. These correspond to the case $F_1 < K$, $F_2 \ldots F_{n-1} == K$, and $F_n < K$. The third term models leaves where all features except F_1 are K, with one leaf for F_1 taking each of the values in $1 \ldots K-1$.

Note, that this expression represents the total number of combinations i.e. the resource requirement for this example under SRAM. With TCAM, the combinations corresponding to the same leaf are collapsed using wildcards. Thus we rewrite the expression as:

$$T_N = \begin{cases} \sum_{n=2}^{N}\lg^n(K1) + \sum_{n=2}^{N-1}\lg^2(K1) + K1 + T_{N-1} & N > 1 \\ K1 & N = 1 \end{cases} \quad (4)$$

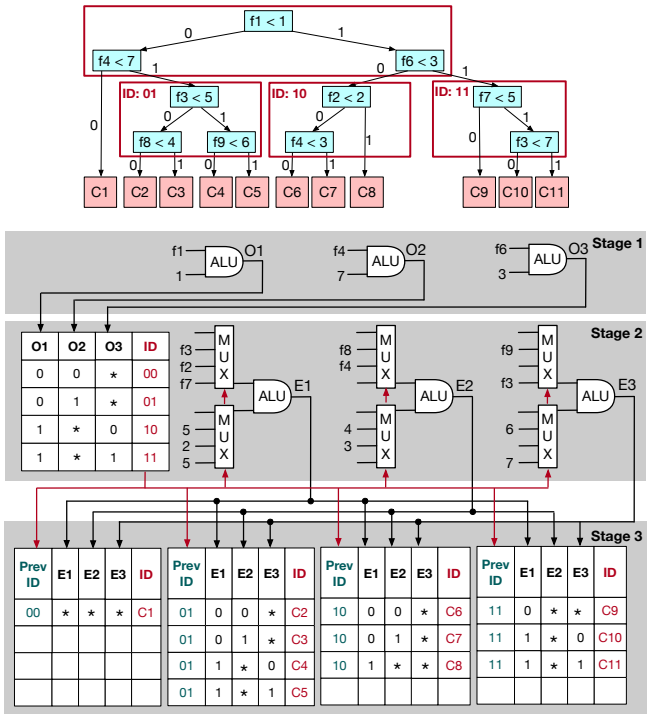


Figure 16: Illustrating the inputs of the mapped representative decision tree from Figure 9 can be configured by the control plane at runtime to implement a different decision tree within the same class.

Solving the recurrence gives the total number of rules that must be provisioned.

A.3 Evaluation

A.3.1 Impact of number of features

Figure 11 had shown the resource usage for different schemes with 10 features. We next vary the number of features, and show the maximum tree depth that can be supported with different schemes. Results are presented in Figure 17 for both SRAM and TCAM.

We make several points. First, `Leo` can support up to 10 features for $k=3$, and 15 features for $k=1$ (§5). Since the memory requirements for implementing the decision tree logic do not grow with the number of features supported, the choice of k determines the maximum depth. While we have not explored, other optimizations are possible with `Leo`– e.g., by alternating layers with $k=1$ and $k=3$, there is potential to achieve lower depth when the number of features is higher than 10. `pFor/SwTree-NT` can support up to 15 features, and the maximum depth is the same across features. In contrast, while `IIsy-NT` is not subject to the constraint on header fields (§5) as it does not perform ALU operations, its memory requirements do grow exponentially with the number of features. Consequently, the tree depth that it can support becomes smaller as the number of features is increased. `Leo` typically

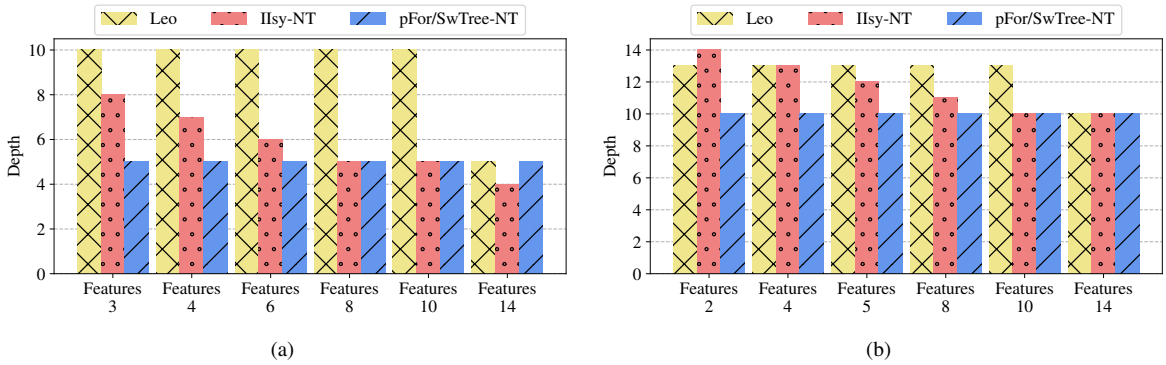


Figure 17: Maximum tree depth with different schemes as number of features is varied using (a) SRAM and (b) TCAM.

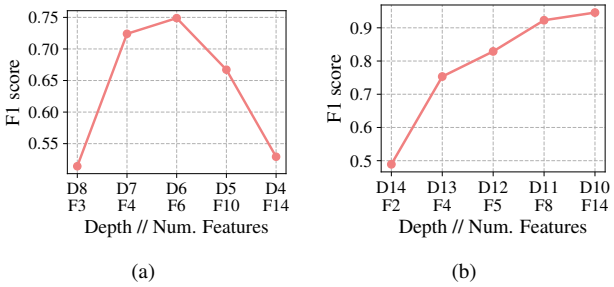


Figure 18: F1 accuracies for different settings with `IIsy-NT` for (a) SRAM; and (b) TCAM for `CICIDS`. The best result is shown in Figure 12.

supports higher depth trees than `IIsy-NT` with SRAM, and comparable or higher depth trees with TCAM up to about 15 features. Beyond 15 features, and if TCAMs were used, `IIsy-NT` may be viable, while `Leo` and `pFor/SwTree-NT` are not – however, in our accuracy experiments, `Leo` already performs very well with TCAM. Further, per-flow state requirements are the likely bottleneck (§7.3) before hitting 15 features. Finally, `IIsy-NT` does not handle transients, and the additional memory overheads if transients were addressed may further limit its performance.

Figure 18 presents a sweep of classification accuracy with `IIsy-NT` for different combinations of $(|F|, D)$ for the `CICIDS` dataset for SRAM and TCAM respectively. Here $|F|$ is the number of features supported, and D is the maximum depth achievable for $|F|$. For SRAM, the best accuracy is obtained with 6 features (depth 6), while with TCAM, the best accuracy is obtained with 14 features (depth 10). These values are reported in Figure 12 in the main text.

A.3.2 Breakdown of F1 scores per class

`CICIDS` dataset. Figure 19(a) presents a breakdown of F1 score per class for different schemes for the `CICIDS` dataset for SRAM. For all classes, `Leo` outperforms `IIsy-NT` and `pFor/SwTree-NT`. While `Leo` sees an F1 score higher than 0.95 for all classes besides OtherMalicious, `IIsy-NT` sees F1 scores falling to 0.24 and 0.28 for two classes while `pFor/SwTree-NT` sees even lower minimum F1 scores. The performance is particularly poor for the OtherMalicious class – this is because the tree depths supported by `IIsy-NT` and `pFor/SwTree-NT` are too small to allow effective categorization of traffic into this class. `Leo` is close to the Control Plane solution for all but the OtherMalicious class which tends to be challenging because of a mix of different attack patterns – but even here, `Leo` significantly outperforms other schemes.

Figure 19(b) presents a similar breakdown with TCAM implementations, and Figure 22 presents a zoomed in version. Although all schemes perform much better with TCAM (as the trees supported are larger), `Leo` still performs better than `IIsy-NT` and `pFor/SwTree-NT`, while performing close to Control Plane. The benefits are particularly strong for the OtherMalicious category.

`NB15` dataset (binary classification) Figure 20 shows a similar breakdown for UNSW-NB15 for both SRAM and TCAM. All schemes perform better since this experiment focuses on a binary classification (and much better with TCAM since they all achieve larger tree depths) – however, `Leo` continues to perform better than `pFor/SwTree-NT` and `IIsy-NT`, while performing close to Control Plane.

`NB15` dataset (multi-class classification) We have also taken the UNSW-NB15 dataset, and trained all approaches for a classification problem involving multiple classes. Figure 21 present a breakdown of the F1 scores per class. Once again, `Leo` performs better than `IIsy-NT` and `pFor/SwTree-NT` (with the benefits stronger in the SRAM case), and performs close to Control Plane.

A.3.3 Other Results/Discussion

Features used in classification. Table 2 presents a set of features used in classification. To implement stateful features, and fit them into 8 and 16 bit budgets. we employed discretization. Packet length metrics were divided by 64 (using bit

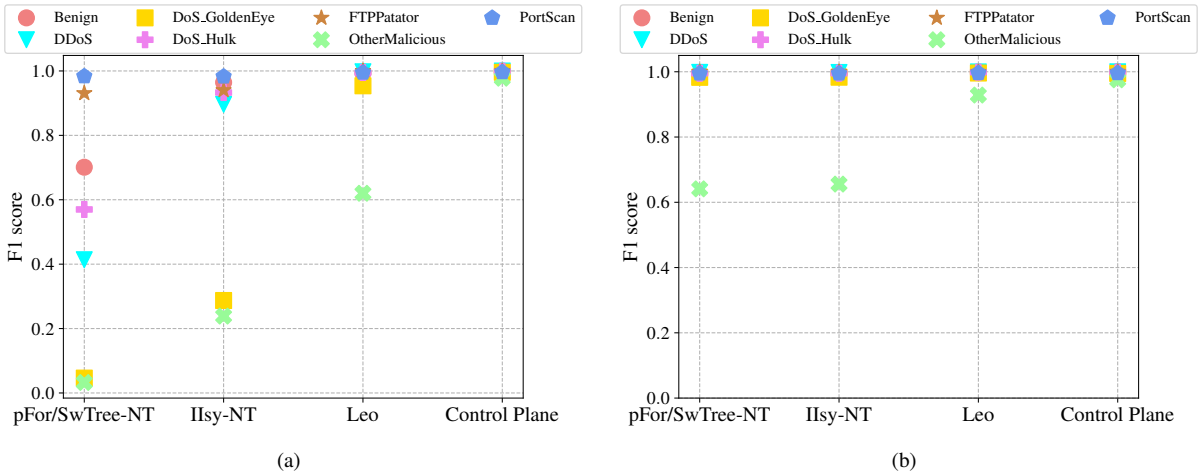


Figure 19: CICIDS-2017 classification accuracy broken down per class for (a) SRAM and (b) TCAM.

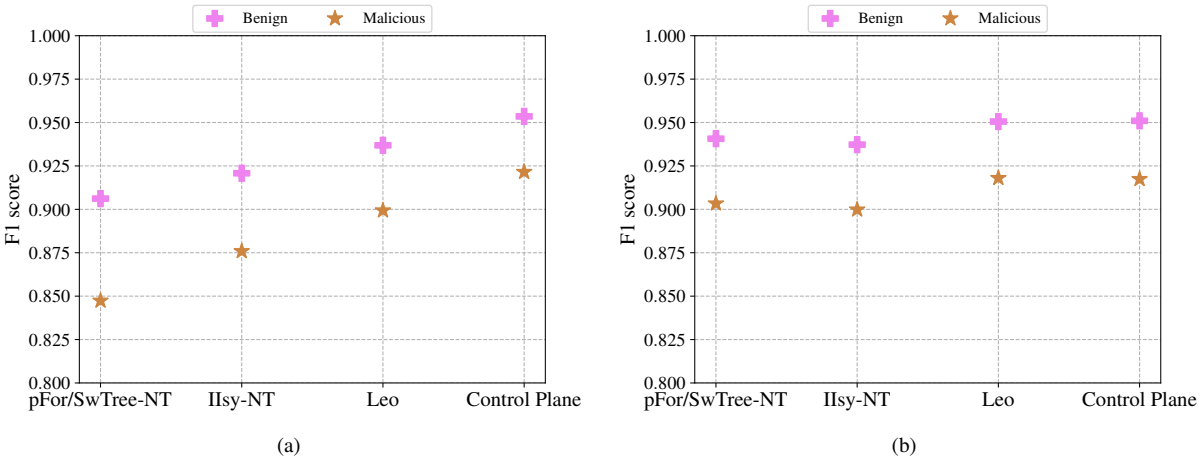


Figure 20: UNSW-NB15 classification accuracy broken down per class in a benign vs malicious scenario with (a) SRAM and (b) TCAM.

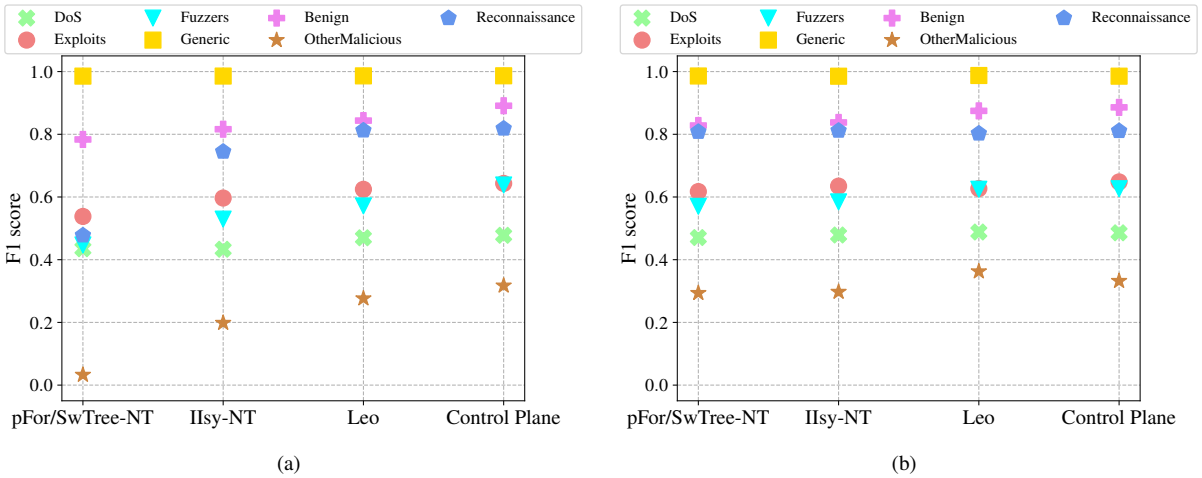


Figure 21: UNSW-NB15 classification accuracy broken down per class (for 7 classes) with (a) SRAM and (b) TCAM.

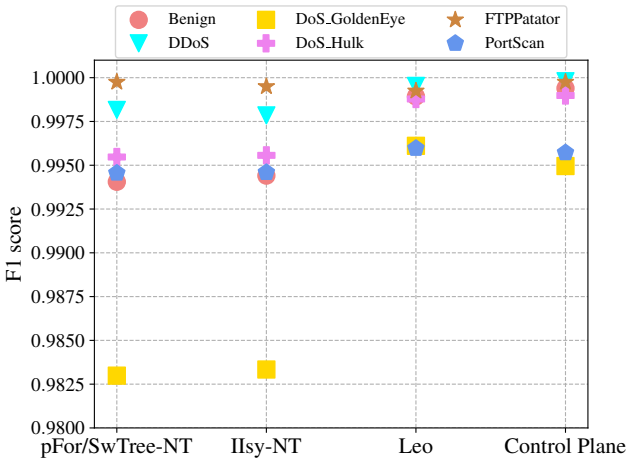


Figure 22: Zooming into the top 6 classes in Figure 19(b).

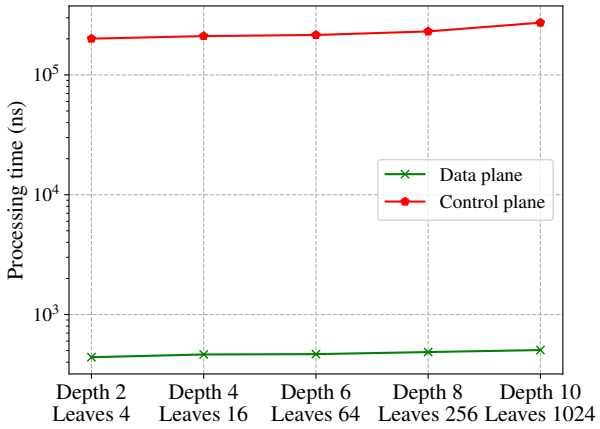


Figure 23: Average time to classify packets when routed through the data plane versus via the control plane.

shifts) which allowed for sizes between 64 bytes and 16K to be represented with 8 bits. Flow size metrics were also divided by 64. Using 16 bits, this lets us represent flow sizes between 64 bytes and 4 MB. Inter-Arrival Time is represented as two 8 bit values per flow, one to track the last timestamp, and the other to keep track of the running minimum. The timestamp itself involves extracting an appropriate 8 bits based on the granularity of measurements. Taking the last 8 bits (bits 0 to 7) would capture timestamps up to 255 nanonseconds (ns) in 1 ns granularity. However, extracting a different set of 8 bits allows a different granularity – e.g., extracting bits 2 to 9 would capture timestamps from 4 ns to 1024 ns in 4 ns granularities.

Comparing the number of leaves with `Leo` and `Leo-NT` in Figure 13. The impact on the number of leaves is initially more than a factor of 2 before converging to exactly 2. This is because, where the leaf constraint is higher, the tables handling the deeper levels span multiple stages (`Leo-NT`). Doubling these tables for `Leo` would require more stages than what are available. However, as the leaf constraint reduces, the tables shrink, freeing up more stages and thus allowing the double-sized tables to grow.

	Feature Name	CICIDS-2017 SRAM	CICIDS-2017 TCAM	UNSW-NB15 SRAM	UNSW-NB15 TCAM
Stateless	Destination Port	✓	✓		
	Forward TTL			✓	✓
	Backward TTL			✓	✓
Stateful 8-bit	Backward packet length min.	✓	✓		
	Backward packet length max.		✓		
	Forward segment size min.		✓		
	Forward initial. window advertisement			✓	✓
Stateful 16-bit	Forward flow size	✓		✓	✓
	Forward initial bytes	✓			
	Backward initial bytes	✓			
	Flow IAT min.		✓		
	Backward IAT min.		✓		
	(Dst. IP, Src. Port) count			✓	✓
	(Src. IP, Dst. Port) count			✓	✓
Total (stateful)		56			

Table 2: A breakdown of the features used for the 1.04M flows data points in Figure 14.

Sequence Abstractions for Flexible, Line-Rate Network Monitoring

Andrew Johnson
Princeton University

Ryan Beckett
Microsoft Research

Xiaoqi Chen
Princeton University

Ratul Mahajan
University of Washington

David Walker
Princeton University

Abstract

We develop FLM, a high-level language that enables network operators to write programs that recognize and react to specific packet sequences. To be able to examine every packet, our compilation procedure can transform FLM programs into P4 code that can run on programmable switch ASICs. It first splits FLM programs into a state management component and a classical regular expression, then generates an efficient implementation of the regular expression using SMT-based program synthesis. Our experiments find that FLM can express 15 sequence monitoring tasks drawn from prior literature. Our compiler can convert all of these programs to run on switch hardware in way that fit within available pipeline stages and consume less than 15% additional header fields and instruction words when run alongside switch programs.

1 Introduction

Many network management tasks involve recognizing and reacting to a user-defined sequence of packets. Such *sequence monitors* can enforce security policies, prioritize traffic, mitigate attacks, ensure protocol compliance, and more. For example, they can identify and de-prioritize video flows using a fingerprint based on successive packets to improve the network for other traffic [23, 24] or verify that network clients faithfully implement protocols such as the Dynamic Host Configuration Protocol (DHCP) by observing the protocol exchange [28].

Ideal sequence monitors are *i*) *flexible*: can express a broad range of monitoring tasks; and *ii*) *line-rate*: can perform all processing directly in the data plane (hardware). Being line-rate allows sequence monitors to analyze all traffic passing through the switch, without needing the switch CPU or a remote server. Switch CPUs cannot process all packets at line rate, and using remote servers incurs high network overhead and reaction delays.

Existing sequence monitors sacrifice either flexibility or line-rate processing. Systems such as Aragog [28] can express most sequence monitoring tasks, but they run entirely in software (i.e., are not line rate). Programmable switches based on Protocol Independent Switch Architecture (PISA) [5] enable hardware-based sequence monitoring, but are programmed in languages such as P4 [4, 26] that are too low level, making it hard to express and debug sophisticated tasks [15, 29, 32]. Hybrid systems like Marple and Sonata [11, 21] run only partially on switches. Their core abstractions focus on data transformations such as filter, map, and fold operations rather than packet sequences. They are line-rate only if strong restrictions are placed on the allowed functions or if hardware could be redesigned [21].

In this paper, we present an abstraction of a packet sequence *pattern* that is both flexible and compiled directly to PISA-based hardware. It enables line-rate sequence monitoring without mirroring traffic to the switch CPU or a central controller. Recognizing patterns for sequence monitors directly in hardware is difficult due to stringent data access constraints in current programmable switches. PISA-based switches process packets using a series of *stages*. Each stage contains its own local memory and a number of arithmetic and logic units (ALUs) to perform computation. While some sequence monitors such as packet counting fit this architecture naturally, others that require tracking state across multiple packets are significantly more challenging to realize.

Our system, called FLM, allows programmers to write sequence monitors using a high-level, pattern-based language. Patterns are specified as regular expressions over packets, with the added ability to record packet parameters for later use. FLM programs can trigger local switch actions immediately upon matching a pattern, and they can monitor packets at any desired granularity (e.g., flow, host).

We convert FLM programs into imperative code using a novel core data structure, which represents a state machine maintaining a variable environment. We transform operations on this data structure into operations on PISA switch registers by carefully dividing them into variable update and transition code for a deterministic finite automaton (DFA), reflecting the pattern's match progress. Our implementation prioritizes

line rate execution on existing network hardware like the Intel Tofino [13], for any accepted packet sequence. We have formally proven that our compilation process from patterns to pipeline stages preserves the original pattern semantics.

We evaluated FLM by encoding 15 monitoring tasks drawn from prior work [10, 15, 18, 21, 24, 25, 28, 29, 31, 32]. We find that we can express all of these tasks in 10-41 lines of FLM code, which demonstrates the flexibility of FLM. We also find that we can compile all of these tasks to the Intel Tofino switch, which demonstrates FLM's ability to provide line-rate monitoring. All of these tasks fit within the number of stages on the switch and consume less than 15% of additional metadata memory or instruction words when run alongside switch programs.

In summary, this work makes three main contributions:

- FLM, a language to express sequence monitors that can run at line-rate on a switch with a novel definition of a pattern syntax and semantics.
- Provably correct compilation from an FLM program to a state machine representation that runs on PISA hardware using a minimal number of stages.
- Evaluation that shows that FLM can express a wide variety of sequence monitoring tasks and compile them to existing network hardware.

2 Background

Our programming abstractions are broadly applicable to contexts where real-time recognition of event sequences is useful, (programmable ASICs, FPGAs, NICs, or software switches). Our implementation is designed for the PISA model. PISA architectures rely on a series of stages. To achieve a high processing rate and avoid memory access hazards such as contention, each stage has its own nearby memory region. Due to this memory layout, data can only be accessed by a single stage, and can only flow *forward* in the pipeline by changing the packet being processed. Within each stage, the switch is able to perform a Read-Modify-Write instruction on a value stored in its memory, where the modify step is specified by a micro-program called a Register Action. A core problem we tackled is writing a DFA transition function in such a way that it fits into one Register Action in order to be applied to packets at line rate. In this work, we consider a monitor recognizing patterns in packets passing through a single switch *pipeline*. A switch containing multiple pipelines operating in parallel could contain multiple monitors.

To simplify our implementation, we build on Lucid [25], an event-driven programming language for PISA switches. Lucid programs declare *events* (i.e., notifications of data plane packet arrival or network control signals) and corresponding *handlers* (i.e., code to react to events). Lucid programs are compiled to P4 code that runs on PISA switches. Both P4 and Lucid are useful intermediate languages simplify our implementation, but are not critical for FLM's key abstractions.

3 Example walk through

In this section, we walk through an example monitoring task for the DHCP protocol step-by-step to show how FLM enables network programmers to more easily build flexible, line-rate packet sequence monitors.

3.1 DHCP Anomaly Detection

Suppose a network operator wants to verify that DHCP, which enables clients to lease IP addresses from a server, is not being misused. DHCP begins with the client broadcasting a "Discover" message, to which the server responds with an "Offer" message with available IP addresses. The client sends a "Request" message for an address, which is confirmed with an "Acknowledge" from the server. The client is then expected to use the acknowledged IP until the end of its lease.

The operator wants to ensure that clients only use their assigned IP. This monitoring task can be performed on the access switch that processes all client communication, including that with the DHCP server. Misuse would appear as a packet sequence belonging to a client, identified by its MAC (link-layer address), of some number of packets (the DHCP protocol), a DHCP "Acknowledge," and then a packet whose source IP does not match the acknowledged one. Below, we show how FLM's core abstractions help achieve this goal.

Events. FLM programs are written in terms of events. For our task, we can define these two events:

```
event DHCP_Ack(int cip, int cmac);
event IP_Pkt(int sip, int smac);
```

Events are detected by parsing packets that arrive at the switch. From the `DHCP_Ack` packet, the system parses the DHCP message payload and extracts the client's MAC and new IP. Other packets are parsed as generic `IP_Pkts`, from which the source IP (`sip`) and MAC (`smac`) are extracted.

Patterns. FLM patterns are regular expressions over events, including concatenation (`.`) and closure (`*`). To begin to tackle the problem DHCP misuse, a programmer might create the following pattern, which identifies the presence of a `DHCP_Ack` amongst any number of other IP packets:

```
IP_Pkt* . DHCP_Ack . IP_Pkt*
```

Recording parameters. The pattern above would match *any* use of the DHCP server. It recognizes an event sequence, but not the event parameters. Including one character for every possible IP address in the regular expression alphabet would make it too large. Instead, FLM patterns allow for the *binding*

FLM Program

```
entry event DHCP_Ack(int cip, int cmac);
entry event IP_Pkt(int sip, int smac);
spec<2048> dhcp_misuse =
IDX = {
    | DHCP_Ack -> {hash(cmac)}
    | IP_Pkt -> {hash(smac)}
}
DETECT {
     IP_Pkt*
    .DHCP_Ack(@int assigned = cip)
    .(IP_Pkt(sip == assigned))*
    .IP_Pkt(sip != assigned)
} => {
    (...Some user response here...);
};
```

FLM Intermediate Representation

```
re<2048> dhcp_misuse =
     IP_Pkt*
    .DHCP_Ack(@int assigned = cip)
    .(IP_Pkt(sip == assigned))*
    .IP_Pkt(sip != assigned);
entry event DHCP_Ack(int cip, int cmac) {
    idx = hash(cmac);
    if (transition(dhcp_misuse, idx, this)) {
        (...user-defined response...);
    }}
entry event IP_Pkt(int sip, int smac) {
    idx = hash(smac);
    if (transition(dhcp_misuse, idx, this)) {
        (...user-defined response...);
    }}
```

Figure 1: An FLM program that monitors for DHCP misuse, and its translation with explicit state machine transitions.

of parameters to recognize patterns over very large alphabets (e.g., all IP addresses). The operator can record the value of the client's assigned IP in the `DHCP_Ack` messsage by writing:

```
DHCP_Ack(@int assigned = cip)
```

This pattern will match any `DHCP_Ack` packet, and record the value of its `cip` parameter in the new variable `assigned`. To check whether future packets use this IP, the operator can add a *predicate* over the parameters of an `IP_Pkt` by writing:

```
IP_Pkt(sip == assigned)
```

This pattern will match any `IP_Pkt` event whose `sip` parameter equals `assigned`. In our application, we also need the negation (`IP_Pkt(sip != assigned)`). Hence, using binding and predicates, the operator can now construct the following FLM pattern to detect DHCP misuse from a client:

```
IP_Pkt*
.DHCP_Ack(@int assigned = cip)
.IP_Pkt(sip == assigned)*
.IP_Pkt(sip != assigned)
```

Arrays of patterns. The above pattern characterizes an anomaly in the communications with a single client. In reality, an operator wants to monitor many clients. To track multiple clients, one specifies an *array* of patterns (in this case of size 2048), which will all be active simultaneously:

```
spec<2048> dhcp_misuse = ...
```

Next, one must specify a mapping of events to patterns, so all clients with the same MAC are identified and applied to the same pattern. The operator provides an indexing function, which computes an array index from the values carried by each event. For `DHCP_Ack` events, the index is the hash of the client MAC. For `IP_Pkt` events (to catch the *outgoing* packets), the index is the hash of the source MAC.

```
IDX = {
    | DHCP_Ack -> {hash(cmac)}
    | IP_Pkt -> {hash(smac)}
}
```

The expressions for each index calculation are user-defined. If the operator wishes to prevent hash collisions, they can implement algorithms such as probabilistic data structures or detect collisions by storing keys and siphoning overflows to a software controller, as in Sonata and Aragog [11, 28].

Responses. Finally, the operator will want to react to detected misuse somehow—perhaps by blocking the client or logging the anomaly. FLM allows users to react however they choose by invoking an arbitrary Lucid subroutine.

Putting it all together. The left side of Figure 1 shows the combination of all the features above in an FLM program implementing the DHCP specification. The `spec` declaration generates an array of size 2048, where each index represents one copy of the FLM pattern. The `IDX` block identifies the flows, the `DETECT` block contains the pattern, and the block after "`=>`" contains the response.

3.2 Compilation Overview

We compile FLM programs to Lucid and use the Lucid compiler to generate P4. While Lucid frees us from defining event parsers in P4, we must still map our pattern-based programs to PISA stages. This presents two key challenges:

1. To match a pattern, the program must both update the register holding the values of the variables stored in the pattern (for example, storing a particular client IP in the `assigned` variable in the DHCP example) and update the register holding the position in the pattern. However, this requires too much memory and computation to fit into a single stage and register, as it must in order to keep the state of the pattern up-to-date.

2. It is not even clear how to implement an arbitrary pattern that does not contain variable bindings. For example, consider a state machine representing a pattern without

```
1  //Compute the character using the old variables
2  if (event.type == DHCP_Ack) {
3      c = ACK;
4      //On ack, update the store value
5      assigned := event.cip;}
6  if (event.type == IP_Pkt and sip != assigned) {
7      c = IP0;}
8  if (event.type == IP_Pkt and sip == assigned) {
9      c = IP1;}
10 //Synthesized mapping f from character to value
11 f(c){   if (c == ACK) {return 12;}
12         if (c == IP0) {return 0;}
13         if (c == IP1) {return 12;}}
14 //Synthesized mapping g from character to value
15 g(c){   if (c == ACK) {return 2;}
16         if (c == IP0) {return 8;}
17         if (c == IP1) {return 9;}}
18 //Synthesized update function to do transition
19 update_state(curr, x, y) {
20     if(curr + y < 3){
21         return x ⊕ 5;}
22     else{
23         return curr & 3;}}
24 //Update the stored state using the f and g maps
25 state := update_state(state, f(c), g(c));
```

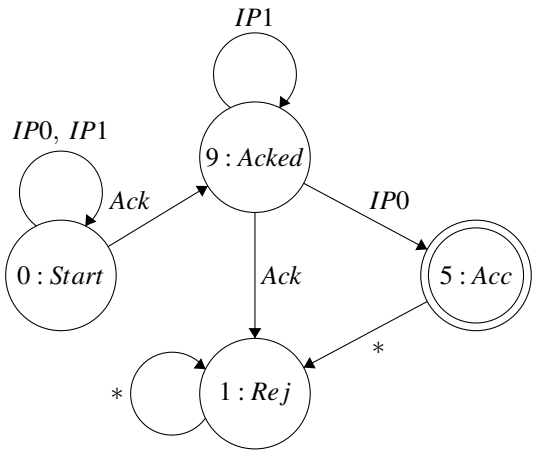


Figure 2: A DFA representation of the translated DHCP FLM pattern the memop for its transition function, and preamble code to compute the input character and variable updates. All integers are 4 bits, and the update function uses addition overflow to model the transitions correctly.

any variable bindings. A simple implementation of its transitions would take the form of:

```
transition (state, input):
    if(state == s1 and input == i1):
        return s2;
    else if (state == s1 and input == i2):
        return s3;
...
```

This will contain more branches than are allowed in a single stage for most machines. However, the transition must fit into a single stage in order to read and write at line rate. The reason for this is that the transition depends on the current state of the machine, which is only available after reading a register. In PISA, registers can only be accessed once per pipeline pass, so the transition must occur at the same time as the read.

Solution overview. We solve these challenges by carefully compiling an FLM program in a series of steps. In a preprocessing step, we transform the FLM program to an intermediate representation. This step inserts explicit "transition" statements into each event handler that will be compiled away later, and keeps the FLM pattern definition as a global definition. In the first step, we break a pattern into a series of variable updates and predicate computations that computes an input character from a finite set determined by the event type and predicate evaluations. This solves the first challenge above, as we can move all of the variable storage and predicate computation into earlier stages. In the second step, we transform a pattern without bindings into a classical regular expression over the input characters from the first step, and synthesize an implementation of the corresponding DFA in a single stage. Condensing the transition function into a single stage solves problem 2, but is difficult as it requires searching through all possible state numberings and bit-wise ALU operations for one that satisfies all transitions. We offload this hard work to an automated SMT solver.

Preprocessing on DHCP. We translate the DHCP FLM program into an intermediate representation that allows for more control of the inputs to an underlying state machine. The core data structure is `re<size>`, which defines an array of state machines with `size` indices that match an FLM pattern. The pattern is copied from the high-level program. To interact with it, the expression `transition(name, idx, ev)` applies the event `ev` to the state machine at index `idx`, and evaluates to a boolean indicating whether or not the sequence of events applied to it so far matches the FLM pattern `name`. For the DHCP example, in the handler of each event, we compute the index using the expressions provided in the high-level program (hashing the MAC address). Then, we add `transition(dhcp_misuse, idx, this)`, where `this` represents the event for the current handler. If that returns true, we run the user-defined response code. The right side of Figure 1 shows the intermediate representation with explicit transition statements in event handlers.

Step 1 for DHCP. The next two steps compile the remaining FLM pattern and `transition` statements from the inter-

mediate representation into simple assignments and register operations. We will refer to the pseudocode in Figure 2, which represents the implementation of one `transition` statement. First, we separate out the variable bindings. For the DHCP example, it is enough to store the parameter of the first `DHCP_Ack` event in the variable `assigned`. Furthermore, we wish to compute an input character `c` from a finite alphabet by evaluating the predicates in the FLM pattern. This alphabet is composed of all of the event *types* of the original, followed by bit strings representing the values of the predicates. Because `IP_Pkt` appears with a predicate, it is expanded to the letters `IP_Pkt1` and `IP_Pkt0`. `DHCP_Ack` does not appear with one, so it stays as is. These translations are shown in Lines 1–9 of Figure 2.

Step 2 for DHCP. In this step, we will translate the FLM pattern into a classical regex over the alphabet described above, and then implement its transition in a single stage. To translate the pattern, events that appear with a predicate become *unions* of any event with the same type where that predicate is true; similarly, events without a predicate are unions of any event of their respective type. Other constructs such as concatenation and closure remain as they are. For this example, the translated alphabet is the set {`DHCP_Ack`, `IP_Pkt0`, `IP_Pkt1`} and the classical regex is:

```
(IP_Pkt0 + IP_Pkt1)*
.DHCP_Ack
.IP_Pkt1*
.(IP_Pkt0)
```

Next, we translate this classical regex into a DFA and synthesize its implementation in a single pipeline stage. This is required as a naive implementation of the transitions would not fit in the limited computation available in one stage. Instead, we search through all of the state numberings and bit-wise operations to find ALU operations that complete all transitions correctly. On the right side of Figure 2, we show a picture of the DFA representing the above regex. To implement it, we take advantage of the fact that a single register read-modify-write action can take up to two arguments computed in prior stages. Given a DFA, we search for the following:

- A mapping from states to integers, as shown by numbers preceding each state in Figure 2 (e.g. *Start* is numbered 0).
- Two mappings (`f` and `g`) from the alphabet to integers that will be used as inputs to the read-modify-write instruction. These are shown on lines 11 and 15 of Figure 2. Because they can be computed in earlier stages, we can use lookup tables to implement them, which are not available when updating the DFA state.
- A read-modify-write instruction that implements the transition function of the DFA using operations available on the switch and results from the `f` and `g` mappings. This is shown on line 19 of Figure 2.

The code in Figure 2 is laid out on the switch to compute `c`, `f`, and `g`. In Figure 1, the `re` definition is replaced with a register definition, and each `transition` statement is replaced with the code in Figure 2: it first reads and updates the variables at the current index, then computes the input character and its corresponding mapping values, and finally applies the transition function to the state register with those values. It outputs whether the result represents an accepting state in the DFA (in this example, result was 5 for "Acc").

4 FLM Language Definitions

In this section, we describe the FLM language, provide its regular-expression-like syntax, and define the language's semantics over packet traces. In section 6, we prove that our compiler translations are correct: the low-level switch program correctly implements the high-level pattern semantics.

4.1 FLM Language

The FLM language is a wrapper to provide access to the expressive FLM patterns. An FLM program consists of:

1. A name and size, written `spec<i> myname = ...` where `i` is the number of replicated state machines.
2. An `IDX = {...}` block which determines which index to use for each event.
3. Optionally, a `DATA {...}` block that declares one or more registers to be used for a stateful response to a sequence match (for example, counting matches).
4. A `DETECT{pat} => {response}` block, where `pat` is an FLM pattern and `response` is Lucid code indicating what to do when the pattern is recognized.

Finally, to recognize properties such as liveness or timeouts, such as detecting half-open TCP queries [21], we provide a special event called `maintenance`. This event is guaranteed to eventually visit every state machine in a spec, so it acts as a final event to match a pattern that might never observe any more packets arriving. More about maintenance events is included in the Appendix.

4.2 Syntax and Semantics of Patterns

In each FLM program, there is a finite set A of event *types*, such as `DHCP_Ack` and `IP_Pkt`. An *event* is a pair of an event type $a \in A$ and an integer z, written $a\langle z \rangle$. While this only includes events with a single parameter, it generalizes easily to any number of parameters. The top of Figure 3 shows the syntax of an FLM pattern. We allow *predicates* over the parameters of events ($a\langle p \rangle$) to denote events of type a whose parameter satisfies p. We also allow *binding* parameters in events ($a\langle @y; p \rangle$) for use in predicates.

Patterns (and their contained predicates) are evaluated under an *environment*. An *environment* E is a mapping from variables to integers, with its domain denoted by $Dom(E)$.

FLM pattern $\boxed{R}$

p		*predicate*
$r ::=$	$\emptyset$	*empty set*
	$\mid \varepsilon$	*empty string*
	$\mid a\langle p\rangle$	*event*
	$\mid a\langle @y; p\rangle.r_1$	*binding event*
	$\mid r_1.r_2$	*concatenation*
	$\mid (r_1)^*$	*closure*
	$\mid r_1 + r_2$	*union*
	$\mid r_1 \,\&\, r_2$	*intersection*

Denotational Semantics $\boxed{[\![r]\!]_E : \text{Set of strings}}$

$$
\begin{aligned}
[\![\emptyset]\!]_E &= \emptyset \\
[\![\varepsilon]\!]_E &= \{\varepsilon\} \\
[\![a\langle p\rangle]\!]_E &= \bigcup_{z\in\mathcal{Z}} \{a\langle z\rangle \mid [\![p(z)]\!]_E\} \\
[\![a\langle @y; p\rangle.r]\!]_E &= \bigcup_{z\in\mathcal{Z}} (\{a\langle z\rangle \mid [\![p(z)]\!]_E\} \circ [\![r]\!]_{E,y\leftarrow z}) \\
[\![r+s]\!]_E &= [\![r]\!]_E \cup [\![s]\!]_E \\
[\![r\&s]\!]_E &= [\![r]\!]_E \cap [\![s]\!]_E \\
[\![r.s]\!]_E &= [\![r]\!]_E \circ [\![s]\!]_E \\
[\![r^*]\!]_E &= \bigcup_{n\in\mathcal{N}} [\![r^n]\!]_E
\end{aligned}
$$

Auxillary definitions

$$
\begin{aligned}
R \circ S &= \{r.s \mid r \in R \wedge s \in S\} \\
r^0 &= \varepsilon \\
r^{i+1} &= r.r^i
\end{aligned}
$$

Figure 3: Technical cheat sheet. Definitions for FLM patterns and their derivatives.

The empty environment is denoted by ".". A *predicate* p is a function from integer to boolean that may contain one or more free variables, and is *closed under an environment* E if the free variables of p are contained in $Dom(E)$. The evaluation of a predicate p applied to an integer z under an environment E is denoted by $[\![p(z)]\!]_E$, and exists if p is closed under E.

In this section (except for the DHCP pattern, for continuity), we use lambda notation to define predicates. For example, $\lambda x.(x \geq 10)$ is a predicate that returns true if the given integer is at least 10. We use the standard semantics of lambda functions. Finally, an FLM pattern r is *closed under an environment* E if all of its free variables appear in E.

On the bottom of Figure 3, we show the semantics of an FLM pattern. Each FLM pattern defines a set of strings of events that belong to its language. A binding has a scope for its variable. Predicates within the scope can use the variable. For example, $[\![a\langle @y; \lambda x.true\rangle.(b\langle \lambda x.(x == y)\rangle)]\!]$. is the set of any event of type a (the predicate is always true) followed by one of type b with the same parameter (e.g. $a\langle 12\rangle.b\langle 12\rangle$). Constructors of FLM patterns are defined similarly to those of classical regular expressions; $[\![r+s]\!]_E$ and $[\![r\&s]\!]_E$ represent union and intersection of the sets $[\![r]\!]_E$ and $[\![s]\!]_E$, respectively, and $[\![r^*]\!]_E$ represents zero or more copies of $[\![r]\!]_E$.

4.3 FLM Intermediate Representation

The FLM intermediate representation simplifies the higher-level language features to leave just the patterns. It includes two new features not present in Lucid:

1. `re<i> myname = pat` is a statement that defines an array of `i` finite state machines named `myname`, which each recognize the FLM pattern `pat`.

2. `transition(myname, idx, ev)` is an expression that applies a transition with the event `ev` to the state machine at index `idx` of `myname`. It evaluates to `true` if the state machine is in an accepting state (the pattern has been recognized), and `false` otherwise.

An FLM program is transformed into the definition of a state machine with the same pattern, size and name. Then, at the beginning of each event handler, the compiler adds the following code:

```
if (transition(myname, idx, this)) {
    response;
}
```

`myname` is the name of the state machine, `this` is the event to transitioned with (the current event for the handler), `idx` is computed using the `IDX` block, and `response` is the user-defined response.

5 From FLM patterns to Regular Expressions

We showed in section 3 how to build a DFA and some preamble code for the DHCP example. Here, we describe more generally how to translate an FLM pattern into a regular expression, which we translate to a DFA in section 7. Due to hardware restrictions, we cannot complete all of the actions necessary to store variables, evaluate predicates, and transition pattern state machines in one stage, so our plan is to carefully separate those operations into a series of stages. First, we lift variable bindings out of patterns, then remove predicates to reduce the problem to implementing a finite state machine over a finite alphabet where events are paired with bits representing the predicates in a pattern.

5.1 Lifting out variable bindings

A binding FLM pattern has the form $b\langle @y; p\rangle.r$. These may occur deep within a pattern, posing a problem for implementing the variable bindings in a pipeline stage before the pattern state. In order to place bindings in an earlier stage, binding occurrences must depend only on the incoming event and environment, not the state of the pattern. We move the bindings to the top-level of a pattern while preserving its semantics by introducing a new form of patterns:

$$b\langle @y\rangle \triangleright r$$

Intuitively, this construct binds the first occurrence of an event with type b's value to the variable y, and then proceeds with matching r. The key aspect of the $\triangleright$ syntax is that it separates the binding out from the rest of the pattern. Our goal is to move these bindings all the way to the top-level using *rewrite rules*, to get a pattern that is written as a series of bindings, followed by a pattern without any variable changes at all. We show how this works on the DHCP example:

```
IP_Pkt*
.DHCP_Ack(@int assigned = cip)
.(IP_Pkt(sip == assigned))*
.IP_Pkt(sip != assigned)
```

The first rule converts an "@" binding into a "▷" one in place.

```
IP_Pkt*
.DHCP_Ack(@int assigned = cip)▷
    (DHCP_Ack
    .IP_Pkt(sip == assigned)*
    .IP_Pkt(sip != assigned))
```

The second rule moves the binding up by one level.

```
DHCP_Ack(@int assigned = cip)▷
    (IP_Pkt*
    .DHCP_Ack
    .IP_Pkt(sip == assigned)*
    .IP_Pkt(sip != assigned))
```

This pattern has the same semantics as the original, but is in *prefix form*: a binding followed by a *binding-free* pattern.

Definition 1. *An FLM pattern s is binding-free if it contains neither $\triangleright$ nor @.*

In the last DHCP example above, the binding-free pattern is the portion after the $\triangleright$.

Definition 2. *An FLM pattern r is in prefix form if it is written as $B \triangleright s$, where B is a series of bindings ($b_1\langle @y_1\rangle \triangleright b_2\langle @y_2\rangle \ldots$), and s is binding-free.*

The last version of the DHCP pattern above is in prefix form. FLM patterns in prefix form cannot contain @ bindings, as they are all converted to the $\triangleright$ syntax. The semantics of the $\triangleright$ operator is the union of two sets. The first covers cases when the binding is required. In this case, the binding $b\langle @y\rangle \triangleright r$ should bind the value of the first occurrence of event b to the variable y. The second covers cases when the variable y is *not* used to match the pattern. For example, the pattern:

$$b\langle @y\rangle \triangleright (a\langle \lambda x.true\rangle + (b\langle \lambda x.true\rangle.a\langle \lambda x.x == y\rangle))$$

is meant to define either any string with a single event of type a, or a sequence of an event of type b followed by a where their parameters match. However, in the case of a single a event, the binding is not needed. For this case, we quantify over *all* possible values for y when defining it, which ensures the value of y does not matter for matching.

Definition 3.

$$\begin{aligned}[\![b\langle @y\rangle \triangleright r]\!]_E \quad &= \{w_1.b\langle z\rangle.w_2 \in [\![r]\!]_{E,y\leftarrow z} \mid b \notin w_1\} \\ &\cup \{w \mid b \notin w \text{ and } \forall z.w \in [\![r]\!]_{E,y\leftarrow z}\}\end{aligned}$$

In section 6, we show that if the rewrite rules can transform a regular expression into a new one that is in prefix form, it can always be implemented in a pipeline. A full list of rewrite rules is contained in the Appendix. For all of these rules, we show that if r is rewritten to r', then for all environments E under which r is closed, $[\![r]\!]_E = [\![r']\!]_E$.

Unimplementable patterns The rewrite rules are not complete. Some patterns cannot be easily rewritten into prefix form to be recognized in a pipeline. For example, the following pattern fails to reach prefix form:

$$a\langle \lambda x.true\rangle.a\langle @y\rangle.a\langle \lambda x.x == y\rangle$$

This is meant to define a sequence of three events of type a, where the parameters of the second and third events match. This is semantically well defined, but it cannot be implemented easily because the variable updates happen *before* accessing the pattern state. When an a event arrives, the decision to record y must be made without knowledge of previous events. Our rewrite rules would reject this pattern. This could arise in some compound monitors: for example, extending the DHCP example to bind the unassigned `sip` and check that it is subsequently used as part of another pattern, such as one for a generic TCP handshake. This would fail because the binding would not be with the first `IP_Pkt`. We included some explicit rejected patterns in the Appendix.

5.2 Translating events

Now, we will focus on recognizing a *binding-free* pattern by translating it into a DFA over a new alphabet.

Alphabet. The alphabet is formed by combining all of the event *types* of the pattern with all possible combinations of values of the predicates. As shown in subsection 3.2, the alphabet for the DHCP example is {`DHCP_Ack`, `IP_Pkt0`, `IP_Pkt1`}. `IP_Pkt` appears with a bit representing the value of the associated predicate, and `DHCP_Ack` remains as it is because it does not appear with a predicate.

In general, the alphabet for the translation of an FLM pattern r is defined as follows, where bin(n) is the binary representation of a natural n, $P(a)$ is the list of all of the predicates in r for event type a, and A is all of the event types:

$$\{a.\text{bin}(n) \mid a \in A \wedge n < 2^{len(P(a))}\}$$

Events. We need to take a single *concrete* event $a\langle z\rangle$ and output a character in the new alphabet. To do this, we define the *letter* translation T_l, which keeps the event type and appends each predicate's evaluation under the given environment. In our example, consider an `IP_Pkt` event where `sip = 10.0.0.1`. If the environment contains the mapping from `assigned` to `10.0.0.1`, then the translated letter is `IP_Pkt1`. Otherwise, it is `IP_Pkt0`.

Definition 4. *The event translation*

$$T_l(a\langle z\rangle, E, P) = a[\![p_1(z)]\!]_E \ldots [\![p_n(z)]\!]_E$$

5.3 Eliminating predicates from patterns

Now, we translate a binding-free pattern into a classic regular expression over a finite alphabet by eliminating all remaining predicates. The translation, called T_{re}, maps events with predicates to *unions* of characters with the same event and the corresponding predicate being true. The formal definition is in the Appendix. For example, `IP_Pkt(sip == assigned)` is translated to the pattern `IP_Pkt1`. Because `IP_Pkt` alone specifies no predicates, it is translated to the pattern `(IP_Pkt0 + IP_Pkt1)`. Union, intersection, concatenation, and closure all just apply the translation recursively. The full translation of the DHCP pattern is:

```
(IP_Pkt0 + IP_Pkt1)*
.DHCP_Ack
.IP_Pkt1*
.IP_Pkt0
```

6 Translation Correctness

In this section, we present the theorem of correctness for our translations. Intuitively, this means that a *translated* string of events is accepted by a *translated* regular expression exactly when the original string of events is in the language of the original pattern, assuming that the variable updates are performed correctly. We first introduce the concept of *derivatives*, which formalize what should happen when a single event is processed. Then, we state our main theorem, which relates a series of derivatives to processing using our translations.

6.1 FLM Pattern Derivatives

Derivatives of FLM patterns formalize what happens when one event arrives. We will define one for binding-free FLM patterns and one for lists of bindings. A derivative of a pattern, D_{re}, is taken with an event and an environment. It outputs a new pattern, which represents the *remaining* pattern to be matched. We illustrate this by example with the DHCP pattern matching a string of events on the left of Figure 4. The last pattern is ε, which means that the original DHCP pattern accepts the string of events. The full derivative rules used are shown on the right of Figure 4. This also shows the binding derivative, which takes a series of bindings, an event, and an environment, and outputs a new binding and environment with the correct variable updates.

Formally, we show that the outputted pattern of a derivative contains all the strings that would form word in the language of the input pattern when concatenated to the input event:

Theorem 1. *For all a, z, s, E where s is binding-free and closed under E:*

$$[\![D_{re}(a\langle z\rangle; s; E)]\!]_E = \{w \mid a\langle z\rangle.w \in [\![s]\!]_E\}$$

6.2 Correctness Theorem

Our translations are correct if, after we translate an FLM pattern into a DFA with T_{re}, and feed it a string of events translated with T_l, the DFA accepts only when the original string of events is in the language of the original pattern.

We show that the relation between a pattern s and its translated DFA via T_{re} is preserved when transitioning the DFA using characters translated with T_l. In particular, taking the FLM pattern derivative of s with an event and then translating it to a DFA is the same as transitioning $T_{re}(s)$ with a translated event. At the end of a string of events, we test whether the *translated DFA* is accepting, which is equivalent to string being in the language of the original pattern.

To state this formally, we define the translation of a *word* (a string of events), which repeatedly applies T_l and D_{bind} to transform the word into a string of finite-alphabet characters, given a list of bindings, an inital environment, and a list of predicates. The translated example word from Figure 4 would be `DHCP_Ack.IP_Pkt1.IP_Pkt0`.

Definition 5. *The word translation*

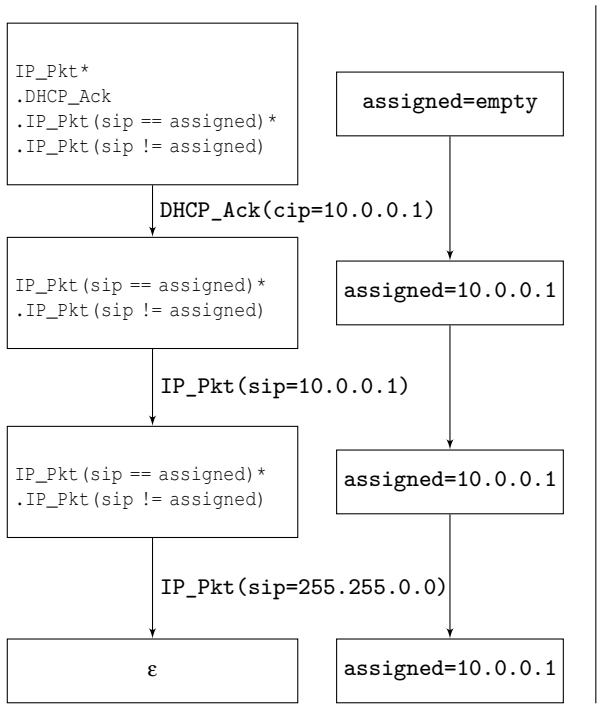
$$T_w(a_1\langle z_1\rangle \ldots a_n\langle z_n\rangle, B, E, P) =$$
$$T_l(a_1\langle z_1\rangle, E, P).T_w(a_2\langle z_2\rangle \ldots a_n\langle z_n\rangle, B', E', P)$$

Where $B', E' = D_{bind}(a\langle z\rangle; B; E)$

The word translation of ε is ε. The rewrite rules described in subsection 5.1 preserve the semantics of patterns. They also preserve a property we call *implementability*, which means that the derivative of a pattern with events that are not binding is semantically equivalent no matter the assignments to unbound variables. This property holds if input patterns bind variables using the *first* occurrence of an event type, and always use variables after they are bound. We show an unimplementable pattern in section 5. For the technical definition, see the Appendix. Finally, we have our correctness theorem, which states we can check whether a *translated* word is in the language of a *translated* regular expression to determine if a word matches a pattern. $Asgn0(B)$ simply assigns 0 to each variable of B, so that there are never undefined variables. The proof is by induction on the length of a word and is contained in the Appendix.

Theorem 2. *For any word w, binding list B, pattern s, environment E, and predicates P, if $B \triangleright s$ is in prefix form, closed under E, and implementable, then:*

$$T_w(w, B, (E, Asgn0(B)), P) \in L(T_{re}(s, P)) \iff w \in [\![B \triangleright s]\!]_E$$



FLM pattern Derivative $D_{re} : Event \to R \to E \to (R,E)$

$$
\begin{aligned}
D_{re}(a\langle z\rangle;\emptyset;E) &= \emptyset \\
D_{re}(a\langle z\rangle;\varepsilon;E) &= \emptyset \\
D_{re}(a\langle z\rangle;b\langle p\rangle;E) &= \begin{cases} \varepsilon & \text{if } a == b \text{ and } [\![p(z)]\!]_E \\ \emptyset & \text{otherwise} \end{cases} \\
D_{re}(a\langle z\rangle;r.s;E) &= D_{re}(a\langle z\rangle;r;E).s + v(r).D_{re}(a\langle z\rangle;s;E) \\
D_{re}(a\langle z\rangle;r+s;E) &= D_{re}(a\langle z\rangle;r;E) + D_{re}(a\langle z\rangle;s;E) \\
D_{re}(a\langle z\rangle;r\,\&\,s;E) &= D_{re}(a\langle z\rangle;r;E)\,\&\,D_{re}(a\langle z\rangle;s;E) \\
D_{re}(a\langle z\rangle;r^*;E) &= D_{re}(a\langle z\rangle;r;E).r^*
\end{aligned}
$$

Nullability $v : R \to R$

$$
\begin{aligned}
v(\varepsilon) &= \varepsilon \\
v(\emptyset) &= \emptyset \\
v(a\langle p\rangle &= \emptyset \\
v(r.s) &= v(r).v(s) \\
v(r+s) &= v(r) + v(s) \\
v(r\&s) &= v(r)\&v(s) \\
v(r^*) &= \varepsilon
\end{aligned}
$$

Binding update $D_{bind} : Event \to B \to E \to (B,E)$ Let $B', E' = D_{bind}(a\langle z\rangle, B, E)$ in the following definitions:

$$
\begin{aligned}
D_{bind}(a\langle z\rangle, \varepsilon, E) &= \varepsilon, E && \text{no bindings} \\
D_{bind}(a\langle z\rangle, b\langle @y\rangle \triangleright B, E) &= b\langle @y\rangle \triangleright B', E' && if\, a \neq b \\
D_{bind}(a\langle z\rangle, a\langle @y\rangle \triangleright B, E) &= B', (E', y \leftarrow z) && if\, a = b
\end{aligned}
$$

Figure 4: The progress of matching a string with the DHCP example. The left column contains the binding-free patterns, and the right tracks the environment. The arrows indicate the pattern derivative with the incoming event and current environment. Acceptance is indicated by the empty string, ε.

```
1  memop template (st, f, g):
2    b1 = [st,0] + [f,g,0] [==, !=, <, >] c1;
3    b2 = [st,0] + [f,g,0] [==, !=, <, >] c2;
4    if (b1 [||, &&] b2):
5      return [st,c3] [|,&,+,⊕] [f,g,c4];
6    else:
7      return [st,c5] [|,&,+,⊕] [f,g,c6];
```

Figure 5: A syntax template for a single register action to be synthesized. Blue-bubbled brackets represent choices between the expressions in the brackets. Red-bubbled brackets represent choices between the operators in the brackets. Each c_i represents a constant chosen by the synthesizer.

7 DFA Synthesis

The previous section reduced the problem of matching FLM patterns to matching specially constructed regular expressions, but it is still not clear how to do this at line rate. In this section, we show how to synthesize code to perform the classical regex derivative (a DFA transition) in order to use theorem 2. We use SMT-based synthesis to fit a transition function into at most four register actions, the maximum allowed on the Tofino.

7.1 Synthesis goal

To implement a DFA's transition function within the allowed register actions, the synthesizer will attempt to cleverly assign numbers to the DFA states and alphabet while generating a short function composed of a fixed number of simple instructions like bitwise operations, arithmetic operations, and conditional branches. The function will calculate the next state given the current state and input event without the need to enumerate DFA transitions.

We take as input a DFA and a bitvector length l. A DFA is a five-tuple (Q,Σ,δ,q_0,F), where: $Q = \{q_0,q_1,\cdots\}$ is a finite set of states with initial state q_0, $\Sigma = \{\sigma_0,\sigma_1\cdots\}$ is a finite set of alphabet symbols, $\delta : Q \times \Sigma \to Q$ is the transition function, and $F \subseteq Q$ is the set of accepting states. For the DHCP example from subsection 3.1, the DFA has $Q = \{Start, Acked, Acc, Rej\}$, $\Sigma = \{Ack, IP0, IP1\}$, $F = \{Acc\}$, and δ as shown in Figure 2.

We output a function to implement the state machine's transition on the Tofino or any other hardware whose memory update is at least as expressive. Specifically, we output:

- A mapping R from Q to $\{0,\ldots,2^l-1\}$ that uniquely numbers the states, and by convention we fix $R(q_0) = 0$.
- Two mappings f and g from Σ to $\{0,\ldots,2^l-1\}$. These are passed as arguments to the register actions.

- A mapping *whichop* from Σ to $\{0,1,2,3\}$ that indicates which register action each character will use.
- Up to four register actions that take the values of the maps R, f, and g on the current state q and input character σ, and output $R(\delta(q,\sigma))$ for all the letters in Σ that use them. Furthermore, the functions must follow the syntax from Figure 5 in order to fit into a single register action.

An example of correct outputs for the DHCP example is also shown in Figure 2 (all characters use the same function).

7.2 Synthesis implementation

To come up with these outputs, we use an SMT solver to do syntax-guided synthesis [1]. We make one bitvector variable for each state (the values of R), two bitvectors for each letter (f and g), and booleans to determine *whichop*. To make the templates, we make boolean indicator variables for which operations, comparisons, and boolean comparisons are used. Then, we encode the templates as constraints over the state variables. If a satisfying assignment is found, we read it to get the output. An interesting problem is to find the best template for synthesis. This is discussed more in subsection 9.3.

8 Implementation

We implement the FLM compiler atop the Lucid framework [19, 25] using approximately 1500 lines of OCaml available on GitHub[1]. We implemented DFA synthesis code in the compiler in OCaml using the z3 SMT solver [30]. It first transforms each FLM pattern into prefix form and translates it to a classical regular expression. Then, it converts the pattern into a DFA and runs syntax-guided synthesis to generate the corresponding mappings and memops, expressed as an intermediate representation Lucid program. This program is subsequently compiled using the existing Lucid framework's backend and vendor-provided P4 compiler (`bf-p4c`) to generate the final data plane program binary. We use `ocamlc` 4.14.0, `z3` 4.11.2, and `bf-p4c` 9.13.0.

9 Evaluation

We evaluate FLM by using it to implement a diverse set of sequence monitoring tasks of interest to network operators. We identified 15 such tasks from prior work, and implemented them alongside a Lucid program that used the same events. All of the monitored patterns are listed in the Appendix.

Figure 6a shows these programs and summarizes our results, including the lines of code needed to express the examples, the synthesized DFA complexity and synthesis time, and the stages used. We are able to express each of the 15 tasks in the FLM language, pointing to its flexibility. The table shows lines of code as a proxy for ease of use. We see that FLM programs are short and all tasks are expressed in a few 10s of lines. In contrast, when translated to Lucid these programs are 5-10x bigger, which is a proxy for implementation effort of expressing these tasks directly in P4.

Our compiler is able to compile each of the programs to the Intel Tofino, which demonstrate the line-rate monitoring capabilities of FLM.

9.1 Compilation time

Figure 6b shows the compilation time for each program on an AWS EC2 t3.medium server with 4GB memory and 2 vCPU (unlimited burst). We break the total compilation time into the frontend (Lucid compiling) and backend (P4 compiling). Note that the Lucid time *includes* the synthesis time from Figure 6a. We see that most programs need little time (seconds) to complete our compilation steps. Although the DFA synthesis step depends on the complexity of the pattern and is theoretically NP-Hard, all tasks finish in a few minutes.

Programs decorated with FLM have slower compilation time for both the Lucid compiler backend and the vendor P4 compiler. This is caused by the additional statements added to the IR and the resulting overhead for optimizing the pipeline layout, and mostly depends on the complexity of the original program. The complexity of the pattern (aside from additional variables) does not affect the backend's and vendor compiler's compiling time. All implementations take roughly the same time to compile once synthesized.

9.2 Hardware Resource Utilization

Figure 6c shows the hardware resource usage of the Tofino binaries for each program. The three most relevant metrics for FLM are the number of pipeline stages used (Figure 6a), the percentage of metadata fields (PHV) allocated, and the percentage of instruction words (VLIW) allocated.

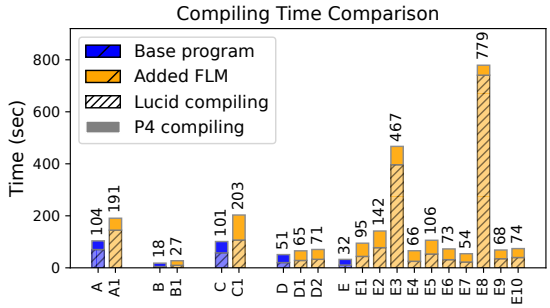
Depending on the complexity of pre-processing involved in translating events (packets) to letters in the pattern, FLM compiles programs into 6-10 stages, all comfortably fitting within the Intel Tofino v1 (12 stages). The DFA itself always uses a single stage, regardless of the complexity of the pattern. The additional stages take care of the Lucid event handling and control flow, as well as predicate computation. Because there are data dependencies between stages arising from control flow, these stages are not always "full;" additional unrelated programs can fit alongside them without using more stages, as the preprocessing steps are parallelized with existing logic.

Meanwhile, FLM programs use reasonable resources: adding FLM to an existing Lucid program only consumes 1-15% additional PHV and VLIW, much of which is for setup (parsing and pre-processing). Interestingly, the resourge usage for some tasks goes *down* with added FLM monitoring due to the additional stage usage.

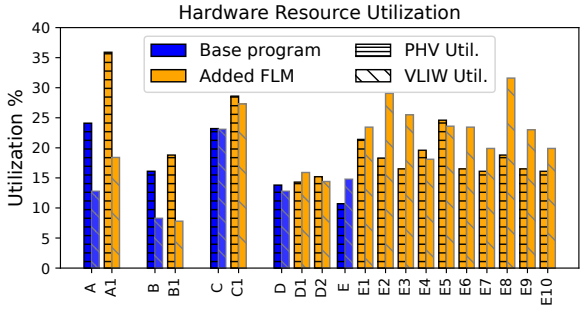
[1] https://github.com/PrincetonUniversity/lucid/tree/SpecRegex

	Lines of code			DFA	FLM	Stages
	Lucid	IR	P4	(Q,Σ)	Time(s)	
A: Cuckoo Firewall [25]	171	388	1795	-	-	10
A1: Cuckoo Insertion	+14	+268	+616	(9,5)	7.6	+0
B: Stateful Firewall [25]	34	91	578	-	-	6
B1: Firewall Correctness	+10	+100	+308	(4,4)	.8	+4
C: SipHash [29]	185	260	2072	-	-	8
C1: SipHash Rounds	+30	+125	+981	(17,3)	19.9	+0
D: Chain Replication [31]	126	223	1319	-	-	10
D1: Sequence Numbers	+10	+75	+299	(4,2)	.5	+0
D2: Double Write	+11	+85	+336	(5,2)	1.3	+0
E: Simple RIP [25]	121	219	1158	-	-	8
E1: DHCP Anomaly [3]	+24	+160	+1184	(4,3)	.6	+0
E2: Fingerprint [24]	+41	+224	+2264	(10,8)	14.2	+2
E3: Port Knocking [10]	+19	+168	+1548	(6,16)	114.4	-1
E4: DNS TTL [21]	+15	+118	+830	(4,2)	.4	+2
E4: DNS Tunneling [21]	+18	+191	+1382	(13,5)	10.7	+1
E6: SwiSh Local [32]	+13	+147	+1123	(7,3)	1.0	+0
E7: NetChain [15]	+13	+89	+521	(4,2)	.4	+0
E8: Paxos Recovery [28]	+25	+279	+1048	(5,32)	15.7	+0
E9: ATP Sequence [18]	+23	+147	+668	(6,8)	6.0	+0
E10: ATP JobID [18]	+35	+140	+662	(19,3)	11.4	+0

(a) The results from compilation of our 15 example monitors. The gray rows show the base programs. The following white rows show one or more monitors that we added to the base programs. For the base programs, we show the lines of code of the source program (Lucid), the intermediate representation before compiling to P4 (IR), and the resulting P4 program (P4), as well as the number of stages used. For the monitoring tasks, we show the *additional* lines of code and stages added by FLM as well as the size of the DFA in terms of states (Q) and the alphabet (Σ) as well as the time required to compile FLM to Lucid.



(b) The compilation time of each example, with the base programs in blue and the monitors in orange.



(c) The PHV and VLIW utilization of each compiled program, with the base programs in vlue and the monitors in orange.

9.3 DFA Synthesis

We measured the size of each DFA in our examples, noted in the "DFA" column of Figure 6a as a pair $(|Q|, |\Sigma|)$ to show the number of states and the size of the alphabet. These varied from small automata with only a few states and symbols to ones with dozens of states and alphabet characters. The step of synthesizing an implementation, included in the "FLM Time", was relatively quick for all of them. In prior work [6], we further evaluated the difficulty of state machine synthesis.

We found that it was possible to generate implementations for state machines representing simple networking tasks such as tracking a TCP handshake. We also tested combinations of state machines via union (tracking multiple at once) or concatenation (tracking one after another) via video fingerprinting examples. While union was usually easier than concatenation, there were many examples of the same size DFA (as measured by states and alphabet) where some worked and some did not. There are other qualities of DFA complexity that show up only in some fingerprinting patterns. For example, states with many incoming or outgoing edges place many constraints on the same variables and make a solution more difficult to find.

Finally, we tried five different synthesis templates of varying complexity, ranging from simple assignment to using Tofino-specific tricks such as representing states with two integers in a paired array and updating both simultaneously. We also varied the choices of bitwise operators available in each template. We found that the template shown in Figure 5 was the best balance between expressiveness (ability to represent a wide variety of DFAs) and synthesis time.

10 Related Work

The syntax for FLM programs is inspired by Aragog [28], a system that focused on recognizing issues in distributed systems by specifying regex-like patterns. The patterns were checked by a *global* verifier that had specific events forwarded to it by all the systems. Aragog operates entirely in the control plane, whereas our work focuses on recognizing packet sequences appearing on a single data plane switch at line rate.

Many works use data plane switches to recognize regular expressions appearing in packets for the purposes of content inspection. Some early work appeared before P4 was released [20, 22]. More recent work has built on further hardware advances, and includes frameworks specialized for matching strings in a packet [14, 27]. DBVal [17] focuses on verifying the data plane execution of a single packet. DeepMatch [12] focuses on searching for regular expressions within the payload of packets, and developed some techniques to hold state between payloads for a single flow. Our work differs in that it focuses on patterns of *sequences* of packets, where all the computation happens in a single stage used once

per packet, rather than a series of stages that can search for patterns *within* packet content. We also allow more expressive patterns with the binding of event values and predicates.

Our patterns draw inspiration from previous work on parametric verification [3], as well as studies of various forms of automata for wide-ranging applications. Timed automata [2] can record the timing of events and place time constraints on transitions. Symbolic automata [8, 9] permit a very wide variety of predicates on transitions without memory other than their state, while register automata [16] can record characters in registers, and check only equality. Recently, symbolic register automata [7] were proposed, which combines the two.

Our preliminary paper [6] presented an algorithm for synthesizing DFA transitions using SMT solvers. FLM extends that core with a new pattern-based language and compiler that separates bindings and predicates from classical regular expressions which are turned into DFAs.

11 Discussion

Limitations. One limitation for recognizing FLM patterns, and the reason for the rewrite rules, is the difficulty of correctly updating variables under the constraints on PISA switches. Other hardware or computation models might provide an easier path to computing the environment derivative (on a general purpose CPU with unlimited memory, it could just be computed directly), which would increase the number of FLM patterns that are recognizable. Still, we show in our evaluations that the subset of implementable patterns is expressive enough for many applications. An interesting research question for the future is to characterize which FLM patterns can be implemented under which computation models, and how best to do so while minimizing resource use.

Another limitation comes from the amount of computation available in a single stage, which dictates the size of implementable DFAs. Very long or complicated patterns could translate to DFAs that are too large to fit into one read-modify-write action. For more details, see the paragraph on synthesis hardness above. Our work is only platform-specific in that the template defined in Figure 5 is tailored to compilation on the Intel Tofino. Other hardware that has different computation available in a single stage would likely permit a more or less expressive template. A solution to this that applies for some DFAs is to carefully use more than one stage to implement it. For preliminary details about this approach, see the Appendix. As hardware improves, we hope to see both more computation available within a stage and more stages, alleviating this restriction from two angles.

Scalability and flexibility. In our examples, we used a variety of indexing functions to represent individual state machines, including per-flow, per-port, and per-MAC. However, in some networks, there is not enough memory available on current hardware to store this many values, risking index collisions. Other works [21] have dealt with this problem using complex data structures, sampling, or grouping flows to be considered together. While FLM does not employ any of these by default, it is flexible enough that a programmer could implement any of these techniques to compute an index before applying transitions to the FLM state machine array. Furthermore, if the high-level language is too restricting, a programmer can write code directly in the FLM intermediate representation, allowing them to intersperse arbitrary code for how the index is computed, which event is used to transition the state machine, and where in the control flow to transition.

Monitoring scope. We built our FLM compiler to target the Intel Tofino, but the core ideas are not reliant on any particular piece of hardware. The main requirement for an FLM program is a single pipeline with atomic updates and persistent memory that is updated per-packet. This applies to any switch implementing the PISA architecture.

We did not solve the problem of recognizing patterns in a distributed system of switches, instead focusing on how to properly compile to a single pipeline. A switch with multiple pipelines would implement multiple FLM monitors independently. For most monitoring tasks, properly configuring how ports map to pipelines (essentially slicing the index space across monitors) should preserve the monitor's reliability by sending packets intended for the same state machine through the same pipeline. We consider distributed monitoring an interesting task for future research.

12 Conclusion

We introduce FLM, a programming language that uses new abstractions to recognize and react to user-defined packet and event sequences at a switch. We develop a compilation procedure that transforms FLM programs into a series of match-action tables and register update functions, using a combination of rewrite rules and SMT-based program synthesis. Our evaluation using 15 sequence monitoring finds that FLM is flexible and supports line-rate processing on current networking hardware. This work raises no ethical concerns.

Acknowledgments

We would like to thank our shepherd, Macro Chiesa, and all of the anonymous reviewers for their insightful comments. This material is based upon work supported by the National Science Foundation Grants CNS-2007073, FMiTF-2219862, and FMiTF-2219863. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] Étienne André, Michał Knapik, Didier Lime, Wojciech Penczek, and Laure Petrucci. *Parametric Verification: An Introduction*, pages 64–100. Springer Berlin Heidelberg, Berlin, Heidelberg, 2019.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, jul 2014.

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM 2013*, pages 99–110, 2013.

[6] Xiaoqi Chen, Andrew Johnson, Mengying Pan, and David Walker. Synthesizing state machines for data planes. In *Proceedings of the Symposium on SDN Research*, SOSR '22, page 81–88, New York, NY, USA, 2022. Association for Computing Machinery.

[7] Loris D'Antoni, Tiago Ferreira, Matteo Sammartino, and Alexandra Silva. Symbolic register automata. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 3–21, Cham, 2019. Springer International Publishing.

[8] Loris D'Antoni, Zachary Kincaid, and Fang Wang. A symbolic decision procedure for symbolic alternating finite automata. *Electronic Notes in Theoretical Computer Science*, 336, 10 2016.

[9] Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 47–67, Cham, 2017. Springer International Publishing.

[10] Rennie Degraaf, John Aycock, and Michael Jacobson. Improved port knocking with strong authentication. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 10–pp. IEEE, 2005.

[11] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 357–371, New York, NY, USA, 2018. Association for Computing Machinery.

[12] Joel Hypolite, John Sonchack, Shlomo Hershkop, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. Deepmatch: Practical deep packet inspection in the data plane using network processors. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, page 336–350, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Intel. Intel tofino series programmable ethernet switch asic. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html, 2022.

[14] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. Fast string searching on PISA. In *Proceedings of the 2019 ACM Symposium on SDN Research (SOSR 19)*, pages 21–28, 2019.

[15] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, 2018.

[16] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[17] K Shiv Kumar, Ranjitha K, P S Prashanth, Mina Tahmasbi Arashloo, Venkanna U., and Praveen Tammana. Dbval: Validating p4 data plane runtime behavior. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, SOSR '21, page 122–134, New York, NY, USA, 2021. Association for Computing Machinery.

[18] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. Atp: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761, 2021.

[19] Devon Loehr and David Walker. Safe, modular packet pipeline programming. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

[20] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. Fast regular expression matching using small tcams for network intrusion detection and prevention systems. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, page 8, USA, 2010. USENIX Association.

[21] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 85–98, New York, NY, USA, 2017. Association for Computing Machinery.

[22] Derek Chi-Wai Pao and Xing Wang. Multi-stride string searching for high-speed content inspection. *Comput. J.*, 55:1216–1231, 2012.

[23] Julien Piet, Dubem Nwoji, and Vern Paxson. Ggfast: Automating generation of flexible network traffic classifiers. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 850–866, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. Beauty and the burst: Remote identification of encrypted video streams. In *USENIX Security Symposium*, pages 1357–1374, 2017.

[25] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. In *ACM SIGCOMM 2021*, pages 731–747, 2021.

[26] The P4 Language Consortium. $P4_{16}$ language specification. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html, November 2018.

[27] Shicheng Wang, Menghao Zhang, Guanyu Li, Chang Liu, Zhiliang Wang, Ying Liu, and Mingwei Xu. Bolt: Scalable and cost-efficient multistring pattern matching with programmable switches. *IEEE/ACM Transactions on Networking*, pages 1–16, 2022.

[28] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. Aragog: Scalable runtime verification of shardable networked systems. In *Operating systems and implementations (OSDI)*, October 2020.

[29] Sophia Yoo and Xiaoqi Chen. Secure keyed hashing on programmable switches. In *Proceedings of the ACM SIGCOMM 2021 Workshop on Secure Programmable network INfrastructure*, pages 16–22, 2021.

[30] z3. z3 solver, publisher = GitHub, howpublished = https://github.com/z3prover/z3/, urldate=2022-06-28, year=2022.

[31] Lior Zeno, Dan R. K. Ports, Jacob Nelson, and Mark Silberstein. Swishmem: Distributed shared state abstractions for programmable switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 160–167, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Lior Zeno, Dan RK Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. {SwiSh}: Distributed shared state abstractions for programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 171–191, 2022.

13 Appendix

13.1 Translating patterns

The full definition of translating patterns uses *BigOr*, which translates an individual event and predicate.

Definition 6. $BigOr(b\langle p\rangle, P) =$
$\bigvee(\{b\} \circ \{bin(i) \mid i < 2^{len(P)}$ *and the bit representing* p *is* $1\})$

$BigOr(b\langle p_1\rangle, [p_1, p_2]) = (b10 + b11)$, as above. Similarly, $BigOr(b\langle p_2\rangle, [p_1, p_2]) = (b01 + b11)$. The other pattern translations are defined recursively. All of the constructors $(+, \&, *, .)$ stay the same, as they have similar semantics for classical regular expressions and patterns, and we translate all of the event patterns with *BigOr*.

Definition 7. *The FLM pattern translation* $T_{re}(r, P)$*:*

$$\begin{aligned}
T_{re}(\varepsilon, P) &= \varepsilon \\
T_{re}(\emptyset, P) &= \emptyset \\
T_{re}(a\langle p_i\rangle, P) &= BigOr(a\langle p_i\rangle, P) \\
T_{re}(s_1.s_2) &= T_{re}(s_1, P).T_{re}(s_2, P) \\
T_{re}(s_1 + s_2) &= T_{re}(s_1, P) + T_{re}(s_2, P) \\
T_{re}(s_1 \,\&\, s_2) &= T_{re}(s_1, P) \,\&\, T_{re}(s_2, P) \\
T_{re}(s^*) &= (T_{re}(s, P))^*
\end{aligned}$$

13.2 Implementability

The following is a list of the rewrite rules we use:

- $b\langle @y; p\rangle.r \xrightarrow{rw} b\langle @y\rangle \triangleright b\langle p\rangle.r$. This rule always applies, and is how we introduce the $\triangleright$ operator. It splits a binding into two parts, and removes it from the expression.
- $r.(b\langle @y\rangle \triangleright s) \xrightarrow{rw} b\langle @y\rangle \triangleright (r.s)$. This rule only applies if b and y do not appear in r. This ensures that no word in r contains an event b, mirroring the semantics of $\triangleright$.
- $(b\langle @y_1\rangle \triangleright s_1) + (b\langle @y_2\rangle \triangleright s_2) \xrightarrow{rw} b\langle @y_1\rangle \triangleright (s_1 + [y_1/y_2]s_2)$. Here $[y_1/y_2]s_2$ denotes the capture-avoiding substitution of y_1 for y_2 in s_2. This rule only applies when s_2 contains no references to y_1. An equivalent rule applies for &.
- $(a\langle @y_1\rangle \triangleright s_1) + (b\langle @y_2\rangle \triangleright s_2) \xrightarrow{rw} a\langle @y_1\rangle \triangleright b\langle @y_2\rangle \triangleright (s_1 + s_2)$. This rule applies if s_1 contains no occurrences of y_2, s_2 contains no occurrences of y_1, and $a \neq b$. An equivalent rule applies for &.
- $(b\langle @y\rangle \triangleright s_1) + s_2 \xrightarrow{rw} b\langle @y\rangle \triangleright (s_1 + s_2)$. This rule applies if s_2 contains no occurrences of y. An equivalent rule applies for &.

These contain side conditions that ensure rewritten expressions have the same semantics after rewriting. For example, we check that variables are not contained in out-of-scope expressions before hoisting bindings to an outer scope. However, they also preserve a key property of FLM patterns that can be written *without* the new binding form that uses $\triangleright$. We call this property *implementability*. First, we define a shorthand for an environment defined over the variables in a list of bindings.

Definition 8. *An environment* G *is compatible with a binding list* B *if* $Dom(G) = Range(B)$*. That is,* G *contains one assignment for each variable in* B*. We will use the metavariable* G *for compatible environments to differentiate from general environments denoted by* E*.*

For example, the environment $(y_1 \leftarrow 1, y_2 \leftarrow 15)$ is compatible with $b_1\langle @y_1\rangle \triangleright b_2\langle @y_2\rangle$. We will write $Asgn0(B)$ to mean the environment that assigns every variable in B to 0. Next, we define implementability, which intuitively means that the pattern derivatives are not affected by assignments to variables in a binding list for event types that do not appear in the bindings.

Definition 9. *An FLM pattern in prefix form* $B \triangleright s$ *is implementable if and only if for any event type* $a \notin B$*, any integer* z*, any environment* E *where* $B \triangleright s$ *is closed under* E*, and any two environments* G_1 *and* G_2 *which are compatible with* B*:*

$$[\![D_{re}(a\langle z\rangle; s; E, G_1)]\!]_{E,G_2} = [\![D_{re}(a\langle z\rangle; s; E, G_2)]\!]_{E,G_2}$$

All of the patterns we introduce in the main paper are implementable. An example pattern which does not have this property is:

$$b\langle @y\rangle \triangleright a\langle \lambda x.(x == y)\rangle.b\langle \lambda x.true\rangle$$

This pattern is semantically well-defined: it is the set of events of type a followed by b with equal parameters. However, the value of y is not known when the event a appears, and so it cannot be implemented at line-rate (in general, this type of pattern would require some look-back capability). This fails the implementability property because the assignment to y will change the result of D_{re} for events of type a. We show that our rewrite rules preserve this property: if $r \xrightarrow{rw} r'$ and r is implementable, then so is r'.

Now, we define a theorem that captures the semantic meaning of *both* the binding and pattern derivatives. Intuitively, it is similar to theorem 1, but uses an environment produced by D_{bind} instead of a constant one.

Theorem 3. *$\forall B, s, a, z, E$: if $B \triangleright s$ is in prefix form, closed under E, and implementable, then:*

$$[\![D_{re}(a\langle z\rangle; s; E', Asgn0(B'))]\!]_{E', Asgn0(B')} = \{w | a\langle z\rangle.w \in [\![B \triangleright s]\!]_E\}$$

Where $B', E' = D_{bind}(a\langle z\rangle; B; E)$

As an example, consider the one from section 5, where $p_1 = \lambda x.x \geq 10$ and $p_2 = \lambda x.x == y$:

$$b\langle @y\rangle \triangleright (b\langle p_1\rangle.b\langle p_2\rangle)$$

Starting with an empty environment and an event $b\langle 12\rangle$, $D_{bind}(b\langle 12\rangle; b\langle @y\rangle; .) = (.; (y \leftarrow 12))$. Taking the pattern derivative using the new environment:

$$D_{re}(b\langle 12\rangle; b\langle \lambda x.x \geq 10\rangle.b\langle p_1\rangle; y \leftarrow 12) = b\langle p_2\rangle$$

The semantics of this remaining pattern when $y \leftarrow 12$ contains just $b\langle 12\rangle$, which is correct according to the theorem: the only string starting with $b\langle 12\rangle$ in $[\![b\langle p_1\rangle.b\langle p_2\rangle]\!]_{y\leftarrow 12}$ is $b\langle 12\rangle b\langle 12\rangle$.

13.3 Example patterns

Below is a full list of the example patterns that were used for evaluation. They range from simple checks to more complicated patterns about high-level protocols.

A1 Cuckoo Firewall [25]

```
(ip_pkt(@int saved_src=src, @int saved_dst=dst)
        .(((cuckoo_insert(fst_src==saved_src) && cuckoo_insert(fst_dst==saved_dst))*
              .(cuckoo_insert(!(fst_src==saved_src)) || cuckoo_insert(!(fst_dst==saved_dst)))
            )||((cuckoo_insert(fst_src==saved_src) && cuckoo_insert(fst_dst==saved_dst))
            .(cuckoo_insert(fst_src==saved_src) && cuckoo_insert(fst_dst==saved_dst))
            .(cuckoo_insert(fst_src==saved_src) && cuckoo_insert(fst_dst==saved_dst))
            .(cuckoo_insert(fst_src==saved_src) && cuckoo_insert(fst_dst==saved_dst)))))
```

This checks whether the cuckoo firewall insertion algorithm is working properly.

B1 Stateful FW Timeout [25]

```
ip_pkt (@int start_time = Sys.time(); ip#tos == TOS_TRUSTED)
    .(ip_pkt(ip#tos == TOS_TRUSTED)
        || ((ip_pkt(!(ip#tos == TOS_TRUSTED))
            && ip_pkt(Sys.time() - start_time < 10000))))*
    .((ip_pkt(!(ip#tos == TOS_TRUSTED))
            && ip_pkt(!(Sys.time() - start_time < 10000))))
```

This is a general specification of firewall correctness. It checks that packets from inside (`TOS_TRUSTED`) are allowed out, and that return packets are not allowed back in past the timeout threshold.

C1 SipHash [29]

```
iptcp_to_server_syn
.siphash_intermediate
.siphash_intermediate
.siphash_intermediate
.siphash_intermediate
```

```
.siphash_intermediate
.siphash_intermediate
.siphash_intermediate
.siphash_intermediate
.siphash_intermediate
.siphash_intermediate
.siphash_intermediate
.siphash_intermediate
.siphash_intermediate
.iptcp_craft_out_dummy
```

This checks whether or not the siphash implementation is completing the proper number of hashing rounds.

D1 Chain replication 1 [31]

```
write(@int saved_seq=seq)
.write(seq<saved_seq)
```

This checks whether there are write events with sequence numbers out of order.

D2 Chain replication 2 [31]

```
write()
.write()
.ack()
```

This checks whether there are two write events to the same index before the first one is `ACK`ed.

E1 DHCP Anomaly (section 3)

```
IP_Pkt*
.DHCP_Ack(@int assigned = cip)
.(IP_Pkt(sip == assigned))*
.IP_Pkt(sip != assigned)
```

This is the example from the paper.

E2 Fingerprint ([24])

```
S1.S5.S1.S4.S7.S8.S1.S2
```

This represents one example video fingerprint. `S1` - `S8` denote different packet sizes. The fingerprint is 8 packets in sequence, with the denoted sizes.

E3 Port Knocking (len=4) [10]

```
ip_in(dport==1234)
.ip_in(dport==5678)
.ip_in(dport==9012)
.ip_in(dport==3456)
```

This pattern represents an example port knocking sequence. `dport` is the destination port of a packet.

E4 DNS TTL Change Count [21]

```
 DNS_packet_fwd(@int<<32>> fst_ttl = ttl)
.(DNS_packet_fwd(fst_ttl != ttl))
```

This checks whether the ttl of packets changes when using DNS by recording the first in a flow and checking the second. The intent is to count the number of TTL changes.

E5 DNS Tunneling [21]

```
DNS_resp(@int d1=dip, @int d2=dip2)
.(  (ip_packet(dip!=d1) && ip_packet(dip!=d2))
    .(ip_packet(dip!=d1) && ip_packet(dip!=d2))
    .(ip_packet(dip!=d1) && ip_packet(dip!=d2))
    .(ip_packet(dip!=d1) && ip_packet(dip!=d2))
    .(ip_packet(dip!=d1) && ip_packet(dip!=d2))
    .(ip_packet(dip!=d1) && ip_packet(dip!=d2))
    .(ip_packet(dip!=d1) && ip_packet(dip!=d2))
    .(ip_packet(dip!=d1) && ip_packet(dip!=d2))
    .(ip_packet(dip!=d1) && ip_packet(dip!=d2))
```

This checks that, upon receiving a DNS response, the receiver goes on to contact the requested IP. Tunneling is suspected if the receiver of a DNS response never uses the information.

E6 SwiSh Local View [32]

```
 (S1||S2||S3)*
 .( (S2.S1)||
    (S3.S2)||
    (S1.S3))
```

This denotes any of the switches in a chain of 3 SwiShMem switches forwarding an update to the wrong neighbor.

E7 NetChain [15]

```
NetChainUpdate(@int v=version)
.NetChainUpdate(version < v)
```

This shows an algorithm running improperly, as detected by having an earlier version after a newer one.

E8 Paxos Recovery [28]

```
Paxos(@int a=l1,@int c=l2,@int e=l3; ty==RECOVER).
(
    (Paxos(l1<a) && Paxos(ty==RECOVER)) ||
    (Paxos(l2<c) && Paxos(ty==RECOVER)) ||
    (Paxos(l3<e) && Paxos(ty==RECOVER))
).
Paxos(ty==RECOVERED)
```

This shows the sequence of exchanges of a Paxos recovery.

E9 ATP sequence [18]

```
ATP_add(@int x = cnt)
.ATP_add(cnt==x+1)
.ATP_add(cnt==x+2)
.ATP_add(cnt==x+3)
```

This shows a sequence of 4 consecutive add events with an increasing count variable.

E10 ATP JobID [18]

```
ATP_add(@int saved_jobid=jobid)
.(ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
```

```
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_release(jobid==saved_jobid))
```

This shows a sequence of 16 consecutive add events with the same job ID.

Unimplementable patterns Below are a few patterns that are rejected:

R1 DHCP Example Reject

```
IP_Pkt*
.DHCP_Ack(@int assigned = cip)
.(IP_Pkt(sip == assigned))*
.IP_Pkt(@int unassigned = sip; sip != assigned)
.IP_Pkt(sip == unassigned)
.IP_Pkt(sip == unassigned)
```

This pattern is rejected because the binding of the variable `unassigned` will not be on the *first* occurrence of the event `IP_Pkt`. A similar limitation applies to any compound, related patterns. It is meant to represent a device that uses the *same* wrong IP three times.

R2 ATP Sequence Version 2

```
(ATP_add(@int x = cnt; cnt == x+1))*
```

This pattern has two problems. First, the variable x is used before it is defined (the predicate `cnt == x+1` does not have the variable `x` in scope). Second, bindings can never occur under a closure. It is meant to loosely specify a similar pattern to E9, but of any length.

R3 Chain Replication Reject

```
write(@int saved_seq=seq)
write(@int saved_seq2=seq; seq > saved_seq)
.write(seq<saved_seq2)
```

This pattern is rejected for a reason similar to R1: the binding of variable `saved_seq2` will not occur on the first `write` event (even though the first `write` event does bind a variable). This does not apply when binding two variables with the *same* event (see E5).

13.4 Extensions

In this section, we go over various extensions to FLM to improve its expressiveness and usability.

13.4.1 Maintenance events

Some patterns, such as those that wait for a timeout, can be difficult to express as a sequence. Consider a sequence to check that received requests are replied to with decisions in a timely manner:

```
request(@t1 = Sys.time())
.request*
.decision(Sys.time() - t1 > Threshold)
```

Here, we are ignoring the indexing and response code to focus on the pattern. This pattern seems reasonable to detect *late* decisions, but what if a decision never comes? Intuitively, that should be a violation as well, but if there is never another packet for this flow, one will never be reported. The problem is that examples such as timeouts are examining a liveness property. Our solution to this is to add *maintenance events*, which do not represent incoming network events. Instead, they are guaranteed to visit every index of an array of FLM patterns eventually. To fix the above example, we can write the following, using `maintenance` for the new events.

```
maintenance*
.request(@t1 = Sys.time())
.(maintenance(Sys.time() - t1 <= Threshold) + request)*
.(decision(Sys.time() - t1 > Threshold) +
(maintenance(Sys.time() - t1 > Threshold)))
```

The placement of maintenance events will not interfere with detecting late decisions, but it will now also detect decisions that never arrive because the maintenance event will eventually come along and match the last disjunction. These can be implemented easily with a packet that repeatedly circulates through the array one index at a time, perhaps with a delay to reduce overhead.

13.4.2 Longer patterns

Some sequence monitors are too complicated to be expressed using the original syntax, causing the compiler to reject them. This could be for one of two reasons:

1. The pattern is unable to be rewritten into prefix form, usually because the programmer desires to bind variables using the same event type in two places. This would violate the rewrite rule for concatenation.

2. The pattern is able to be rewritten successfully, but an implementation of the transition function for the translated DFA cannot be found. The synthesis algorithm might either time out, or return "unsat."

The solution to both of these is to pay more stages for the ability to implement a pattern. To do so, we introduce *unambiguous concatenation*, which allows a pattern to be split into two parts that can be implemented sequentially.

Unambiguous concatenation We say that two FLM patterns in prefix form are *unambiguously concatenated* with a semantic condition that allows us to split it across stages. First, we define the *prefixes* of a pattern, which are all the prefixes of any accepted word:

Definition 10. *The prefixes of an FLM pattern r, denoted prefix(r), is the set:*

$$\{u|\ \exists E, v \text{ such that } u.v \in [\![r]\!]_E\}$$

Next, we define the *continuations* of a pattern, which are all the words which can be appended to an accepted word to get another accepted word:

Definition 11. *The Continuations of an FLM pattern r, denoted continuation(r) is the set:*

$$\{v|\ \exists E, u, w \text{ such that } u \in [\![r]\!]_E \text{ and } u.v.w \in [\![r]\!]_E\}$$

As a simple example with the pattern $a.b^*$, the prefixes and continutations can be defined with the languages of the following patterns:

$$prefix(a.b^*) = \varepsilon + a.b^*$$
$$continuation(a.b^*) = b^*$$

Finally, we say that two patterns r_1 and r_2 are unambiguously concatenated, denoted $r_1!!r_2$, if the prefixes of r_1 and the continuations of r_2 only intresect with ε, or more formally:

$$r_1!!r_2 \iff (prefix(r_1) \cap continuation(r_2)) \setminus \{\varepsilon\} = \emptyset$$

Using the above example, $a.b^*!!a.b^*$ is a valid unambiguous concatenation. ε is excluded because it is always both a prefix and continuation of any non-empty language.

Implementation The unambiguous concatenation condition lets us compile a single pattern into multiple stages, which will either allow a programmer to reuse events for variable bindings or compile a larger pattern to switch actions. In principle, there could be many patterns connected with !!. We assume that this is at the top level, and call each concatenated pattern a *section*. We compile a series of sections in three steps, after compiling each section individually:

1. For each DFA except the last section, add one new state, called "done." Add a self-loop for every character to "done." For each transition from any accepting state to the rejecting state, replace it with a transition to "done."

2. For each section's code except the first, add a clause to only run it if the previous section's state was "done."

3. In each section's DFA, if any later sections are not nullable, change all of its accepting states to non-accepting ones. The `transition` statement returns whether or not the last section run ended in an accepting state.

Step one allows us to track when each section has finished matching characters. This is the step where the unambiguous concatenation condition is important. The transitions from accepting states to non-reject states correspond to the *continuations* of a section, while the transitions from the initial state correspond to its *prefixes*. The condition guarantees that the prefixes of a later section do not coincide with the continuations of an earlier one, so we never miss a transition.

Step two ensures that we are running the patterns in sequence, not in parallel. Note that if there are new variables to be bound, they are only bound *after* previous sections are "done."

Step three ensures that the series of sections only accepts when the current string matches the unambiguously concatenated pattern. We leave accepting states if all later states are nullable because otherwise, we would miss some strings that only match the earlier sections.

13.5 Proofs of Theorems

Theorem 4. *"Derivatives commute":*
$\forall a, z, s, E, P$ where s is binding free, both s and each predicate in P is closed under E, and $preds(s) \subseteq P$:

$$D_{clas}(T_l(a\langle z\rangle, E, P); T_{re}(s, P)) = T_{re}(D_{re}(a\langle z\rangle; s; E), P)$$

Proof. By induction on the structure of s □

Theorem 5. *For all a, z, s, E where s is binding-free and s is closed under E:*

$$[\![D_{re}(a\langle z\rangle; s; E)]\!]_E = \{w' | a\langle z\rangle.w' \in [\![s]\!]_E\}$$

Proof. Proof by induction on the structure of s. Base cases:

1. $\emptyset$:

$$[\![D_{re}(a\langle z\rangle; \emptyset; E)]\!]_E$$
$$= [\![\emptyset]\!]_E = \emptyset = \{w | a\langle z\rangle.w \in \emptyset\}$$

2. ε:

$$[\![D_{re}(a\langle z\rangle; \varepsilon; E)]\!]_E$$
$$= [\![\emptyset]\!]_E = \emptyset = \{w | a\langle z\rangle.w \in \{\varepsilon\}\}$$

3. $a\langle p\rangle : a\langle z\rangle \in [\![a\langle p\rangle]\!]_E$ iff $[\![p(z)]\!]_E$. So, $\{w | a\langle z\rangle.w \in [a\langle z\rangle]_E\} = \{\varepsilon\}$ if $[\![p(z)]\!]_E$ and $\emptyset$ else, which is the derivative.

Induction cases:

1. r + s:

$$[\![D_{re}(a\langle z\rangle; r+s; E)]\!]_E$$
$$= [\![D_{re}(a\langle z\rangle; r; E)]\!]_E \cup [\![D_{re}(a\langle z\rangle; s; E)]\!]_E$$

   By induction:

$$= \{w | a\langle z\rangle.w \in [\![r]\!]_E\} \cup \{w | a\langle z\rangle.w \in [\![s]\!]_E\}$$
$$= \{w | a\langle z\rangle.w \in [\![r]\!]_E \cup [\![s]\!]_E\}$$
$$= \{w | a\langle z\rangle.w \in [\![r+s]\!]_E\}$$

2. r & s: Very similar to above, substituting & and $\cap$ for $+$ and $\cup$.

3. s^*:
   The following together show that $[\![D_{re}(a\langle z\rangle; s^*; E)]\!]_E = \{w | a\langle z\rangle.w \in [\![s^*]\!]_E\}$

(a) $\{w|a\langle z\rangle.w \in [\![s^*]\!]_E\} \subseteq [\![D_{re}(a\langle z\rangle; s^*; E)]\!]_E$:
If $a\langle z\rangle.w \in [\![s^*]\!]_E$, then $a\langle z\rangle.w \in [\![s^i]\!]_E$ for some smallest natural i. i cannot be 0. Since $a\langle z\rangle.w \in [\![s^i]\!]_E = [\![s.s^{i-1}]\!]_E$, $a\langle z\rangle.w = w_1.w_2$ s.t. $w_1 \in [\![s]\!]_E \wedge w_2 \in [\![s^{i-1}]\!]_E$. s_1 cannot be ε, otherwise i could be smaller. So, $w_1 = a\langle z\rangle.w_1'$ and $w = w_1'.w_2$.
By induction,

$$[\![D_{re}(a\langle z\rangle; s; E)]\!]_E = \{w|a\langle z\rangle.w \in [\![s]\!]_E\}$$

So, $w_1' \in [\![D_{re}(a\langle z\rangle; s; E)]\!]_E$. $w_2 \in [\![s^*]\!]_E$, so $w \in [\![D_{re}(a\langle z\rangle; s; E).s^*]\!]_E = [\![D_{re}(a\langle z\rangle; s^*; E)]\!]_E$

(b) $[\![D_{re}(a\langle z\rangle; s^*; E)]\!]_E \subseteq \{w|a\langle z\rangle.w \in [\![s^*]\!]_E\}$:
If $w \in [\![D_{re}(a\langle z\rangle; s^*; E)]\!]_E$, then: $w = s_1.s_2$, where:

$$s_1 \in [\![D_{re}(a\langle z\rangle; s; E)]\!]_E \text{ and } s_2 \in [\![s^*]\!]_E$$

By induction, $a\langle z\rangle.s_1 \in [\![s]\!]_E$. So, $s_1.s_2 \in [\![s^*]\!]_E$

4. *r.s*:
The following together show that $[\![D_{re}(a\langle z\rangle; r.s; E)]\!]_E = \{w|a\langle z\rangle.w \in [\![r.s]\!]_E\}$

(a) $\{w|a\langle z\rangle.w \in [\![r.s]\!]_E\} \subseteq [\![D_{re}(a\langle z\rangle; r.s; E)]\!]_E$:
If a word $a\langle z\rangle.w \in [\![r.s]\!]_E$, then by definition $a\langle z\rangle.w = w_1.w_2$ s.t. $w_1 \in [\![r]\!]_E \wedge w_2 \in [\![s]\!]_E$. By definition, $[\![D_{re}(a\langle z\rangle; r.s; E)]\!]_E = [\![D_{re}(a\langle z\rangle; r; E).s]\!]_E \cup [\![v(r).D_{re}(a\langle z\rangle; s; E)]\!]_E$. There are two cases
   i. $w_1 = a\langle z\rangle.w_1'$. Then by induction, $w_1' \in [\![D_{re}(a\langle z\rangle; r; E)]\!]_E$, so $w_1'.w_2 \in [\![D_{re}(a\langle z\rangle; r; E).s]\!]_E$
   ii. $w_1 = \varepsilon$, and $w_2 = a\langle z\rangle.w_2'$. Since $w_1 = \varepsilon \in [\![r]\!]_E$, $v(r) = \varepsilon$. By induction, $w_2' \in [\![D_{re}(a\langle z\rangle; s; E)]\!]_E$, so $w \in [\![v(r).D_{re}(a\langle z\rangle; s; E)]\!]_E$

This includes all possibilities for any word in $\{w|a\langle z\rangle.w \in [\![r.s]\!]_E\}$.

(b) $[\![D_{re}(a\langle z\rangle; r.s; E)]\!]_E \subseteq \{w|a\langle z\rangle.w \in [\![r.s]\!]_E\}$:
If $w \in [\![D_{re}(a\langle z\rangle; r.s; E)]\!]_{D_{bind}(a\langle z\rangle; r.s; E)}$, then either:
   i. $w \in [\![D_{re}(a\langle z\rangle; r; E).s]\!]_E$. By induction, $w = w_1'.w_2$ such that $a\langle z\rangle.w_1' \in [\![r]\!]_E$ and $w_2 \in [\![s]\!]_E$. So, $a\langle z\rangle.w \in [\![r.s]\!]_E$.
   ii. $w \in [\![v(r).D_{re}(a\langle z\rangle; s; E)]\!]_E$.Then either $v(r) = \emptyset$ and there are no such w, or $v(r) = \varepsilon$ and by induction, $a\langle z\rangle.w \in [\![s]\!]_E$. Since r is nullable, $[\![s]\!]_E \subseteq [\![r.s]\!]_E$.

This includes all possibilities for any word in $[\![D_{re}(a\langle z\rangle; r.s; E)]\!]_E$

□

Lemma 1. *For any $B \triangleright b_1\langle @y_1\rangle \triangleright b_2\langle @y_2\rangle \triangleright B' \triangleright s$ in prefix form and any E it is closed under:*

$$[\![B \triangleright b_1\langle @y_1\rangle \triangleright b_2\langle @y_2\rangle \triangleright B' \triangleright s]\!]_E = [\![B \triangleright b_2\langle @y_2\rangle \triangleright b_1\langle @y_1\rangle \triangleright B' \triangleright s]\!]_E$$

Proof. By induction on the length of B preceding the two to be exchanged. In the base case, expand the definitions twice. In the induction case, just exchange the smaller list. □

Lemma 2. *If $b\langle @y\rangle \triangleright B \triangleright s$ is implementable, then so is $B \triangleright s$.*

Proof. By expanding the semantics to get some value for y, and then using impl property on the compatible environments with B. □

Lemma 3. *If $B \triangleright s$ is implementable, then for any $a \notin B,z$, E where $B \triangleright s$ is closed, and compatible environment G, $B \triangleright D_{re}(a\langle z\rangle; s; (E,G))$ is implementable as well.*

Proof. By induction on the structure of s. □

Theorem 6. *Restatement of Theorem 3: For all a,z,B,s,E where $B \triangleright s$ is in prefix form (B is a non-redundant list of bindings and s is binding-free) and implementable:*

$$[\![B' \triangleright D_{re}(a\langle z\rangle; s; E', Asgn0(B'))]\!]_{E', Asgn0(B')} = \{w'|a\langle z\rangle.w' \in [\![B \triangleright s]\!]_E\}$$

where $B', E' = D_{bind}(a\langle z\rangle, B, E)$

Proof. By induction on the number of bindings in B. When B has no bindings, just use 1. When B has a binding with event b, proceed by cases on whether $a \in B$.

1. If $a \in B$, then by 1, we can assume without loss of generality that it is the first binding in B (i.e. $B = a\langle @y\rangle \triangleright B' \triangleright s$. Also, since B is not redundant, $D_{bind}(a\langle z\rangle; B'; E, y \leftarrow z) = (B', E, y \leftarrow z)$. Then $D_{bind}(a\langle z\rangle; B; E) = (B', (E, y \leftarrow z))$. Next, we can expand the definition of $[\![B \triangleright s]\!]_E$:

$$[\![a\langle @y\rangle \triangleright B' \triangleright s]\!]_E = \underbrace{\{w_1.a\langle z'\rangle.w_2 \in [\![B' \triangleright s]\!]_{E,y\leftarrow z'} \text{ and } a \notin w_1\}}_{\mathbf{S1}} \cup \underbrace{\{w | \forall z'.w \in [\![B' \triangleright s]\!]_{E,y\leftarrow z'} \text{ and } a \notin w\}}_{\mathbf{S2}}$$

   There are clearly no $a\langle z\rangle.w \in \mathbf{S2}$, since they cannot contain the event a. Furthermore, if any word $a\langle z\rangle.w = w_1.a\langle z\rangle.w_2 \in \mathbf{S1}$, w_1 must be ε. So:

$$\begin{aligned}&\{w' | a\langle z\rangle.w' \in [\![a\langle @y\rangle \triangleright B' \triangleright s]\!]_E\}\\ &= \{w' | a\langle z\rangle.w' \in \mathbf{S1}\}\\ &= \{w' | a\langle z\rangle.w' \in [\![B' \triangleright s]\!]_{E,y\leftarrow z}\}\end{aligned}$$

   We now want to show that this set is equal to $[\![B' \triangleright D_{re}(a\langle z\rangle; s; E, y \leftarrow z, Asgn0(B'))]\!]_{E,y\leftarrow z,Asgn0(B')}$, which is true directly by the induction hypothesis. :

$$\begin{aligned}&[\![B' \triangleright D_{re}(a\langle z\rangle; s; E, y \leftarrow z, Asgn0(B'))]\!]_{E,y\leftarrow z,Asgn0(B')}\\ &\qquad = \{w' | a\langle z\rangle.w' \in [\![B' \triangleright s]\!]_{E,y\leftarrow z}\} \quad \text{By induction}\end{aligned}$$

2. If $a \notin B$, then $B = b\langle @y\rangle \triangleright B' \triangleright s$. $D_{bind}(a\langle z\rangle; B; E) = (B, E)$, since a doesn't appear in the bindings. So, we can expand the definition of $[\![B \triangleright s]\!]_E$::

$$[\![b\langle @y\rangle \triangleright B' \triangleright s]\!]_E = \underbrace{\{w_1.b\langle z'\rangle.w_2 \in [\![B' \triangleright s]\!]_{E,y\leftarrow z'} \text{ and } b \notin w_1\}}_{\mathbf{S1}} \cup \underbrace{\{w | \forall z'.w \in [\![B' \triangleright s]\!]_{E,y\leftarrow z'} \text{ and } b \notin w\}}_{\mathbf{S2}}$$

   Next, we can expand the definition of $[\![b\langle @y\rangle \triangleright B' \triangleright D_{re}(a\langle z\rangle; s; E, Asgn0(b\langle @y\rangle \triangleright B'))]\!]_{E,Asgn0(b\langle @y\rangle \triangleright B')}$:

$$\begin{aligned}&[\![b\langle @y\rangle \triangleright B' \triangleright D_{re}(a\langle z\rangle; s; E, Asgn0(b\langle @y\rangle \triangleright B'))]\!]_{E,Asgn0(b\langle @y\rangle \triangleright B')}\\ &= [\![b\langle @y\rangle \triangleright B' \triangleright D_{re}(a\langle z\rangle; s; E, y \leftarrow 0, Asgn0(B'))]\!]_{E,y\leftarrow 0,Asgn0(B')}\\ &= \underbrace{\{w_1.b\langle z'\rangle.w_2 \in [\![B' \triangleright D_{re}(a\langle z\rangle; s; E, y \leftarrow 0, Asgn0(B'))]\!]_{E,y\leftarrow z',Asgn0(B')} \text{ and } b \notin w_1\}}_{\mathbf{S3}}\\ &\cup \underbrace{\{w | \forall z'.w \in [\![B' \triangleright D_{re}(a\langle z\rangle; s; E, y \leftarrow 0, Asgn0(B'))]\!]_{E,y\leftarrow z',Asgn0(B')}\}}_{\mathbf{S4}}\end{aligned}$$

   Because $b\langle @y\rangle \triangleright B' \triangleright s$ is implementable, so is $B' \triangleright D_{re}(a\langle z\rangle; s; E, y \leftarrow 0, Asgn0(B'))$ by using 2 and 3. Using this, we have the following in both **S3** and **S4**:

$$\begin{aligned}&[\![B' \triangleright D_{re}(a\langle z\rangle; s; E, y \leftarrow 0, Asgn0(B'))]\!]_{E,y\leftarrow z',Asgn0(B')}\\ &= [\![B' \triangleright D_{re}(a\langle z\rangle; s; E, y \leftarrow z', Asgn0(B'))]\!]_{E,y\leftarrow z',Asgn0(B')}\end{aligned}$$

   This lets us use the induction hypothesis on **S3** and **S4**, since $B' \triangleright D_{re}(a\langle z\rangle; s; E, y \leftarrow 0, Asgn0(B'))$ is implementable by 2 and 3:

$$[\![B' \triangleright D_{re}(a\langle z\rangle; s; E, y \leftarrow z', Asgn0(B'))]\!]_{E,y\leftarrow z',Asgn0(B')} = \{w | a\langle z\rangle.w \in [\![B' \triangleright s]\!]_{E,y\leftarrow z'}\}$$

   Plugging this into the definitions above gives the following:

$$\begin{aligned}&[\![b\langle @y\rangle \triangleright B' \triangleright D_{re}(a\langle z\rangle; s; E, Asgn0(b\langle @y\rangle \triangleright B'))]\!]_{E,Asgn0(b\langle @y\rangle \triangleright B')}\\ &= \underbrace{\{w' = w_1.b\langle z\rangle.w_2 | a\langle z\rangle.w' \in [\![B' \triangleright s]\!]_{E,y\leftarrow z'} \text{ and } b \notin w_1\}}_{\mathbf{S3'}}\\ &\cup \underbrace{\{w' | \forall z'.a\langle z\rangle.w' \in [\![B' \triangleright s]\!]_{E,y\leftarrow z'} \text{ and } b \notin w\}}_{\mathbf{S4'}}\end{aligned}$$

   This concludes the proof, since these two sets are the same as **S1** and **S2**, except for the requirement that the words start with $a\langle z\rangle$. Finally, it is clear that b is not in a word $a\langle z\rangle.w$ if b is not in w.

□

Lemma 4. *For all a, z, E, and B:*

$$D_{bind}(a\langle z\rangle; B; E, Asgn0(B)) = B', (E', Asgn0(B')$$

Where $B', E' = D_{bind}(a\langle z\rangle; B; E)$

Proof. Observe that the initial values in E do not matter for D_{bind}. If $a \in B$, then afterwards it is not in B, and each variable that was paired with a now has a value in E'. These values would have overwritten the 0 added from $Asgn0(B)$. The values of all other variables do not change. If $a \notin B$, then neither B nor E change. □

Lemma 5. *For all a, z, E, B, s where* $B \triangleright s$ *is closed under E:*

$$[\![B \triangleright s]\!]_E = [\![B \triangleright s]\!]_{E, Asgn0(B)}$$

Proof. By induction on the length of B.
Base case: When B is empty, $Asgn0(B)$ adds no variables, so they are the same.
Induction step: $B = b\langle @y\rangle \triangleright B'$:

$$\begin{aligned}
[\![B \triangleright s]\!]_{E, Asgn0(B)} &= \{w_1.b\langle z\rangle.w_2 \in [\![B' \triangleright s]\!]_{E, Asgn0(B), y\leftarrow z} | b \notin w_1\} \cup \{w | b \notin w \text{ and } \forall z. w \in [\![B' \triangleright s]\!]_{E, Asgn0(B), y\leftarrow z}\} \\
&= \{w_1.b\langle z\rangle.w_2 \in [\![B' \triangleright s]\!]_{E, y\leftarrow z, Asgn0(B')} | b \notin w_1\} \cup \{w | b \notin w \text{ and } \forall z. w \in [\![B' \triangleright s]\!]_{E, y\leftarrow z, Asgn0(B')}\} \\
\text{By induction:} \quad &= \{w_1.b\langle z\rangle.w_2 \in [\![B' \triangleright s]\!]_{E, y\leftarrow z} | b \notin w_1\} \cup \{w | b \notin w \text{ and } \forall z. w \in [\![B' \triangleright s]\!]_{E, y\leftarrow z}\} \\
&= [\![B \triangleright s]\!]_E
\end{aligned}$$

□

Theorem 7. *Restatement of Theorem 2: For any word w, bindings B, environment E, and predicates P, if an FLM pattern* $B \triangleright s$ *is in prefix form,* $B \triangleright s$ *is closed under E, and* $B \triangleright s$ *is implementable, then:* $T_w(w, B, (E, Asgn0(B)), P) \in L(T_{re}(s, P))$ *if and only if* $w \in [\![B \triangleright s]\!]_E$.

Proof. By induction on the length of w.

For the base case, T_{re} does not change the length of accepted words. So, $\varepsilon \in T_{re}(s, P) \iff \varepsilon \in [\![B \triangleright s]\!]_E$ (ε is the only word of length 0).

For the induction step: $w = a\langle z\rangle.w'$; $s' = D_{re}(a\langle z\rangle; s; E, Asgn0(B))$; and $B'; E' = D_{re}(a\langle z\rangle; B; E)$.
Starting with:

$$T_w(a\langle z\rangle.w', B, (E, Asgn0(B)), P) \in L(T_{re}(s, P))$$

By the definition of T_w and applying 4, This is equivalent to:

$$T_l(a\langle z\rangle, (E', Asgn0(B')), P).T_w(w', B', (E', Asgn0(B'), P) \in L(T_{re}(s, P))$$

Where $B', (E', Asgn0(B') = D_{bind}(a\langle z\rangle, B, (E, Asgn0(B)))$. By the definition of the classic derivative, this is equivalent to:

$$T_w(w', B', (E', Asgn0(B'), P) \in L(D_{clas}(T_l(a\langle z\rangle, (E', Asgn0(B'), P); T_{re}(s, P))))$$

By 4, this is equivalent to:

$$w' \in [\![B' \triangleright D_{re}(a\langle z\rangle, s, (E', Asgn0(B'))]\!]_{(E', Asgn0(B')}$$

By 6, this is true if and only if:

$$a\langle z\rangle.w' \in [\![B \triangleright s]\!]_{E, Asgn0(B)}$$

Finally, we can use 5 to get the result:

$$w \in [\![B \triangleright s]\!]_E$$

□

Lemma 6. *If $r \xrightarrow{rw} r'$ and r is implementable, then for all E such that r and r' are closed under E, $[\![r]\!]_E = [\![r']\!]_E$.*

Proof. By cases on which rewrite rule is used.

1. $r = b\langle @y; p\rangle.s \xrightarrow{rw} b\langle @y\rangle \triangleright b\langle p\rangle.s$

$$[\![b\langle @y\rangle \triangleright b\langle p\rangle.s]\!]_E$$
$$= \{w_1.b\langle z\rangle.w_2 \in [\![b\langle p\rangle.s]\!]_{E,y\leftarrow z} | b \notin w_1\} \cup \{w | \forall z.w \in [\![b\langle p\rangle.s]\!]_{E,y\leftarrow z} \wedge b \notin w\}$$

The right-hand set is empty, since any word in it must start with a b event.Similarly, w_1 must be ε for any word in the left-hand set.

$$= \{b\langle z\rangle.w_2 \in [\![b\langle p\rangle.s]\!]_{E,y\leftarrow z}\}$$
$$= \bigcup_{z\in\mathcal{Z}} (\{b\langle z\rangle | [\![p(z)]\!]_E\} \circ [\![r]\!]_{E,y\leftarrow z})$$
$$= [\![b\langle @y; p\rangle.s]\!]_E$$

2. $r = B_1 \triangleright (s_1.(b\langle @y\rangle \triangleright s_2)) \xrightarrow{rw} (B_1 \triangleright b\langle @y\rangle \triangleright s_1.s_2)$
In this case, we know that $b\langle @y\rangle \triangleright s_2$ and $B_1 \triangleright s_1$ are implementable, that $b \notin B_1 \triangleright s_1$, and that $y \notin B_1$. We can now show the two are equivalent by induction on the length of B_1. In the base case, we use the fact that y is out of scope in s_1:

$$[\![s_1.(b\langle @y\rangle \triangleright s_2)]\!]_E$$
$$= \{w_1.w_2 | w_1 \in [\![s_1]\!]_E \wedge w_2 \in [\![(b\langle @y\rangle \triangleright s_2)]\!]_E\}$$
$$= \{w_1.w_2 | (\forall z.w_1 \in [\![s_1]\!]_{E,y\leftarrow z}) \wedge w_2 \in [\![(b\langle @y\rangle \triangleright s_2)]\!]_E\}$$
$$= \{w_1.w_2 | (\forall z.w_1 \in [\![s_1]\!]_{E,y\leftarrow z}) \wedge (w_2 \in \{w_1'.b\langle z\rangle.w_2' \in [\![s_2]\!]_{E,y\leftarrow z} | b \notin w_1'\} \cup \{w | \forall z.w \in [\![s_2]\!]_{E,y\leftarrow z} \wedge b \notin w\})\}$$

Because we are quantifying over z for w_1, it is clear that any $w_1.w_2$ in this set must also be in $[\![b\langle @y\rangle \triangleright s_1.s_2]\!]_E$, depending on which set w_2' is in:

$$[\![b\langle @y\rangle \triangleright s_1.s_2]\!]_E$$
$$= \{w_1''.b\langle z\rangle.w_2'' \in [\![s_1.s_2]\!]_{E,y\leftarrow z} | b \notin w_1''\} \cup \{w' | \forall z.w' \in [\![s_1.s_2]\!]_{E,y\leftarrow z} \wedge b \notin w'\}$$

If w_2' is in the right-hand set, then $w_1'' = w_1.w_1'$ and $w_2'' = w_2'$. Otherwise, $w' = w_1.w$. The same expansions show membership the other way; if a word $w_1''.b\langle z\rangle.w_2'' \in [\![s_1.s_2]\!]_{E,y\leftarrow z}$, then a prefix of $w_1'' \in [\![s_1]\!]_{E,y\leftarrow z} = [\![s_1]\!]_E$, and the rest is in the right-hand set above. Otherwise, a prefix of w' is in s_1 and the rest is in the left-hand set above.

The induction case follows directly from expanding the definition:

$$[\![b_1\langle @y_1\rangle \triangleright B_1' \triangleright (s_1.(b\langle @y\rangle \triangleright s_2)]\!]_E$$
$$= \{w_1.b\langle z\rangle.w_2 \in [\![B_1' \triangleright (s_1.(b\langle @y\rangle \triangleright s_2)]\!]_{E,y_1\leftarrow z} | b \notin w_1\} \cup \{w | \forall z.w \in [\![B_1' \triangleright (s_1.(b\langle @y\rangle \triangleright s_2)]\!]_{E,y_1\leftarrow z} \wedge b \notin w\}$$

Using induction:

$$= \{w_1.b\langle z\rangle.w_2 \in [\![B_1' \triangleright b\langle @y\rangle \triangleright (s_1.s_2)]\!]_{E,y_1\leftarrow z} | b \notin w_1\} \cup \{w | \forall z.w \in [\![B_1' \triangleright b\langle @y\rangle \triangleright (s_1.s_2)]\!]_{E,y_1\leftarrow z} \wedge b \notin w\}$$

3.
$$(b\langle @y_1\rangle \triangleright s_1) + (b\langle @y_2\rangle \triangleright s_2) \xrightarrow{rw} b\langle y_1\rangle \triangleright (s_1 + [y_1/y_2]s_2)$$

Here, we assume both $b\langle @y_1\rangle \triangleright s_1$ and $b\langle @y_2\rangle \triangleright s_2$ are implementable. The proof follows directly from expanding the two $\triangleright$ expressions, and substituting the variables in s_2.

$$[\![b\langle @y_1\rangle \triangleright (s_1 + [y_1/y_2]s_2)]\!]_E$$
$$= \{w_1.b\langle z\rangle.w_2 \in [\![s_1 + [y_1/y_2]s_2]\!]_{E,y_1\leftarrow z} | b \notin w_1\} \cup \{w | \forall z.w \in [\![s_1 + [y_1/y_2]s_2]\!]_{E,y_1\leftarrow z} \wedge b \notin w\}$$
$$= \{w_1.b\langle z\rangle.w_2 \in [\![s_1]\!]_{E,y_1\leftarrow z} \cup [\![[y_1/y_2]s_2]\!]_{E,y_1\leftarrow z} | b \notin w_1\} \cup \{w | \forall z.w \in [\![s_1]\!]_{E,y_1\leftarrow z} \cup [\![[y_1/y_2]s_2]\!]_{E,y_1\leftarrow z} \wedge b \notin w\}$$
$$= \{w_1.b\langle z\rangle.w_2 \in [\![s_1]\!]_{E,y_1\leftarrow z} \cup [\![s_2]\!]_{E,y_2\leftarrow z} | b \notin w_1\} \cup \{w | \forall z.w \in [\![s_1]\!]_{E,y_1\leftarrow z} \cup [\![s_2]\!]_{E,y_2\leftarrow z} \wedge b \notin w\}$$
$$= [\![(b\langle @y_1\rangle \triangleright s_1) + (b\langle @y_2\rangle \triangleright s_2)]\!]_E$$

4.
$$(b\langle @y_1\rangle \triangleright s_1) + (c\langle @y_2\rangle \triangleright s_2) \xrightarrow{rw} b\langle @y_1\rangle \triangleright c\langle @y_2\rangle \triangleright (s_1 + s_2)$$
We assume that $b\langle @y_1\rangle \triangleright s_1$ and $c\langle @y_2\rangle \triangleright s_2$ are implementable. The proof follows directly from expanding the definitions of the two $\triangleright$ expressions.

$$\begin{aligned}
&[\![(b\langle @y_1\rangle \triangleright s_1) + (c\langle @y_2\rangle \triangleright s_2)]\!]_E \\
&= [\![b\langle @y_1\rangle \triangleright s_1]\!]_E \cup [\![c\langle @y_2\rangle \triangleright s_2]\!]_E \\
&= S1 : \{w | w = w_1.b\langle z_1\rangle.w_2 \in [\![s_1]\!]_{E,y_1\leftarrow z_1} \wedge b \notin w_1\} \cup S2 : \{w | \forall z.w \in [\![s_1]\!]_{E,y_1\leftarrow z} \wedge b \notin w\} \\
&\cup S3 : \{w | w = w_1.c\langle z_2\rangle.w_2 \in [\![s_2]\!]_{E,y_2\leftarrow z_2} \wedge c \notin w_1\} \cup S4 : \{w | \forall z.w \in [\![s_2]\!]_{E,y_2\leftarrow z} \wedge c \notin w\}
\end{aligned}$$
Expanding the other definition:
$$\begin{aligned}
&[\![b\langle @y_1\rangle \triangleright c\langle @y_2\rangle \triangleright (s_1 + s_2)]\!]_E \\
&= \{w | w = w_1.b\langle z_1\rangle.w_2 \in [\![c\langle @y_2\rangle \triangleright (s_1 + s_2)]\!]_{E,y_1\leftarrow z_1} \wedge b \notin w_1\} \cup \{w | \forall z.w \in [\![c\langle @y_2\rangle \triangleright (s_1 + s_2)]\!]_{E,y_1\leftarrow z_1} \wedge b \notin w\}
\end{aligned}$$
After expanding this to another 4 sets, it is clear that $[\![(b\langle @y_1\rangle \triangleright s_1) + (c\langle @y_2\rangle \triangleright s_2)]\!]_E \subseteq [\![b\langle @y_1\rangle \triangleright c\langle @y_2\rangle \triangleright (s_1 + s_2)]\!]_E$ by cases on which set ($S1, S2, S3, S4$) a word is in, and similarly in the reverse direction.

5.
$$(b\langle @y\rangle \triangleright s_1) + s_2 \xrightarrow{rw} b\langle @y\rangle \triangleright (s_1 + s_2)$$
This proof is very similar to the above with one fewer expansion, because the binding-free s_2 is always implementable.

The rules (equivalent to 3,4, and 5) for & have similar proof structure, but are simplified by using intersection rather than union. □

Lemma 7. *If $r \xrightarrow{rw} r'$ and r is implementable, then so is r'.*

Proof. By cases on which rewrite rule is used.

1. $r = b\langle @y; p\rangle.s \xrightarrow{rw} b\langle @y\rangle \triangleright b\langle p\rangle.s$
We assume that s is binding free, so that $b\langle @y\rangle \triangleright b\langle p\rangle.s$ is in prefix form. Now, we show that it is implementable, for any $a \neq b$, the derivative of s is the same no matter the environment, since the following does not depend on z_1 at all:
$$\begin{aligned}
&D_{re}(a\langle z\rangle; b\langle p\rangle.s; E, y \leftarrow z_1) \\
&= D_{re}(a\langle z\rangle; b\langle p\rangle; E, y \leftarrow z_1).s + \nu(b\langle p\rangle).D_{re}(a\langle z\rangle; s; E, y \leftarrow z_1) \\
&= \emptyset.s + \emptyset.D_{re}(a\langle z\rangle; s; E, y \leftarrow z_1) \\
&= \emptyset
\end{aligned}$$

2. $r = (B_1 \triangleright s_1).(b\langle @y\rangle \triangleright s_2) \xrightarrow{rw} (B_1 \triangleright b\langle @y\rangle \triangleright s_1.s)$
In this case, we know that $b\langle @y\rangle \triangleright s_2$ and $B_1 \triangleright s_1$ are implementable, that $b \notin B_1 \triangleright s_1$, and that $y \notin B_1$. Now, we can expand the expression, for some $a\langle z\rangle \notin B_1 \triangleright b$ and compatible environments G_1, G_2 to $B_1 \triangleright b$. We will write these as $G_1', y \leftarrow z_1$ and $G_2', y \leftarrow z_2$, separating out the compatible environments to B_1 and B_2, and expand definitions:
$$\begin{aligned}
&[\![D_{re}(a\langle z\rangle; s_1.s_2; (E, G_1))]\!]_{E,G_2} \\
&= [\![D_{re}(a\langle z\rangle; s_1; (E, G_1)).s_2 + \nu(s_1).D_{re}(a\langle z\rangle; s_2; (E, G_1))]\!]_{E,G_2} \\
&= [\![D_{re}(a\langle z\rangle; s_1; (E, G_1)).s_2]\!]_{E,G_2} \cup [\![\nu(s_1).D_{re}(a\langle z\rangle; s_2; (E, G_1))]\!]_{E,G_2} \\
&= [\![D_{re}(a\langle z\rangle; s_1; ((E, y \leftarrow z_1), G_1')).s_2]\!]_{(E,y\leftarrow z_2),G_2'} \cup [\![\nu(s_1).D_{re}(a\langle z\rangle; s_2; ((E, G_1'), y \leftarrow z_1))]\!]_{(E,G_2'),y\leftarrow z_2}
\end{aligned}$$
On the left, because y is not in scope in s_1, its value does not change the semantics of its derivative, so we can replace z_1 with z_2. Similarly, on the right we can replace G_1' with G_2' because those variables are out of scope in s_2:
$$= [\![D_{re}(a\langle z\rangle; s_1; ((E, y \leftarrow z_2), G_1')).s_2]\!]_{(E,y\leftarrow z_2),G_2'} \cup [\![\nu(s_1).D_{re}(a\langle z\rangle; s_2; ((E, G_2'), y \leftarrow z_1))]\!]_{(E,G_2'),y\leftarrow z_2}$$

$$= [\![D_{re}(a\langle z\rangle; s_1; ((E, y \leftarrow z_2), G_1'))]\!]_{(E,y\leftarrow z_2),G_2'} \circ [\![s_2]\!]_{(E,y\leftarrow z_2),G_2'} \cup [\![v(s_1]\!]_{(E,G_2'),y\leftarrow z_2} \circ [\![D_{re}(a\langle z\rangle; s_2; ((E, G_2'), y \leftarrow z_1))]\!]_{(E,G_2'),y\leftarrow z_2}$$

Finally, we can apply the inductive hypothesis for s_1 and s_2 and roll back up the definitions to get the desired result:

$$= [\![D_{re}(a\langle z\rangle; s_1; ((E, y \leftarrow z_2), G_2'))]\!]_{(E,y\leftarrow z_2),G_2'} \circ [\![s_2]\!]_{(E,y\leftarrow z_2),G_2'} \cup [\![v(s_1]\!]_{(E,G_2'),y\leftarrow z_2} \circ [\![D_{re}(a\langle z\rangle; s_2; ((E, G_2'), y \leftarrow z_2))]\!]_{(E,G_2'),y\leftarrow z_2}$$

$$= [\![D_{re}(a\langle z\rangle; s_1; ((E, y \leftarrow z_2), G_1')).s_2]\!]_{(E,y\leftarrow z_2),G_2'} \cup [\![v(s_2).D_{re}(a\langle z\rangle; s_2; ((E, G_2'), y \leftarrow z_2))]\!]_{(E,G_2'),y\leftarrow z_2}$$

$$= [\![D_{re}(a\langle z\rangle; s_1; ((E, y \leftarrow z_2), G_1')).s_2 + v(s_2).D_{re}(a\langle z\rangle; s_2; ((E, G_2'), y \leftarrow z_2))]\!]_{(E,G_2'),y\leftarrow z_2}$$

$$= [\![D_{re}(a\langle z\rangle; s_1.s_2; E, G_2)]\!]_{E,G_2}$$

3.

$$(b\langle @y_1\rangle \triangleright s_1) + (b\langle @y_2\rangle \triangleright s_2) \xrightarrow{rw} b\langle y_1\rangle \triangleright (s_1 + [y_1/y_2]s_2)$$

Here, we assume both $b\langle @y_1\rangle \triangleright s_1$ and $b\langle @y_2\rangle \triangleright s_2$ are implementable. Then, we can expand definitions, writing $G_1 as (y_1 \leftarrow z_1)$ and G_2 as $(y_1 \leftarrow z_2)$:

$$[\![D_{re}(a\langle z\rangle; s_1 + [y_1/y_2]s_2; E, y_1 \leftarrow z_1)]\!]_{E,y_1\leftarrow z_2}$$

$$= [\![D_{re}(a\langle z\rangle; s_1; E, y_1 \leftarrow z_1) + D_{re}(a\langle z\rangle; [y_1/y_2]s_2; E, y_1 \leftarrow z_1)]\!]_{E,y_1\leftarrow z_2}$$

$$= [\![D_{re}(a\langle z\rangle; s_1; E, y_1 \leftarrow z_1)]\!]_{E,y_1\leftarrow z_2} \cup [\![D_{re}(a\langle z\rangle; [y_1/y_2]s_2; E, y_1 \leftarrow z_1)]\!]_{E,y_1\leftarrow z_2}$$

On the left, we can apply the induction hypothesis directly. On the right, we undo the substitution and use it, then roll back up the definition.

$$= [\![D_{re}(a\langle z\rangle; s_1; E, y_1 \leftarrow z_1)]\!]_{E,y_1\leftarrow z_2} \cup [\![D_{re}(a\langle z\rangle; s_2; E, y_2 \leftarrow z_1)]\!]_{E,y_2\leftarrow z_2}$$

$$= [\![D_{re}(a\langle z\rangle; s_1; E, y_1 \leftarrow z_2)]\!]_{E,y_1\leftarrow z_2} \cup [\![D_{re}(a\langle z\rangle; s_2; E, y_2 \leftarrow z_2)]\!]_{E,y_2\leftarrow z_2}$$

$$= [\![D_{re}(a\langle z\rangle; s_1; E, y_1 \leftarrow z_2)]\!]_{E,y_1\leftarrow z_2} \cup [\![D_{re}(a\langle z\rangle; [y_1/y_2]s_2; E, y_1 \leftarrow z_2)]\!]_{E,y_1\leftarrow z_2}$$

$$= [\![D_{re}(a\langle z\rangle; s_1 + s_2; E, G_2)]\!]_{G_2}$$

4.

$$(b\langle @y_1\rangle \triangleright s_1) + (c\langle @y_2\rangle \triangleright s_2) \xrightarrow{rw} b\langle @y_1\rangle \triangleright c\langle @y_2\rangle \triangleright (s_1 + s_2)$$

We assume that $b\langle @y_1\rangle \triangleright s_1$ and $c\langle @y_2\rangle \triangleright s_2$ are implementable, and expand definitions, writing G_1 as $y_1 \leftarrow z_1, y_2 \leftarrow z_2$ and G_2 as $y_1 \leftarrow z_1', y_2 \leftarrow z_2'$:

$$[\![D_{re}(a\langle z\rangle; s_1 + s_2; E, y_1 \leftarrow z_1, y_2 \leftarrow z_2)]\!]_{E,y_1\leftarrow z_1',y_2\leftarrow z_2'}$$

$$= [\![D_{re}(a\langle z\rangle; s_1; E, y_1 \leftarrow z_1, y_2 \leftarrow z_2) + D_{re}(a\langle z\rangle; s_2; E, y_1 \leftarrow z_1, y_2 \leftarrow z_2)]\!]_{E,y_1\leftarrow z_1',y_2\leftarrow z_2'}$$

$$= [\![D_{re}(a\langle z\rangle; s_1; E, y_1 \leftarrow z_1, y_2 \leftarrow z_2)]\!]_{E,y_1\leftarrow z_1',y_2\leftarrow z_2'} \cup [\![D_{re}(a\langle z\rangle; s_2; E, y_1 \leftarrow z_1, y_2 \leftarrow z_2)]\!]_{E,y_1\leftarrow z_1',y_2\leftarrow z_2'}$$

y_1 is out of scope in s_2, and y_2 is out of scope in s_1. So, we can assign any value to them without changing the derivative:

$$= [\![D_{re}(a\langle z\rangle; s_1; E, y_1 \leftarrow z_1, y_2 \leftarrow z_2')]\!]_{E,y_1\leftarrow z_1',y_2\leftarrow z_2'} \cup [\![D_{re}(a\langle z\rangle; s_2; E, y_1 \leftarrow z_1', y_2 \leftarrow z_2)]\!]_{E,y_1\leftarrow z_1',y_2\leftarrow z_2'}$$

Finally, we can apply the induction hypothesis for the other variables, using the implementability property. Then, we roll back up the definitions.

$$= [\![D_{re}(a\langle z\rangle; s_1; E, y_1 \leftarrow z_1', y_2 \leftarrow z_2')]\!]_{E,y_1\leftarrow z_1',y_2\leftarrow z_2'} \cup [\![D_{re}(a\langle z\rangle; s_2; E, y_1 \leftarrow z_1', y_2 \leftarrow z_2')]\!]_{E,y_1\leftarrow z_1',y_2\leftarrow z_2'}$$

$$= [\![D_{re}(a\langle z\rangle; s_1 + s_2; E, y_1 \leftarrow z_1', y_2 \leftarrow z_2')]\!]_{E,y_1\leftarrow z_1',y_2\leftarrow z_2'}$$

5.

$$(b\langle @y\rangle \triangleright s_1) + s_2 \xrightarrow{rw} b\langle @y\rangle \triangleright (s_1 + s_2)$$

This proof is very similar to the above, because the binding-free s_2 is always implementable.

$$[\![D_{re}(a\langle z\rangle; s_1 + s_2; E, y \leftarrow z_1)]\!]_{E,y\leftarrow z_2}$$

$$= [\![D_{re}(a\langle z\rangle; s_1; E, y \leftarrow z_1)]\!]_{E,y\leftarrow z_2} \cup [\![D_{re}(a\langle z\rangle; s_2; E, y \leftarrow z_1)]\!]_{E,y\leftarrow z_2}$$

y is out of scope in s_2, so we can replace its value with anything without changing the derivative.

$$= [\![D_{re}(a\langle z\rangle; s_1; E, y \leftarrow z_1)]\!]_{E,y\leftarrow z_2} \cup [\![D_{re}(a\langle z\rangle; s_2; E, y \leftarrow z_2)]\!]_{E,y\leftarrow z_2}$$

Now, we apply the induction hypothesis for s_1, and get the require result:

$$= [\![D_{re}(a\langle z\rangle; s_1; E, y \leftarrow z_2)]\!]_{E,y\leftarrow z_2} \cup [\![D_{re}(a\langle z\rangle; s_2; E, y \leftarrow z_2)]\!]_{E,y\leftarrow z_2}$$

$$= [\![D_{re}(a\langle z\rangle; s_1 + s_2; E, y \leftarrow z_2)]\!]_{E,y\leftarrow z_2}$$

The & forms of rules 3, 4, and 5 above have very similar proofs, replacing $+$ and $\cup$ with & and $\cap$. □

OctoSketch: Enabling Real-Time, Continuous Network Monitoring over Multiple Cores

Yinda Zhang[†], Peiqing Chen[⋆], Zaoxing Liu[⋆]
[⋆]*University of Maryland,* [†]*University of Pennsylvania*

Abstract

Sketching algorithms (sketches) have emerged as a resource-efficient and accurate solution for software-based network monitoring. However, existing sketch-based monitoring makes sacrifices in online accuracy (query time accuracy) and performance (handling line rate traffic with low latency) when dealing with distributed traffic across multiple cores. In this work, we present OctoSketch, a software monitoring framework that can scale a wide spectrum of sketches to many cores with high online accuracy and performance. In contrast to previous systems that adopt straightforward sketch merges from individual cores to obtain the aggregated result, we devise a continuous, change-based mechanism that can generally be applied to sketches to perform the aggregation. This design ensures high online accuracy of the aggregated result at any query time and reduces computation costs to achieve high throughput. We apply OctoSketch to nine representative sketches on three software platforms (CPU, DPDK, and eBPF XDP). Our results demonstrate that OctoSketch achieves about 15.6× lower errors and up to 4.5× higher throughput than the state-of-the-art.

1 Introduction

While telemetry on dedicated switching hardware continues to be important, the deployment of monitoring capabilities in software platforms is increasing with the transition towards virtualized deployments and "white-box" capabilities [1–3]. The ability to monitor network traffic within software platforms has become a key enabler for many network management applications, including load balancing, anomaly detection, and performance diagnosis [4, 5]. Over the years, the volume of traffic processed by each server has seen a substantial increase (*e.g.*, 10G to 100G). As a result, efficient monitoring of traffic across multiple cores emerges as a pressing demand.

When monitoring traffic distributed across cores, downstream applications often entail high-fidelity *aggregated* measurement results. For instance, in-network caching systems [6, 7] require real-time measurement of the *hot* objects to determine what to cache. Similarly, a load balancer needs aggregated flow statistics (*e.g.*, heavy hitters, top-k flows) to decide where to assign flows (to cores or nodes) [7, 8]. In this scenario, sketch algorithms naturally emerge as a promising solution. This is because recent theoretical advances [9] have shown that many sketches, such as Count-Min [10], Count Sketch [11], and UnivMon [12], have intrinsic *mergeability*: Independent sketches monitoring different traffic partitions can be merged in a way such as *sum* and *max* to obtain aggregated results without losing accuracy guarantee. With this property, a natural design for sketch-based multicore monitoring is to run an independent sketch per core and periodically merge the entire sketches from all cores for aggregation.

While we have seen significant recent progress in sketch algorithms [12–17] and their implementations [18,19], we argue that sketch-based multicore monitoring systems remain impractical. Many existing systems, such as Elastic Sketch [13], NitroSketch [14], and HeteroSketch [20], adopt the above "sketch-merge" design to scale their systems to distributed settings. However, these solutions raise two significant issues in practice. First, this approach does not retain accuracy at any given time – if the *aggregator* responsible for answering the query has not yet obtained the most recent sketches from cores, the aggregated result is stale, missing the measurement over the current traffic. A recent study [21] shows that an inaccurate and stale result can lead to a significant load imbalance in a load balancer. Second, if we consider an undesirable extreme to frequently merge sketches for fresh results, the packet performance will be significantly reduced, and it is difficult for each core to achieve high line rates (§2.2).

Ideally, we want a multicore monitoring system to meet three requirements. First, we need high accuracy whenever applications query a measurement result. This is essentially *online accuracy*, which enables real-time telemetry support for applications. Second, we expect *resource efficiency* to achieve line rates with low resource overhead since CPU and memory resources are shared among other services [22]. Finally, we need *generality*, covering a broad range of traffic metrics for various application requirements [12, 13, 23].

In this paper, we present **OctoSketch**, a software framework to perform high-performance and real-time monitoring on multiple CPU cores that meets the above three requirements. In OctoSketch, we argue for a continuous, change-based design to aggregate the sketch results among multiple cores. In contrast to periodically merging entire sketches,

OctoSketch keeps track of individual counter changes in the sketch for every packet and sends the counter difference-Δ (from the last update) when the change is sufficiently large (*e.g.*, over a threshold to meet the accuracy requirement). By this simple-yet-effective design, each worker still maintains a sketch but intends to send the Δ that could affect the overall accuracy of the aggregated result. In this way, communication between the worker and the aggregator becomes a "continuous" stream of worker-aggregator messages, which are tiny in space but carry the most critical and timely information. Compared to previous efforts [9, 12, 13, 20], OctoSketch eliminates potentially wasted counter updates when merging the sketches (*e.g.*, many counters do not change significantly since the last update) to improve online accuracy and optimize resource footprints.

With the continuous, change-based message passing scheme, OctoSketch is able to realize dynamic resource allocation policies toward different systems and user objectives. For example, within a CPU resource budget, OctoSketch can be configured to achieve the best (possible) online accuracy; or if the accuracy requirement is loose (*e.g.*, $\ll$ than best possible accuracy), OctoSketch can be configured to save CPU cycles instead of using all the aggregator CPU trying to reach an unnecessary accuracy. OctoSketch realizes these policies by (1) maintaining a *shared buffer* between workers and the aggregator, (2) adjusting worker message frequency via dynamic counter update thresholds, and (3) letting the aggregator learn the sending rate of the workers via the queue occupancy in the shared buffer, as detailed in §4.3.

To the best of our knowledge, OctoSketch is the first work to: (a) propose the integration of a continuous, change-based update mechanism with various sketches to enable the practical adoption of sketches in the multi-core scenario. Our goal is to improve the accuracy and performance of distributed sketches across cores to meet various measurement objectives; (b) analytically prove that OctoSketch can retain the same asymptotic error bounds as in the ideal case in which traffic is not distributed. That is, when querying OctoSketch, the aggregator can provide a result that represents the aggregation of traffic from all cores (by a bounded error) at any given time; and (c) provide practical end-to-end design and implementation of this idea on three popular software platforms. We further show that, compared to existing sketch-merge approaches, OctoSketch reduces message passing overhead by up to four orders of magnitude for the same accuracy.

We apply OctoSketch to a wide range of state-of-the-art sketches (*e.g.*, Count-Min Sketch [10], UnivMon [12], Elastic Sketch [13], and CocoSketch [16]) and demonstrate its performance on the CPU, Intel DPDK library [24], and eBPF XDP [25]. Our experiments show that OctoSketch achieves around 15.6× smaller errors than previous sketch-merge techniques at query time. Moreover, OctoSketch also reaches up to 4.5× higher throughput and up to 1.9× reduction in CPU utilization in DPDK. We apply OctoSketch to two common use cases: load balancer and key-value cache. Our results show that OctoSketch achieves a 3.15× lower imbalance rate and a 13% higher cache hit rate than the sketch-merge baseline. We have open-sourced OctoSketch and other baseline algorithms on https://github.com/Froot-NetSys/OctoSketch.

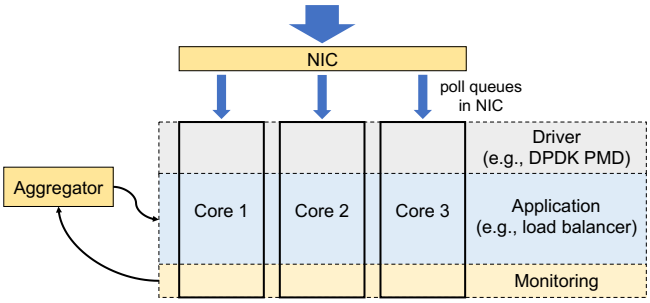


Figure 1: Multicore monitoring problem in a server

Ethics: This work does not raise ethical issues.

2 Background and Motivation

In this section, we first describe how distributed traffic over multiple cores brings new challenges to online monitoring. We then discuss existing efforts and their limitations in scaling monitoring capabilities to distributed settings.

2.1 Multicore monitoring problem

Since the latest NIC designs [26] for data centers have already reached 100Gbps/port, network applications supporting such a high speed require multiple CPU cores. For instance, a DPDK-enabled Open vSwitch [27] requires 6 to 10 CPU cores to reach 100Gbps. Hence, we call such a network application using multiple cores a *multicore application*.

Multicore applications often require timely and accurate traffic measurement results, such as heavy hitters, distinct flows, and entropy [4, 5]. Figure 1 shows a typical workflow for monitoring traffic distributed over multiple cores. The NIC will distribute the incoming packets into multiple Rx queues, and each CPU core will poll packets from one or more queues via driver (*e.g.*, DPDK polling mode driver [24]). The resource in each core is shared between drivers, applications, and monitoring programs.

In contrast to a non-distributed monitoring setting, this multicore workflow for a single server poses three significant challenges:

- ***Online accuracy:*** Many applications need to obtain accurate aggregated statistics in an online fashion. For example, cache-based load balancers rely on real-time large flow measurements to determine hot items to cache [6, 7]. We define the *online accuracy* according to the statistics computed over the traffic from the end of the last measurement to the current query time. Due to the distributed nature of the traffic, there can be a significant error or delay to query aggregated statistics at any given time for two reasons: (1) the current aggregated result is stale, and (2) frequent ag-

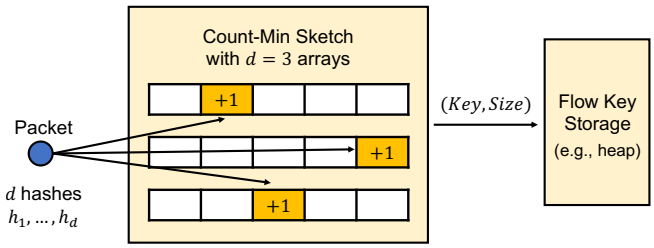


Figure 2: Example of the Count-Min sketch

gregation from all the cores is expensive to compute.

- ***Resource efficiency:*** The resources (*e.g.*, CPU and memory space) used by monitoring should keep up with line-rate requirements and leave ample room for applications.
- ***Generality in metrics of interest:*** Similar to other network monitoring systems [12,13], a multicore monitoring system shall measure multiple traffic metrics to support a range of applications (*e.g.*, DDoS detection [5], load balancing [4], and traffic engineering [28]).

2.2 Prior Solutions and Limitations

Background of Sketches. Sketches have emerged as a promising network monitoring solution due to their high resource efficiency and accuracy guarantees. At a high level, sketches are a kind of approximate data structure that can estimate various statistics online. For example, there are (1) Count-Min sketch [10] and Count sketch [11] for detecting heavy hitters (flows with large sizes), (2) Locher sketch [29] for detecting superspreaders (Source IPs that connect to many Destination IPs), (3) CocoSketch [16] for querying flow size on multiple keys, and (4) UnivMon [12] and Elastic sketch [13] for supporting a range of these tasks instead of a specialized sketch per task. As shown in prior work [12–14, 30–32], these sketches can often provide better accuracy-memory tradeoffs than traditional sampling-based techniques [33, 34].

To illustrate the insertion process of sketches, we use the Count-Min sketch for heavy-hitter detection as an example. As shown in Figure 2, the Count-Min sketch typically consists of multiple arrays of counters. For the insertion of each packet, it computes a set of independent hash values based on the flow key (*e.g.*, Source IP) of the packet and updates the corresponding counter in each array. Moreover, Count-Min Sketch needs additional data structures (*e.g.*, heap) to store flow keys whose estimated sizes are large. We further discuss the background of sketches and their applications in §A.

Key-based partition. One way to apply sketches in the multicore scenario is to divide different keys to different cores based on hashing. For each core, we can maintain a sketch for tracking the keys hashed to it and the keys in different sketches never overlap. However, strictly pinning a key to a core can lead to degraded packet performance. For example, when the skewness of the network traffic varies, key-based partition often leads to high load imbalance among cores.

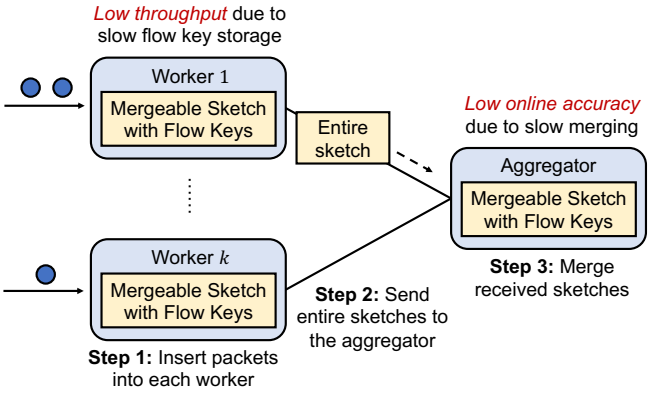


Figure 3: Merging sketches from multiple cores to an aggregator (sketch-merge)

Splitting large flows into multiple cores will help balance the load, as shown in existing load-balancing algorithms [35]. Moreover, depending on the definition of the key, it is sometimes infeasible to pin a key to a core. For instance, a NIC may distribute packets based on the 5-tuple while the application may want to measure the keys based on user-level information (*e.g.*, the key-value pair). In this case, it is hard to guarantee all packets associated with a key will reach the same core. Moreover, some multiple applications require measuring results based on multiple key definitions, and we cannot guarantee key distribution across different keys, as shown in [16]. As a result, OctoSketch does not make any assumptions about the way of distributing keys to accommodate various network applications.

Merging sketches from multiple cores (sketch-merge). Another natural solution is to merge multiple sketches from different cores into one for query periodically, where the merged sketch preserves the same accuracy guarantee [9]. For instance, Sketch 1 measuring flow set A can be merged with Sketch 2 measuring flow set B (*e.g.*, *sum* or *max* of the two counter arrays) to obtain statistics about a combined flow set $A \cup B$, as long as Sketches 1 and 2 share the same configuration. Most existing solutions leverage this mergeability to scale to multi-site (*e.g.*, multiple cores or servers) in diverse domains, including UnivMon [12], HeteroSketch [20], and FetchSGD [36].

However, periodically merging entire sketches from multiple cores brings large penalties on online accuracy and overall throughput. We use Figure 3 as an example to illustrate the workflow and its issues: Given k *workers*, each worker maintains one sketch to handle its own traffic. If one worker has received enough packets or a certain time interval is reached, it will send its sketch with heavy flow keys to the *aggregator* and recreate a new one. Once the aggregator receives a new sketch from a worker, the aggregator merges the sketch to its own (aggregated) sketch, and reports aggregated results such as heavy hitters and distinct counts. Unfortunately, the merging process can bring significant overheads to both the

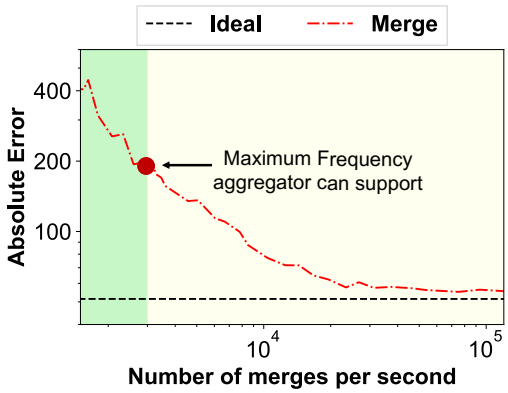


Figure 4: Frequency of sketch-merge vs. Error

aggregator and the workers as follows:

- **Bottleneck in the aggregator:** In practice, the aggregator is stuck in a fundamental dilemma: To achieve high online accuracy, the aggregator has to frequently merge multiple sketches (*e.g.*, merging at every millisecond), which can be prohibitively expensive to compute; otherwise, while infrequent and fixed-time merges save resources, the queried result at the aggregator may be stale by up to a fixed time window. To understand this problem, we consider a hypothetical non-merging design where all packets are processed by a single core with unlimited CPU cycles and cache, and use its accuracy as a reference of the ideal accuracy. As depicted in Figure 4, the aggregator needs to merge around 3×10^4 sketches per second to achieve comparable accuracy to the ideal case under 150 million packets per second throughput. However, a 2.35GHz CPU core can only support up to 3×10^3 merges per second. [1]
- **Bottleneck in the worker:** Each worker core keeps an entire sketch instance, including its counter structure and flow key storage. Prior efforts (*e.g.*, [13, 14]) have demonstrated that flow key storage is one of the performance bottlenecks in sketches. Maintaining such flow key storage per core is computation-heavy considering the high-volume traffic.

3 System Overview

We now describe the high-level design of OctoSketch and highlight the key ideas to achieve the multicore requirements.

3.1 OctoSketch Workflow

As illustrated in Figure 5, OctoSketch has two main components: multiple monitoring *workers* for ingesting the traffic

[1] The setting is the same as in §6 and §7. We use Count-Min sketch and CAIDA dataset [37], and show the absolute errors of heavy hitter detection. In this experiment, we have 16 workers and 1 aggregator. The frequency of merging is set at each worker. Since the aggregator is receiving and aggregating sketches from all workers, the aggregator is the bottleneck and can only accommodate 3×10^3 merges per second, while each worker can still send more merges. When increasing the merge frequency beyond the maximum rate the aggregator can support, we use a buffer to store sketches and aggregate the results later for evaluating the errors.

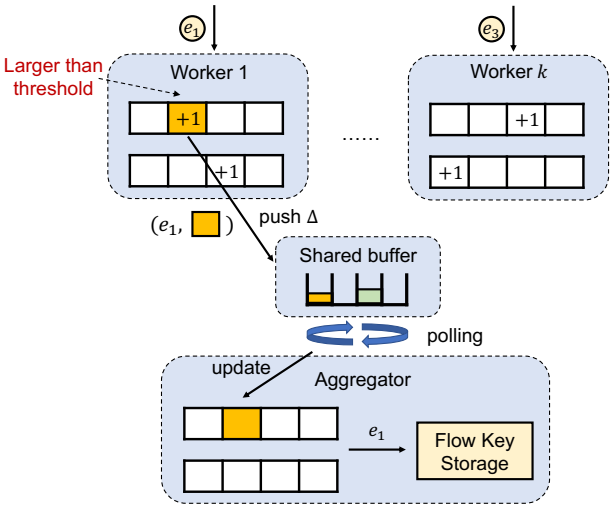


Figure 5: Workflow of OctoSketch

Query Type	Sketch	Acc.	Thp.
Flow Size	Count-Min [10]	9.32×	3.85×
	Count Sketch [11]	9.04×	3.22×
Cardinality	LogLog [38]	54.93×	1.29×
	HyperLogLog [39]	38.97×	1.29×
Super-Spreader	Locher Sketch [29]	4.06×	4.51×
Quantile	DDSketch [40]	4.29×	0.92×
Multi-Key	CocoSketch [16]	37.25×	1.01×
General	UnivMon [12]	13.55×	2.63×
	ElasticSketch [13]	14.03×	0.93×

Table 1: Applicability of the OctoSketch on latest sketches

and one *aggregator* for answering queries and overall control. By default, we allocate each worker and aggregator a separate core/thread. Thanks to the lightweight design of OctoSketch, one aggregator is sufficient to achieve online accuracy (§7). There is also a *shared buffer* for workers and aggregators to prevent inter-thread message losses from bursts.

Data ingestion in the workers: Each worker is a monitoring program running in a CPU core that is responsible for processing its own portion of the traffic from a NIC Rx queue and maintaining relevant sketch data structures (*e.g.*, hash-indexed counter arrays) for later aggregation. In OctoSketch, the goal of the workers is to provide timely and accurate counter updates to the aggregator to answer queries, while minimizing the CPU and memory footprints. When the worker sends an update to the aggregator, it will insert the update into a shared buffer. Unlike prior work that sends the entire sketch as an update, OctoSketch adapts to actual workloads and sends only lightweight, change-based counter updates (Δs). We describe this distributed update mechanism in detail in §4.1.

Query estimation in the aggregator: The aggregator fetches updates from all workers via the shared buffer and updates its own aggregated data structures to compute the statistics of interest. We describe the aggregation procedure in detail

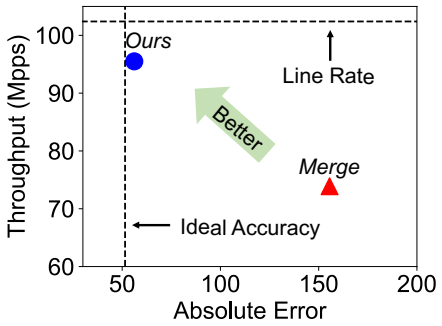


Figure 6: Online accuracy vs. throughput

in §4.2. The query results will then be shared with other software processing libraries or online applications for their management purposes.

Supported sketches (Table 1): OctoSketch is a general framework that support all *linearly mergeable* sketches and thus can answer a broad spectrum of sketch-based queries. To demonstrate the performance of OctoSketch, we apply OctoSketch to 9 popular sketches as shown in Table 1. We note that OctoSketch cannot make sketches that are fundamentally not mergeable (*e.g.*, RAP [41] for detecting heavy hitters) work in the multicore scenario.

3.2 Key Ideas

To realize the OctoSketch workflow, we have three key ideas. By leveraging these ideas, OctoSketch can achieve near-ideal online accuracy and line-rate throughput, outperforming the baseline solution as summarized in Figure 6.

Idea 1: ***Adopting a continuous, change-based mechanism where each worker only sends "sufficiently changed" counters to the aggregator.***

This idea achieves significantly better online accuracy compared to merging sketches (*e.g.*, all the sketch counters and the corresponding flow key storage). The fundamental reason behind the inefficiency of sketch-merge is that if we look at the value of each counter, not all counters are equally important to query accuracy. Due to the skewness of the network workload, the counter values in the sketch are also heavy-tailed. As shown in Figure 7, if we use the Count-Min sketch to find heavy hitters, more than 80% of the counter values do not change in the aggregator, while there are also counters whose changes are larger than 10^3. In other words, the aggregator and the workers waste most computations to merge counters which are not important and delay the updates from important counters.

Inspired by prior change-based solutions [42–45], OctoSketch essentially "amortizes" a large sketch merging operation into a series of individual counter change notifications to achieve online accuracy. Each notification only contains the counters that changed above a threshold. In this way, we can save the computation of merging small counters and use the resources for merging large counters more frequently. In other words, we only send counters with the most critical *informational value* for online accuracy. Our analysis in §5 shows that estimation accuracy is bounded by a certain range from ideal accuracy.

Compared to prior change-based designs with hash tables or other deterministic structures, OctoSketch aims to enable the use of diverse sketches for efficient multicore monitoring by addressing several additional challenges. First, sketches introduce extra errors on top of the errors caused by the change-based updates. The change thresholds must be determined based on the unified error bounds. Second, there are several key parameters regarding the sketch data structures and communication between different parties. We need to dynamically fine-tune the parameters to allocate the resources among workers and aggregators for various objectives. Third, for different sketches, we need to analyze and reconstruct the data structures for better performance.

Idea 2: ***Adaptive resource allocation between workers and aggregator under various objectives.***

With Idea 1, OctoSketch provides an improved tradeoff between communication and online accuracy. As a multicore system, a key question is how OctoSketch can allocate resources among workers and aggregator to meet various system objectives. For instance, an example objective is to achieve the best possible online accuracy with a fixed CPU budget (e.g., 50%). Users can also specify an accuracy requirement (e.g., 95%) while the objective is to minimize total CPU utilization. To meet these objectives, we adaptively allocate resources using dynamic counter change thresholds based on traffic rates. In particular, we set a universal threshold for all workers to adjust the resources: (1) When the packet arrival rate on a worker is high, the counter changes are significant, and the worker needs to send updates via the work-aggregator channel. (2) When the packet arrival rate is low, the worker does not send updates until sufficient counter changes are made, saving worker/aggregator resources for other worker-aggregator channels.

Therefore, to achieve the best possible online accuracy, the aggregator just needs to poll counter updates from workers (via concurrent queues) as much as possible within a CPU limit. When the queue length is small, the aggregator will decrease the threshold to receive more frequent counter updates. If we only want to fulfill an accuracy requirement that is lower than the best possible accuracy, we can derive a fixed threshold to meet the error bound based on the analysis in §5.

Idea 3: ***Reconstructing workers and aggregator to remove redundant data structures.***

To scale to multiple cores, prior solutions need to maintain flow key storage in each worker and aggregator, which is compute-heavy (as shown in §2.2). We observe that Idea 1 and 2 enable the opportunity to reconstruct sketches and remove flow key storage in the workers. Specifically, since

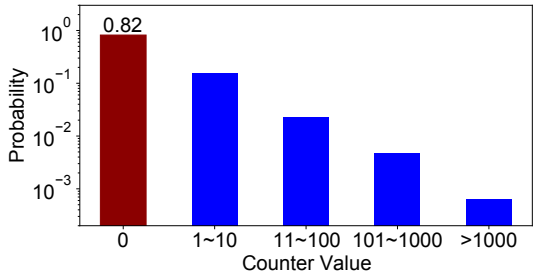


Figure 7: Counter value distribution when sketch-merge

Symbol	Description
k	the number of workers
τ	the threshold shared by all workers
e	the flow key of a packet
$f(e)$	the real size of flow e
$\widehat{f}(e)$	the estimated size of flow e
d	the number of arrays in the sketch
l	the number of counters in each array
$h_i(.)$	the hash function for the ith array
$C_i[j]$	the jth counter in the ith array
q_j	the shared queue between worker j and aggregator

Table 2: Symbols and notations

the traffic statistics are calculated based on aggregated flows, only the aggregator needs to keep a flow key storage and the workers can remove their flow key storage. Therefore, each worker eliminates the performance bottleneck of maintaining complex data structures (*e.g.*, heap) and only sends potential heavy flow keys to the aggregator. Due to the efficient continuous, change-based mechanism, the counter and key updates will not create additional burdens on the aggregator.

4 Detailed Design

In this section, we first use the Count-Min sketch [10] as an example to describe how OctoSketch works in both workers (§4.1) and the aggregator (§4.2). Then, we extend the design of OctoSketch to other sketches. We summarize the frequently used symbols and notations in Table 2.

4.1 Worker

Data structure: In each worker, OctoSketch maintains a Count-Min sketch with d arrays of l counters without flow key storage. Since OctoSketch guarantees that all counters in every worker are always smaller than a given threshold τ (*e.g.*, 128), we only need $\lceil \log \tau \rceil$ bits (*e.g.*, 8) for each counter instead of using 32-bit counters. Thus, we can effectively save 4× memory without losing accuracy. In our experiments, we find that setting τ to 128 is often sufficient to handle the highest achievable throughput ($\approx$ 1400Mpps) for 16 workers.

Sketch insertion: Algorithm 1 describes how to insert a packet into each worker. The original operations of the Count-Min sketch are shown in black color, while the newly added operations by OctoSketch are shown in green color. In Line 3-5, when the Count-Min sketch updates a counter in each of its arrays, the worker will check whether the counter is larger than the threshold τ. If the counter value is above τ in a packet insertion, the worker generates a message containing a vector of <the flow key of the packet, the row and column indices of the counter, and the counter value> and pushes it into a concurrent queue between worker and aggregator. Each worker has a corresponding queue to avoid competition among workers, and the counter will be cleared out.

Algorithm 1: Sketch insertions on worker j.

```
Input: A packet with flow key e
1 foreach i (1 ⩽ i ⩽ d) do
2     C_i[h_i(e)] ← C_i[h_i(e)] + 1;
3     if C_i[h_i(e)] ⩾ τ then
4         q_j.enqueue(e, i, h_i(e), C_i[h_i(e)]);
5         C_i[h_i(e)] ← 0;
6 end
```

Figure 8 illustrates an example of the insertion step. To insert packet e_1, the worker first updates a counter in each array. Because the accessed counter in the second array is larger than the threshold 10, OctoSketch pushes the tuple <the flow key (e_1), the 2D counter index (the 2nd array, the 4th counter), counter value (10)> into the shared buffer and sets the counter to 0. While the aggregator can get the counter index via hashing the flow key, we choose to send the index directly to reduce the hash computation cost for the aggregator.

4.2 Aggregator

Data structure: The aggregator needs to maintain a $d \times l$ Count-Min sketch that is same-sized as other sketches in the workers, and a min-heap to record heavy flow keys.

Algorithm 2: Sketch insertions on the aggregator.

```
1 foreach i (1 ⩽ i ⩽ k) do
2     (e, j, p, V) ← q_i.dequeue();
3     C_j[p] ← C_j[p] + V;
4     if C_j[p] > heap.min then
5         f̂(e) ← min_{k∈[d]} C_k[h_k(e)];
6         heap.insert(e, f̂(e));
7 end
```

Insertion to the aggregated sketch: As shown in Algorithm 2, the aggregator keeps polling the items (the 4-tuple) from the k shared (concurrent) queues. For each item from the queue, the aggregator first updates the corresponding counter based on the index and value, and then updates the flow key heap. If the new counter is larger than the minimum value recorded in the heap, the aggregator will obtain the estimated

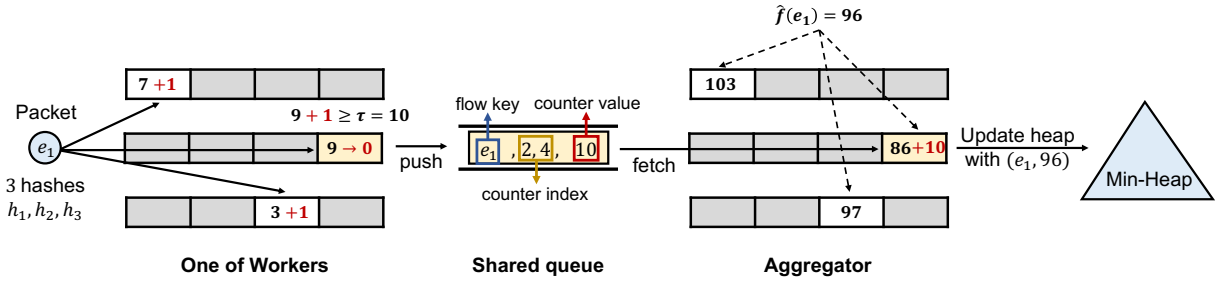


Figure 8: The insertion step in OctoSketch

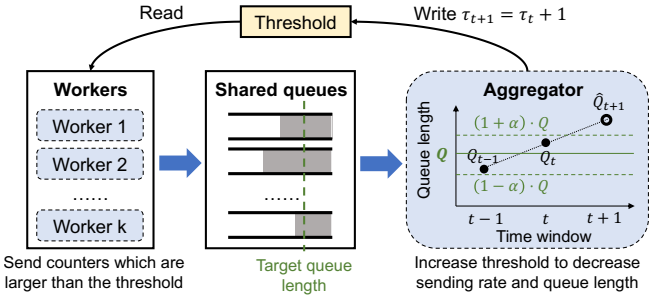


Figure 9: Adaptive thresholds

flow size e based on the query method of the Count-Min sketch (minimum counter value in the d accessed counters) and insert e into the heap. For example, as shown in Figure 8, if a tuple $< e_1, 2, 4, 10 >$ is in the shared queue, the aggregator will increase the 4th counter in the 2nd array by 10. Note if the counter after updating is larger than the minimum value record in the heap, the aggregator will update the heap with the tuple $(e_1, 96)$, where 96 is the estimated size of e_1.

4.3 Resource Allocation

We conceptually consolidate k workers as a single worker and k shared queues as a single queue to address the resource allocation problem. All k workers share the same threshold τ. The threshold is an atomic value. Ideally, the aggregator should adjust the threshold τ to match its receive rate with the total sending rate of k workers. Different from a distributed system where servers are interconnected by the network, the aggregator in a single server can quickly access the variables of other cores and obtain a global view to adjust the threshold.

The aggregator periodically modifies the threshold (*e.g.*, every 100μs). To ensure that the total queue length is small, the aggregator should set a target queue length Q (*e.g.*, 10). [2] For each time window t, the aggregator measures the total queue length Q_t as a reference mark for adjusting the threshold. We assume the sending rate is stable in that short period, so we can predict the queue length for the next time window based on the previous one. Specifically, as shown in Equation 1, according to Q_{t-1} and Q_t, we calculate the expected queue length $\widehat{Q}_{t+1}$ for the next time window $t+1$ corresponding to the threshold τ.

$$\widehat{Q}_{t+1} = Q_t + (Q_t - Q_{t-1}) \tag{1}$$

As shown in Equation 2, if the expected queue length $\widehat{Q}_{t+1}$ is too small, the aggregator will decrease the threshold to increase the send rate and the queue length; if $\widehat{Q}_{t+1}$ is too large, the aggregator will increase the threshold to decrease the queue length.

$$\tau_{t+1} = \begin{cases} \tau_t - 1, & \widehat{Q}_{t+1} < (1-\alpha)\cdot Q \\ \tau_t + 1, & \widehat{Q}_{t+1} > (1+\alpha)\cdot Q \\ \tau_t, & \text{Otherwise} \end{cases} \tag{2}$$

Policies for resource allocation:

- *Best possible accuracy:* The policy we describe above can make OctoSketch quickly adapt to dynamic packet arrival rates. However, it will use up 100% of CPU resources in the aggregator to improve accuracy as much as possible, which may not be ideal. Alternatively, we can consider two additional policies to reduce the aggregator CPU utilization.
- *Optimizing CPU usage for an accuracy target:* As we will prove in §5, the error bound of OctoSketch depends on the threshold τ. Based on an accuracy target, we can calculate the corresponding threshold τ' based on Equation 3 and set it as the lower bound of the threshold. In particular, the threshold will not decrease if it is smaller than τ'. In this policy, we can free up some extra computation resources when there is not much traffic.
- *Setting CPU usage limit:* We can also add a scheduling policy that forces the aggregator to use at most p% (*e.g.*, 50%) of the CPU. We leverage the Linux program CPU limit [46] to monitor and control CPU usage within p%.

4.4 Extension to Other Sketches

With some additional considerations, we can apply OctoSketch to a broad spectrum of sketches. In this work, we consider 8 additional sketches and scale them into multiple cores with OctoSketch. We summarize the key point of applying OctoSketch to these sketches and defer the details of each sketch to Appendix C.

[2] The target queue length should be larger than 0. Otherwise, the aggregator tends to keep increasing the threshold to ensure that the sending rate is always lower than the receiving rate.

Supporting sketches for estimating Cardinality: The merging process of cardinality-related sketches (*e.g.*, HyperLogLog [39], Locher sketch [29]) is different from the Count-Min sketch. Given the received counter A and the corresponding counter B in the aggregator, instead of summing up the counters, these sketches should set $B = \text{Max}(A,B)$. To apply OctoSketch to these kinds of sketches, every time the counter is updated, the worker never zeros it out but only sends the counter whose value change is large enough to the aggregator. Take HyperLogLog as an example. If the counter before updating is C and is C' after updating, the worker will send the counter if $\left|2^C - 2^{C'}\right| \geqslant 2^\tau$. We use 2^C instead of C, because the estimation provided by the HyperLogLog is based on the 2^C, which is the same as that of LogLog and Locher sketch. In Appendix B, we prove that, after applying OctoSketch, the cardinality-related sketches can achieve the same accuracy as the ideal one if the cardinality is sufficiently large.

Handling counters with flow keys: Unlike Count-Min sketch where the sketch structure only contains counters, some complex sketches (*e.g.*, CocoSketch [16] and Elastic sketch [13]) has a flow key corresponding to every counter. For these sketches, OctoSketch will send both the key and the counter to the aggregator and set the counter to zero if the counter is large enough. For each <key, counter> pair, the aggregator inserts the key into the sketch using the same insertion logic as the original sketch.

Handling negative counter values: For sketches such as the Count Sketch [11] and UnivMon [12], each counter value can be positive or negative. For these sketches, when OctoSketch checks whether the counter is large enough, we use the absolute value instead.

Benefits of applying OctoSketch (Table 1): The actual benefits vary among sketches as shown in Table 1. The throughput benefit of OctoSketch often comes from the heavy key storage. Because some sketches (*e.g.*, ElasticSketch [13]) proposed their own heavy key storage to speed up, OctoSketch cannot further optimize their throughput. The throughput may decrease due to the additional overhead of the concurrent queue. However, such a gap is often small (< 10%), and OctoSketch can still improve their accuracy.

5 Analysis

In this section, we first show that, after applying OctoSketch, sketches can still achieve the same error bounds as that of the ideal accuracy after receiving enough packets. Then, we analyze the tradeoff between online accuracy and communication cost. We prove that OctoSketch requires up to four orders of magnitude fewer worker-aggregator messages than prior sketch-merge approach for the same accuracy.

5.1 Error Bound

We show the error bound of OctoSketch for the Count-Min sketch in this section. The detailed proofs and analysis of OctoSketch for Count sketch and HyperLogLog are deferred to Appendix B. These three sketches provide three typical kinds of accuracy guarantees which are also used by other sketches (*e.g.*, Elastic sketch, Locher sketch, and UnivMon). To analyze the worst-case guarantee, we assume that the threshold is always τ_{max} and denote $\tau \equiv \tau_{max}$.

Let $\widehat{f}(e)$ be the estimated flow size of flow e for OctoSketch, $f(e)$ be the real flow size of flow e, k' be the maximum number of workers that a flow may pass by, and $L_1 = \sum_e f(e)$ which is the total number of packets in the traffic.

Theorem 1. *For OctoSketch for Count-Min sketch, let* $d = \log_2 \delta^{-1}$ *and* $l = 2\varepsilon^{-1}$. *For any flow e and any traffics whose* $L_1 > \varepsilon^{-1} k' \tau$,

$$\Pr\left[\left|\widehat{f}(e) - f(e)\right| > \varepsilon L_1\right] < \delta \tag{3}$$

Proof sketch: Suppose that $\widehat{f'}(e)$ is the estimated size of the Count-Min sketch working in a single core. Note that $\left|\widehat{f}(e) - f(e)\right| \leqslant \left|\widehat{f'}(e) - f(e)\right| + \left|\widehat{f}(e) - \widehat{f'}(e)\right|$, where the first part $\left|\widehat{f'}(e) - f(e)\right|$ is the original error from the Count-Min, while the second part $\left|\widehat{f}(e) - \widehat{f'}(e)\right|$ is the additional error brought by OctoSketch. For the first part, we can reuse the result in the paper of Count-Min sketch [10]. The analysis of the additional error caused by OctoSketch is shown in Appendix B. We show that such an additional error is finally marginal compared with that of the Count-Min sketch.

Interpretation: This theorem shows that OctoSketch can still achieve the same asymptotic error bounds as perform sketches in a single location after receiving enough packets. The number of packets depends on the maximum number of workers a flow may pass by (k') and the threshold of each worker (τ). In our experiments, setting τ to 2^7 is enough to achieve around 1400Mpps for 16 workers. In that case, after receiving 1M packets, OctoSketch can guarantee the same asymptotic error bounds for $\varepsilon > 2^{-9} \approx 0.2\%$. Note that flows may not appear on all 16 workers. If each flow is distributed to most two workers (*i.e.*, $k' = 2$), after 1M packets, OctoSketch can guarantee the same error bounds of $\varepsilon > 2^{-12} \approx 0.024\%$. Moreover, using this theorem, we can also compute the required threshold τ based on the accuracy requirement ε, trace length L_1, and k'. Specifically, we can set the threshold as $\tau = \frac{\varepsilon L_1}{k'}$ (4).

5.2 Communication and Accuracy tradeoff

We want to guarantee that the accuracy of the aggregated result over multiple cores is close to the ideal accuracy. We first formally define this accuracy goal in the following.

Definition 1 (Accuracy Goal)**.** *Given a sketch* S_1 *which works in a single core and a sketch* S_2 *which is the aggregated result over multiple cores, suppose that* $\widehat{f_i}(e)$ *is the estimated flow size of e for sketch* S_i. *We want to ensure that, at any time,*

for any flow e, $\left|\widehat{f}_1(e) - \widehat{f}_2(e)\right| < \Delta$*, where* Δ *is a predefined parameter.*

Note that the cost to achieve the accuracy goal varies for different sketches. In this section, we use the Count-Min sketch [10] and Count sketch [11] as a case study to show the superiority of OctoSketch over sketch-merge. To calculate the cost, we consider the number of counters the aggregator should process to achieve the accuracy goal. Suppose that for the Count-Min/Count sketch in each worker and the aggregator, there are d arrays, each with l counters.

Theorem 2. *To achieve the accuracy goal, sketch-merge needs to send* $O\left(\Delta^{-1}k \cdot N \cdot dl\right)$ *counters, while OctoSketch needs to send* $O\left(\Delta^{-1}k \cdot N \cdot d\right)$ *counters.*

Interpretation: We can see that OctoSketch needs to send l times fewer counters to achieve the accuracy goal compared to sketch-merge. In prior work [12,13], there are often more than thousands of counters in each array, *i.e.*, $l > 10^3$. Therefore, OctoSketch can send much fewer counters to achieve the same accuracy guarantee. Note that the computation cost needed in the aggregator is proportional to the number of counters sent. It also validates our experimental results that sketch-merge needs too much computation to achieve the similar accuracy. In experiments, we set $d = 3$ and $l = 2^{16}$. If we set Δ to 100, for 16 workers ($k = 16$), OctoSketch needs to send at most $0.48N$ counters to achieve the accuracy goal, *i.e.*, OctoSketch only needs to send at most 1 counter to the aggregator per 2 packets. Meanwhile, sketch-merge needs around 3×10^4 counters per packet to achieve the same accuracy.

6 Implementation

We implement OctoSketch using C++ and use xxHash [47] as the hashing library. All experiments are run on CloudLab [48]. We have open-sourced the artifact on GitHub [49].

Shared buffer: We use the concurrent queue [50] to be the shared buffers between workers and the aggregator. The queue is lock-free and uses atomic operations to achieve high throughput. In addition, it can dynamically allocate memory according to the number of items in the queue.

CPU: We implement OctoSketch on a machine with an AMD EPYC 7452 32-Core Processor at 2.35GHz and 128 GB ECC memory. We pre-process the input traces and distribute them to different workers. Each worker only needs to read the flow keys in memory and insert them into the sketch. In that way, we can focus on measuring the performance of sketches without the impacts from other applications.

DPDK: We also integrate OctoSketch with DPDK (version 21.11) [24]. Each OctoSketch worker is integrated with the polling mode thread in DPDK. Our testbed has two servers that are the same as the CPU implementation. Each server is equipped with a Mellanox ConnectX-5 Ex 100G NIC [26]. One server generates high-speed TCP traffic using pktgen-dpdk, while another server runs DPDK to receive packets and process them using OctoSketch. We use multiple cores to receive packets. Each worker corresponds to one core and one Rx queue, and it should extract packets from the Rx queue and insert them into the sketch. Thus, we can measure the overhead of sketches compared with DPDK.

eBPF XDP: We also integrate OctoSketch with XDP [25] in Linux kernel 5.15.0 using *SKB* mode. Our testbed has two servers that are the same as the DPDK implementation. We load the OctoSketch worker into the kernel to process packets, and the aggregator works in the user space. We use the *bpf_ringbuf* provided in the XDP library to send data from kernel space to user space. Note that in our setting, we cannot implement sketch-merge as it needs to send too much data (the whole sketch and heavy flow keys) at a time.

7 Evaluation

Our experiments compare the OctoSketch to the baseline sketch-merge approach and demonstrate that:

- OctoSketch can always maintain high online accuracy at any query time.
- OctoSketch can achieve high throughputs on all tested platforms, and it scales linearly with the number of workers.
- OctoSketch can achieve high resource efficiency on a representative high-performance packet processing library (Intel DPDK).

7.1 Experimental Methodology

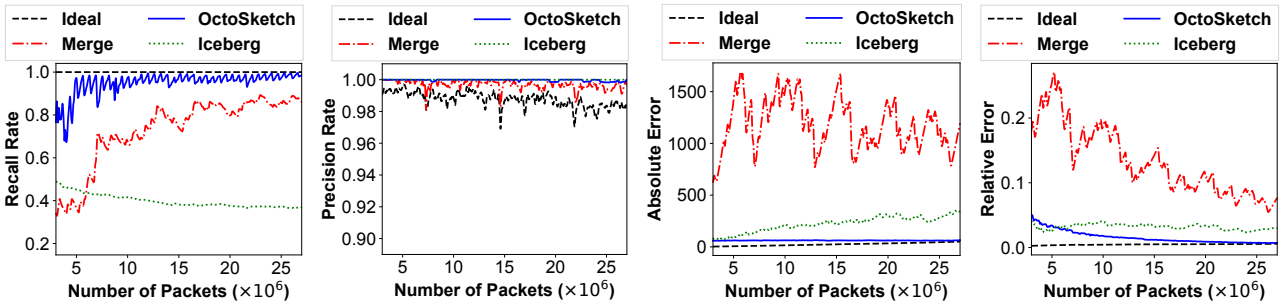
Traces: We mainly use two datasets:

- We use CAIDA traces [37] collected in the Equinix-Chicago monitor in 2018 in for our experiments by default. We use <Source IP, Destination IP> as the flow key.
- We generated 11 datasets that follows the Zipf [51] distribution with various skewness using Web Polygraph [52].

Metrics: We evaluate the following six performance metrics.

- *Absolute Error:* $\frac{1}{|Q|}\sum_{e\in Q}\left|f(e) - \widehat{f}(e)\right|$, where $f(e)$ is the real size, $\widehat{f}(e)$ is the estimated size, and Q is the query set.
- *Relative Error:* $\frac{1}{|Q|}\sum_{e\in Q}\frac{|f(e)-\widehat{f}(e)|}{f(e)}$, where $f(e)$ is the real value, $\widehat{f}(e)$ is the estimated value, and Q is the query set.
- *Recall Rate:* The ratio of the number of correctly reported flows to the number of correct flows.[3]
- *Miss Rate:* The ratio of the number of missing correct flows to the number of correct flows, which is $1 - RR$.
- *Precision Rate:* The ratio of the number of correctly reported flows to the number of reported flows.
- *F1 Score:* F1 Score is $2 \cdot (RR \cdot PR)/(RR + PR)$, where RR is the recall rate, and PR is the precision rate.
- *Throughput*: For CPU, we use million insertions per second (Mips). For DPDK and XDP, we use million packets per second (Mpps). The throughput numbers are the average value among 100 trials.

[3] Correct flows are the real heavy hitters in the traffic, and correctly reported flows are the real heavy hitters in the reported ones.



(a) Recall Rate (b) Precision Rate (c) Absolute Error (d) Relative Error

Figure 10: Accuracy of Count-Min Sketch at different query time.

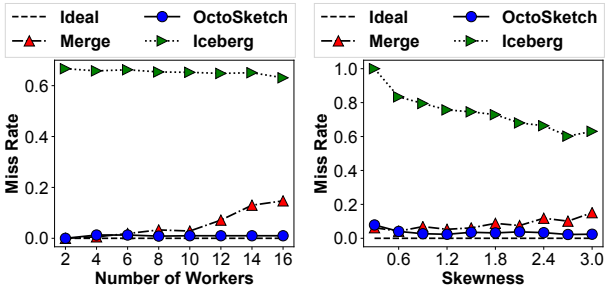
Sketches: We apply OctoSketch to all 9 sketches in Table 1. Due to space limitations, we show 4 representative sketches in this section: Count-Min Sketch [10] for detecting heavy hitters, Locher Sketch [29] for measuring super-spreaders, CocoSketch [16] for arbitrary partial key queries, and UnivMon [12] for generic flow monitoring.

Baselines: (1) *Ideal accuracy* is the accuracy of the sketch that works in a single core and measures the whole traffic. (2) *Sketch-merge* is merging the entire sketches as shown in §2.2. (3) *Iceberg* [53] uses the Count-Min sketch to find global heavy hitters. Instead of utilizing the sketch mergeability, Iceberg only uses the Count-Min sketch as a local heavy hitter estimator. Each worker in the Iceberg only sends the local heavy flow keys to the aggregator. The aggregator will ask every other worker for the statistics of these flow keys and aggregate the result. Once all worker answers, the aggregator can decide on the global heavy hitters. As shown in its paper [53], such a multi-round communication can reduce communication overhead.

Parameters: We refer to the individual paper to configure the sketch parameters [10, 12, 13, 16, 29]. For instance, in Count-Min sketch and Locher sketch, we use 3 arrays of 2^{16} counters per array. For CocoSketch, we use 2 arrays as suggested in their paper, and there are 2^{16} buckets per array. We describe the detailed parameters and configurations in Appendix D.1.

7.2 Online Accuracy

F1 scores at different query times (Figure 10a-10b&20): The F1 Score of OctoSketch is often close to the ideal accuracy and is 36.5% higher than sketch-merge. The sketch-merge needs to wait longer to get the sketches from all workers and converge to a relatively accurate result. Therefore, its F1 score is often lower than 0.5 before processing 5 million packets. Then we look at the recall (Figure 10a) and precision (Figure 10b) of the F1 score, respectively. The recall of OctoSketch is usually lower than the ideal accuracy, while the precision is often higher. It is because the counter in the aggregator is often smaller than that of the ideal accuracy since there is some information left in each worker. As a result, OctoSketch tends to underestimate compared to the ideal accuracy, resulting in a higher precision but a lower recall. The recall rate of Iceberg is around 54% lower than OctoSketch. Its recall rate is low because the multi-round communication to all workers increases the delay (staleness) in the aggregator, which makes the aggregator miss many heavy flows.



(a) Accuracy vs. # of workers (b) Accuracy vs. Skewness

Figure 11: Accuracy with different parameters

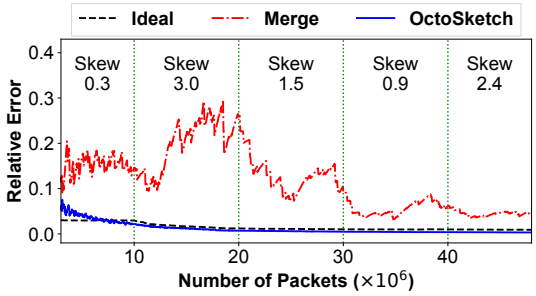
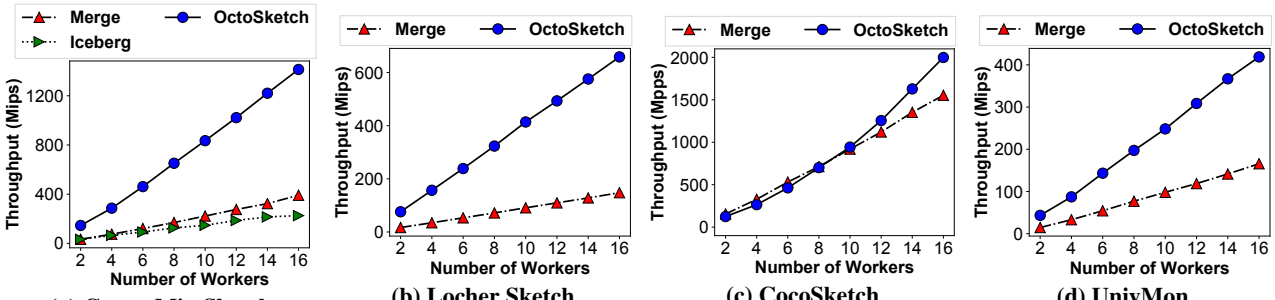


Figure 12: Dynamically changing workloads

Errors at different query times (Figure 10c-10d&21): OctoSketch can keep low errors in any query time. The gap between the relative error of OctoSketch and the ideal accuracy is often less than 0.05 and continues to decrease with more packets received, which matches our theoretical analysis in §5.1. The relative error of sketch-merge is 15.6× larger than OctoSketch, and its absolute error is 41.9× larger than OctoSketch. In addition, the accuracy of sketch-merge changes significantly over time: error drops abruptly once the aggregator merges a sketch, and keeps increasing until the aggregator merges another one. In contrast, OctoSketch's continuous mechanism maintains stable accuracy. The relative error of Iceberg is 2.6× larger than OctoSketch.



(a) Count-Min Sketch (b) Locher Sketch (c) CocoSketch (d) UnivMon

Figure 13: Throughput on CPU

Accuracy with different numbers of workers (Figure 11a): The miss rate of sketch-merge on the Count-Min sketch increases when scaling to more workers. It is because the time it takes to merge sketches in the aggregator has to increase as the number of workers increases. With 16 workers, the miss rate of sketch-merge for the Count-Min sketch is around 14.7× larger than that of OctoSketch, while the difference between the miss rate of OctoSketch and the ideal one is less than 2%. The Iceberg algorithm uses multi-round communication, which further increases the complexity and waiting time of the aggregator. As a result, its miss rate is around 12.0× larger than sketch-merge and 70.3× larger than OctoSketch.

Accuracy with different skewness (Figure 11b): OctoSketch for the Count-Min sketch achieves low miss rates on different skewnesses, while the error of sketch-merge increases with increasing skewness. It also verifies our statement shown in §3.2 that sketch-merge is not friendly to heavy-tailed workloads. The miss rate of sketch-merge is around 9.1× larger than OctoSketch, while the miss rate of Iceberg is around 21.0× larger.

Dynamically changing workloads (Figure 12): We simulate dynamic workloads by combing five datasets with different skewness ($0.3 \rightarrow 3.0 \rightarrow 1.5 \rightarrow 0.9 \rightarrow 2.4$). We find that OctoSketch for the Count-Min sketch is robust to dynamically changing workloads and maintains similar accuracy to the ideal. However, we note that OctoSketch does not address the inherent accuracy issues of the sketches. If the original sketch used is not capable of handling dynamically changing workloads, OctoSketch does not mitigate this limitation, but does not worsen the accuracy. Nevertheless, we observe that many sketches [10, 11, 13, 39] tend to exhibit a certain level of robustness to such dynamic workloads.

7.3 Throughput

CPU Throughput (Figure 13): OctoSketch achieves high CPU throughput and scales well with more workers. In the Count-Min sketch, with 16 workers, OctoSketch can achieve around 1400Mips. The multi-round communication in Iceberg increases the computation overhead of each worker and thus decreases the throughput. The throughput of Iceberg is around 1.4× lower than sketch-merge and 5.3× lower than OctoSketch. Compared to sketch-merge, OctoSketch can achieve 3.85×, 4.5×, and 2.63× higher throughput for Count-Min sketch, Locher sketch, and UnivMon, respectively. It is worth noting that the throughput of OctoSketch can sometimes increase faster than linear scaling with more workers. This is because the counter update threshold may also increase with more workers. Therefore, each worker costs less to send counters and the total throughput increases faster. For CocoSketch, due to the extra overhead of OctoSketch to send counters, the throughput sometimes is slightly lower than sketch merge, but the gap between them is less than 10%.

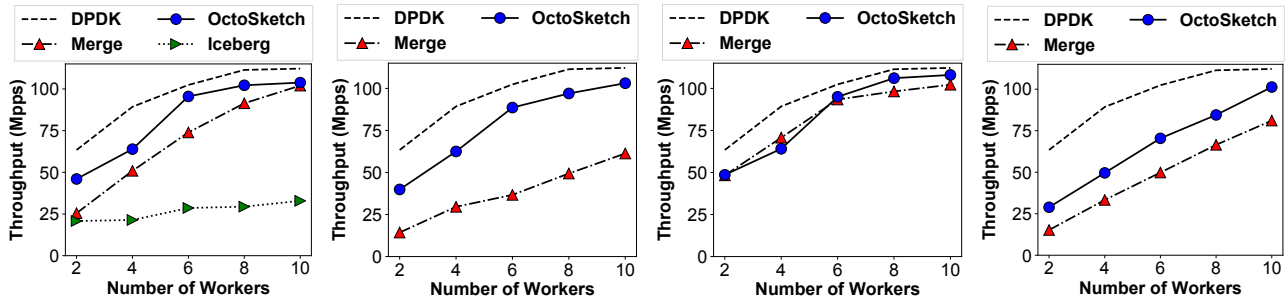
DPDK Throughput (Figure 14): OctoSketch achieves high throughput in DPDK. For the six tested sketches, OctoSketch can often reach ≈100Mpps with 10 workers. Sketch-merge often needs at least 2 more workers to achieve similar throughput as OctoSketch for Count-Min sketch, Locher sketch, and UnivMon. Specifically, OctoSketch accelerates the Locher sketch to achieve 97.0Mpps in 8 workers, while sketch-merge only achieves 61.3Mpps using 10 workers. One exception is CocoSketch: the throughput of OctoSketch is similar to that of the sketch-merge. This is because CocoSketch does not need additional heavy key storage, and thus OctoSketch does not further optimize throughput. Iceberg reaches only 32.9Mpps with 10 workers, because Iceberg needs multiround communication between the aggregator and all workers and additional workers can bring communication overheads.

eBPF XDP Throughput (Figure 15): As we cannot implement sketch-merge in XDP as shown in §6, we compare the throughput of OctoSketch to that of the XDP with sketches. The throughput of OctoSketch scales linearly with the number of workers. The throughput of OctoSketch with the Count-Min sketch is about 85% of the XDP, and the throughput of OctoSketch with the CocoSketch achieves 92% of the XDP.

7.4 CPU Utilization and Stability

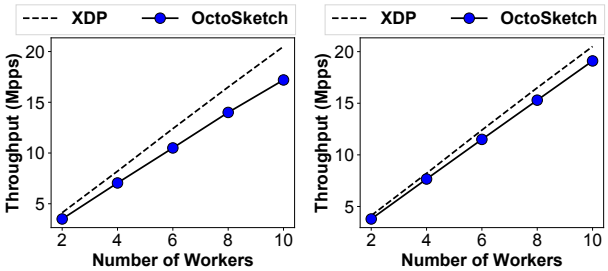
In this experiment, we show how OctoSketch can quickly adapt to varying packet arrival rates in DPDK.

CPU usage (Figure 16): The sketch CPU utilization is often reduced after applying OctoSketch. For the Count-Min sketch, the CPU usage of OctoSketch is around 1.75 and 3.34 times lower than sketch-merge and Iceberg, respectively. As shown in §7.3, OctoSketch incurs more CPU cycles when applied to CocoSketch due to the cost of sending counters. But the



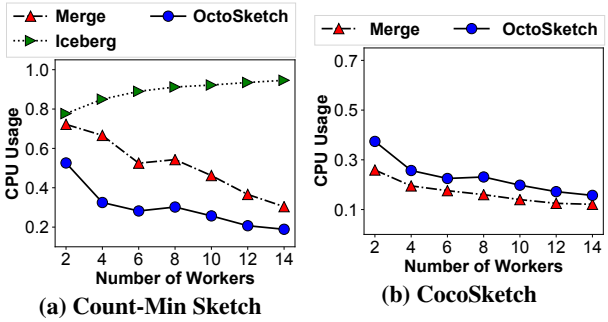
(a) Count-Min Sketch (b) Locher Sketch (c) CocoSketch (d) UnivMon

Figure 14: Throughput on DPDK



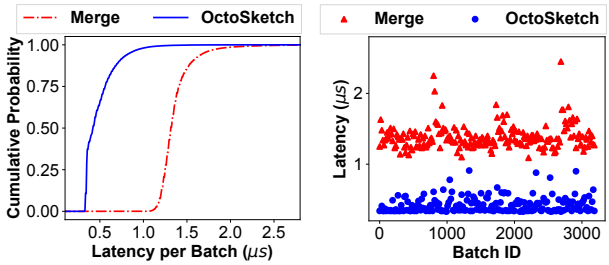
(a) Count-Min Sketch (b) CocoSketch

Figure 15: Throughput on XDP



(a) Count-Min Sketch (b) CocoSketch

Figure 16: CPU usage on DPDK



(a) CDF of Latency (b) Mini-batch Latency

Figure 17: Latency on DPDK

difference in CPU usage is usually less than 7%. Note that the CPU usage of sketches often decreases with more workers because the overhead of performing packet processing under DPDK will be increased when dealing with large volumes of traffic. The CPU usage of Iceberg increases with more workers due to the extra multi-round communication between workers and the aggregator.

Packet processing latency (Figure 17): We measure the latency needed for the Count-Min sketch in mini-batches (32 packets). We do not show per-packet latency as it is too fine-grained (≈ 10ns) for servers to measure. As shown in Figure 17a, for OctoSketch, more than 65% of the batches can be finished in 0.5μs, and over 98% of batches can be finished in 1μs. Moreover, the median latency of sketch-merge is around 3 times larger than that of the OctoSketch. Figure 17b shows that sketch-merge has larger jitters than OctoSketch. Every time sketch-merge creates a new sketch, there will be frequent sketch and flow key operations in the next few batches, leading to large latency. The jitters in OctoSketch are mainly caused by the *atomic* operations in the concurrent queue. Overall, the latencies and jitters in OctoSketch are significantly smaller than those of sketch-merge.

Adaptive thresholds (Figure 18): To illustrate the adaptiveness of OctoSketch, we use 10 workers to run Count-Min sketches with DPDK, set the lower bound of the thresholds to 8, and dynamically change the packet arrival rate. The upper figure shows that OctoSketch can quickly adapt to the varying arrival rates of packets. Moreover, we notice that the highest threshold used is still smaller than 20, *i.e.*, 8-bit is enough for each counter. The bottom figure shows the total queue length of the 10 shared queues. We find that the large queue length is often caused by the threshold decrease. Even if the packet arrival rate is stable, OctoSketch will still dynamically adjust the threshold. Sometimes, the threshold is too low, leading to a longer queue. However, OctoSketch can quickly adjust the threshold to decrease the queue length. We can see the queue length is relatively stable around the target length and is usually well below 400. Therefore, the size of the shared buffer needed by OctoSketch is often smaller than 10KB. Such a size is much smaller than that of the sketch-merge. For sketch-merge, we need at least 768KB (the size of a Count-Min sketch) more memory for merging.

7.5 Case Study: Load balancer

In this section, we show the benefit of OctoSketch for the Intel dynamic load balancer [35]. We also conduct a case study on the key-value cache and defer the results to Appendix §D.3.

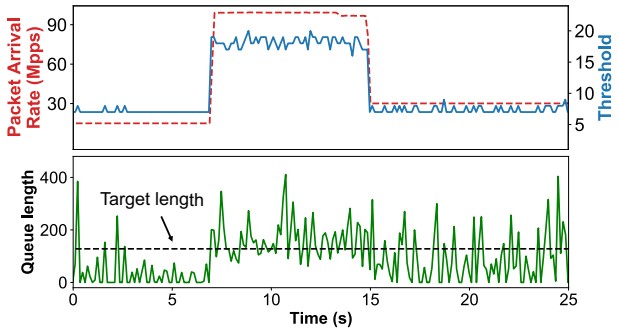
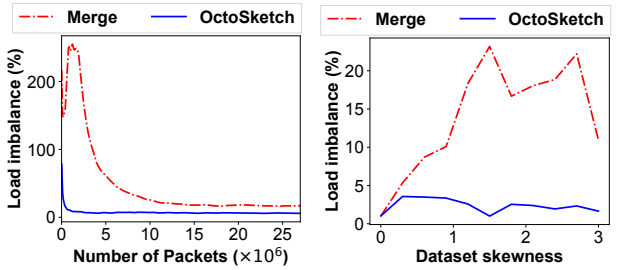


Figure 18: Adaptive thresholds



(a) Over time **(b) Over skewness**

Figure 19: Load balancer with different monitoring tools.

Initially, all traffic is distributed through RSS (based on hash). The aggregator aggregates the statistics over multiple cores and detects large flows. Once a flow is detected as a large flow, the aggregator sets a flow director rule for this flow in the network adaptor. Then, the large flow will be redirected and redistributed to multiple cores with the help of Intel DLB [54] which maintains the order of packets. In this experiment, we run simulations on the CPU due to the lack of Intel DLB hardware support in our testbed and measure load imbalance rate as the ratio of the number of packets processed by the most loaded worker relative to the least loaded worker [55]. The ideal imbalance rate should be 0.

Load balance over time (Figure 19a): OctoSketch helps Intel load balancer to achieve low imbalance rates. With fewer than 1M packets, the imbalance rate from OctoSketch is lower than 10%. However, sketch-merge needs more than 10M packets to reach a low imbalance rate and its imbalance rate can be large ($> 100\%$) before that. This is because sketch-merge spends more time detecting new large flows than OctoSketch, resulting in more packet losses. Moreover, after processing more than 20M packets, the imbalance rate of OctoSketch is still $3.15\times$ lower than that of sketch-merge.

Load balance over skewness (Figure 19b): We also show the imbalance rates on processing different packet traces with varying skewness. OctoSketch maintains low imbalance rates regardless of the skewness, while sketch-merge's imbalance rate is about $8.89\times$ larger when the skewness is larger than 1.2. Note that larger skewness does not always lead to larger imbalances. The imbalance rate also depends on how fast the algorithm can detect large flows and redistribute them.

8 Other Related work

Mergeability of sketches: As discussed in §2.2, most existing solutions (*e.g.*, CocoSketch [56], Beaucoup [15], HeteroSketch [20], and FetchSGD [36]) leverage the mergeability to apply sketches to distributed systems. These works often send the whole sketch to the aggregator, and the aggregator can get the aggregated result based on the merged sketch. There are also some works [57] that try to explore the operations that can be supported by merge. Specifically, prior works mainly use merge to get the statistics over union ($A \cup B$), while [57] discusses how to use merge to support other operations like intersection, Jaccard similarity, and relative complement.

Continuous monitoring model: In addition to the linear mergeability [9] discussed in §2.2, there are also theoretical works about continuous monitoring for distributed systems. However, some of these efforts [58] assume that every worker has a lot of resources to record all flow sizes, while some of them [59, 60] only focus on a specific sketch of a task. In addition, these works [53] all aim to minimize the total communication cost, and failed to consider the computation cost. As shown in §2.2, the main bottleneck in the multicore scenario is the computation cost.

Real-time telemetry: Timeliness is an important property in distributed monitoring. Trumpet [61] is an event monitoring system in which users define network-wide events. There is a centralized controller which installs triggers at end-hosts where triggers test for local conditions, and the controller aggregates these signals and tests for the presence of specified network-wide events. They use a hash table in each end-host and assume that the end-host can process all packets with full accuracy. However, in our high-volume, multicore context, each core in a server does not have enough computation and memory resources to do so.

9 Conclusions

Today's networked applications require multicore packet processing and distributed flow monitoring. While sketches have emerged as a resource-efficient and highly accurate measurement primitive, fundamental limitations remain. Prior distributed solutions merge sketches from all cores and bring significant accuracy degradation and resource overhead. In this paper, we propose OctoSketch to scale sketches to multicore scenarios using a continuous, change-based mechanism to aggregate multicore measurements. OctoSketch can be applied to a broad range of sketches and answer various sketch-based queries. Our evaluation shows that OctoSketch can achieve significantly higher accuracy and throughput than prior sketch-merge techniques.

Acknowledgments: We thank the anonymous reviewers and our shepherd Behnaz Arzani for their thorough comments and feedback. This work was supported in part by NSF grants CNS-2107086, SaTC-2132643, CNS-2106946, and the Red Hat Collaboratory award.

References

[1] Georgios P. Katsikas, Tom Barbette, Dejan Kostic, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *NSDI 2018*, pages 171–186. USENIX Association, 2018.

[2] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: a framework for NFV applications. In *SOSP 2015*, pages 121–136. ACM, 2015.

[3] Brocade vyatta 5400 vrouter. http://www.brocade.com/products/all/networkfunctions-virtualization/product-details/5400-vrouter/index.page.

[4] DPDK-Based Load Balancer. https://dpdksummitapac2021.sched.com/event/hdLm.

[5] Seyed Kaveh Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic ddos defense. In *USENIX Security 2015*, pages 817–832. USENIX Association, 2015.

[6] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *SOSP 2017*, pages 121–136. ACM, 2017.

[7] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *FAST 2019*, pages 143–157. USENIX Association, 2019.

[8] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proc. of SIGCOMM*, 2017.

[9] Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff M. Phillips, Zhewei Wei, and Ke Yi. Mergeable summaries. In *PODS 2012*, pages 23–34. ACM, 2012.

[10] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 2005.

[11] Moses Charikar, Kevin C. Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 2004.

[12] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *ACM SIGCOMM*, 2016.

[13] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: adaptive and fast network-wide measurements. In *SIGCOMM 2018*, pages 561–575. ACM, 2018.

[14] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: robust and general sketch-based monitoring in software switches. In *SIGCOMM 2019*. ACM, 2019.

[15] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 226–239, 2020.

[16] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: high-performance sketch-based measurement over arbitrary partial key query. In *SIGCOMM 2021*, pages 207–222. ACM, 2021.

[17] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *ICNP 2018*, pages 313–323. IEEE Computer Society, 2018.

[18] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison. Elastic switch programming with p4all. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 168–174, 2020.

[19] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. SketchLib: Enabling efficient sketch-based monitoring on programmable switches. In *USENIX NSDI*, 2022.

[20] Anup Agarwal, Zaoxing Liu, and Srinivasan Seshan. {HeteroSketch}: Coordinating network-wide monitoring in heterogeneous and dynamic networks. In *In USENIX NSDI)*, pages 719–741, 2022.

[21] Peiqing Chen, Yuhan Wu, Tong Yang, Junchen Jiang, and Zaoxing Liu. Precise error estimation for sketch-based flow measurement. In *ACM Internet Measurement Conference (IMC)*, 2021.

[22] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. SCREAM: sketch resource allocation for software-defined measurement. In *CoNEXT 2015*, pages 14:1–14:13. ACM, 2015.

[23] Hao Zheng, Chen Tian, Tong Yang, Huiping Lin, Chang Liu, Zhaochen Zhang, Wanchun Dou, and Guihai Chen. Flymon: enabling on-the-fly task reconfiguration for network measurement. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 486–502, 2022.

[24] Data plane development kit. https://www.dpdk.org/.

[25] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: fast programmable packet processing in the operating system kernel. In *CoNEXT 2018*, pages 54–66. ACM, 2018.

[26] ConnectX-5 Ex. https://www.mellanox.com/related-docs/oem/dell/PB_ConnectX-5_Ex_Card_dell.pdf.

[27] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. The design and implementation of open vswitch. In *NSDI 15*, pages 117–130. USENIX Association, 2015.

[28] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: fine grained traffic engineering for data centers. In *Co-NEXT '11*. ACM, 2011.

[29] Thomas Locher. Finding heavy distinct hitters in data streams. In *SPAA 2011*, pages 299–308. ACM, 2011.

[30] Qun Huang, Patrick P. C. Lee, and Yungang Bao. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *SIGCOMM 2018*, pages 576–590. ACM, 2018.

[31] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *SIGCOMM 2017*, pages 113–126. ACM, 2017.

[32] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *NSDI 2013*. USENIX Association, 2013.

[33] Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. https://tools.ietf.org/html/rfc3176.

[34] Sajad Shirali-Shahreza and Yashar Ganjali. Flexam: flexible sampling extension for monitoring and security applications in openflow. In *HotSDN 2013*, 2013.

[35] Intel dynamic load balancer (intel dlb) - accelerating elephant flow. https://builders.intel.com/docs/networkbuilders/intel-dynamic-load-balancer-intel-dlb-accelerating-elephant-flow-technology-guide-1677672283.pdf.

[36] Daniel Rothchild, Ashwinee Panda, Enayat Ullah, Nikita Ivkin, Ion Stoica, Vladimir Braverman, Joseph Gonzalez, and Raman Arora. Fetchsgd: Communication-efficient federated learning with sketching. In *ICML 2020*, volume 119, pages 8253–8265. PMLR, 2020.

[37] The CAIDA UCSD Anonymized Internet Traces. http://www.caida.org/data/passive/passive_dataset.xml.

[38] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities (extended abstract). In *ESA 2003*, volume 2832 of *Lecture Notes in Computer Science*, pages 605–617. Springer, 2003.

[39] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156, 2007.

[40] Charles Masson, Jee E. Rim, and Homin K. Lee. Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *Proc. VLDB Endow.*, 12(12):2195–2205, 2019.

[41] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Randomized admission policy for efficient top-k and frequency estimation. In *INFOCOM 2017*, pages 1–9. IEEE, 2017.

[42] Abhinandan Das, Sumit Ganguly, Minos N. Garofalakis, and Rajeev Rastogi. Distributed set expression cardinality estimation. In *VLDB 2004*, pages 312–323. Morgan Kaufmann, 2004.

[43] Minos N. Garofalakis. Distributed data streams. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 883–890. Springer US, 2009.

[44] Brian Babcock and Chris Olston. Distributed top-k monitoring. In *SIGMOD 2003*, pages 28–39. ACM, 2003.

[45] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-wide heavy hitter detection with commodity switches. In *SOSR 2018*, pages 8:1–8:7. ACM, 2018.

[46] cpulimit. https://github.com/opsengine/cpulimit.

[47] xxHash Library. http://www.xxhash.com/.

[48] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *ATC*, pages 1–14, July 2019.

[49] Source code related to OctoSketch. https://github.com/Froot-NetSys/OctoSketch.

[50] Fast Concurrent Queue. https://github.com/cameron314/readerwriterqueue.

[51] David MW Powers. Applications and explanations of Zipf's law. In *Proc. EMNLP-CoNLL*. Association for Computational Linguistics, 1998.

[52] Alex Rousskov and Duane Wessels. High-performance benchmarking with web polygraph. *Software: Practice and Experience*, 34(2):187–211, 2004.

[53] Emmanuelle Anceaume, Yann Busnel, Nicolo Rivetti, and Bruno Sericola. Identifying global icebergs in distributed streams. In *SRDS 2015*, pages 266–275. IEEE Computer Society, 2015.

[54] Queue management and load balancing on intel architecture. https://builders.intel.com/docs/networkbuilders/SKU-343247-001US-queue-management-and-load-balancing-on-intel-architecture.pdf.

[55] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire Jr., and Dejan Kostic. RSS++: load and state-aware receive side scaling. In *CoNEXT 2019*, pages 318–333. ACM, 2019.

[56] Ruijie Miao, Yinda Zhang, Zihao Zheng, Ruixin Wang, Ruwen Zhang, Tong Yang, Zaoxing Liu, and Junchen Jiang. High-performance sketch-based measurement over arbitrary partial key query. *IEEE/ACM Transactions on Networking*, 2023.

[57] Jakub Lemiesz. On the algebra of data sketches. *Proc. VLDB Endow.*, 14(9):1655–1667, 2021.

[58] Ram Keralapura, Graham Cormode, and Jeyashankher Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *SIGMOD 2006*, pages 289–300. ACM, 2006.

[59] Ke Yi and Qin Zhang. Optimal tracking of distributed heavy hitters and quantiles. In *PODS 2009*, pages 167–174. ACM, 2009.

[60] Zengfeng Huang, Ke Yi, and Qin Zhang. Randomized algorithms for tracking distributed count, frequencies, and ranks. In *PODS 2012*, pages 295–306. ACM, 2012.

[61] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *ACM SIGCOMM 2016*, pages 129–143. ACM, 2016.

[62] Sketch in DPDK. https://github.com/DPDK/dpdk/commit/db354bd2e1f82294ddeb667a1dbca27a325d1eb4.

A Background of sketches

Here we give a background of sketching algorithms and the applications of sketches on software platforms.

Sketch example: Count-Min sketch. The Count-Min sketch [10] maintains d arrays of counters, with w counters in each array. When a packet arrives, the Count-Min sketch calculates d independent hash values from the *flow identity* of the packet (*e.g.*, 5-tuple, source IP, and destination IP). Each hash function offers an independent counter position in each array, and the Count-Min sketch subsequently increases the associated counter per array. When querying the size of a flow, its estimation is given by the minimum value among the associated counters for that flow.

Applications of sketches: Performing telemetry under high-performance packet processing is an important application for sketches. As an example, DPDK has included sketching algorithms in their library to provide an efficient way to profile the traffic for heavy hitters [62]. In addition, prior works [4] apply sketches to a DPDK-based Load Balancer. Specifically, they use sketches to detect heavy hitters and distribute them to multiple idle worker cores for parallel packet processing. Moreover, prior system work also applies sketches to super-spreader/DDoS detection [32] and key-value cache [7, 21].

B Analysis

Theorem 1. *For OctoSketch for Count-Min sketch, let* $d = \log_2 \delta^{-1}$ *and* $l = 2\varepsilon^{-1}$. *For any flow* e *and any traffic whose* $L_1 > \varepsilon^{-1} k' \tau$,

$$\Pr\left[\left|\widehat{f}(e) - f(e)\right| > \varepsilon L_1\right] < \delta \tag{4}$$

Proof. Suppose that $\widehat{f'}(e)$ is the estimated size of the Count-Min sketch that works in a single core. Note that $\left|\widehat{f}(e) - f(e)\right| \leqslant \left|\widehat{f'}(e) - f(e)\right| + \left|\widehat{f}(e) - \widehat{f'}(e)\right|$, where the first part $\left|\widehat{f'}(e) - f(e)\right|$ is the original error from the Count-Min sketch, while the second part $\left|\widehat{f}(e) - \widehat{f'}(e)\right|$ is the additional error brought by the OctoSketch.

For the first part, given $d = \log_2 \delta^{-1}$ and $l = 2\varepsilon^{-1}$, the error bound of the Count-Min sketch [10] as shown in its paper is that

$$\Pr\left[\left|\widehat{f'}(e) - f(e)\right| > \varepsilon L_1\right] < \delta. \tag{5}$$

Then, we consider the second part, *i.e.*, the gap of estimated flow size between OctoSketch for Count-Min sketch and the Count-Min sketch that works in a single core. Note that for each worker, there is at most τ flow size information left in the worker's sketch at any time. The total gap between each counter in the aggregator's sketch and the single core baseline is at most $k' \cdot \tau$. Because the gap between the estimation is not larger than the gap between each counter, we can get that $\left|\widehat{f}'(e) - \widehat{f}(e)\right| < k'\tau$. Given $L_1 > \varepsilon^{-1}k'\tau$, we can ensure that $\left|\widehat{f}'(e) - \widehat{f}(e)\right| < \varepsilon L_1$.

Because the Count-Min sketch only overestimates the flow size, we can get that $\widehat{f}'(e) \geqslant f(e)$. Because OctoSketch will always have some flow size information in the worker not updated to the aggregator, we have the query result from the aggregator $\widehat{f}(e) \leqslant \widehat{f}'(e)$. Based on 5 and $\widehat{f}(e) \leqslant \widehat{f}'(e)$, we have

$$\Pr\left[\widehat{f}(e) - f(e) > \varepsilon L_1\right] < \delta. \tag{6}$$

Based on $\widehat{f}'(e) \geqslant f(e)$, $\widehat{f}'(e) \geqslant \widehat{f}(e)$, and $\left|\widehat{f}'(e) - \widehat{f}(e)\right| < \varepsilon L_1$, we have $f(e) - \widehat{f}(e) < \varepsilon L_1$. Based on 6 and $f(e) - \widehat{f}(e) < \varepsilon L_1$, we can get the result. □

Theorem 2. *To achieve the accuracy goal, sketch-merge needs to send $O\left(\Delta^{-1}k \cdot N \cdot dl\right)$ counters, while OctoSketch needs to send $O\left(\Delta^{-1}k \cdot N \cdot d\right)$ counters.*

Proof. We first consider the number of counters needed by the sketch-merge technique. There are $d \cdot l$ counters for each sketch. To guarantee that the gap of estimated size is smaller than Δ, sketch-merge needs to send the whole sketch ($d \cdot l$ counters) to the aggregator for every $\frac{\Delta}{k}$ packets. Therefore, sketch-merge needs to send the sketch at least $\frac{k \cdot N}{\Delta}$ times. As a result, sketch-merge needs to send $O\left(\frac{k \cdot N}{\Delta} \cdot d \cdot l\right)$ counters.

For OctoSketch, to guarantee the error, OctoSketch needs to set the threshold as $\frac{\Delta}{k}$. Note that each packet only accesses one counter in the array. Because there are N packets in the traffic, each array sends at most $\frac{k \cdot N}{\Delta}$ counters. As a result, OctoSketch needs to send $O\left(\frac{k \cdot N}{\Delta} \cdot d\right)$ counters. □

Let $L_k = \sqrt[k]{\sum_e f^k(e)}$ be the k-th norm of the frequency vector of the traffic. Then, we show the error bound of the OctoSketch for the Count sketch.

Theorem 3. *For OctoSketch for the Count sketch, let $d = O(\log_2 \delta^{-1})$ and $l = 8\varepsilon^{-2}$. For any flow e and any traffics whose $L_2 > 2\varepsilon^{-1}k'\tau$,*

$$\Pr\left[\left|\widehat{f}(e) - f(e)\right| > \varepsilon L_2\right] < \delta \tag{7}$$

Proof. Similar to that of the Count-Min sketch, we first analyze the error brought by the Count sketch. Suppose that $\widehat{f}'(e)$ is the estimated size of the Count sketch that works in a single core. Given $d = O(\log_2 \delta^{-1})$ and $l = 8\varepsilon^{-2}$, based on the error bound of the Count sketch [11], we have $\Pr\left[\left|\widehat{f}'(e) - f(e)\right| > \frac{\varepsilon}{2}L_2\right] < \delta$. Then we analyze the error brought by OctoSketch. Given the threshold τ of each worker and $L_2 > 2\varepsilon^{-1}k'\tau$, we can make sure that

$$\left|\widehat{f}'(e) - \widehat{f}(e)\right| < k'\tau < \frac{\varepsilon}{2}L_2$$

Based on the triangle inequality, we have

$$\Pr\left[\left|\widehat{f}(e) - f(e)\right| > \varepsilon L_2\right] < \delta$$

□

Then, we show the error bound of the OctoSketch for HyperLogLog. Because HyperLogLog works for different query tasks from the Count-Min sketch and the Count sketch, we define different symbols for HyperLogLog. Let $\widehat{Z}$ be the estimated cardinality of HyperLogLog, $\widehat{Z}'$ be the estimated cardinality of the OctoSketch for HyperLogLog, m be the number of counters used in the HyperLogLog, α_m is the constant used by HyperLogLog given m, $C[i]$ be the ith counter in the HyperLogLog, and $C'[i]$ be the ith counter in the OctoSketch for HyperLogLog. As shown in the paper [39], α_m is always smaller than 0.73 regardless of m, and

$$\widehat{Z} = \alpha_m m^2 \left(\sum_{i=1}^{m} 2^{-C[i]}\right)^{-1}$$

Theorem 4. *If $\widehat{Z} > 2\alpha_m m^2 2^{\tau-2}$, we have*

$$\widehat{Z} = \widehat{Z}'$$

Proof. OctoSketch can guarantee that, for any i, $\left|2^{C[i]} - 2^{C'[i]}\right| < 2^{\tau}$, and $C'[i] = \tau - 1$ by default. As all counters are integers, we can get that

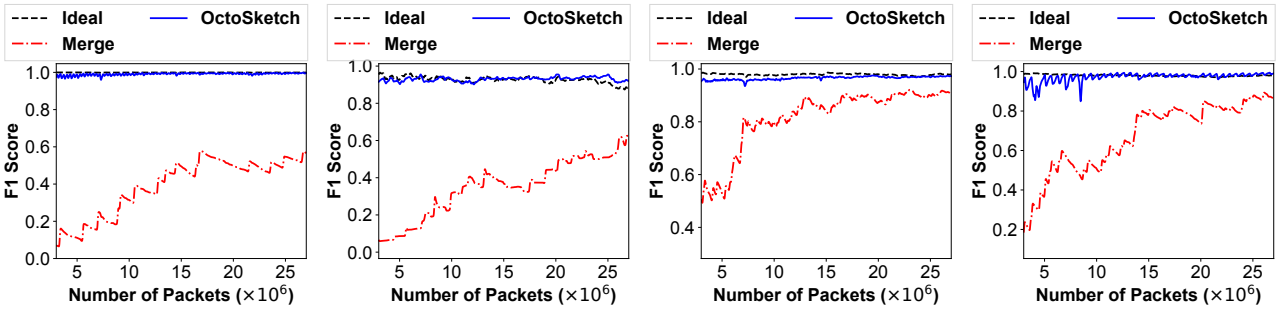
$$C'[i] = \begin{cases} 0, & C[i] < \tau - 1 \\ C[i], & C[i] \geqslant \tau - 1 \end{cases}$$

Therefore, if all $C[i] \geqslant \tau - 1$, we can guarantee that $\widehat{Z} = \widehat{Z}'$. If there is any $C[i] < \tau - 1$,

$$\widehat{Z} \leqslant \alpha_m m^2 \left(2^{-(\tau-2)}\right)^{-1} = 2\alpha_m m^2 2^{\tau-2}$$

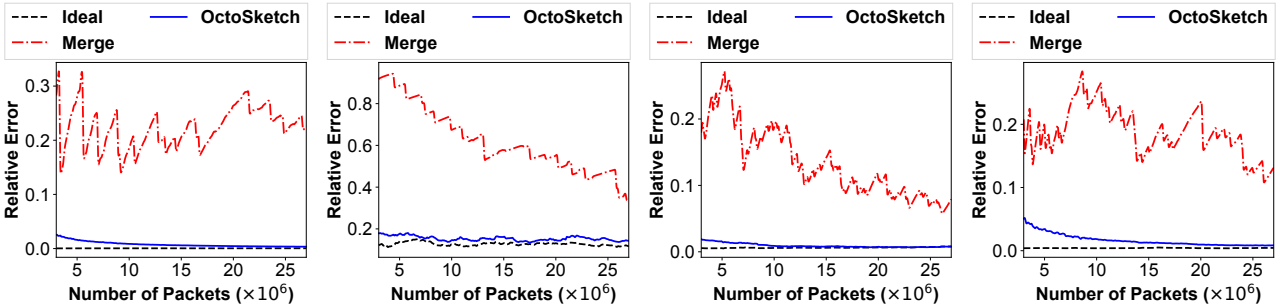
□

Different from the Count-Min sketch, not all packets can change the value of the accessed HyperLogLog. As the property of HyperLogLog, if there are N distinct flow keys mapped to the counter, the counter will only be updated $O(\log N)$ times. In other words, the number of updates is much smaller than the number of packets in the workloads. For example, there are about 30 million packets in 1 minute in CAIDA traces [37], while there are only around 1 million distinct flow keys, and only part of them can update the accessed counter. Therefore, in our experiments, the threshold $\tau = 2$ is often enough for the aggregator to process all updates.



(a) CocoSketch (b) Locher Sketch (c) Elastic Sketch (d) UnivMon

Figure 20: F1 scores on other sketches



(a) CocoSketch (b) Locher Sketch (c) Elastic Sketch (d) UnivMon

Figure 21: Relative errors on other sketches

C Extensions to other sketches

In this section, we show how we apply OctoSketch to the other eight sketches in detail.

Count sketch [11] and UnivMon [12]: There are also multiple arrays of counters in the Count sketch. Count sketch offers unbiased size estimation to each flow and thus each counter value in the Count sketch can be positive or negative. Therefore, as discussed in §4.4, OctoSketch uses the absolute value to check whether the counter is large enough. Moreover, UnivMon is composed of multiple Count sketches, and the insertion process is the same for each Count sketch.

LogLog [38], HyperLogLog [39], and Locher sketch [29]: There is only one array of small counters (*e.g.*, 4-bit counter) for LogLog and HyperLogLog. Note that the insertion logics of LogLog and HyperLogLog are the same, and they mainly differ in the query method. Therefore, we apply OctoSketch to both of them in the same way. The details of applying OctoSketch to HyperLogLog are discussed in §4.4. Locher sketch is comprised of multiple arrays of HyperLogLog [39] estimators, and the insertion process is the same for each HyperLogLog.

DDSketch [40]: There is one array of counters in the DDSketch. As the insertion logic of the DDSketch is similar to that of the Count-Min sketch, we apply OctoSketch to the DDSketch in the same way.

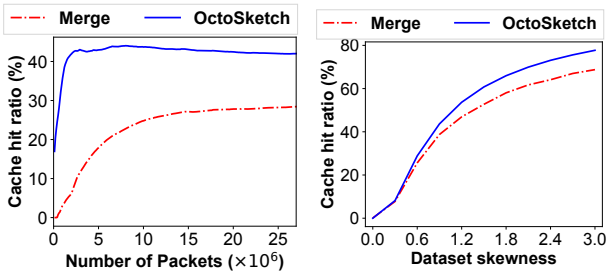
CocoSketch [16]: CocoSketch is composed of a number of (*e.g.*, 2) arrays of buckets, and each bucket stores a key and a counter. CocoSketch does not need an additional heap to record flow keys. Therefore, OctoSketch for CocoSketch also does not need the heap in the aggregator. As discussed in §4.4, OctoSketch will send the whole bucket (key and counter) to the aggregator, if the counter is large enough. In the aggregator, OctoSketch uses the same insertion logic as CocoSketch to update its own aggregated sketch.

Elastic sketch [13]: There are two parts in the Elastic sketch: a *heavy part* and a *light part*. The light part is a Count-Min sketch. The heavy part is an array of buckets to record heavy flow keys, where each bucket also contains a key and a counter similar to that of the CocoSketch. Elastic sketch also does not need an additional heap to record heavy flow keys. OctoSketch for Elastic sketch should keep both the heavy part and the light part in both workers and aggregator and does not need the heap in the aggregator. To apply OctoSketch, the light part's insertion logic is the same as that of the OctoSketch-optimized Count-Min sketch. For the heavy part, OctoSketch will send the flow key and the counter to the aggregator if the counter in the bucket is sufficiently large as shown in §4.4. In the aggregator, OctoSketch uses the same insertion logic as Elastic sketch to update its own aggregated sketch.

D Evaluation

D.1 Parameters

Sketch parameters: For Count-Min sketch and Locher sketch, we use 3 arrays of 2^{16} counters per array. For CocoSketch, we use 2 arrays as suggested in their paper, and there are 2^{16} buckets per array. For Elastic sketch, only one Count-Min sketch array is used (3×2^{16} counters) based on



(a) Over time **(b) Over skewness**

Figure 22: Cache hit ratio

the paper. For UnivMon, we use 6 Count sketches. We ensure that sketch-merge and OctoSketch use the same configuration (and memory) in the aggregator. As shown in §4, for OctoSketch, the sketch in each worker can use a smaller counter size without accuracy loss. In experiments, each counter in the worker of OctoSketch is 8-bit, while each counter in the worker of sketch-merge is 32-bit. Specifically, the Count-Min sketch for sketch-merge in workers needs 768KB memory, while it only needs 192KB memory for the workers of OctoSketch. For the resource allocation policy in OctoSketch, we set $\alpha = 0.25$. We set the cycle of sketch-merge based on the maximum frequency the aggregator can support.

D.2 Figures

Due to space limitations, we move the Figure 20 and 21 for §7.2 to the appendix.

D.3 Case Study: Key-Value Cache

In this section, we show the benefit of OctoSketch for a key-value cache. Similar to the [21], we run experiments on the simulator of DistCache [7]. Specifically, the traffic is distributed over multiple workers, and the aggregator aggregates the statistics over multiple workers. Once an object is detected as a hot object on the aggregator, it is cached. If an object is no longer recognized as a hot object, it will be offloaded from the cache. We calculate the cache hit ratio while using OctoSketch and sketch-merge for hot object detection.

Cache hit ratio over time (Figure 22a): OctoSketch can achieve fast detection and a high cache hit ratio. With less than 2M packets, the cache hit ratio of OctoSketch is higher than 40%. However, sketch-merge needs more than 10M packets to converge. It indicates that sketch-merge spends much time detecting new hot objects which may lead to many cache misses. In addition, due to the different detection rates on hot objects, the cache hit ratio of OctoSketch is still 13% higher than sketch-merge after 20M packets.

Cache hit ratio over skewness (Figure 22b): We also show the final cache hit ratio on datasets with different skewness. The cache hit rate is higher with larger skewness. The cache hit ratio of OctoSketch is around 9% higher than that of the sketch-merge when the skewness is larger than 1.5.

NR-Surface: NextG-ready μW-reconfigurable mmWave Metasurface

Minseok Kim[†] Namjo Ahn[†] Song Min Kim[*]

KAIST

Abstract

Metasurface has recently emerged as an economic solution to expand mmWave coverage. However, their pervasive deployment remains a challenge, mainly due to the difficulty in reaching the tight 260ns NR synchronization requirement and real-time wireless reconfiguration while maintaining multi-year battery life. This paper presents `NR-Surface`, the first real-time reconfigurable metasurface fully compliant with the NR standard, operating at 242.7 μW for a 2.1-year lifetime on an AA battery. `NR-Surface` incorporates (i) a new extremely low-power (14KHz sampling) reconfiguration interface, NarrowBand Packet Unit (NBPU), for synchronization and real-time reconfiguration, and (ii) a highly responsive and low-leakage metasurface designed for low-duty cycled operation, by carefully leveraging the structure and the periodicity of the NR beam management procedure in the NR standard. `NR-Surface` is prototyped and evaluated end-to-end with NR BS built on srsRAN to demonstrate diverse usage scenarios including multiple `NR-Surface` per BS, multiple UE per `NR-Surface`, and 3D beamforming. Around-the-corner UE evaluations showcase `NR-Surface` efficacy under different user mobility patterns (20.3dB gain) and dynamic blockage (22.2dB gain).

1 Introduction

With the rich bandwidth spanning multi-GHz, millimeter-wave (mmWave) is a key technology to breakthrough spectrum shortage in the sub-6GHz, towards enhanced mobile broadband (eMBB) under the exponential growth of mobile data in the 5G and 6G cellular services. However, the benefit of mmWave communication comes at the cost of the technical hurdle of substantial path loss – limiting the coverage, especially under complex indoor scenarios with significant obstructions, and hindering reliability for mobile users. To address this, various approaches have been introduced, including dense deployment of base stations (BS) [39, 60], relays [21, 58, 62], and distributed antennas [48, 57, 59, 72]. While simple and straightforward, installing a large body of costly transceivers is a hurdle to pervasive deployment and widespread of mmWave technology.

More recently, metasurface has emerged as an economic solution to expand mmWave coverage. Diverse designs were introduced including reconfigurable metasurfaces that adapt to varying channel and user mobility [22, 23], and low-cost 3D printed metasurface [64], where they are commonly designed to operate by passively reflecting the BS signal without RF chain for signal generation. This significantly simplifies the metasurfaces' RF structure compared to the full-fledged mmWave transceiver for lower manufacturing costs. However, pervasive deployment of the metasurfaces still remains a challenge, mainly due to the lack of a metasurface synchronization mechanism (to NR infrastructure BS) and real-time wireless reconfiguration interface, especially with multi-year battery lifetime to minimize deployment and maintenance costs.

This paper presents `NR-Surface`, the first real-time reconfigurable metasurface fully compliant with the NR standard operating at μW regime for a 2.1-year battery lifetime. For this `NR-Surface` tackles the challenges imposed by the NR standard at only 242.7μW power: (i) 260ns timing synchronization with the BS [10, 13] and (ii) in-time metasurface reconfiguration following the beam management procedure [11, 12]. `NR-Surface` co-designs extremely low-power (15KHz sampling rate) wireless reconfiguration interface and highly responsive and low-leakage varactor-based metasurface. The reconfiguration interface is overlayed on top of NB-IoT (on the NR guard band). This imposes no hardware change on the BS while minimizing disruption to the cellular traffic. `NR-Surface`'s metasurface achieves low-latency reconfiguration within the NR cyclic prefix (preventing communication error) with 10% low-duty cycling. `NR-Surface` performance is evaluated end-to-end with NR BS built on USRP X300 with srsRAN [70], where it offers 20.3dB and 22.2dB gain for around-the-corner UEs under mobility and dynamic blockage, respectively. Support for multiple `NR-Surface` per BS and multiple-UE per `NR-Surface` were also evaluated for diversified practical scenarios.

[†]Co-primary student authors.
[*]Corresponding author. (songmin@kaist.ac.kr)

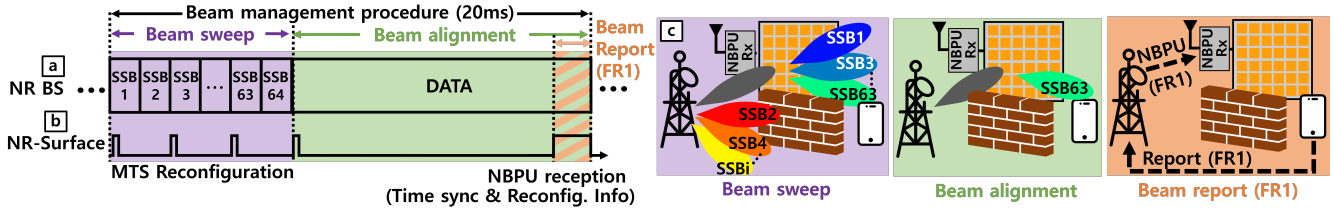


Figure 1: (a) NR beam management procedure and (b) the corresponding NR-Surface operation, offering NR-compliance. (c) Illustrative view of NR-Surface operation in the beam management procedure.

NR-Surface presents a reconfiguration interface of NarrowBand Packet Unit (NBPU), embedded on NR NB-IoT. NBPU is an asymmetric communication having NB-IoT with 180KHz bandwidth as the transmitter (BS side) and custom-built NBPU Rx with 15KHz bandwidth as the receiver (NR-Surface side). BS carefully selects the NB-IoT symbol to be interpreted as an OOK NBPU symbol at the NBPU Rx, via aliasing and harmonics from the envelope detector. Equivalent time sampling is applied to achieve 260ns synchronization accuracy meeting the NR requirement, by leveraging the insight that the arrival timing of the NBPU symbol is easily inferable from the periodic nature of the NR beam management procedure (repeated every 20ms). NBPU also delivers timing and phase for metasurface reconfiguration. The beam management procedure offers another energy-saving opportunity, where the metasurface reconfiguration takes merely 3.8% of the time. The metasurface simply holds the same configuration during the rest of the period. With varactor-based cells and GPIO-controlled control, NR-Surface metasurface is designed for efficient low-duty cycling with 3ps low-latency reconfiguration and a negligible 280nW power leakage when the configuration is maintained.

As the first NR-controlled real-time reconfigurable mmWave metasurface, NR-Surface co-designs a new low-duty-cycled metasurface and BS-controlled wireless interface to collaboratively achieve NR-compliance at μW power dissipation. We believe NR-Surface is a practical step towards the pervasive deployment of mmWave technology in our everyday lives. Our contributions are as follows:

- To the best of our knowledge, NR-Surface is the first real-time reconfigurable, NR-compliant, and low-power mmWave metasurface towards pervasive deployment.
- NR-Surface is immediately deployable and incurs a minimum cost, as it does not require any hardware change on the NR BS.
- We prototype NR-Surface on PCB (metasurface on Rogers RO4003C substrate) and perform diverse real-world evaluations under diversified usage settings, including multiple NR-Surface, mobile users, and under dynamic blockages.

2 Background

NR beam management procedure. NR mmWave communication follows the beam management procedure depicted in Figure 1(a) that incorporates the entire communication cycle – beam sweep, report, and alignment (i.e., data communication). During beam sweep in the first 5ms of the period, the BS transmits up to 64 Synchronization Signal Blocks (SSB) with different beam patterns, one of which is selected by each UE and fed back to the BS via beam report. Finally, the uplink and downlink data is exchanged during data beam alignment through the selected beam patterns. We note that the beam management procedure iterates exactly every 20ms and beam sweep, report, and alignment timings are known in advance. This offers the opportunity to integrate NR-Surface into the procedure in low-power, saving energy by being active only at known duration in the beam management procedure.

While mmWave in NR is known for its high data rate, the link frequently suffers from low reliability due to beam misalignments and blockages, especially in dynamic and mobile scenarios. To alleviate this, NR features inter-band Carrier Aggregation (CA) that integrates the high-rate FR2 (mmWave) data plane with a robust FR1 control plane [8], as in existing 5G platforms including Qualcomm [65–67] and MediaTek modems [53]. This is used to integrate NR-Surface control into the NR standard, where NR-Surface is controlled via the robust FR1 (sub-6GHz) in an energy-efficient manner.

NB-IoT. NB-IoT is an NR protocol for massive Machine Type Communication (mMTC) use cases, designed to serve IoT devices. To enable lightweight communication, NB-IoT uses the FR1 band and occupies only 180KHz bandwidth which is narrower than the minimum bandwidth of the underlying NR. This narrowband property allows NB-IoT to utilize the guardband of the FR1 band. Furthermore, NB-IoT only encodes data via 4QAM OFDM with 12 subcarriers, considering the limited performance of low-end devices. NR-Surface leverages the NB-IoT guardband mode to integrate the control plane of NR-Surface into the existing NR control plane.

3 Design Overview

Figure 2 illustrates NR-Surface prototype, comprising 900MHz antenna, reconfiguration interface, low-power MCU, and 24GHz metasurface. The 900MHz antenna captures the NR-Surface's control channel, named NarrowBand Packet Unit (NBPU), embedded in the NR FR1. NBPU Rx syn-

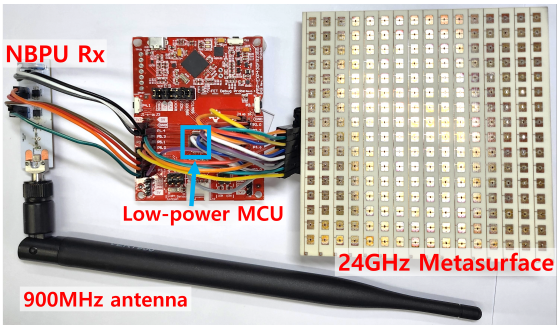


Figure 2: NR-Surface prototype implementation consisting of low-power reconfiguration interface with antenna, FR2 metasurface, and MCU.

chronizes with the NBPU and decodes the reconfiguration info at μW-regime. The low-power MCU reconfigures the 24GHz metasurface via GPIO to steer the received beam based on the decoded info. Figure 1(b) provides an overview of NR-Surface operation aligned with the NR beam management procedure, explaining the illustrative view of NR-Surface operation in Figure 1(c). During the beam sweep (Figure 1(c) left), the 24GHz metasurface is reconfigured at the allocated SSBs, sweeping the beams beyond the blockages. In the beam alignment (Figure 1(c) middle), the metasurface steers the beam toward UE's direction as reported in the previous procedure. NBPU Rx acquires reconfiguration info and timing for the next beam management procedure, based on the UE's beam report (Figure 1(c) right). During this whole procedure, our MCU enters idle mode to save energy when neither receiving NBPU nor reconfiguring the metasurface (Figure 1(b)).

NR-Surface design consists of two main parts enabling NR-compliant operation while at μW-consumption. The first part, discussed in Section 4, is the NBPU design for tight ns-synchronization with NR BS. To achieve this, the channel is co-designed with the μW-regime Rx to maximize the SNR and hence the synchronization accuracy, with signals selected to constructively add when passing through the low-end Rx. Section 5 presents a low-power metasurface design, tailored for real-time reconfigurations with short delays. NR-Surface implements duty-cycling to further save energy down to 242.7μW, by hardware components designed to consume only 6.4μW during the idle mode, maintaining configurations efficiently until known wake-up timings derived from the NR periodicity.

4 μW Synchronization with NR Base Station

Per the NR standard [10, 13], a tight time synchronization of under $\pm$260ns between BS and NR-Surface is required to prevent FR2 communication errors. Doing so lets NR-Surface reconfiguration fall within FR2 cyclic prefix (CP) of 585ns (i.e., 260$\times$2 with 65ns margin for multipath), thereby preventing error. For this NR-Surface establishes an FR1 control channel with the BS for a tight and reliable synchronization and reconfiguration. To minimize the disruption on the ongoing NR traffic and maintain compatibility, this channel is built on NB-IoT on the FR1 guardband and readily available on NR BS.

The control is a downlink-only (BS $\rightarrow$ NR-Surface) channel carrying the synchronization (i.e., preamble) and reconfiguration info, where BS uses NB-IoT for Tx. Meanwhile, NR-Surface aims at μW-regime and is unable to equip NB-IoT Rx that consumes 75mW [41] to limit the battery life to only two days on a single AA [4]. For this we present NarrowBand Packet Unit (NBPU), an asymmetric control channel overlayed on NB-IoT. In particular, by emulating with NB-IoT, NBPU is designed as a single-carrier 15KHz OOK. NR-Surface uniquely leverages aliasing from the bandwidth asymmetry between the 180KHz NBPU Tx (i.e., NB-IoT in the BS) and 15KHz NBPU Rx, to achieve 234ns synchronization (therefore meeting that NR requirement) and decoding with only 119.3μW for 2.1-year battery lifetime with a single AA.

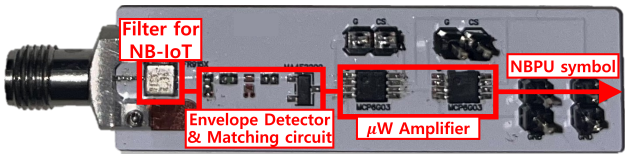


Figure 3: NBPU Rx to yield NBPU symbol from NB-IoT, comprising a filter for NR guardband operation, envelop detector for zero-power downconversion, and μW amplifiers to boost SNR.

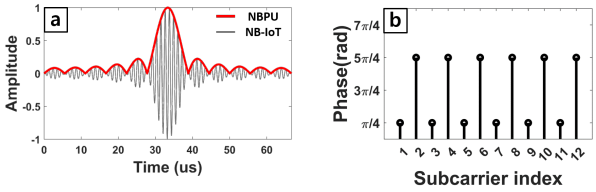


Figure 4: (a) The selected NBPU symbol with the maximum SNR and the corresponding (b) NB-IoT symbol (4QAM OFDM with 12 subcarriers).

4.1 NBPU Embedding in NB-IoT

NBPU Symbol. Here we discuss generating an NBPU symbol from an NB-IoT symbol – 12 subcarrier 4QAM OFDM with 15KHz spacing (from 15KHz to 180KHz). NBPU Rx in Figure 3 leverages the envelope detector for zero-power downconversion. The envelope comprises multiple harmonics, where the 1st order harmonics dominate over higher order harmonics ($\sim$37.3dB in our experiments). From this, the envelope is modeled entirely as the set of 1st-order harmonics – Each component has the frequency and phase from the difference between a pair of NB-IoT subcarriers, where the envelope detector yields the summation of all harmonic components from all subcarriers pairs.

Letting Δf denote 15KHz subcarrier spacing and ϕ represent one of 4QAM phases, $\{\frac{\pi}{4}, \frac{3\pi}{4}, \frac{5\pi}{4}, \frac{7\pi}{4}\}$, the envelope becomes $\sum_{k=1}^{11}\sum_{i=k+1}^{12}\cos(2\pi k\Delta f t + \phi_i - \phi_{i-k})$ where $k\Delta f$ and $(\phi_i - \phi_{i-k})$ denote the frequency and phase difference between ith and $i-k$th subcarriers. This indicates that the phase of the envelope is determined solely by the combinations of the NB-IoT subcarrier phases.

Figure 4 depicts the NBPU symbol and the corresponding NB-IoT symbol. Among all possible NB-IoT envelopes, we use the one with the maximum power as the NBPU symbol for the maximum SNR and thus, the best synchronization performance. This is achieved by selecting an NB-IoT symbol such that its 1st harmonics (i.e., subcarrier difference from the envelope detector) with equal frequencies align in phase; All 1st harmonics are constructively added in the frequency domain (and in the time domain from the Parseval's theorem) to yield the maximum power. To see why, without loss of generality, let all odd number subcarriers have phases $\pi/4$ and the other subcarriers have phases $5\pi/4$ (as shown in Figure 4(b)). This makes harmonics' phases $k\pi$ for $2\pi k\Delta f$ frequency component. These harmonics yield an envelope of $\sum_{k=1}^{11}(12-k)\cos(2\pi k\Delta f t + k\pi)$. Here, the amplitude of each frequency component, $k\Delta f$, is maximized to $(12-k)$, the sum of all harmonics. Figure 4(a) illustrates the NBPU symbol and its NB-IoT symbol whose 12 subcarrier phases are carefully chosen.[1] As shown in Figure 3, the NBPU symbol is extracted with the implemented NBPU Rx board, which consists of the filter for extracting NB-IoT band, the envelope detector for downconversion, and 34μW consuming amplifier.

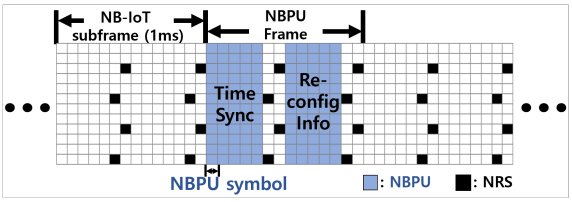


Figure 5: An NBPU Frame fits into one NB-IoT subframe. It has sync and reconfiguration info. (i.e., timing and phase).

NBPU Frame. As depicted in Figure 5 an NBPU frame is fit into a single NB-IoT subframe. An NBPU frame involves 5 NBPU symbols for synchronization and another 5 for the reconfiguration info, where two columns following the synchronization and reconfiguration info are unused as they are preoccupied with the mandatory Narrowband Reference Signal (NRS) for channel sounding. We note that one NBPU frame is inserted every 20 NB-IoT subframes to align with the periodic NR beam management procedure repeating every 20ms (Details in Section 5).

[1] Implementing the target NB-IoT subcarrier phases requires careful payload selection and reverse engineering of the channel coding. Details are in Appendix A.

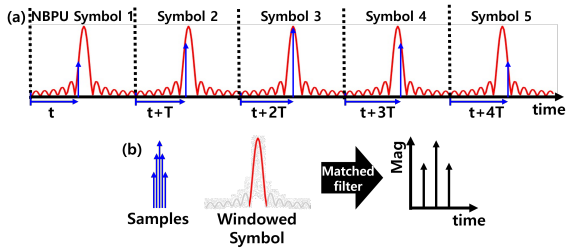


Figure 6: (a) Equivalent time sampling utilizes the accumulated sampling time offset T where t is common offset for all five symbols. (b) The windowed symbol makes the computationally complex matched-filtering light.

4.2 *ns*-synchronization

Meeting the NR requirement of $\pm$260ns synchronization accuracy requires the corresponding sampling rate of 3.84Msps [63]. However, running the ADC at this rate consumes mW-level power in low-power SAR ADCs [17, 73], which is unaffordable for `NR-Surface`. Instead, `NR-Surface` leverages repeated symbols from which offset (T in Figure 6) samples mimic the higher sampling rate, or equivalently, obtain fine-grained samples – a technique known as the Equivalent-Time Sampling [61]. Specifically, as in Figure 6(a), five NBPU symbols are repeated back-to-back with a single sample per symbol (14Ksps sampling rate); By enforcing an offset of $T = 260$ns between consecutive samples (using MCU clock), the five samples collectively mimic a 3.84Msps sampling with only 14Ksps ADC – cutting down the ADC power consumption by 274$\times$ compared to using 3.84Msps ADC.

`NR-Surface` runs matched filtering on the samples for optimal detection and synchronization [63]. In particular, as in Figure 6(b), matched filtering is performed only on a central 780ns window out of 66.7μs symbol. This (i) significantly reduces the computation and energy consumption, and (ii) maintains the synchronization performance as the signal power is concentrated in the central portion of the symbol. This requires first finding the center of the symbol so as to sample within the window – Upon system bootup, `NR-Surface` simply runs Equivalent-Time Sampling across the entire symbol from which the center is found by matched filtering. This takes 1.2s to complete (derivation in Appendix B) and is a one-time task as the NBPU Time Sync repeats exactly every 20ms following the beam management protocol in NR standard. `NR-Surface` achieves 234ns average synchronization error under 0dB SNR (>30m [15]), as evaluated in Section 7.5.

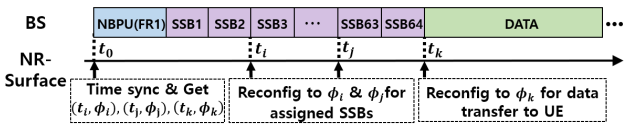


Figure 7: Interoperation overview to control FR2. BS transmits NBPU to synchronize t_0 and inform reconfiguration such as $(t_i, \phi_i), (t_j, \phi_j), (t_k, \phi_k)$. Then, `NR-Surface` reconfigures in the right time and beam.

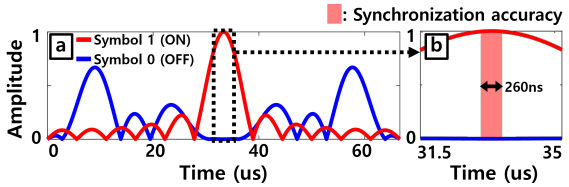


Figure 8: (a) Two NBPU symbols indicating 0 and 1. (b) They are OOK symbol pairs within the synchronization accuracy (red box).

5 μW Real-time Reconfiguration

NR-Surface needs to be tightly coupled with the NR beam management procedure for interoperability. As shown in Figure 7, upon receiving the NBPU frame, NR-Surface performs synchronization to obtain t_0 and extracts reconfiguration info, $(t_i, \phi_i), (t_j, \phi_j), (t_k, \phi_k)$. Including one NBPU symbol used for synchronization, We design two NBPU symbols (illustrated in Figure 8(a)), enabling low-power and robust communication of reconfiguration info. Specifically, two NBPU symbols are interpreted as 15KHz OOK symbol pairs by NBPU Rx, and our synchronization accuracy allows us to sample only the central part (red box in Figure 8(b)).[2] This asymmetric communication (180KHz $\rightarrow$ 15KHz) allows NBPU Rx to focus on only the constructively/destructively added portion of each NBPU symbol. This offers an additional 13dB SNR gain, which helps robust communication even with the passive envelope detector.

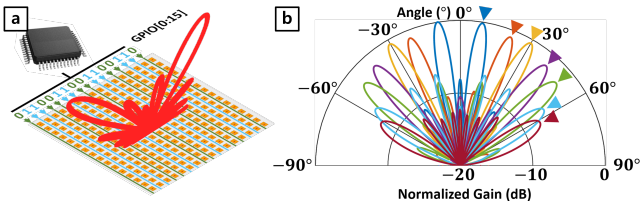


Figure 9: (a) GPIO-controlled metasurface structure and (b) example beam patterns. The main lobes are marked with triangles.

5.1 *ns*-reconfigurable Metasurface

Figure 9(a) illustrates the metasurface consisting of a 16x16 array of unit-cells, reconfigured in real-time at NR-synchronized timings derived from NBPU. In satisfying the NR synchronization requirement, the reconfiguration should finish within the margin between the NR requirement and NR-Surface accuracy. NR-Surface achieves this with the varactor-based unit-cell design and 1-bit digital output (GPIO) control, whose short response time allows NR-Surface to reconfigure beam patterns in less than 10ns while consuming only negligible energy ($\sim$ 12pJ in our implementation). From this, NR-Surface forms narrow beam patterns towards various directions to cover a wide area. Figure 9(b) plots example beam patterns that NR-Surface provides with 16 columns, whose half-power beam width is 6.3°. Moreover, $<$ 260ns NR synchronization requirement is still satisfied at a low SNR of $+$0dB, including both the synchronization error from NBPU and reconfiguration delay.

[2] Since NB-IoT can be turned on/off in the subframe scale, the NBPU symbol is needed to represent OFF state, whose NB-IoT subcarrier phase is $[\pi/4, \pi/4, 5\pi/4, 5\pi/4, 5\pi/4, 5\pi/4, \pi/4, \pi/4, \pi/4, \pi/4, 5\pi/4, 5\pi/4]$.

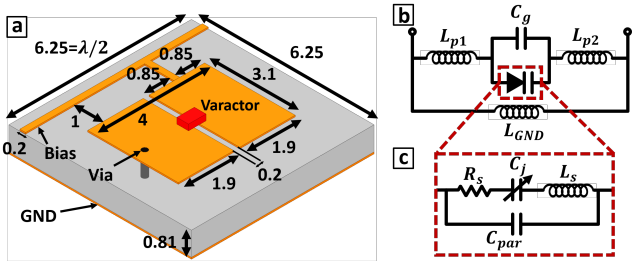


Figure 10: NR-Surface (a) unit-cell design and equivalent circuit models for (b) unit-cell and (c) varactor diode. Dimensions are in mm.

Figure 10(a) illustrates NR-Surface unit-cell design achieving 1-bit phase shift in 10ns while consuming only $\sim$ 12pJ. Unit-cell consists of two metal plates and a single varactor diode (Figure 10(b)), whose rectangular structure has been shown to be effective as a phase shifter [69, 75]. Specifically, the MAVR-011020-1411 varactor diode [5] is placed, consisting of a varying capacitance from $C_j = 0.22$pF to $C_j = 0.04$pF at 0V to 15V bias, respectively, with a series resistance of $R_s = 13\Omega$ (Figure 10(c)). The GPIO interface is directly connected to provide bias voltage across the diode, allowing fast reconfigurations within 10ns owing to short rise/fall times of the GPIO interface ($<$ 7ns in our implementation) and the small time constant $R_sC_j = 3$ps of the diode. Furthermore, the unit-cell only consumes $\sim$ 12pJ during reconfiguration, mainly due to small energy stored and dissipated at the varactor diode. However, the reduced voltage range of GPIO limits the minimum capacitance of the diode to $C_j = 0.08$pF at $3.3V$. Hence, the key design parameters such as length, width, and gap of metal plates are chosen, such that the varactor diode in parallel still shifts the phase by 180° over the reduced capacitance range.

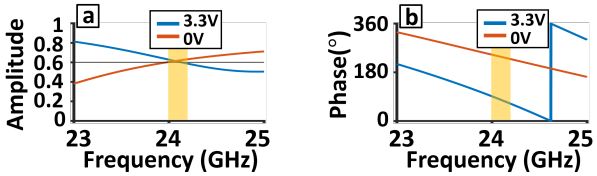


Figure 11: (a) Amplitude and (b) phase of the reflection coefficient when 0V and 3.3V are applied.

Figure 10(a) shows the fine-tuned dimensions aimed at 1-bit phase shift, from simulations conducted with Ansys HFSS [1] on the RO4003C substrate with $\varepsilon_r = 3.55$. The amplitude and phase of the reflection coefficient are plotted

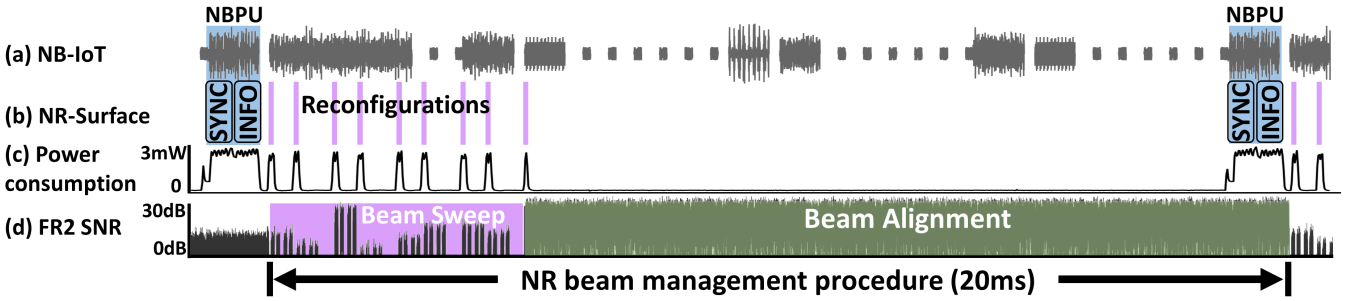


Figure 12: Overall operation of NR-Surface. (a) NB-IoT containing the NBPU frame, (b) NR-Surface duty-cycled operation synchronized with NR BS, (c) Measured power consumption, and (d) FR2 signal with improved SNR from aligned beams.

in Figure 11(a) and (b), respectively, showing NR-Surface unit-cell design achieves $\sim 180°$ phase shift with 0.6 amplitude across the entire 200MHz channel. We note the simulation and the implementation are carried out on 24GHz ISM band which shares similar RF properties with 28GHz NR mmWave, for the prototype evaluation. Figure 9(b) plots the example beam patterns simulated with HFSS, whose main lobe is steered towards different directions from 10° to 70° from the normal direction. The beam width in the example is 6.3°, however, this can be as narrow as 1.3° for 80-column metasurface, leveraging large availability of GPIO (up to 80 interfaces in typical low-power MCUs [54, 71, 74]). We note NR-Surface can also be extended to provide 3D beam patterns as evaluated in Section 7.9 with separately controlled unit-cells. To this end, NR-Surface reconfigures beam patterns towards a wide range of $\pm 70°$, where 256 unit-cells consume only a negligible amount of $\sim 3nJ$. While these energy-efficient reconfigurations facilitate low-power operation, the rest of the period should also be made low-power as well since ns-reconfigurations occupy a small portion of the period.

5.2 Duty-cycling NR-Surface

The final piece to μW operation is NR-synchronized low duty-cycling, by leveraging the two opportunities from the beam management procedure: (i) The synchronization and reconfiguration only take 10% and NR-Surface is kept idle with the beam pattern maintained for the rest of the time. Also, (ii) the idle/wakeup times are known a priori, as beam management procedure repeats every 20ms. This makes in-time mode switch possible even in NR-Surface's low-end hardware with slow wake up (e.g., 10μs for MCU). In particular, all hardware components (i.e., NBPU Rx, MCU, and metasurface) are carefully designed to collaboratively achieve idle power consumption of 6.4μW. Specifically, the varactor-based unit-cell maintains the phase at an extremely low 72nW leakage. In the meantime, the bias voltage via GPIO is kept stable under low power modes supported by typical MCUs at hundreds of nW range [54, 71, 74] and NBPU Rx is turned off. The MCU and NBPU Rx wakes up only at the time when the NBPU frame is expected.

5.3 Putting It All Together

With duty-cycling, μW synchronization, and reconfiguration NR-Surface achieves 242.7μW consumption for a 2.1-year lifetime on a single AA battery. Figure 12 plots the overall operation from the experiment results encompassing synchronization, reconfiguration, and beam alignment, as well as the corresponding power consumption. In Figure 12(a), (b) NR-Surface synchronizes with NBPU and decodes reconfiguration info. As shown in Figure 12(c) this step involves the heaviest computation to consume near half of the entire power (119.3μW). Metasurface reconfiguration and the corresponding short power bursts in Figure 12(b) and (c), respectively, take place in accordance with beam sweep and alignment in Figure 12(d). The short bursts sum up to 117μW. NR-Surface then keeps the beam towards the UE for the rest of the period. This is when the duty-cycling is in effect, consuming only 6.4μW to maintain the metasurface configuration. We note NR-Surface is reconfigured only at half-subframes for non-colocated scenarios (i.e., NB-IoT and FR2 BS in separate devices) which increments synchronization error by 260ns, leveraging longer CPs of 1106ns at half-subframe boundaries to avoid signal losses. Collectively, battery-powered operation facilitates pervasive deployment without wall plugs, in dynamic indoor scenarios as well as outdoors. NR-Surface real-time operations in dynamic environments are evaluated in Section 7.1.

5.4 Multi NR-Surface

Multiple NR-Surface for a single BS may be necessary under complex environments with a large number of blockages and blind spots. For this, a separate set of SSBs is allocated to each NR-Surface and the same NR-compliant operation is applied. The BS tracks which SSB (i.e., beam pattern) corresponds to which NR-Surface, and transmits multiple NBPUs for independent reconfiguration. In particular, they are embedded in different frames of a single NB-IoT spectrum, or extended to multiple NB-IoT spectrums in other NR guardbands for a large number of deployments. Beam patterns of up to 64 limit the number of NR-Surface that can cooperate with a single BS, however, exceeding beam patterns can be leveraged by other BS to benefit from dense NR-Surface deployment. We evaluated the real-time multiple operations in Section 7.3.

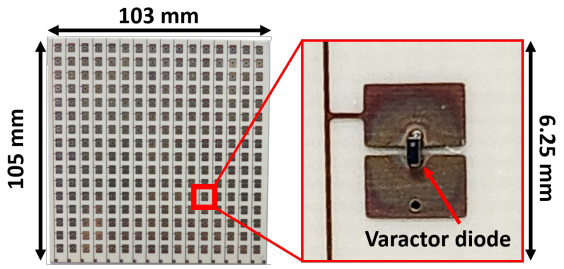


Figure 13: Metasurface and a unit-cell.

6 Prototype Implementation

We fabricate a prototype of NR-Surface and implement the real-time reconfiguration inside the NR deployment setup.

6.1 Hardware Configuration

Metasurface. Figure 13 shows the fabricated metasurface with the RO4003C substrate [6] of 32mil thickness, consisting of 16x16 unit-cells. Each column is connected to a single bias line and configured by a GPIO interface of the MCU. The unit-cell is magnified in the red box, showing the MAVR-011020-1411 [5] varactor mounted on the center.

NBPU Rx. NR-Surface employs a fully operational NBPU Rx aiming at a high SNR for reliable synchronization and decoding while minimizing the power consumption down to μW regime. Figure 3 shows the NBPU Rx consisting of a filter, envelope detector, and two amplifiers. NBPU Rx first filters out signals in passband using the TFR915X [56] Surface-Acoustic-Wave filter, whose center frequency is 915MHz with the 3dB-bandwidth 320KHz suitable for the NBPU's guard-band operation of the NB spectrum. We note the passband filter is particularly important for the NBPU, as the envelope detector down-converts all signals into the same baseband.

NBPU Rx employs a MA4E2200 [46] Schottky diode to yield the envelope of the filtered NBPU signal. The diode is matched to the antenna with an L-matching network consisting of two inductors, 22nH in parallel and 48nH in series, reducing the impedance mismatch loss to -4dB. To boost SNR, NBPU Rx employs 2-stage μW-consuming MCP6G03 [55] amplifiers to boost the signal strength by +68dB in total, which can be controlled from +68dB to +40dB to avoid distortions at large signal strengths. We note the amplifier yields +14dB SNR on the typical operating scenario [15], suitable for tight synchronization which we demonstrated in Section 7.5.

Power consumption. NR-Surface consume the most energy during the tight synchronization and decoding of NBPU, up to 2.4mW consisting of 655μW for amplifier, 294μW for ADC, 111μW for the synchronization timer, and 1.36mW for the synchronization and decoding logics. Leveraging duty-cycling, the power-hungry components are only active during a duration of NBPU which is much shorter than the entire 20ms period. This allows NR-Surface to significantly reduce power consumption to 119.3μW by 20.4x. Meanwhile, reconfigurations consume 1.67mW from the synchronization timer and reconfiguration logic which converts beam patterns into GPIO states. This is also reduced with duty-cycling, down to 117μW by 13.9x when 8 SSBs are assigned to NR-Surface beam patterns. During the low-power mode, NR-Surface consumes only 6.4μW to hold configurations, which consists of 5.9μW for the low-power mode standby current, 0.5μW for a timer for reconfiguration timings, and 0.072μW for power dissipated at unit-cells. NR-Surface power consumption during the 20ms period sums up to a total 242.7μW, which provides a 2.1-year lifetime from a single AA battery.

6.2 NR Evaluation Setup

We consider a general NR FR2 scenario. The BS is configured to transmit NB-IoT frames in FR1 containing the NBPU, and data packets in FR2 of large bandwidth.

FR2 transmissions. The BS employs USRP X300 series [29] software-defined radio platform, transmitting NR-compatible signals generated from the MATLAB 5G Toolbox [49]. The baseband signal is configured with 50MHz channel bandwidth, which consists of 64 SSBs for the beam sweeping followed by 15ms of data. The signal is continuously generated with a period of 20ms. For FR2 communication, the baseband signal is upconverted by ADMV1013 [19] to mmWave ISM band centered at 24.125GHz for the prototype evaluation, with +5dBm transmit power. The FR2 signal is transmitted by the 24GHz microstrip patch array antenna [27] with a fixed beam direction and +22dBi directivity.

The UE is also equipped with the same microstrip patch array antenna to receive the FR2 signal, which is downconverted by ADMV1014 [18]. The UE employs USRP X300 series to process the baseband frame of bandwidth 50MHz, running a custom GNURadio [28] block. The UE measures SNRs of SSBs in the beam sweeping, which is reported to the BS using the 905MHz spectrum within the ISM band. We note the RF characteristics of 900MHz ISM spectrum are also similar to NR low-band spectrum.

FR1 transmissions. The BS employs the same USRP running srsRAN [70], an open-source software implementation of an O-RAN compliant base station. The BS is configured to 900MHz ISM band centered at 915MHz for the prototype evaluation, with 180KHz bandwidth for the NB-IoT signal. We provide NB-IoT payload bits such that the generated NB-IoT signal contains the NBPU. On the other hand, the BS receives the beam report from the UE and extracts the best SSB index, whose center frequency is configured to 905MHz within the ISM band. This SSB index is immediately reflected to the next NBPU transmission. For the transmission, the BS is equipped with a VERT900 [30] antenna of +3dBi gain.

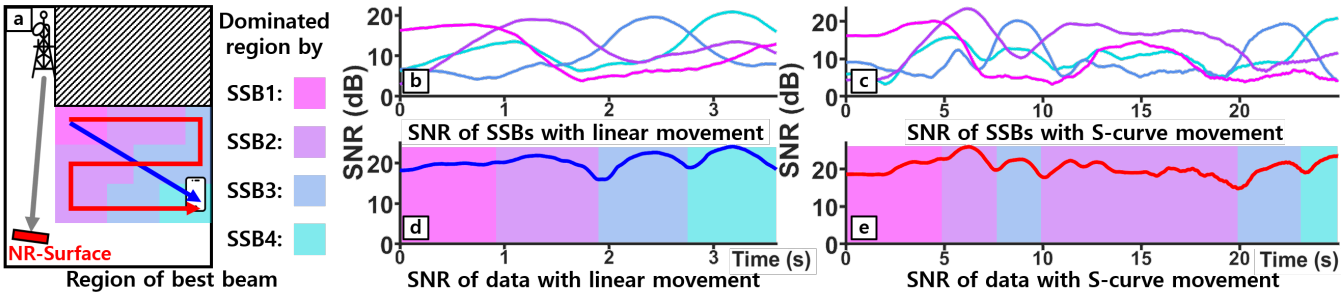


Figure 14: NR-Surface's real-time reconfiguration performance under mobile UE scenario. (a) In the indoor experiment setting, NR-Surface can increase coverage with multiple beams. Under various UE movements (e.g., linear & S-curve), the best beam is changed by the location (b,c) and NR-Surface reflects it in real time (d,e).

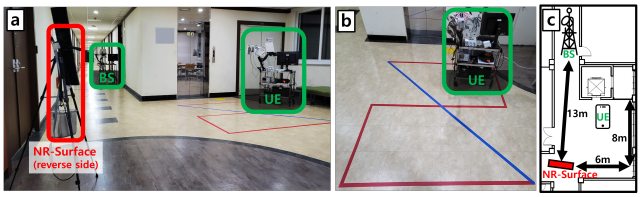


Figure 15: (a) Experiment setting for the mobile environment in the corridor and (b) the two tracks of mobile UE moved where the detailed scale is in (c).

7 Evaluation

In this section, we perform experiments under a dynamic environment to show that NR-Surface is ready to be deployed in practice. Specifically, we demonstrate the practicality of NR-Surface system by deploying NR-Surface in mobile and rapidly changing environments such as human movement scenarios. We measure various performances required for practicality such as SNR gain, latency, coverage, robustness of NBPU, and power consumption.

7.1 Dynamic Environment

NR-Surface is ready for practical deployment in NR scenario, achieving both the high average SNR gain (max 22.2dB) with fast beam tracking performance (< 20ms reconfiguration delay). In order to utilize NR-Surface in real life, NR-Surface must be able to quickly change the best beam direction in the dynamic environment that the mobile UE can experience. Since the environment around the handheld device is rapidly changed due to human movement, such as irregular movement or blockage by the human body, we evaluate the performance of NR-Surface reconfiguration in both scenarios.

SNR gain in mobile UE environment. We evaluate the SNR gain in real-time reconfiguration of NR-Surface with respect to the UE mobility. The evaluation setup is shown in Figure 15 (a) where the BS cannot support UE without NR-Surface since the LoS path is blocked by the corridor. NR-Surface is placed at the intersection to improve coverage of such UEs in the blind spot, which is tightly synchronized to the FR2 frame and sweeps beam configurations during the SS burst. The FR2 UE moves around the corridor following two paths in Figure 15 (b), namely "linear" in which UE moves at a typical human walking speed of 1m/s with constant distances to NR-Surface, and "S-curve" which UE pass through multiple points with different distance to NR-Surface. The detailed scale of our deployment is shown in Figure 15 (c).

As a reference, the region in Figure 14 (a) is colored depending on which beam configuration provides maximum SNR. Figure 14(b) and Figure 14(c) plot the dynamic SNR variation of 4 SSBs for the movement path "linear" and "S-curve", respectively, smoothed with window 10 over measurements taken once every 20ms. Specifically, UE with the linear movement passes through from SSB1 to SSB4 in sequence, and UE with the S-curve movement passes through in sequence as SSB1 → SSB2 → SSB3 → SSB2 → SSB3 → SSB4. The SNR variation plots confirm that the best SSB changes as the UE passes through regions.

Figure 14 (d) and Figure 14 (e) illustrate the SNR of data, measured every 20ms for the movement path "linear" and "S-shape" respectively. In response to the change of optimal beam configuration when the UE crosses the virtual boundary, the UE reports the best SSB to NR BS every 20ms as a single fixed beam would quickly deteriorate. The colored backgrounds in these figures indicate the timing boundary of the best SSB changes. This is applied to NR-Surface for real-time reconfiguration, such that the data SNR always follows the largest SNR among SSBs. NR-Surface achieves high average SNR gain of 18.3dB and 20.3dB in each UE movement, respectively, without significant SNR drop at crossing boundaries.

SNR gain in blockage changing environment. Considering the human body can suddenly block the established beam direction, NR-Surface should be able to rapidly update the best beam to cope with complex environments due to dynamically changing blockages. In the NR beam management procedure, the UE experiences serious SNR drop when the LoS path is blocked by the dynamic environment. NR-Surface deployment can recover the SNR drop by reconfiguring beam patterns in real time.

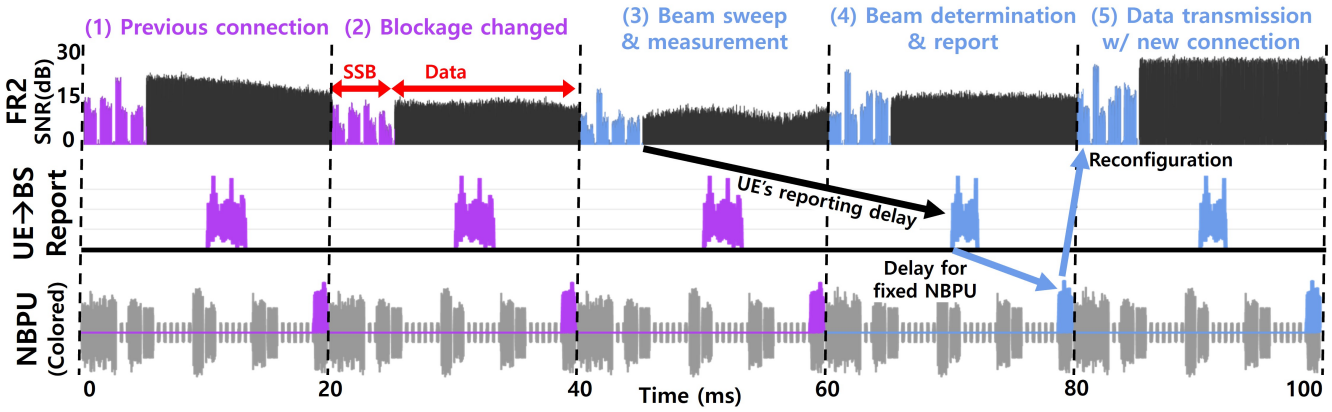


Figure 16: Real-world measurements of transmitted signals at the moment of blockage change. (1)-(3) Due to the dynamic blockage, the data SNR drops and the best SSB changes during the beam sweep. (4) The best SSB is determined and reported to the BS, which is also delivered to NR-Surface within 20ms. (5) NR-Surface is reconfigured to recover data SNR gain.

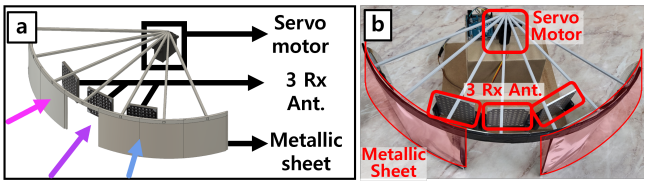


Figure 17: (a) outline of both blockage and UE design and (b) its implementation.

We setup the UE and blockage as in Figure 17(a)-(b), which has three microstrip patch antennas for Rx and movable metallic sheets via servo motor. Each three Rx antenna of UE is directed in different directions to check the best beam between multiple Rx beams, where each antenna is correlated to one Rx beam of digital beamforming. For an accurate evaluation, the movement of blockage is mechanically controlled to minimize errors from manual manipulation. By rotating the servo motor accurately, the metallic sheets always block two antennas and allow only one antenna can receive the signal from NR-Surface. Thus, after the blockage environment is changed, UE finds the best SSB during the beam sweep step among multiple Rx beamforming. Then, UE reports the best SSB index to BS, which reconfigures NR-Surface in real time.

Figure 18 (a) - (c) shows our evaluation setup, where the line-of-sight between BS and UE is blocked by the corridor. Each figure shows that only one antenna is allowed to receive a signal while the other is blocked by metallic sheets. We control the servo motor to change blockage quickly within 60ms, in (a) → (b) → (c) → (b) → (a) sequence where the servo motor is paused for 400ms in each state. Figure 18(d) plots the SNR gain of data every 20ms, where colored regions show the beam index in NBPU. SNR drops slightly between regions due to fast servo movements, and a stable SNR at paused durations. NR-Surface can establish a high SNR path with 22.2dB average SNR by fast recovery.

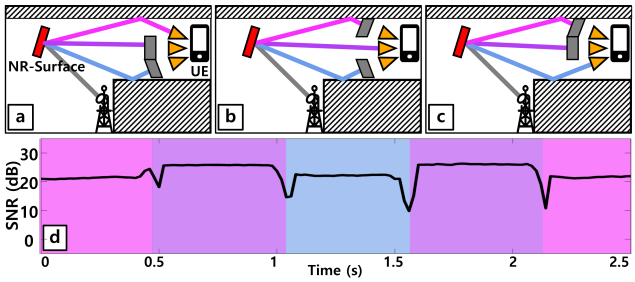


Figure 18: (a) - (c) show a scenario that only one antenna among 3 is not blocked. (d) shows the achieved SNR by the blockage environment change.

Time domain analysis for beam establishment. To get a closer look at what control sequence actually exists at the moment of blockage change, Figure 16 shows what the received signals by UE, BS, and NR-Surface look like by zooming in on the moment of blockage change from Figure 18 (c) to (b). The FR2 SNR plot demonstrates the strongest SSB is changed during (1)-(3) due to the new blockage setting. Thus, the SNR gain of the data at (2) is lower than the previous data at (1). Then, during the next beam sweep step at (3), UE newly determines the best SSB among 3 Rx antennas. After UE's beam determination, UE reports the best SSB index to BS via the FR1 channel in (4), where a few milliseconds (25 ms - 29 ms in our implementation) delay occurs due to UE's processing time and allocated UL time. Note that this UE-driven delay can be shorter when UE has a powerful signal processing unit or BS allocates resources more frequently to UE. After BS gets a report from UE, BS transmits NBPU to control, shown as the blue solid line in (1). The fixed time of NBPU causes a max 20 ms delay, which is short enough to recover the blocked path by human movement. Finally, NR-Surface is reconfigured to the new SSB index in (5).

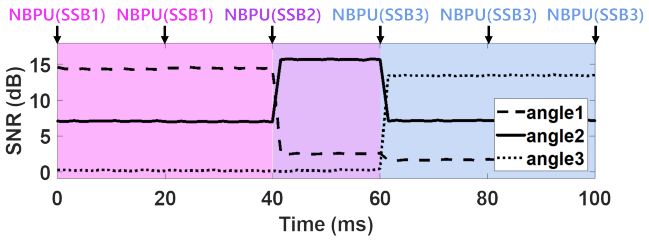


Figure 19: SNR of three beam patterns when NR-Surface is reconfigured in real-time. NR-Surface continuously receives NBPU and reconfigures beam patterns every 20ms.

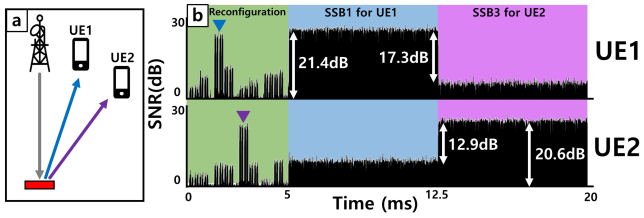


Figure 20: (a) NR-Surface operation scenario with multiple UEs (b) SNR fluctuation of the received FR2 per each UE.

Real-time configuration analysis. NR-Surface real-time reconfiguration can support highly dynamic environments where the best beam pattern changes every 20ms period of the NR beam management procedure. We employ the same metal-sheet blockage and a fast servo that changes blockage within 20ms, such that the UE reports different beam patterns in each NR beam management procedure. This is also delivered to NR-Surface via NBPU every 20ms. Figure 19 plots the SNR of three antennas placed at the main lobe of beam patterns, which demonstrates NR-Surface being continuously reconfigured every 20ms to support dynamic environments in real-time.

7.2 Multi UE Operation

To demonstrate multi-UE operation, we place two UEs at different locations such that each UE can select different SSBs for its data transmission (Figure 20(a)). UE1 and UE2 are scheduled to receive data from 5ms to 12.5ms, and 12.5ms to 20ms, respectively. NR-Surface receives the schedule info along with SSB indices. Figure 20(b) illustrates the SNR of the received FR2 signal at each UE. During the beam sweeping, UEs report different SSBs as their best SSB index, which are SSB1 and SSB3 for UE1 and UE2 (triangles in Figure 20), respectively. NR-Surface is reconfigured to SSB1 at 5ms for the UE1, such that the UE1 achieves +21.4dB SNR during its data transmission. Then NR-Surface is reconfigured at 12.5ms such that the UE2 achieves +20.6dB SNR from 12.5ms to 20ms. These SNRs are +17.3dB and +12.9dB better when compared to NR-Surface is reconfigured to only a single beam pattern of the other UE, for UE1 and UE2, respectively.

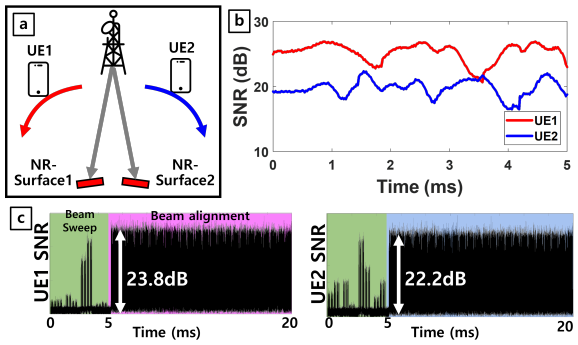


Figure 21: (a) Multiple NR-Surface scenario with multiple moving UEs (b) SNR fluctuation of the received FR2 per each UE (c) Received SNR of each UE, while both users moved during 5 sec.

7.3 Multi NR-Surface

NR-Surface supports deployment scenarios where one BS controls multiple NR-Surface. Figure 21(a) illustrates the evaluation setup where two NR-Surface are placed and two UEs move around NR-Surface. Note the NR BS reconfigures its beam towards multiple NR-Surface, where the evaluation setup comprises two fixed antennas at the BS for demonstration purposes. Two NBPUs are placed in the same NB-IoT guardband at different subframes, with different IDs in the reconfiguration info to control two NR-Surface separately. Figure 21(b) plots the data SNR of each UE, showing each NR-Surface is concurrently reconfigured in real-time to track each UE, respectively, with an average 22.5dB SNR gain. Figure 21(c) plots the example 20ms period during the movement, showing each UE experiences different SSB as their best beam pattern. This is separately reflected to two NR-Surface such that both UE achieve high data SNR. The evaluation demonstrates the capability of multiple NR-Surface deployment in complex environments, providing a large number of SSBs to cover multiple blind spots.

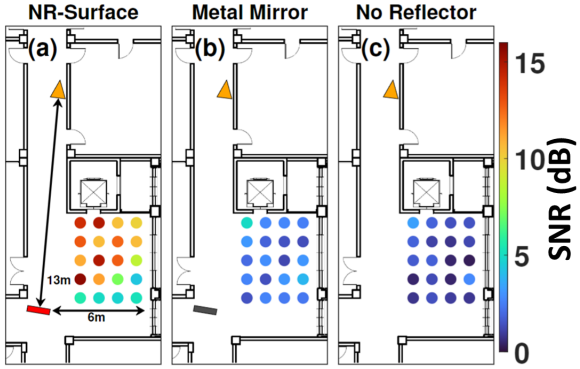


Figure 22: Comparison of SNR at various Rx locations (dots) between (a) NR-Surface placed at red rectangle (b) metal mirror of the same size with NR-Surface (c) no reflector placed. The Tx is placed at orange triangle steered towards NR-Surface.

7.4 Coverage Extension

We evaluate SNR improvements of NR-Surface in various UE locations, compared to a naive metal reflector and no reflector being placed. Figure 22(a) illustrates the measurement setup, a typical around-the-corner scenario with a strong blockage between Tx and Rx. Tx to reflector distance is fixed at 13m, and the transmit power is configured to 10dB less than previous experiments to emulate more practical attenuation. The SNR is measured at 4x5 UE locations on a rectangular grid of 1m spacing, with a maximum Rx to reflector distance is 7.3m. NR-Surface and metal mirror are placed at the intersection facing the Tx, where NR-Surface is reconfigured to steer the best beam possible while the metal mirror is fixed. Figure 22(a)-(c) show NR-Surface can achieve on average 7.6dB gain than the SNR with a metal mirror, and 8.8dB gain than the SNR without reflectors. Note the SNR improvement diminishes at UE locations whose steering angle exceeds 70° from the normal direction, showing NR-Surface can support wide range of steering angles up to 140° in practice.

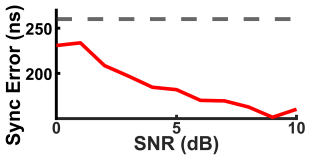


Figure 23: NBPU Synchronization error versus SNR.

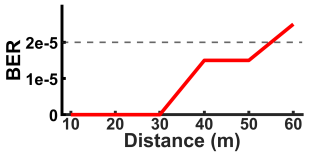


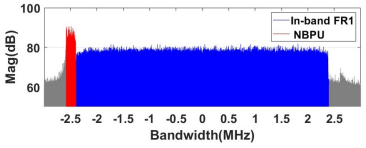
Figure 24: NR-Surface BER versus distance to the BS.

7.5 μW Synchronization Accuracy

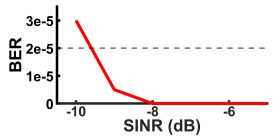
We evaluate NR-Surface synchronization accuracy which is necessary to ensure compliance with NR standards. Figure 23 plots the average synchronization error with respect to the SNR over 400 NBPUs, measured as the difference between the boundary of the NBPU transmission and the rising edge of the GPIO interface in NR-Surface. Specifically, we connect a high sampling rate ($\sim 1Gsps$) oscilloscope to both NR-Surface Rx and GPIO interface. NR-Surface successfully achieves 234ns synchronization error at low SNR of +0dB, which is much lower than SNR in typical operation scenarios [15]. From this, reconfigurations satisfy the NR requirements of $<$ 260ns including $<$ 10ns reconfiguration delay, even under the typical clock drift $<$ 20ns [3, 7] quartz crystal occurring every 20ms period. NR-Surface achieves this high accuracy in low-power by the equivalent time sampling, which mimics 3.84MHz ADC with a low-end 14KHz ADC.

7.6 NBPU BER Measurements

We evaluate NR-Surface demodulation performance and corresponding reliability of NR-Surface reconfigurations. Figure 24 plots the average BER of NR-Surface with respect to the distance to the NR BS, averaged over 200,000 symbols. Specifically, NR BS is placed in the indoor corridor and transmits NBPU at +16dBm, the maximum transmit power of NB-IoT in small cell BS. NR-Surface is equipped with the +11dBi Yagi antenna towards the BS, assuming antenna direction is known during the installation. NR-Surface achieves $< 2 \times 10^{-5}$ BER up to 50m distance, which corresponds to 99.99% satisfying high data-rate applications by NR KPIs [14], assuming 5 bits are used for reconfiguration info. NR-Surface achieves this in low-power by asymmetric NBPU OOK symbol design maximizing SNR while reducing bandwidth for low-power NR-Surface Rx.



(a) Coexistence of NBPU and in-band FR1.

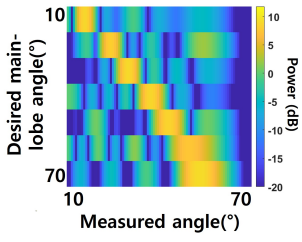


(b) BER versus SINR in NR guardband

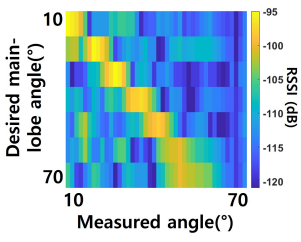
Figure 25: Evaluation setup and BER performance against NR interference.

7.7 Interference Robustness

We evaluate NR-Surface robustness against the NR interference. NBPU is implemented on top of NB-IoT, which is placed at guardbands of the NR spectrum. Figure 25(a) plots the fully utilized 5MHz NR spectrum consisting of heavy NR traffic (blue) and NBPU (red) in the guardband, placed with only a 70KHz gap between the NR traffic. Note the NB-IoT in the guardband is allowed to boost transmit power by +6dB compared to the NR spectrum. Figure 25(b) plots the BER with respect to the SINR. The SINR drops down to $-$8dB when the NR spectrum is fully utilized. However, NR-Surface successfully achieves $< 2 \times 10^{-5}$ BER even at maximum interference (up to $-$10dB). NR-Surface achieves this by a narrow passband filter of 320KHz bandwidth to suppress the interference by $-$18dB, such that NR-Surface still achieves low BER and high reliability in practical NR deployments.



(a) Simulated beam pattern



(b) Measured beam pattern

Figure 26: Beam patterns with desired main-lobe angles from 10° to 70°.

7.8 Beam Pattern Validation

Figure 26(a) and (b) illustrate the simulated and measured beam patterns generated from NR-Surface using 1-bit GPIO outputs, respectively. Power simulation and measurement results are arranged such that each row represents a single beam pattern desired to be steered towards $+10°$ to $+70°$. Note that the measurement resolution is 2°. The power measurement matrix has a similar appearance to the simulation, having a strong diagonal demonstrating each beam pattern is successfully steered to the desired angle. NR-Surface successfully covers up to 140° from $-70°$ to $+70°$ using symmetric patterns, where side-lobes of the beam pattern are suppressed.

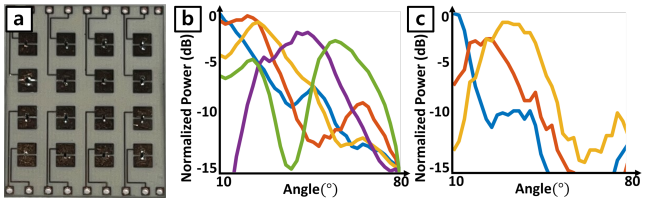


Figure 27: (a) Fabricated 3D metasurface and (b) measured 3D Beam pattern of NR-Surface steered towards different azimuth angles and (c) different elevation angles direction (smoothed with window=10°).

7.9 Extension to 3D Beamforming

To demonstrate the 3D beamforming capability, we fabricate a 4x4 metasurface whose unit-cells are separately controlled. Figure 27(a) shows the fabricated metasurface, whose bias lines are separately connected to different bias terminals. Figure 27(b),(c) illustrates measured beam patterns steered toward different azimuth angles and different elevation angles. We note the valid beam patterns with strong main-lobe are drawn in Figure 27(b),(c). NR-Surface successfully steers beam towards azimuth angles from $+10°$ to $+60°$, covering 120° from $-60°$ to $+60°$ using symmetric patterns. With the same 3D metasurface, NR-Surface can also steer beams toward elevation directions from $+10°$ to $+40°$, covering 90° from $-45°$ to $+45°$ using symmetric patterns. We note the slightly narrower elevation coverage can be compensated with multiple NR-Surface deployments. NR-Surface successfully steers the beam to different directions in 3D, which can be further extended to narrower and stronger beam patterns when implemented with larger metasurfaces.

8 Related Works

mmWave communication is considered to be the future of mobile communications, on which extensive research has been conducted [16,26,32,36,51]. However, mmWave communication suffers from high attenuation, such that antenna arrays for effective beamforming are necessary [20,31,34,35,37,47,50]. In efforts to increase the coverage and reliability of mmWave systems, approaches such as dense deployment of base stations (BS) [39,60], dual-band wireless network architecture with sub-6GHz [77], relays [21,58,62], and distributed antennas [48,57,59,72] has been proposed. While simple and straightforward, these solutions require installing a large body of costly transceivers, hindering pervasive applicability and widespread of mmWave technology.

Metasurface [79,80] has emerged as an economic solution for aiding mmWave beamforming for reliable communication with extended coverage. Specifically, NR-Surface cost (i.e., $\sim$ \$600) is much cheaper than COTS repeater solutions pricing $\sim$ \$3000 [2]. Furthermore, NR-Surface still has a similar cost to the existing mmWave metasurface with the same or smaller size [33,76,78] which cost \$500 – \$800. They support dynamic channel [22,23], MIMO [81], or even utilize low-cost 3D printing [64]. A recent metasurface, mmWall [24], integrates link layer discovery protocol of mmWave networks into the design. Nevertheless, real-time, NR-compliant MAC layer design still remains a challenge. In the given context, NR-Surface prioritizes seamless integration with NR standards, presenting distinct challenges such as achieving tight synchronization in low-power consumption.

Another line of research [25,40,43–45,52,68] focuses low power communication for sustainable deployment. Recent works [38,42] are capable of demodulating network symbols with mW level power consumption. NR-Surface uniquely implements the low power Rx for metasurface control, to operate with 247 μW end-to-end for practical deployment.

9 Conclusion

We present NR-Surface, an NR-compliant metasurface reconfigured in real-time within the NR standard. NR-Surface operates at μW-regime achieving a 2.1-year lifetime on an AA battery, consisting of the metasurface and NR-compliant Rx, which are duty-cycled leveraging the unique features of the NR standards. NR-Surface end-to-end system is implemented on srsRAN and evaluated on diverse scenarios from dynamic environments, multiple UEs to 3D beamforming achieving up to 22.2dB gain.

Acknowledgements

We thank our shepherd, Prof. Haitham Hassanieh, and the anonymous reviewers for their constructive comments. This research was supported in part by the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2024-2020-0-01787) supervised by the IITP(Institute of Information & Communications Technology Planning & Evaluation), the IITP grant funded by the Korea government (MSIT) (No.2021-000269, Development of sub-THz band wireless transmission and access core technology for 6G Tbps data rate, %33), and the National Research Foundation of Korea(NRF) grant funded by the Korea government (MSIT) (No. RS-2023-00213816). We would like to acknowledge the technical support from ANSYS Korea.

References

[1] Ansoft high frequency structure simulator. https://www.ansys.com/Products/Electronics/ANSYS-HFSS.

[2] Debunking the myth that millimeter wavelengths aren't viable for 5g. https://movandi.com/debunking-the-myth-that-millimeter-wavelengths-arent-viable-for-5g/.

[3] Ecx-31b smd crystal datasheet. https://www.mouser.kr/datasheet/2/122/ecx_31b-15952.pdf.

[4] Energizer max aa. https://energizer.com/wp-content/uploads/2023/12/e91_max_na.pdf.

[5] Mavr-011020-1411. https://www.macom.com/products/product-detail/MAVR-011020-1411.

[6] Rogers ro4003c. https://www.rogerscorp.com/advanced-electronics-solutions/ro4000-series-laminates/ro4003c-laminates.

[7] Quartz crystal technology overview. Technical report, Diodes, November 2018.

[8] 5G mmWave Deployment Best Practices. Technical report, GSMA, November 2022.

[9] 3GPP. Evolved Universal Terrestrial Radio Access (E-UTRA); Physical layer procedures. Technical Specification (TS) 36.213, 3rd Generation Partnership Project (3GPP), 03 2023. Version 17.5.0.

[10] 3GPP. NR; Base Station (BS) radio transmission and reception. Technical Specification (TS) 38.104, 3rd Generation Partnership Project (3GPP), 06 2023. Version 18.2.0.

[11] 3GPP. NR; Multiplexing and channel coding. Technical Specification (TS) 38.212, 3rd Generation Partnership Project (3GPP), 09 2023. Version 18.0.0.

[12] 3GPP. NR; Physical channels and modulation. Technical Specification (TS) 38.211, 3rd Generation Partnership Project (3GPP), 06 2023. Version 17.5.0.

[13] 3GPP. NR; Requirements for support of radio resource management. Technical Specification (TS) 38.133, 3rd Generation Partnership Project (3GPP), 06 2023. Version 18.2.0.

[14] 3GPP. Service requirements for the 5G system. Technical Specification (TS) 22.261, 3rd Generation Partnership Project (3GPP), 06 2023. Version 19.3.0.

[15] 3GPP. Study on channel model for frequencies from 0.5 to 100 GHz. Technical report (TR) 38.901, 3rd Generation Partnership Project (3GPP), 01 2024. Version 17.1.0.

[16] Talal Ahmad, Ramón Agüero, and Lakshminarayanan Subramanian. Learning congestion state for mmwave channels. In *Proceedings of the 3rd ACM Workshop on Millimeter-wave Networks and Sensing Systems*, pages 19–25, 2019.

[17] Analog Devices. A/D Converters (ADC). https://www.analog.com/en/product-category/analog-to-digital-converters.html.

[18] Analog Devices. *24 GHz to 44 GHz, Wideband, Microwave Downconverter, ADMV1014*, 2018. Available at https://www.analog.com/media/en/technical-documentation/data-sheets/ADMV1014.pdf, Rev. A.

[19] Analog Devices. *24 GHz to 44 GHz, Wideband, Microwave Upconverter, ADMV1013*, 2018. Available at https://www.analog.com/media/en/technical-documentation/data-sheets/ADMV1013.pdf, Rev. B.

[20] Kang Min Bae, Hankyeol Moon, Sung-Min Sohn, and Song Min Kim. Hawkeye: Hectometer-range subcentimeter localization for large-scale mmwave backscatter. In *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '23, 2023.

[21] Qi Bi. The proximity radio access network for 5g and 6g. *IEEE Communications Magazine*, 60(1):67–73, 2022.

[22] Kun Woo Cho, Yasaman Ghasempour, and Kyle Jamieson. Towards dual-band reconfigurable metasurfaces for satellite networking. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 17–23, 2022.

[23] Kun Woo Cho, Mohammad H Mazaheri, Jeremy Gummeson, Omid Abari, and Kyle Jamieson. mmwall: A reconfigurable metamaterial surface for mmwave networks. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, pages 119–125, 2021.

[24] Kun Woo Cho, Mohammad H Mazaheri, Jeremy Gummeson, Omid Abari, and Kyle Jamieson. {mmWall}: A steerable, transflective metamaterial surface for {NextG}{mmWave} networks. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1647–1665, 2023.

[25] A Eid, J Hester, and M Tentzeris. mm-wave tunnel diode-based rectifier for perpetual iot. In *2020 IEEE International Symposium on Antennas and Propagation and North American Radio Science Meeting*, pages 1495–1496. IEEE, 2020.

[26] Aline Eid, Jimmy Hester, and Manos M Tentzeris. A scalable high-gain and large-beamwidth mm-wave harvesting approach for 5g-powered iot. In *2019 IEEE MTT-S International Microwave Symposium (IMS)*, pages 1309–1312. IEEE, 2019.

[27] Eravant. *K-Band Microstrip Patch Array Antenna, 12° x 12°, SAM-2432432212-KF-L1*, 2022. Rev 1.4.

[28] Ettus Research. GNURadio. https://www.gnuradio.org/.

[29] Ettus Research. *USRP™ X300 and X310, X Series*. Available at https://www.ettus.com/wp-content/uploads/2019/01/X300_X310_Spec_Sheet.pdf.

[30] Ettus Research. *WSS016 Antenna With SMA(M), VERT900*. Available at https://kb.ettus.com/images/2/2b/ettus_research_vert900_datasheet.pdf.

[31] Yasaman Ghasempour, Muhammad K Haider, Carlos Cordeiro, Dimitrios Koutsonikolas, and Edward Knightly. Multi-stream beam-training for mmwave mimo networks. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 225–239, 2018.

[32] Moinak Ghoshal, Z Jonny Kong, Qiang Xu, Zixiao Lu, Shivang Aggarwal, Imran Khan, Yuanjie Li, Y Charlie Hu, and Dimitrios Koutsonikolas. An in-depth study of uplink performance of 5g mmwave networks. In *Proceedings of the ACM SIGCOMM Workshop*, pages 29–35, 2022.

[33] Jean-Baptiste Gros, Vladislav Popov, Mikhail A. Odit, Vladimir Lenets, and Geoffroy Lerosey. A reconfigurable intelligent surface at mmwave based on a binary phase tunable metasurface. *IEEE Open Journal of the Communications Society*, 2:1055–1064, 2021.

[34] Muhammad Kumail Haider, Yasaman Ghasempour, Dimitrios Koutsonikolas, and Edward W Knightly. Listeer: Mmwave beam acquisition and steering by tracking indicator leds on wireless aps. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 273–288, 2018.

[35] Haitham Hassanieh, Omid Abari, Michael Rodriguez, Mohammed Abdelghany, Dina Katabi, and Piotr Indyk. Fast millimeter wave beam alignment. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 432–445, 2018.

[36] Suraj Jog, Zikun Liu, Antonio Franques, Vimuth Fernando, Haitham Hassanieh, Sergi Abadal, and Josep Torrellas. Millimeter wave wireless network on chip using deep reinforcement learning. In *Proceedings of the SIGCOMM'20 Poster and Demo Sessions*, pages 70–72. 2020.

[37] Suraj Jog, Jiaming Wang, Junfeng Guan, Thomas Moon, Haitham Hassanieh, and Romit Roy Choudhury. {Many-to-Many} beam alignment in millimeter wave networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 783–800, 2019.

[38] Zerina Kapetanovic, Ali Saffari, Ranveer Chandra, and Joshua R Smith. Glaze: Overlaying occupied spectrum with downlink iot transmissions. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 3(4):1–21, 2019.

[39] Mirza Golam Kibria, Kien Nguyen, Gabriel Porto Villardi, Kentaro Ishizu, and Fumihide Kojima. Next generation new radio small cell enhancement: Architectural options, functionality and performance aspects. *IEEE Wireless Communications*, 25(4):120–128, 2018.

[40] Atsutse Kludze and Yasaman Ghasempour. {LeakyScatter}: A {Frequency-Agile} directional backscatter network above 100 {GHz}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 375–388, 2023.

[41] Lauridsen. An empirical nb-iot power consumption model for battery lifetime estimation. In *2018 IEEE 87th Vehicular Technology Conference (VTC Spring)*, pages 1–5, 2018.

[42] Haofan Lu, Mohammad Mazaheri, Reza Rezvani, and Omid Abari. A millimeter wave backscatter network for two-way communication and localization. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 49–61, 2023.

[43] Charles Lynch, Ajibayo O Adeyeye, Aline Eid, Jimmy GD Hester, and Manos M Tentzeris. 5g/mm-wave fully-passive dual rotman lens-based harmonic mmid for long range microlocalization over wide angular ranges. *IEEE Transactions on Microwave Theory and Techniques*, 71(1):330–338, 2023.

[44] Charles A Lynch, Ajibayo O Adeyeye, Aline Eid, JGD Hester, and Manos M Tentzeris. 5g/mm-wave next generation rfid systems for future iot applications. In *2021 IEEE International Conference on RFID Technology and Applications (RFID-TA)*, pages 77–80. IEEE, 2021.

[45] Charles A Lynch, Ajibayo O Adeyeye, Aline Eid, Jimmy Hester, and Manos M Tentzeris. Ultra-long-range dual rotman lenses-based harmonic mmld's for 5g/mm-wave lot applications. In *2022 IEEE/MTT-S International Microwave Symposium-IMS 2022*, pages 32–35. IEEE, 2022.

[46] MACOM. *Surface Mount Zero Bias Schottky Diodes, MA4E2200*. Available at https://cdn.macom.com/datasheets/MA4E2200%20Series.pdf, Rev. V10.

[47] Sohrab Madani, Suraj Jog, Jesus O Lacruz, Joerg Widmer, and Haitham Hassanieh. Practical null steering in millimeter wave networks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 903–921, 2021.

[48] Saad Mahboob and Rodney G Vaughan. Fiber-fed distributed antenna system in an fpga software defined radio for 5g demonstration. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67(2):280–284, 2019.

[49] MathWorks. Matlab 5g toolbox, 2023.

[50] Mohamamd Mazaheri, MohammadSadegh Vafaeinezhad, Mostafa Alizadeh, Mohammad Fakharzadeh, and Safieddin Safavi-Naeini. A low-cost and low-power single bit phased array at ka-band for 5g applications. In *2019 49th European Microwave Conference (EuMC)*, pages 622–625. IEEE, 2019.

[51] Mohammad H Mazaheri, Soroush Ameli, Ali Abedi, and Omid Abari. A millimeter wave network for billions of things. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 174–186. 2019.

[52] Mohammad Hossein Mazaheri, Alex Chen, and Omid Abari. Mmtag: A millimeter wave backscatter network. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 463–474, 2021.

[53] MediaTek. *MediaTek M80 5G Modem*, 2021.

[54] Microchip. 8-bit PCI and AVR Microcontrollers. https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors/8-bit-mcus.

[55] Microchip Technology. *110 μA Selectable Gain Amplifier, MCP6G03*, 2006. Available at https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/DataSheets/22004b.pdf.

[56] Microchip Technology. *Signal Conditioning 915.0 MHz SAW resonator, TFR915X*, 2017. Available at https://www.mouser.kr/datasheet/2/268/TFR915X-1665563.pdf, V1.2.

[57] Arno Moerman, Joris Van Kerrebrouck, Olivier Caytan, Igor Lima de Paula, Laurens Bogaert, Guy Torfs, Piet Demeester, Hendrik Rogier, and Sam Lemey. Beyond 5g without obstacles: mmwave-over-fiber distributed antenna systems. *IEEE Communications Magazine*, 60(1):27–33, 2022.

[58] Syed Hassan Raza Naqvi and Pin-Han Ho. Achieving 5g nr mmwave indoor coverage under integrated access backhaul. *IEEE Systems Journal*, 15(4):5429–5439, 2021.

[59] Syed Hassan Raza Naqvi, Pin Han Ho, and Limei Peng. 5g nr mmwave indoor coverage with massive antenna system. *Journal of Communications and Networks*, 23(1):1–11, 2021.

[60] Kien Nguyen, Mirza Golam Kibria, Jing Hui, Kentaro Ishizu, and Fumihide Kojima. Minimum latency and optimal traffic partition in 5g small cell networks. In *2018 IEEE 87th Vehicular Technology Conference (VTC Spring)*, pages 1–5, 2018.

[61] NI. *NI-SCOPE*, 2023.

[62] Konstantinos Ntontin and Christos Verikoukis. Relay-aided outdoor-to-indoor communication in millimeter-wave cellular networks. *IEEE Systems Journal*, 14(2):2473–2484, 2019.

[63] J.G. Proakis and M. Salehi. *Fundamentals of Communication Systems*. Pearson Education, 2013.

[64] Kun Qian, Lulu Yao, Xinyu Zhang, and Tse Nga Ng. Millimirror: 3d printed reflecting surface for millimeter-wave coverage expansion. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, MobiCom '22, page 15–28, New York, NY, USA, 2022. Association for Computing Machinery.

[65] Qualcomm. *Snapdragon X65 5G Modem-RF System Product Brief*, 2021.

[66] Qualcomm. *Snapdragon X70 5G Modem-RF System Product Brief*, 2022.

[67] Qualcomm. *Snapdragon X75 5G Modem-RF System Product Brief*, 2023.

[68] Elahe Soltanaghaei, Adwait Dongare, Akarsh Prabhakara, Swarun Kumar, Anthony Rowe, and Kamin Whitehouse. Tagfi: Locating ultra-low power wifi tags using unmodified wifi infrastructure. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 5(1):1–29, 2021.

[69] Xinyun Song, Xudong Bai, and Weiren Zhu. Reconfigurable metasurface for nearly full-range and continuous modulation of reflection, transmission, and absorption. *ACS Applied Electronic Materials*, 4(3):1225–1231, 2022.

[70] srsRAN PROJECT. srsRAN. https://www.srslte.com/.

[71] STMicroelectronics. STM8 8-bit MCUs. https://www.st.com/en/microcontrollers-microprocessors/stm8-8-bit-mcus.html.

[72] Minkyu Sung, Joonyoung Kim, Eon-Sang Kim, Seung-Hyun Cho, Young-Jun Won, Byoung-Chul Lim, Sung-Yeop Pyun, Joon Ki Lee, and Jong Hyun Lee. 5g trial services demonstration: Ifof-based distributed antenna system in 28 ghz millimeter-wave supporting gigabit mobile services. *Journal of Lightwave Technology*, 37(14):3592–3601, 2019.

[73] Texas Instruments. Analog-to-Digital Converters. https://www.ti.com/ko-kr/data-converters/adc-circuit/products.html.

[74] TEXAS INSTRUMENTS. MSP430 microcontrollers. https://www.ti.com/microcontrollers-mcus-processors/msp430-microcontrollers/overview.html.

[75] Shuncheng Tian, Haixia Liu, and Long Li. Design of 1-bit digital reconfigurable reflective metasurface for beam-scanning. *Applied Sciences*, 7(9):882, 2017.

[76] Alexander Wolff, Lars Franke, Steffen Klingel, Janis Krieger, Lukas Mueller, Ralf Stemler, and Marco Rahm. Continuous beam steering with a varactor-based reconfigurable intelligent surface in the Ka-band at 31GHz. *Journal of Applied Physics*, 134(11):114502, 09 2023.

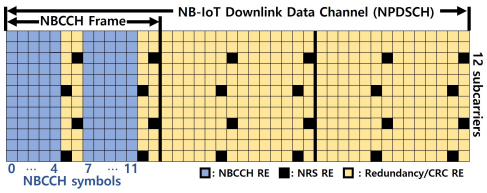
[77] Li Yan, Xuming Fang, Yuguang Fang, Li Hao, Qing Xue, and Chenren Xu. Kf-lstm based beam tracking for uav-assisted mmwave hsr wireless networks. *IEEE Transactions on Vehicular Technology*, 71(10):10796–10807, 2022.

[78] Fei Yang, Fan Xu, Chenxi Liu, Xinyu Yang, Ziqiang Wang, Junwei Wu, and Xiaojian Fu. Two-dimensional beam steering based on compact programmable coding metasurface. *Applied Sciences*, 12(22), 2022.

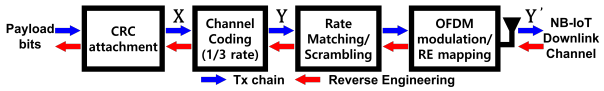
[79] Haobo Zhang, Hongliang Zhang, Boya Di, Kaigui Bian, Zhu Han, Chenren Xu, Daqing Zhang, and Lingyang Song. Rss fingerprinting based multi-user outdoor localization using reconfigurable intelligent surfaces. In *2021 15th International Symposium on Medical Information and Communication Technology (ISMICT)*, pages 167–172. IEEE, 2021.

[80] Yongzhao Zhang, Yezhou Wang, Lanqing Yang, Mei Wang, Yi-Chao Chen, Lili Qiu, Yihong Liu, Guangtao Xue, and Jiadi Yu. Acoustic sensing and communication using metasurface. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1359–1374, 2023.

[81] Yutong Zhang, Boya Di, Hongliang Zhang, Jinlong Lin, Chenren Xu, Daqing Zhang, Yonghui Li, and Lingyang Song. Beyond cell-free mimo: Energy efficient reconfigurable intelligent surface aided cell-free mimo communications. *IEEE Transactions on Cognitive Communications and Networking*, 7(2):412–426, 2021.



(a) NB-IoT downlink frame structure in guardband mode



(b) NB-IoT PHY block diagram

Figure 28: Emulating NBPU with NB-IoT

Appendix A. Emulating NBPU

As shown in Figure 28, NBPU is embedded in the NB-IoT downlink data channel, which constitutes an NB-IoT downlink frame. NBPU should be embedded avoiding Narrowband Reference Signal (NRS), subcarriers with fixed phases, in the 5, 6, 12, and 13th symbols of one subframe. By two NRSs in one symbol, only $\binom{10}{2} = 45$ out of a total $\binom{12}{2} = 66$ harmonics can be controlled, reducing the number of controllable by 68%. NBPU frame with 10 symbols is embedded in the blue box of Figure 28 while the Rx will ignore the 5 and 6th symbols.

The payload bits that can embed NBPU should be found through reverse engineering of the PHY block diagram in Figure 28. In this figure, coded bits $\mathbf{Y}$ and modulated bits $\mathbf{Y}'$ are one-to-one mapping, so its reverse engineering is straightforward. On the other hand, in the 1/3 code rate channel coding, payload bits $\mathbf{X}$ cannot generate arbitrary $\mathbf{Y}$ since the length of $\mathbf{X}$ is shorter than $\mathbf{Y}$ which induces a reduced degree of freedom. To solve this problem, the 240 bits of NBPU can be completely reverse-engineered by adding redundancy bits to $\mathbf{Y}$ such that $\mathbf{X}$ becomes greater than 240 bits. Since the length of payload bits is pre-defined in TS 36.213 [9], we utilize a 256-bit payload, the shortest length among candidates, whose result of PHY block diagram is allocated in 3 subframes.

Appendix B. Sweeping NBPU Symbol

For equivalent time sampling, `NR-Surface` samples with 260ns shifted sampling offset every NB-IoT symbol where symbol length is $66.67\mu s \simeq 256 \times 260$ns. Since the number of symbols transmitted in 20ms is 280 symbols, the accumulated sampling offset after 20ms is $(280 - 256 - 1) \times 260$ns where -1 is for the overflowed sampling offset than symbol length. Thus, after NBPU is transmitted N times (i.e., $N \times 20$ms), `NR-Surface` samples NBPU with $(13 \times N \bmod 256) \times 260$ns sampling offset. Then, the minimum N is 256 to sweep NBPU with 260ns fully. Considering that `NR-Surface` transmits 5 NBPU symbols every 20ms, we can achieve a fully swept symbol 5 times faster (i.e., N becomes 60). Then, the corresponding time is 1.2s ($= 60 \times 20$ms).

Cyclops: A Nanomaterial-based, Battery-Free Intraocular Pressure (IOP) Monitoring System inside Contact Lens

Liyao Li[12*], Bozhao Shang[24], Yun Wu[24], Jie Xiong[3], Xiaojiang Chen[24], Yaxiong Xie[1,†]
1. *University at Buffalo SUNY,* 2. *Northwest University (China)*
3. *University of Massachusetts Amherst and Microsoft Research Asia,*
4.*Shaanxi International Joint Research Centre for the Battery-Free Internet of Things*

Abstract

Intraocular pressure (IOP), commonly known as eye pressure, is a critical physiological parameter related to health. Contact lens-based IOP sensing has garnered significant attention in research. Existing research has been focusing on developing the sensor itself, so the techniques used to read sensing data only support a reading range of several centimeters, becoming the main obstacle for real-world deployment. This paper presents Cyclops, the first battery-free IOP sensing system integrated into a contact lens, which overcomes the proximity constraints of traditional reading methods. Cyclops features a three-layer antenna comprising two metallic layers and a nanomaterial-based sensing layer in between. This innovative antenna serves a dual purpose, functioning as both a pressure sensor and a communication antenna simultaneously. The antenna is connected to an RFID chip, which utilizes a low-power self-tuning circuit to achieve high-precision pressure sensing, akin to a 9-bit ADC. Extensive experimental results demonstrate that Cyclops supports communication at meter-level distances, and its IOP measurement accuracy surpasses that of commercial portable IOP measurement devices.

1 Introduction

Intraocular pressure (IOP) is a critical human vital sign of human health, and thus monitoring IOP is crucial for assessing and managing various diseases, especially glaucoma, the second leading cause of irreversible blindness worldwide [1–4]. Elevated IOP causes permanent damage to the optic nerve inside eyes, ultimately resulting in glaucoma and gradual vision loss. Glaucoma has no cure yet but if we detect Glaucoma in its early stage, we could prevent it from causing permanent vision by lowering the IOP via medicine or surgery. We emphasize that early-stage Glaucoma has no obvious symptoms, making accurate IOP measurement the only feasible solution for timely diagnosis of early-stage Glaucoma.

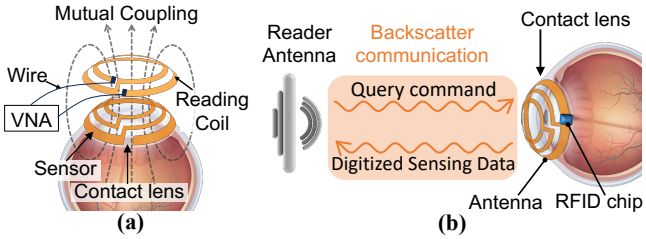


Figure 1. Reading data from contact lens via mutual coupling only supports centimeter level reading range (**a**), while backscatter based communication extends it to meter level(**b**).

Furthermore, continuous and long-term tracking of IOP is necessary for accurate disease diagnosis and effective disease management, since IOP naturally fluctuates in response to various physiological and psychological factors [5, 6], such as sleeping or dramatic emotional fluctuations. Accumulating long-term monitoring data helps eliminate the influence of unrelated factors, enabling accurate diagnosis. Additionally, timely tracking of IOP variations allows for precise and timely interventions, such as initiating medication immediately upon detecting elevated IOP to control the progress of Glaucoma.

Due to the significant impact of IOP on human health, accurate measurement has attracted a large amount of attention, particularly in biomedical engineering. Contact lens-based wearable techniques have emerged as the leading solution because of their lightweight nature, potential for mobility, and high accuracy resulting from direct contact with the eye balls. Two types of pressure sensors that fit in the contact lens have been the main focus of prior works: the piezoresistive sensors [4, 7–15], and the inductively coupled structure based sensors [16–24]. These sensors incorporate a ring structure made of conductive material that conforms to the shape of the contact lens, as depicted in Figure 1. In piezoresistive sensors, the ring structure's resistance is carefully designed to be pressure sensitive, while in coupled structure based sensors, the reactance is pressure sensitive. As a result, pressure can be inferred by measuring the impedance of both sensor types.

Optimizing the sensor to make it more sensitive, biosafe,

*Work conducted during academic visiting at the University at Buffalo.
†Corresponding author

transparent, and easy to manufacture has been the main focus, so in most current solutions, the contact lens only contains pressure sensors. To read the sensing data from the sensors embedded inside the contact lens, a reading coil connected to a vector network analyzer (VNA) is positioned atop the sensors to measure the impedance, just as shown in Figure 1(a). However, this method only supports a very limited data reading range of a few centimeters. Our survey of 20 related works reveals that the maximum reading range reaches only **five centimeters**. Putting a bulky reader in close proximity to eye balls poses a significant hindrance to practicality and emerges as the primary obstacle for real-world deployment.

This paper presents, **Cyclops**, a battery-free contact lens based IOP sensing system that supports long-range data reading, eliminating the requirement for a bulky data reader to be in close proximity to the user. We achieve long-range communication by integrating a backscatter communication system with the pressure sensor, all within the small contact lens. A backscatter communication system requires a sizable metallic antenna. The current pressure sensor features a conductive ring structure. Embedding both the antenna and the ring structure in the contact lens with limited size and a thickness below 200 μm results in mutual coupling and thus interference, impacting both communication and sensing performance.

Instead of directly integrating a backscatter antenna and an existing pressure sensor together, we introduce a novel approach: a dual-purpose antenna that serves as a pressure sensor and a communication antenna concurrently. Our design involves a three-layer sandwich-like antenna structure comprising two *metallic antenna layers* and a central *sensing layer*. The sensing capability arises from the nanomaterial utilized in the middle layer, which exhibits significant variations in intrinsic properties such as *conductivity* and *permittivity* in response to pressure changes. Simultaneously, the sensing layer, integrated into the antenna assembly, influences the overall impedance of the complete three-layer structure, rendering the entire system sensitive to applied pressure and effectively functioning as a pressure sensor. Additionally, through meticulous optimization of the two metallic layers' structure, we can align the antenna's impedance with that of the backscatter chip, enabling long-range communication.

To design a sensing layer that works robustly on a battery-free, purely energy-harvesting powered system, we have the following design goals:

- *Ultra-Low Power Consumption.* Our sensing layer's power consumption must be maintained at an absolute minimum to align harmoniously with the overall power profile of the integrated system.
- *Superior Sensitivity.* The ability of the sensing layer to detect subtle changes in IOP is utmost significance, necessitating an exceptionally high sensitivity level.
- *Thickness, Curvature and Transparency.* The sensing layer's dimensions, including thickness and curvature, must seamlessly match the contact lens. The sensing layer should be transparent to minimize interference to human view.
- *Biocompatibility and Biosafety.* The sensor must exhibit a rigorous level of biocompatibility.

To meet the design goals, we leverage a nanocomposite to fabricate our sensing layer. The nanocomposite consists of *hollow carbon spheres* (HCS) nanoparticles blended in the *polydimethylsiloxane* (PDMS)—the most widely used material for fabricating contact lens. Specifically, we fabricate a flexible HCS-PDMS sensing film with a thickness of tens of micrometers. This nanocomposite's flexibility enables it to conform to the curvature of the contact lens. Moreover, all the materials utilized, including HCS and PDMS, have already demonstrated their biocompatibility.

The two metallic antenna layers create an electric field across the HCS-PDMS sensing film, inducing the quantum tunneling effect between the HCS nanoparticles, resulting in a tunneling current. Pressure affects the film's thickness, changing the nanometer-level spacing between HCS, which, in turn, influences the strength of the tunneling current. The variations in tunneling current effectively change the conductivity of the sensing film, which consequently affects the permittivity of the nanomaterial [25]. We further enhance our HCS-PDMS sensing film's sensitivity to pressure through three optimizations: replacing normal UHCS with Urchin-like HCS (UHCS) for enhanced tunneling current, fine-tuning UHCS density to strike a balance between sensitivity and transparency, and adding microstructures to the surface of UHCS-PDMS sensing film to further boost the sensitivity.

Our dual-purpose three-layer antenna is connected to an RFID chip for backscatter communication. Matching the impedance of the antenna with that of the chip is crucial to maximize communication efficiency. It is, however, challenging to match the chip's impedance with our three-layer antenna whose impedance varies with the applied pressure. To tackle this problem, we propose to make the chip's impedance variable as well, allowing for dynamic real-time matching. We leverage the commercial Magnus S3 RFID chip that has a *self-tuning circuit* to adjust the chip impedance by modifying a 9-bit register. However, the self-tuning circuit has a limited range of *impedance tunability*. To ensure consistent matching, we have optimized the structure of the two metallic antenna layers to align the antenna's impedance tunability with the range of impedance variations induced by pressure, ensuring long-range communication.

To achieve high-resolution IOP sensing, precise measurement of the impedance variations in the three-layer antenna is essential. Traditionally, various methods have been employed to measure antenna impedance variations by analyzing the backscattered signal, such as tracking phase variations to infer impedance changes [26, 27]. It's worth noting that these methods are susceptible to channel dynamics and become ineffective when the human that wears the contact lens moves.

To mitigate the impact of human motion, we choose a different path: directly sampling and digitizing the impedance

of the antenna within the RFID chip. However, this requires a high-resolution analog-to-digital converter (ADC). To tackle this problem, instead of directly measuring the antenna's impedance, we opt to measure the impedance of the chip itself, which can be obtained from the 9-bit register of the self-tuning circuit. Consequently, this 9-bit register effectively serves as a 9-bit ADC, enabling high-resolution sampling and digitization. After data digitization, the reader then directly queries the Cyclops lens to fetch the digitized impedance.

We have developed cost-effective methods to fabricate the UHCS nanoparticles, the UHCS-PDMS nanocomposite sensing film, the three-layer antenna, and the Cyclops-lens. A substantial number of Cyclops lenses have been manufactured, and their IOP measurement performance has been tested in various environments. Extensive experimental results demonstrate that Cyclops achieves a communication distance of 1 meter, significantly surpassing the centimeter-level range of existing solutions. The IOP accuracy, when tested on artificial eyes and real pig eyes, achieves a median accuracy of 0.51 mmHg and 0.5 mmHg, respectively, surpassing that of commercial portable IOP measurement devices. We have also investigated the impact of tears and eyelids. Results show that Cyclops works robustly, achieving median accuracy of 0.55 mmHg, 0.54 mmHg, and 0.6 mmHg, with tears, eyelids, and both of them, respectively.

2 Related Work

IOP Tonometry. Clinical IOP measurement relies on various techniques, including Goldmann applanation tonometry [28], Dynamic contour tonometry [29], Tonopen tonometry [30], and Pneumatonometer [31]. However, these technologies lack the capability of continuous sensing and require trained medical personnel for operation. Implantable IOP biosensors [32–35] inside the anterior chamber, are the most accurate but also probably the most invasive techniques for IOP measurement.

Contact-lens Based IOP Sensor. Contact-lens based techniques are the dominant solution for the wearable IOP sensing design. Diverse types of pressure sensors that fit in the contact lens have been proposed. Piezoresistive sensors [4, 7–15, 36] are one type of sensor whose resistance changes in response to the applied pressure. The main focus of research in this direction has been designing diverse conductive material whose resistance is sensitive to applied pressure, Some examples of the material are Ag nanowires (AgNWs) [37], gold hollow nanowires(AuHNW) [8], self-assembly graphene [36] and Graphene-AgNWs [18]. Inductively coupled sensors [16–24], are sensors whose reactance changes with the applied pressure, so finding the structure whose capacitance or inductance is sensitive to pressure has been the main focus of research in this direction. The dominant data reading technique for both piezoresistive sensors and inductively coupled sensors are the mutual coupling based method as shown in Figure 1(a), which only supports a reading range of several centimeters. Microfluidic sensor [38–43] is a special type of sensing technique that integrates microfluidic channels inside the contact lens. The pressure affects the distribution of liquid inside those microfluidic. However, observing the distribution of liquid requires a high-end camera placed in proximity of the contact lens and many other complicated, bulky, and costly optic infrastructures.

Contact-lens Based Sensing Systems. Besides measuring eye pressure, contact lens based devices are also used for blood glucose monitoring [44–46], integrated pixel display functions [47–51], eye movement sensing [52] and drug delivery [53–56]. The well-known contact lens system Google contact lens [44] utilize sensors to measure blood glucose levels through tears. Two commercial contact lens systems MoJo [47] and InWith [48] support displaying information on the contact lens. All three contact lens system require on-lens batteries. A recent work [53] design a contact lens capable of detecting glucose levels in tears, and drugs can be released from the self-regulated pulsatile drug delivery system through remote communication. Work [52] makes a solid step towards battery-free contact lens-based computing by embedding an RFID chip inside the contact lens. Another work [49] develops a contact lens equipped with the capability to continuously monitor glucose levels in tears. This contact lens allows for tracking of diabetic conditions through and LED pixel display. Although these studies develop the new functionalities on contact lens, most works typically rely on on-lens batteries for power. Cyclops is a battery-free system that utilizes a commercial ThingMagic RFID reader to access the measurement data.

3 Nanomaterial-based Pressure Sensor

Our design of a pressure sensor with low power consumption and high sensitivity is rooted in the quantum tunneling effect occurring between nanoparticles. We expound upon this concept and present our design in the forthcoming sections.

3.1 Primer of Nanoparticles

In this section, we introduce the background of nanoparticles and the quantum tunneling effect between nanoparticles.

Nanoparticles: Hallow Carbon Sphere. We use hollow carbon sphere (HCS) in our system, which are spherical assemblies composed of carbon atoms, uniformly dispersed across the surface to encase an empty interior, as illustrated in Figure 2(a). These structures boast diameters in the range of hundreds of nanometers, with sphere thicknesses measured around 10 nm. This unique architecture imparts exceptional attributes to HCS, including robust electrical conductivity, chemical stability, thermal resilience, and mechanical durability. Furthermore, HCS exhibits great biocompatibility and biosafety [57], making it suitable for bio-related applications.

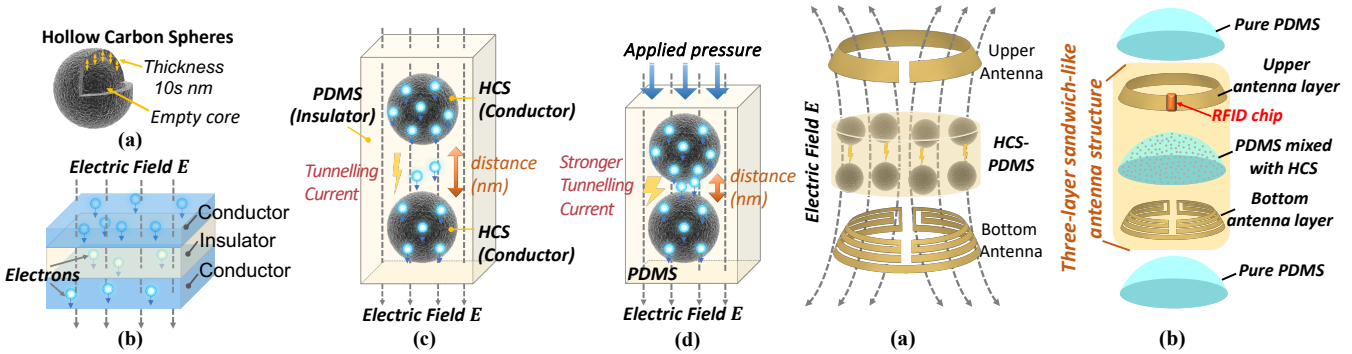


Figure 2. The model of the hollow carbon spheres is shown in **(a)**.The model of the tunneling junction is shown in **(b)**. The quantum tunneling effect between hollow carbon spheres is depicted in **(c)**. Applied imposition of pressure on the construct is illustrated in **(d)**.

Figure 3. Two metallic antenna layers with different structures are used to power the pressure sensor as illustrated in **(a)**. The complete five-layer contact lens antenna structure is shown in **(b)**.

Quantum Tunneling Effect. We leverage a *quantum tunnel junction* to illustrate the phenomenon of quantum tunneling. A typical tunnel junction consists of two electrically conducting layers separated by a thin insulating layer, as depicted in Figure 2(b). Ordinarily, electrons within each conductive layer are unable to traverse the insulating layer and access the adjacent conductive layer. However, the introduction of an external electric field helps electrons overcome this barrier, thereby giving rise to a *quantum tunneling current*. As per quantum theory, the magnitude of this tunneling current hinges on two primary factors: the thickness of the insulating layer and the intensity of the applied electric field.

Quantum Tunneling Effect between HCS. Resembling the tunnel junction, when we embed two electrically conductive HCS inside any insulating material and apply an external electric field, we will observe quantum tunneling current between two HCS particles, just as shown in Figure 2(c). The magnitude of the tunneling current depends on the distance between HCS particles and the intensity of the applied electric field.

3.2 Nanomaterial-based Pressure Sensing

We introduce the design of our nanomaterial-based pressure sensor that leverages the quantum tunneling effect.

3.2.1 Sensing Structure and Sensing Mechanism.

Adhering to the tunneling junction structure, we embed HCS that function as conductors inside the *polydimethylsiloxane* (PDMS), which serves as the insulator, as depicted in Figure 2(c). The HCS particles are dispersed in the PDMS matrix, forming a multilayer structure. The spacing between individual particles is on the nanometer scale. This configuration yields what we term an HCS-PDMS nanocomposite material. Notably, we select PDMS as it is the most widely employed material for manufacturing contact lenses, being both an efficient insulator and showcasing remarkable biocompatibility and biosafety characteristics.

Application of an external electric field to the HCS-PDMS initiates the emergence of a quantum tunneling current between adjacent HCSs, akin to the portrayal in Figure 2(c)(d). An increase in IOP increases the corneal radius of curvature which results in a bi-axial lateral expansion of the HCS-PDMS construct film within the lens. The entire layer undergoes thinning, leading to a proportional reduction in the spacing between the HCSs. As elaborated in previous sections, a decrease in the gap between the conductors amplifies the potency of the quantum tunneling effect, ultimately giving rise to an escalated tunneling current. Consequently, pressure can be measured by assessing the intensity of the resultant tunneling current.

3.2.2 Powering the Pressure Sensor

The proper functioning of our pressure sensor relies on the presence of an external electric field. Conventionally, the predominant method for generating an electric field involves employing electrodes connected to a power source. This potential difference between the electrodes establishes the electric field. However, our contact lens operates without a traditional power source, being driven solely by energy-harvesting mechanisms. As a result, creating such an electric field within a battery-free device poses a significant challenge.

Three-layer Sandwich-like Antenna Structure. In order to energize the sensor on our battery-free device, we introduce a 3-layer sandwich-like antenna structure. This configuration involves two distinct metallic antenna layers, flanking a central HCS-PDMS film (*i.e.*, the pressure sensor) as depicted in Figure 3(a). Notably, the two metallic antenna layers are designed to have different structures. Consequently, when the interrogation signal from the reader interacts with the tag, distinct surface currents are induced on the two antenna layers, resulting in disparate current distributions. This, in turn, naturally engenders a potential difference and establishes an electric field between the two antenna layers. This process successfully applies an external electric field to the HCS-PDMS film, thus powering the sensor.

Complete Five-layer Contact Lens Design. Direct interaction between the metallic antennas and human tissue is

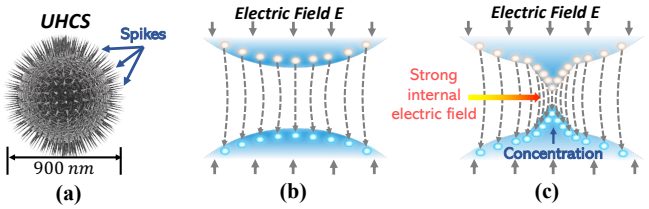


Figure 4. The urchin-like HCS (UHCS) has uniformly distributed spikes on its surface, creating a much stronger internal electric field compared to HCS without spikes.

considered biologically unsafe. In response, we encapsulate both the sensor and the tag antenna with two additional layers of biosafe PDMS. Following this enhancement, a 5-layer contact lens configuration is achieved, as depicted in Figure 3(b). Among those layers, the two pure PDMS layers and two metallic antenna layers have a thickness of 30 μm and 10 μm, respectively. The thickness of the HCS-PMDS sensing film is 60 μm. Consequently, the overall thickness of the complete contact lens amounts to 140 μm.

3.2.3 Maximizing the Sensitivity.

We have made three optimizations to our nanomaterial-based pressure sensor to maximize its sensitivity.

Urchin-like HCS. To generate a stronger tunneling current under the same external electric field, we make a structural modification to the carbon sphere. Specifically, we replace the normal HCS with urchin-like hollow carbon sphere (UHCS) whose structure is depicted in Figure 4(a). The surface of the urchin-like carbon sphere features long, sharp, and uniformly distributed spikes. When subjected to an external electric field, a substantial concentration of free electrons accumulates on these spikes, engendering a markedly intensified internal electric field in contrast to the smooth-surfaced normal HCS, as shown in Figure 4(b) and (c). This augmented internal electric field significantly facilitates electron traversal across the insulating space between the carbon spheres, resulting in an amplified tunneling current. Moreover, the presence of spikes on the carbon sphere serves to diminish the gap between neighboring carbon spheres, particularly at low filling concentrations, thereby further augmenting the tunneling current.

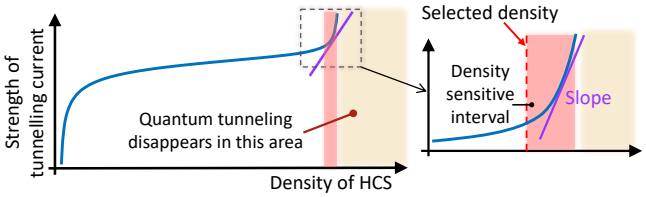


Figure 5. The relationship between the strength of tunneling current and the density of UHCS inside of the PDMS.

Optimizing the UHCS Density. The distance between the carbon spheres, which corresponds to the density or concentration of carbon spheres within the PDMS, holds notable influence over the resultant tunneling current. We have constructed a model and subsequently conducted simulations to discern the connection between the intensity of the quantum tunneling current and the density of UHCS. The simulated outcomes, graphically presented in Figure 5, yield two key observations. Firstly, the quantum tunneling phenomenon disappears when the distance between carbon spheres decreases too much (density surpassing 2.06 $wt.\%$). Secondly, a density-sensitive interval (ranging from 1.72 $wt.\%$ to 2.06 $wt.\%$) emerges, within which even minor fluctuations in density yield substantial variations in the generated tunneling current's strength.

To optimize sensitivity, we configure the density of UHCS at the start of this density-sensitive interval, just as illustrated in Figure 5. This strategic selection ensures that minute fluctuations in pressure yield significant deviations in the magnitude of the resulted tunneling current.

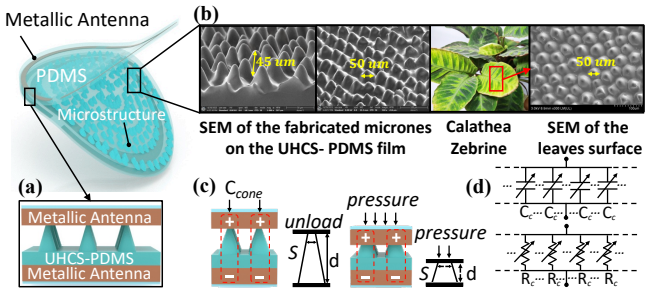


Figure 6. The microstructure of the UHCS-PDMS film is depicted in **(a)**. The scanning electron microscopy (SEM) imaged microstructure is illustrated in **(b)**. The schematic of the pressure sensing is illustrated in **(c)**. Circuit schematic diagram of pressure sensing is depicted in **(d)**.

Bio-inspired Microstructures. To further improve the sensitivity, we introduce microstructures of the UHCS-PDMS sensing film. Specifically, we create a large number of *microcones* on the surface of the sensing film, each with an average diameter of 50 μm and height of 45 μm, just as shown in Figure 6. The apex of each microcone connects to the antenna layer, while the base of the cone interfaces with the surface of the sensing film, as depicted in Figure 6(a). For the purpose of structural stability, the residual space between the antenna and the microcones is filled with pure PDMS.

Each microcone serves as a connection point between two antenna layers, effectively functioning as a resistor. The resistance R_c of such a resistor is given as:

$$R_c = \frac{1}{\sigma S} \tag{1}$$

where σ represents the conductivity of the material, namely the UHCS-PDMS nanocomposite and S denotes the *area of the plate*, which is the contact area between the cone and the upper antenna layer. As per our empirical observations outlined in §6.1, an increase in applied pressure not only expands the contact area S, but also increases the conductivity of the material, primarily due to a more pronounced tunneling

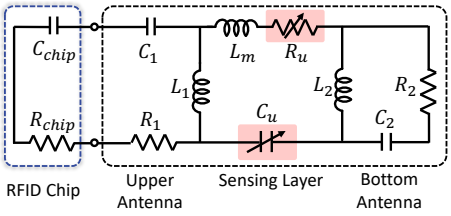


Figure 7. The equivalent circuit of the Cyclops lens antenna, where C_u and R_u represents a variable capacitor and resistor.

effect, consequently resulting in decreased resistance R_c.

The combination of the two metallic antenna layers and the cone in between also forms a parallel plate capacitor, as shown in Figure 6(c), whose capacitance C_{cone} is:

$$C_c = \frac{\varepsilon S}{4\pi d} \tag{2}$$

where ε represents the *permittivity* of the UHCS-PDMS nanocomposite, and d is the distance between two plates, *i.e.*, two antenna layers. When pressure is applied, the entire three-layer antenna gets thinner, leading to an expansion in the contact area S between the cone and the antenna and a reduction in the distance d between the two antenna layers. As per our empirical findings in §6.1, this process also results in increased permittivity ε of the nanocomposite. Consequently, an increase in pressure leads to a corresponding increase in capacitance C_c of each microcone-antenna capacitor.

In summary, the incorporation of microcone introduces an array of pressure-sensitive capacitors and resistors connected in parallel, just as shown in Figure 6(d). Consequently, the overall impedance of the entire sandwich-like structure becomes responsive to pressure, whose resistance can be represented as:

$$R_u = \frac{1}{\sum_{k=1}^{K} \frac{1}{R_{c_k}}} \tag{3}$$

and capacitance is given as:

$$C_u = \sum_{k=1}^{K} C_{c_k} \tag{4}$$

assuming there are K microcones added to the film.

Fabrication of microcones. The fabrication of a microcone structure demands advanced MEMS technology, involving costly equipment and skilled operators. In response, we propose a bio-inspired microcone fabrication technique that notably mitigates fabrication complexity and cost. Our approach stems from the observation that Calathea Zebrine [58] leaves possess a remarkably uniform microcone surface structure, as evidenced in Figure 6(b). Leveraging this natural inspiration, we propose to produce a microcone template utilizing Calathea Zebrine leaves[1]. Subsequently, we replicate the surface pattern from this microcone template onto the sensing film, effectively generating uniform microcone structures.

[1]The detailed microcone template fabrication process using Calathea Zebrine leaf can be found in Appendix §A.1.

3.2.4 Integrated Sensing System

We connect an RFID chip to our three-layer antenna for backscatter communication, just as shown in Figure 3(b). The RFID chip we use is Magnus S3 [59] with a known and fixed chip impedance $Z_{chip} = \mathrm{Re}(3.99\ \Omega) + \mathrm{Im}(91.05\ \Omega)\mathrm{j}$. The equivalent circuit diagram of our sensing system is shown in Figure 7. In such an integrated sensing and communication system, the three-layer antenna functions as both a pressure sensor and a communication antenna. The impedance of the three-layer antenna depends on two categories of factors:

- *Antenna-dependent static components.* All these components are fixed if the structures of those two metallic antenna layers are known. Specifically, C_1 and C_2 represent the *parasitic capacitance* associated with the two antenna layers; R_1 and R_2 denote the *parasitic resistance* of the respective antenna layers; L_1 and L_2 correspond to the *parasitic self-inductance* of two antenna layers; L_m is the *mutual-inductance* between two antenna layers.
- *Pressure-dependent variable components.* The impedance of the sandwich-like three-layer structure C_u and R_u varies when the applied pressure varies.

Antenna Impedance. Based on the equivalent circuit shown in Figure 7(b), we derive the overall impedance of the whole three-layer antenna as:

$$Z_{ant} = -\frac{1}{j\omega C_1} + R_1 + \frac{j\omega L_1 (Z_s + Z_b)}{j\omega L_1 + Z_s + Z_b} \tag{5}$$

where Z_s and Z_b are:

$$Z_s = -\frac{1}{j\omega C_u} + R_u + j\omega L_m, \tag{6}$$

$$Z_b = \frac{\omega^2 R_2 L_2 C_2 + j\omega L_2}{1 + \omega^2 C_2 L_2 - j\omega R_2 C_2}. \tag{7}$$

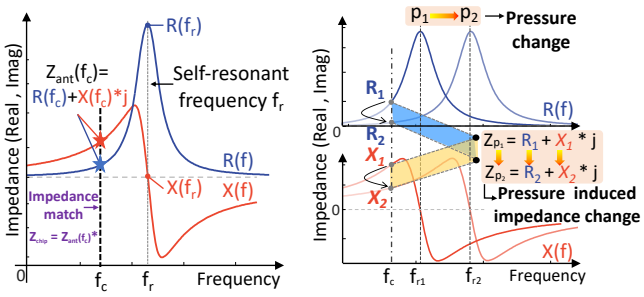
The detailed process of deriving the antenna impedance is presented in Appedix B. According to Eqn. 5, we know that any variations in pressure change the value of the pressure-sensitive resistor R_u and capacitor C_u, which in turn affects Z_s, ultimately changing the impedance Z_{ant} of the entire three-layer antenna. Therefore, our three-layer antenna effectively functions as a pressure sensor whose impedance value is directly associated with the applied pressure.

4 Concurrent Sensing and Communication

In this section, we present our system design that integrates the dual three-layer and the RFID chip into one battery-free IOP sensing system that simultaneously achieves high sensing accuracy and supports long range communication.

4.1 Long-Range Backscatter Tag Design

To support long-range backscatter communication, it is imperative to match the impedance of the commercial RFID chip with the impedance of our three-layer antenna. Since the impedance of an antenna changes with the operating frequency, our objective is to design an antenna that matches its



(a) Impedance matching at f_c **(b)** Impedance varies with pressure

Figure 8. The antenna impedance fluctuates with both frequency and applied pressure. We focus on achieving impedance matching at the operating frequency.

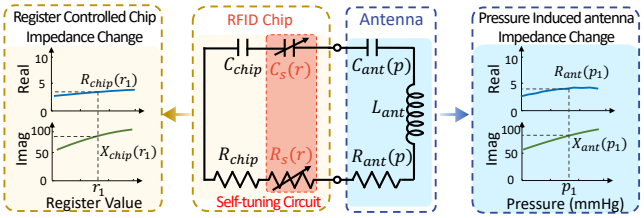


Figure 9. The equivalent circuit of the Cyclops with the self-tuning circuit, where C_s and R_s represents a variable capacitor and resistor values that vary based on the register value.

impedance Z_{ant} with the chip impedance Z_{chip} at its *operating frequency* f_c, *i.e.*, Z_{chip}=$Z_{ant}(f_c)$ just as shown in Figure 8(a). It is essential to emphasize that the operating frequency f_c of our system differs from the *self-resonant frequency*[2] f_r of the antenna because our chip has close-to-zero resistance (R_c=3.99 Ω) and high reactance (X_c=91.05 Ω).

Challenge: Antenna Impedance Fluctuations. When the applied pressure changes, the antenna impedance distribution across frequencies $Z_{ant}(f)$ changes accordingly, as depicted in Figure 8(b). In contrast, the chip impedance Z_{chip} is pressure-oblivious and thus remains constant. As a result, the impedance of the antenna and the chip cannot maintain matching as applied pressures fluctuate, which could significantly impact the communication efficiency of our system.

Solution: Self-tuning Circuit. We leverage the self-tuning circuit inside the commercial Magnus S3 RFID chip to dynamically adjust the impedance of the chip to match the variable antenna impedance, just as shown in Figure 9. The self-tuning circuit effectively functions as a variable capacitor $C_s(r)$ and resistor $R_s(r)$, with r representing the value of a 9-bit self-tuning register. The RFID chip fine-tunes its impedance by adjusting the self-tuning register's value, making the chip impedance also variable, denoted as $Z_{ant}(r)$.

Each time the chip powers on, it initiates a search for the optimal r_m value that achieves impedance matching with the connected antenna. The key insight the chip leverages is that antenna efficiency is at its peak when impedance is well-matched. Consequently, the RFID chip attempts to discover the ideal r_m value that maximizes the RSSI of the received signal, as detailed in [59]. Once this optimal r_m is identified, the chip locks its register value to r_m to ensure continuous impedance matching between the chip and the antenna.

[2]Self-resonant frequency is defined as the frequency where the antenna resistance is maximized and at the same time reactance is zero.

4.2 High-Resolution Pressure Sensing

To obtain precise digital pressure measurements, we need to accurately measure the impedance of the three-layer antenna and subsequently digitize the measured impedance using a high-resolution analog-to-digital converter (ADC). However, several challenges emerge. First, impedance measurement typically relies on a vector network analyzer, a sophisticated and costly equipment unsuitable for integration within a contact lens. Furthermore, even if the contact lens possessed adequate space for housing a high-resolution ADC, the energy constraints of a battery-free device present significant hurdles in powering such a component.

Solution: Effective 9-bit ADC via Impedance Matching. We rely on the real-time impedance matching scheme to achieve simultaneous antenna impedance measurement and digitization. The underlying concept hinges on the fact that the applied pressure, which can be denoted as p, governs the overall impedance of the three-layer antenna, which can be denoted as $Z_{ant}(p)$. In parallel, the RFID chip adjusts its register value r, thus modifying its own impedance $Z_{chip}(r)$, as illustrated in Figure 10(a). To maximize the communication efficiency, the chip searches for r_m that guarantees the state of the impedance matching between the chip and antenna:

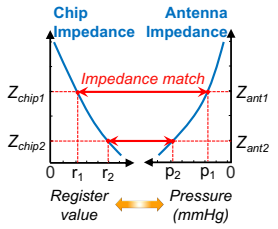
$$Z_{ant}(p) = Z_{chip}(r_m). \quad (8)$$

Eqn. 8 effectively establishes a mapping between the applied pressure p and the chip register value r_m. For instance, Figure 10(b) plots the relationship between the pressure p and register value r_M we collected from our fabricated contact lens, based on which, we can derive the applied pressure p from the chip register value r_m that achieves impedance matching. Moreover, the 9-bit chip register inherently functions as a 9-bit ADC that digitizes the sensing value, granting exceptionally high pressure-sensing resolution.

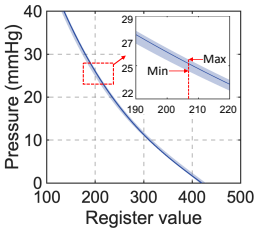
The chip stores the 9-bit register value at a specific location (from 0xC7 to 0xCF for Magnus S3) inside its memory bank. The reader sends a standard C1G2 command, *i.e.*, the *read* command that contains the exact location of the register inside the memory bank, to fetch the value r_m from the chip, based on which, the reader derives the pressure according to the pressure-register mapping in Figure 10(b). Specifically, we employ polynomial curve fitting to establish an average trend line, which forms our test curve for calibration.

4.3 Dual-Purpose Antenna Deisgn

Our three-layer antenna serves a dual purpose, functioning as both a communication antenna and a pressure sensor with



(a) Theoretic mapping.



(b) Mapping in practice.

Figure 10. The impedance matching naturally creates a mapping between the register value r_m and the applied pressure p.

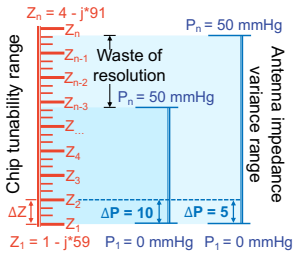


Figure 11. Aligning the antenna impedance variation range with chip's impedance tunability.

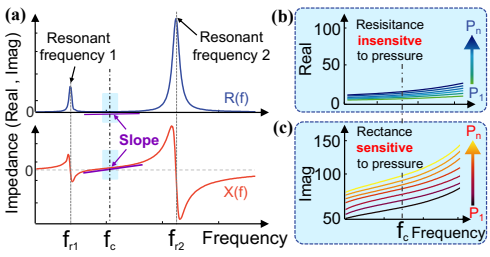


Figure 12. A dual-frequency antenna results in significant pressure-induced reactance variations but minimal resistance variations at the operating frequency.

a sensing range spanning from 0 mmHg to 40 mmHg. To concurrently optimize communication and sensing performance, the antenna's impedance variations under various pressures within the sensing range should align with the chip's *impedance tunability*, *i.e.*, the range of impedance achievable by the chip via modifying the value of the 9-bit register. First, it is crucial to ensure that all potential antenna impedances fall within the tunability range of the chip. Any outlier leads to impedance mismatches, thereby adversely affecting communication efficiency. Furthermore, to make the most of the 9-bit ADC's resolution, the antenna's impedance variance range should occupy the chip's tunability to the greatest extent possible, as visually represented in Figure 11.

The Magnus S3 chip exhibits substantial tunability in reactance, ranging from 59 Ω to 91 Ω, while its resistance tunability is negligible, spanning from 1 Ω to 4 Ω. Hence, our antenna design should prioritize sensitivity to pressure-induced reactance variations while remaining less responsive to changes in resistance. Except for the requirement on the antenna impedance variance, our antenna design must meet two additional criteria. Firstly, it should be sized, shaped, and curved to seamlessly integrate with the contact lens. Secondly, the two metallic antenna layers must exhibit distinct structures to enable the generation of an electric field between them, thus powering the sensor, as illustrated in Figure 3(a).

Dual Resonant Frequency Antenna Structure. We leverage a dual resonant frequency antenna structure to effectively align the antenna impedance variations with the chip's tunability. Just as shown in Figure 12(a), we create two resonant frequencies f_{r1} and f_{r2} on both sides of the working frequency f_c through antenna structural design. We could see that, compared to the single resonant frequency antenna in Figure 8(a), the resistance variations near the operating frequency f_c exhibit a flat slope while the reactance variations have a sharp slope. As shown in Figure 8(b), the applied pressure induces a shift in the impedance distribution of one antenna across frequencies. Such a shift in impedance distribution for our dual-frequency antenna leads to substantial variations in reactance while keeping resistance changes to a minimum just as shown in Figure 12(b) and (c). Such a trend in impedance variations aligns with the chip's tunability.

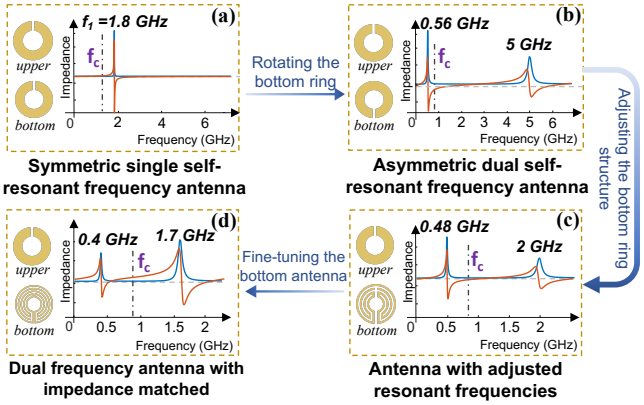


Figure 13. The workflow of creating a dual-frequency antenna structure and confining the antenna's impedance variations within the chip's tunability.

We follow the steps depicted in Figure 13 to create such a dual-frequency antenna and confine its impedance variations within the chip's tunability. Our three-layer antenna consists of two conductive layers separated by an intermediate dielectric layer. We initially start with a completely symmetrical antenna structure, just as shown in Figure 13(a), which exhibits only a single resonant frequency near the operating frequency (0-3 GHz). After experimenting with various methods to create a second resonant frequency on such a small antenna, we surprisingly find out that, the easiest way to break the symmetry is by rotating the bottom layer as shown in Figure 13(b). This asymmetry results in a significant shift in the original self-resonant frequency, moving it from 1.8 GHz to 0.5 GHz, while concurrently inducing the emergence of another self-resonant frequency at 5 GHz.

We also observe that increasing the electric length of the bottom layer significantly lowers the frequency of the second resonant frequency while keeping the first frequency less affected. Therefore, we gradually increased the length of the bottom layer antenna and shifted two resonant frequencies to 0.48 GHz and 2 GHz respectively, as shown in Figure 13(c). Finally, by making fine adjustments to the bottom layer's dimensions, such as the ring width and the gap between adjacent

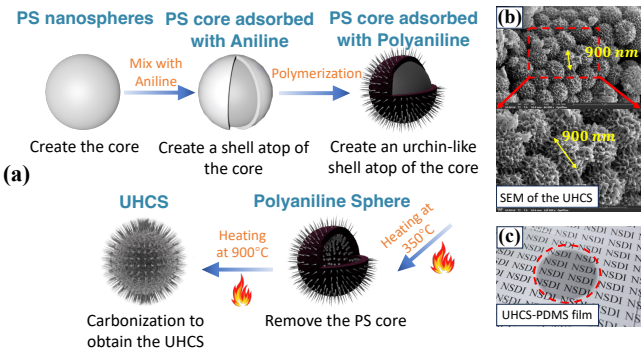


Figure 14. The fabrication process of UHCS is depicted in (**a**). The scanning electron microscopy (SEM) imaged UHCS that we fabricated is illustrated in (**b**). The fabricated UHCS-PDMS nanocomposite film is shown in (**c**).

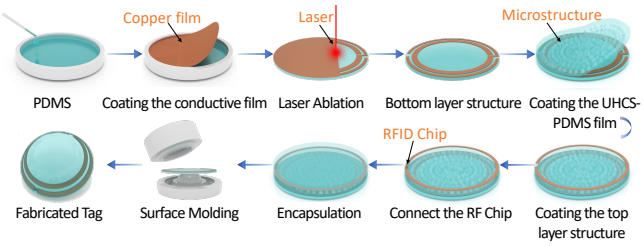


Figure 15. Cyclops contact lens fabrication workflow.

rings[3], we were able to finely tune the impedance distribution to ensure that pressure-induced variations remain within the chip's tunability as shown in Figure 13(d).

5 Implementation and Fabrication

Synthesis of UHCS. We employ the seeded swelling polymerization method [60] to synthesize *polystyrene* (PS) nanospheres. We then mix the fabricated PS nanospheres with *aniline* and add *iron(III) nitrate* to initiate the polymerization of aniline. This procedure results in the formation of an urchin-like *polyaniline* shell atop the PS nanospheres, as shown in Figure 14(a). After the washing and drying, we subject the spiky polyaniline to a temperature of 350°C within an *argon* atmosphere, effectively eliminating the PS core and leaving behind the shell structure. The final step involves subjecting the urchin-like polyaniline shell to a temperature of 900°C to initiate the carbonization of the shell, yielding the UHCS material. A detailed quantitative description of the synthesis process can be found in Appendix §A.2. Figure 14(b) depicts the scanning electron microscope (SEM) images of the fabricated UHCS whose mean sphere diameter is 900 *nm*.

Fabrication of UHCS-PDMS with Microstructure. We disperse the UHCS powder inside the PDMS solution with a mass fraction of 1.75 *wt.%*. Subsequently, we employ a spin coating process to apply the blended solution onto the Calathea Zebrine template with microstructures. The solution

[3]The detailed antenna fine-tuning process is in Appendix C.

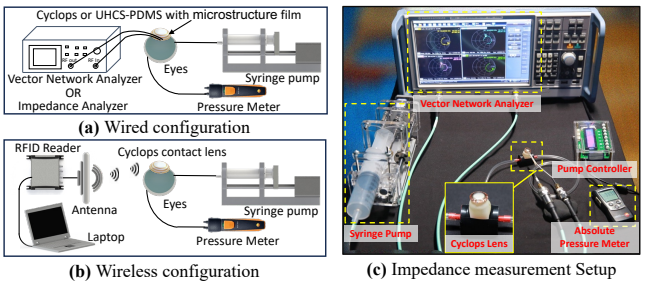


Figure 16. IOP measurement and material characteristic measurement platform for controlled experiments.

and template are then heated at 80°*C* for an hour to produce the UHCS-PDMS film. A detailed UHCS-PDMS film fabrication process can be found in Appendix §A.3. Figure 14(c) shows a circular film of the UHCS-PDMS with a radius of 3 *cm*, showcasing exceptional transparency.

Fabrication of Contact Lens. Figure 15 details the workflow for crafting our contact lens. We start by shaping a copper film using laser engraving machine to match the proposed antenna structure and place it on a pure PDMS film. We then put the UHCS-PDMS film with microstructures atop the first antenna layer. Then, we etch the copper film using a laser engraving machine to create the upper layer antenna, which is connected directly to the RFID chip. We then place the upper layer antenna on the UHCS-PDMS film. To ensure biosafety, we encase the upper layer antenna in an additional pure PDMS film. After a preliminary shaping step of subjecting it at 80°C for 10 minutes, we transfer the structure to a contact lens mold and thermally cure it at 80°C for one hour.

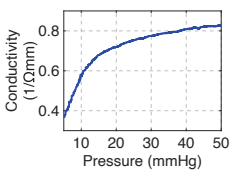
6 Evaluation

Measurement Setup. We fabricate a plastic prosthetic eye, which is a curved 3D eye model featuring an inlet connected to the syringe pump for precise pressure control and an outlet connected to a testo 510 pressure meter [61] to establish the accurate ground truth for applied pressure. We program the syringe pump controller to regulate the inflation speed and thus the applied pressure, as shown in Figure 16(c). We wirelessly collected real-time pressure from the pressure meter.

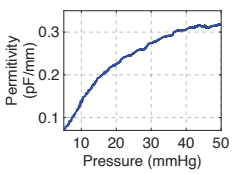
This platform serves as the basis for a variety of measurements. To assess the impedance, conductivity, or permittivity of materials or antennas, we position the fabricated sensing film or antenna atop the eyes and connect the material or antenna to an impedance analyzer (HIOKI IM7587 [62]) or a vector network analyzer (R&S ZNB8 [63]) using wire, just as shown in Figure 16(a) and (c). For evaluating the performance of the developed contact lens, we use a ThingMagic M6 reader [64] to query the chip, as shown in Figure 16(b).

6.1 Hardware Verification

Verification of the Fabricated Nanocomposite. To validate the characteristics of the nanocomposite, we place the UHCS-

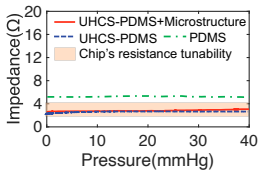


(a) Conductivity.

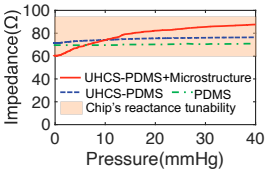


(b) Permittivity.

Figure 17. The properties of intermediate film under varying applied pressure.



(a) Resistance.



(b) Reactance.

Figure 18. The pressure-induced impedance variations of different materials.

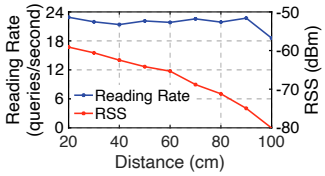


Figure 19. The measured RSS and reading rate at varying reader-chip distances.

PDMS film, containing microstructures with a thickness of 60 μm, between two conductive copper films. We cut the dimensions of the film slightly larger than those of the copper layers to prevent short-circuiting between the upper and lower copper layers. Subsequently, we connect the copper films to an impedance analyzer and measure the conductivity and permittivity under varying pressure conditions. The results are presented in Figure 17, clearly demonstrating that applied pressure influences the conductivity and permittivity of the UHCS-PDMS film. Specifically, higher applied pressure corresponds to greater conductivity or permittivity, thus corroborating the theory outlined in §3.2.3.

Validating the Alignment with Chip's Tunability. We conduct an investigation into the range of pressure-induced impedance variations in our three-layer antenna. Specifically, we connect the antenna to a vector network analyzer and apply varying levels of pressure. For comparison, we replace the pure PDMS film with UHCS-PDMS film without microstructures and UHCS-PDMS with microstructures. We maintain same thickness across all the three films. The measured impedance, along with the chip's impedance tunability, is presented in Figure 18(a) and (b), yielding two notable observations. First, the impedance of both types of UHCS-PDMS film falls within the chip's tunability range, while the resistance of the pure PDMS falls outside of this range. Second, the impedance variation range of the UHCS-PDMS with microstructures covers a significantly wider span of the chip's reactance tunability, indicating higher sensitivity when compared with the pure UHCS-PDMS without microstructures.

Communication Range. In this section, we explore the communication range of our three-layer antenna. We employ our reader to interrogate the RFID chip embedded in the contact lens and record two key parameters: the received signal strength (RSS) and the reading rate, which represents the number of successful pressure queries per second. We conduct these measurements at various distances ranging from 20 cm to 1 m and present the findings in Figure 19. It is evident from the results that as the communication distance increases, the RSS decreases, eventually reaching -80 dBm at a distance of 1 m. Similarly, the reading rate maintains at 22 $queries/second$ within the communication rate and slightly decreases to 19 $queries/second$ when approaching the 1-meter communication range. Beyond this boundary, a stable connection between the reader and the RFID chip becomes unattainable. To our knowledge, achieving a communication range of 1 m is a significant advancement compared to existing contact lens-based IOP measurement systems.

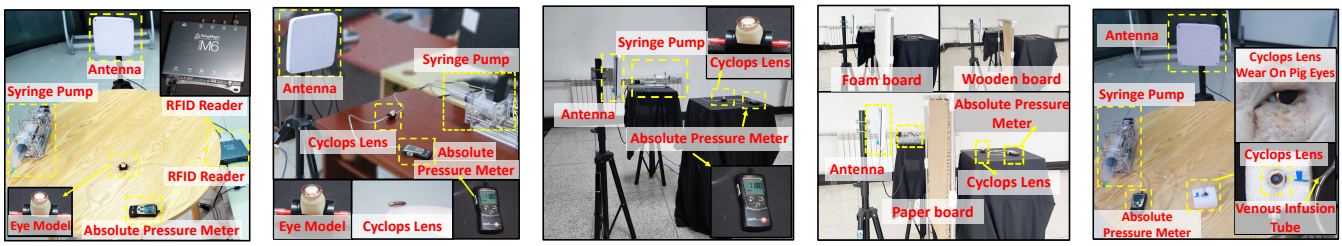
6.2 IOP Measurement Performance

We begin with an end-to-end experiment to gain an insight into how Cyclops works, followed by the verification of IOP measurement performance from various angles.

End-to-end Performance. We use the experimental setup illustrated in Figure 16 to control the applied pressure and also record the ground truth using pressure meter. We use the reader to query the tag for the register value and derive the pressure according to the mapping. We repeat the experiments at various locations, including an office ($5m \times 7m$) with several tables, an activity room ($10m \times 8m$) full of furniture, and a corridor ($20m \times 8m$) with several billboard sets, as shown in Figure 20. At each location, we vary the chip-reader distance from 20 cm to 1 m with a step size of 10 cm.

We plot the CDF of the IOP measurement error in Figure 21, from which we see that the median error is around 0.51 mmHg and the 70% error is around 0.8 mmHg. We also compared our system with commercial portable intraocular pressure (IOP) measurement devices, e.g., the Fa-800 Vet, FUAN [65] and the iCare IC 200 [66] with a respective measurement accuracy of $\pm$1.5 mmHg and $\pm$1.2 mmHg ($\leq$ 20 mmHg). Furthermore, we plot the estimation errors across various IOP ranges in Figure 22. Within the normal human IOP range of 11-20 mmHg, the median error is approximately 0.5581 mmHg, and the 75 % error is about 0.9123 mmHg. Our system demonstrates an even higher accuracy compared to commercial portable IOP measurement devices.

Tracking the IOP Variations. We explore Cyclops's capability to continuously monitor human IOP, which exhibits diurnal fluctuations, typically being lowest in the morning upon waking up and gradually peaking around noon [67]. In our investigation, we utilize a syringe pump to systematically manipulate pressure applied to the tag. The pressure is incrementally increased from 5 mmHg to 28 mmHg and then continuously reduced back to 5 mmHg. This loading and unloading procedure is repeated for **24 hours**. To provide a benchmark, we also employ a stepwise pressure variation pattern. Here, we apply a specific pressure to the contact lens, maintain it for 30 seconds, and then gradually increase or decrease it to reach another specified pressure level. We also test the stepwise pattern for 24 hours to test Cyclops's robustness.



(a) Office room. **(b)** Activity room. **(c)** Corridor. **(d)** NLoS scenario. **(e)** Biological scenario.

Figure 20. The experiment settings. To understand the impact of multipath, we conduct experiments inside the office, an activity room, and a corridor. To understand the NLoS, we block the LoS using various materials. We also conduct our experiments on pig eyes to investigate the biological impact.

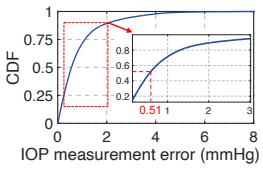


Figure 21. The estimation error for IOP measurement.

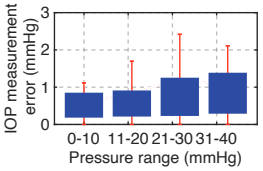
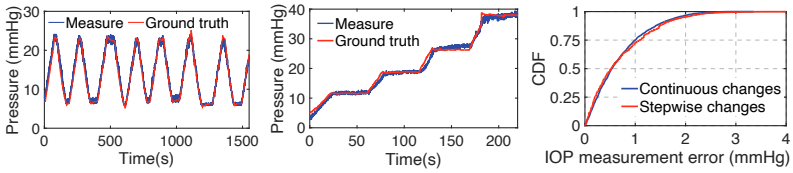


Figure 22. The estimation error for different IOP ranges.



(a) Continuous changes. **(b)** Stepwise changes. **(c)** IOP measurement error.

Figure 23. Performance on tracking the IOP variations. We conduct continuous and stepwise IOP measurement patterns and generate the cumulative distribution function of measurements.

The results depicted in Figure 23(a) and (b) showcase Cyclops's remarkable IOP tracking performance under both continuous and stepwise patterns, exhibiting only minor deviations from the ground truth. Figure 23(c) illustrates the CDF of IOP estimation errors during the 24-hour test. The median IOP tracking errors for the continuous and stepwise patterns are 0.53 mmHg and 0.54 mmHg, respectively. These findings demonstrate Cyclops's precision and robustness in IOP monitoring, particularly during prolonged tracking sessions, highlighting its potential in clinical applications where accurate and continuous IOP measurement is crucial.

Impact of Environment and Distance. We plot the cumulative distribution function of Cyclops's IOP measurement error measured with different reader-chip distances in Figure 24 *lower*, and the median error of each distance in Figure 24 *upper*. We plot the results obtained from the office, the activity room, and the corridor in Figure 24(a), (b) and (c).

The median error in IOP measurements varies between [0.3, 0.75] mmHg, [0.35, 1.02] mmHg, and [0.37, 1.1] mmHg for the office, activity room, and corridor, respectively. Remarkably, there is no significant disparity in IOP measurement performance across these diverse testing environments, despite variations in their multipath profiles. This consistency arises from the inherent separation of the sensing and communication processes within our system. Sensing is primarily achieved through impedance matching, while the reader employs backscatter communication to query the sensing data, specifically the 9-bit register value. In contrast to techniques that rely on analyzing the backscattered signal, such as phase variation calculations, our sensing quality remains largely independent of the received signal quality.

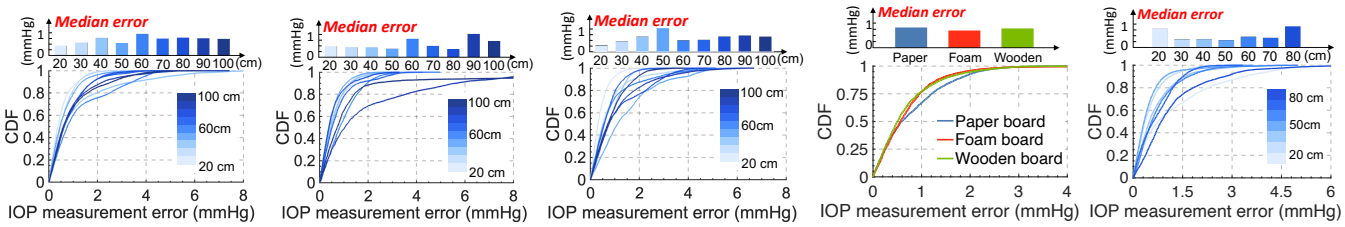
Nonetheless, it is worth noting that our sensing performance is influenced by signal quality to some extent, for example, the median error is smaller when the reader-chip distance is less than 50 centimeters, as shown in Figure 24. This improvement is attributed to enhance the accuracy of impedance matching when the received signal strength is substantially higher. Beyond this range, we do not see a clear trend between the reader-chip distance and the IOP measurement error, demonstrating the decoupling of sensing performance and the communication signal strength.

Impact of NLoS. We introduce various obstructing objects between the tag and the reader antenna to create NLoS scenarios, just as shown in Figure 20(d). The objects include a paper board, a foam board and a wooden board,. The distance between the antenna and the contact lens remains at 70 *cm*. We plot the IOP error in Figure 25. We see that the median errors are 0.562 mmHg, 0.589 mmHg, 0.505 mmHg, when the obstructing objects are a wooden board, a paper board, and a foam, respectively. This experiment further demonstrates that our sensing performance is decoupled with the signal quality.

6.3 Biological Impact on the Performance

We use pig eyes to assess the biological impact on our system's performance, since the structure of pig eyes closely resembles that of human eyes, including intraocular pressure regulation and eye size. We examine Cyclops's performance in various bio-environments[4], to explore the effects of different eye shapes, tear compositions, and the eyelid.

[4]With the approval of university's institutional review board (IRB).



(a) Office room. **(b)** Activity room. **(c)** Corridor.

Figure 24. The measurement error of IOP varies with the distance between the reader and the contact lens in three different environments.

Figure 25. The IOP measurement error in non line-of-sight scenario.

Figure 26. The IOP measurement error on pig eyes.

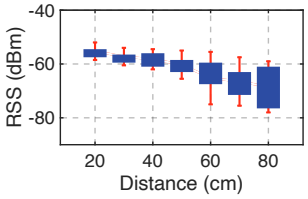


Figure 27. The measured RSS on pig eyes.

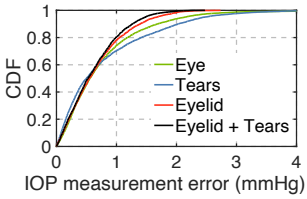
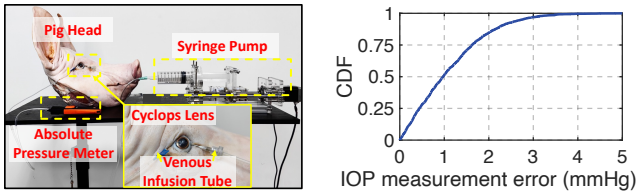


Figure 28. Impact of various biological tissues.



(a) Pig head. **(b)** IOP measurement error.

Figure 29. IOP measurement performance on pig head.

End-to-end Performance on Pig Eyes. To measure IOP on pig eyes, we put the contact lens on the eyes as shown in Figure 20(e). We use two venous infusion tubes to create two channels in the pig's eyes, as shown in Figure 20(e). One tube connects to a syringe pump for pressure control, and another connects to a pressure meter to obtain ground truth. The CDF of the estimation error is shown in the Figure 26. Even though the communication range slightly decreases to 80 cm, Cyclops still achieves high accuracy on pig eyes. The median error varies within the range of [0.33, 0.85] mmHg across distances, which is lower than that of commercial portable IOP measurement devices. The performance in the biological environment validates the efficacy of our approach.

Impact of Eyeball Shapes. Different individuals have varying eyeball shapes. To validate the robustness of our system across different organisms, we place the contact lens on ten different pig eyes and measure the RSS of the backscatter signal. The measurement results are shown in Figure 27. The communication distance ranges are similar for all 10 pig eyes, *i.e.*, 80 cm. Despite the shape variations between pig eyes, the average RSS deviation remains around 3.29 dBm over distance. These results demonstrate that the performance of the Cyclops is consistent across pig eyes of different shapes.

The Impact of Tears and Eyelids. In order to make our user scenario closer to the real biological environment, we also investigate the impact of tears and eyelids. To evaluate the effect of the tears, we drip around 0.05 mL of artificial tear fluid onto the pig eyes and measure the IOP under varying reader-chip distances. We also add eyelids on pig eyes and test its impact. We plot the IOP error measured from the contact lens worn on pure pig eyes, pig eyes with tears, pig eyes with eyelids and pig eyes with both eyelids and tears, in Figure 28, from which we see that the median error is 0.5 mmHg, 0.55 mmHg, 0.54 mmHg and 0.6 mmHg, respectively. With the pig eyelid and tears, we observe that our system achieves nearly the same performance as pure pig eyes, which meets the requirements of real-world IOP measurement.

Performance Evaluation on Pig Head. To evaluate the performance of our system more authentically, we attach the contact lens to the eyes of a pig. We attempt to change the IOP of the pig eyes by inserting a venous infusion tube from the edge of the eyes into the interior, as depicted in Figure 29(a). We also employed a pressure meter to monitor the real-time IOP changes in the pig eye. We display the cumulative distribution function of IOP measurement error from the contact lens under this scenario. As shown in Figure 29(b), the median error is 0.97 mmHg which still demonstrates good performance compared to commodity portable IOP measurement devices.

7 Conclusion

In this work, we introduce Cyclops, a wearable contact lens that can continuously measure intraocular pressure changes over a long distance. Through comprehensive introduction and experimental evaluation, we demonstrate the effectiveness of the system in measuring IOP changes in real-world settings and biological environments.

Acknowledgments

This work is supported by the NSFC A3 Foresight Program Grant 62061146001, and the National Natural Science Foundation of China (62272388, 62372374), the Shaanxi Science and Technology Innovation Team Program under Grant (2024RS-CXTD-05), the Shaanxi Qinchuangyuan Program under Grant (QCYRCXM-2023-103), and the International Cooperation Project of Shaanxi Province (2020KWZ-013). This Work is also supported by Yaxiong Xie's startup funding. We thank our reviewers and shepherd for their insightful feedback which helped improve this paper.

References

[1] Jaimie D Steinmetz, Rupert RA Bourne, Paul Svitil Briant, Seth R Flaxman, Hugh RB Taylor, Jost B Jonas, Amir Aberhe Abdoli, Woldu Aberhe Abrha, Ahmed Abualhasan, Eman Girum Abu-Gharbieh, et al. Causes of blindness and vision impairment in 2020 and trends over 30 years, and prevalence of avoidable blindness in relation to vision 2020: the right to sight: an analysis for the global burden of disease study. *The Lancet Global Health*, 9(2):e144–e160, 2021.

[2] Yih-Chung Tham, Xiang Li, Tien Y Wong, Harry A Quigley, Tin Aung, and Ching-Yu Cheng. Global prevalence of glaucoma and projections of glaucoma burden through 2040: a systematic review and meta-analysis. *Ophthalmology*, 121(11):2081–2090, 2014.

[3] Matthew J Burton, Jacqueline Ramke, Ana Patricia Marques, Rupert RA Bourne, Nathan Congdon, Iain Jones, Brandon AM Ah Tong, Simon Arunga, Damodar Bachani, Covadonga Bascaran, et al. The lancet global health commission on global eye health: vision beyond 2020. *The Lancet Global Health*, 9(4):e489–e551, 2021.

[4] Joohee Kim, Jihun Park, Young-Geun Park, Eunkyung Cha, Minjae Ku, Hyeon Seok An, Kyoung-Pil Lee, Man-Il Huh, Junmo Kim, Taek-Soo Kim, et al. A soft and transparent contact lens for the wireless quantitative monitoring of intraocular pressure. *Nature Biomedical Engineering*, 5(7):772–782, 2021.

[5] Kevin Gillmann, Robert N Weinreb, and Kaweh Mansouri. The effect of daily life activities on intraocular pressure related variations in open-angle glaucoma. *Scientific reports*, 11(1):6598, 2021.

[6] Da Young Shin, Kyoung In Jung, Hae Young Lopilly Park, and Chan Kee Park. The effect of anxiety and depression on progression of glaucoma. *Scientific Reports*, 11(1):1769, 2021.

[7] Guo-Zhen Chen, Ion-Seng Chan, Leo KK Leung, and David CC Lam. Soft wearable contact lens sensor for continuous intraocular pressure monitoring. *Medical engineering & physics*, 36(9):1134–1139, 2014.

[8] Tae Yeon Kim, Jee Won Mok, Sang Hoon Hong, Sang Hoon Jeong, Hyunsik Choi, Sangbaie Shin, Choun-Ki Joo, and Sei Kwang Hahn. Wireless theranostic smart contact lens for monitoring and control of intraocular pressure in glaucoma. *Nature Communications*, 13(1):6801, 2022.

[9] Tae Yeon Kim, Sangbaie Shin, Hyunsik Choi, Sang Hoon Jeong, David Myung, and Sei Kwang Hahn. Smart contact lenses with a transparent silver nanowire strain sensor for continuous intraocular pressure monitoring. *ACS Applied Bio Materials*, 4(5):4532–4541, 2021.

[10] Yushi Zhang, Yufeng Chen, Tianxing Man, Dong Huang, Xiao Li, Hongwei Zhu, and Zhihong Li. High resolution non-invasive intraocular pressure monitoring by use of graphene woven fabrics on contact lens. *Microsystems & nanoengineering*, 5(1):1–8, 2019.

[11] Yu Pang, Yuxing Li, Xuefeng Wang, Chenjie Qi, Yi Yang, and Tian-Ling Ren. A contact lens promising for non-invasive continuous intraocular pressure monitoring. *RSC advances*, 9(9):5076–5082, 2019.

[12] Jiandong Xu, Tianrui Cui, Thomas Hirtz, Yancong Qiao, Xiaoshi Li, Fanhui Zhong, Xiaolin Han, Yi Yang, Sheng Zhang, and Tian-Ling Ren. Highly transparent and sensitive graphene sensors for continuous and non-invasive intraocular pressure monitoring. *ACS applied materials & interfaces*, 12(16):18375–18384, 2020.

[13] Vladimir Laukhin, Irene Sánchez, Ana Moya, Elena Laukhina, Raul Martin, Fernando Ussa, Concepció Rovira, Antón Guimera, Rosa Villa, Jordi Aguiló, et al. Non-invasive intraocular pressure monitoring with a contact lens engineered with a nanostructured polymeric sensing film. *Sensors and Actuators A: Physical*, 170(1-2):36–43, 2011.

[14] Matteo Leonardi, Peter Leuenberger, Daniel Bertrand, Arnaud Bertsch, and Philippe Renaud. First steps toward noninvasive intraocular pressure monitoring with a sensing contact lens. *Investigative ophthalmology & visual science*, 45(9):3113–3117, 2004.

[15] Yanyan Fan, Hailing Tu, Hongbin Zhao, Feng Wei, Yi Yang, and Tianling Ren. A wearable contact lens sensor for noninvasive in-situ monitoring of intraocular pressure. *Nanotechnology*, 32(9):095106, 2020.

[16] Guo-Zhen Chen, Ion-Seng Chan, and David CC Lam. Capacitive contact lens sensor for continuous non-invasive intraocular pressure monitoring. *Sensors and Actuators A: Physical*, 203:112–118, 2013.

[17] Joohee Kim, Jaeyoon Kim, Minjae Ku, Eunkyung Cha, Seoyoung Ju, Won Yeong Park, Ki Hean Kim, Dai Woo Kim, Per-Olof Berggren, and Jang-Ung Park. Intraocular pressure monitoring following islet transplantation to the anterior chamber of the eye. *Nano letters*, 20(3):1517–1525, 2019.

[18] Joohee Kim, Minji Kim, Mi-Sun Lee, Kukjoo Kim, Sangyoon Ji, Yun-Tae Kim, Jihun Park, Kyungmin Na, Kwi-Hyun Bae, Hong Kyun Kim, et al. Wearable smart sensor systems integrated on soft contact lenses for wireless ocular diagnostics. *Nature communications*, 8(1):1–8, 2017.

[19] Cheng Yang, Qianni Wu, Junqing Liu, Jingshan Mo, Xiangling Li, Chengduan Yang, Ziqi Liu, Jingbo Yang, Lelun Jiang, Weirong Chen, et al. Intelligent wireless theranostic contact lens for electrical sensing and regulation of intraocular pressure. *Nature communications*, 13(1):1–15, 2022.

[20] Jinyuan Zhang, Kyunghun Kim, Ho Joong Kim, Dawn Meyer, Woohyun Park, Seul Ah Lee, Yumin Dai, Bongjoong Kim, Haesoo Moon, Jay V Shah, et al. Smart soft contact lenses for continuous 24-hour monitoring of intraocular pressure in glaucoma care. *Nature communications*, 13(1):1–15, 2022.

[21] M Hossein M Kouhani, Jiajia Wu, Arman Tavakoli, Arthur J Weber, and Wen Li. Wireless, passive strain sensor in a doughnut-shaped contact lens for continuous non-invasive self-monitoring of intraocular pressure. *Lab on a Chip*, 20(2):332–342, 2020.

[22] Matteo Leonardi, Elie M Pitchon, Arnaud Bertsch, Philippe Renaud, and Andre Mermoud. Wireless contact lens sensor for intraocular pressure monitoring: assessment on enucleated pig eyes. *Acta ophthalmologica*, 87(4):433–437, 2009.

[23] Mohammad Hossein Mazaheri Kouhani, Arthur Weber, and Wen Li. Wireless intraocular pressure sensor using stretchable variable inductor. In *2017 IEEE 30th International Conference on Micro Electro Mechanical Systems (MEMS)*, pages 557–560. IEEE, 2017.

[24] Chien-Kai Tseng, Yu-Chieh Huang, Shang-Wei Tsai, Guan-Ting Yeh, Chung-Hao Chang, and Jin-Chern Chiou. Design and fabricate a contact lens sensor with a micro-inductor embedded for intraocular pressure monitoring. In *SENSORS, 2012 IEEE*, pages 1–4. IEEE, 2012.

[25] Martin Dressel and George Grüner. Electrodynamics of solids: optical properties of electrons in matter, 2002.

[26] Anandghan Waghmare, Youssef Ben Taleb, Ishan Chatterjee, Arjun Narendra, and Shwetak Patel. Z-Ring: Single-Point Bio-Impedance Sensing for Gesture, Touch, Object and User Recognition. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–18, 2023.

[27] Wei Sun, Yuwen Chen, Yanjun Chen, Xiaopeng Zhang, Simon Zhan, Yixin Li, Jiecheng Wu, Teng Han, Haipeng Mi, Jingxian Wang, et al. MicroFluID: A Multi-Chip RFID Tag for Interaction Sensing Based on Microfluidic Switches. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 6(3):1–23, 2022.

[28] Goldmann applanation tonometry. https://en.wikipedia.org/wiki/Goldmann_Applanation_Tonometer.

[29] Omar S Punjabi, Christoph Kniestedt, Robert L Stamper, and Shan C Lin. Dynamic contour tonometry: principle and use. *Clinical & experimental ophthalmology*, 34(9):837–840, 2006.

[30] SY Hsu, MM Sheu, AH Hsu, KY Wu, JI Yeh, JN Tien, and RK Tsai. Comparisons of intraocular pressure measurements: Goldmann applanation tonometry, noncontact tonometry, tono-pen tonometry, and dynamic contour tonometry. *Eye*, 23(7):1582–1588, 2009.

[31] Tanner J Ferguson, Catherine G Knier, Uttio Roy Chowdhury, Kjerseten J Monson, Michael Greenwood, Russell J Swan, Richard Gorham, John P Berdahl, and Michael P Fautsch. Intraocular pressure measurement with pneumatonometry and a tonometer tip cover. *Ophthalmology and therapy*, 9:127–137, 2020.

[32] Po-Jui Chen, Saloomeh Saati, Rohit Varma, Mark S Humayun, and Yu-Chong Tai. Implantable flexible-coiled wireless intraocular pressure sensor. In *2009 IEEE 22nd International Conference on Micro Electro Mechanical Systems*, pages 244–247. IEEE, 2009.

[33] Jeong Oen Lee, Haeri Park, Juan Du, Ashwin Balakrishna, Oliver Chen, David Sretavan, and Hyuck Choo. A microscale optical implant for continuous in vivo monitoring of intraocular pressure. *Microsystems & nanoengineering*, 3(1):1–9, 2017.

[34] Jeong Oen Lee, Vinayak Narasimhan, Juan Du, Blaise Ndjamen, David Sretavan, and Hyuck Choo. Biocompatible multifunctional black-silicon for implantable intraocular sensor. *Advanced healthcare materials*, 6(4):1601356, 2017.

[35] Amit Todani, Irmgard Behlau, Mark A Fava, Fabiano Cade, Daniel G Cherfan, Fouad R Zakka, Frederick A Jakobiec, Yuqing Gao, Claes H Dohlman, and Samir A Melki. Intraocular pressure measurement by radio wave telemetry. *Investigative ophthalmology & visual science*, 52(13):9573–9580, 2011.

[36] Zhiduo Liu, Gang Wang, Chen Ye, Hongyan Sun, Weihua Pei, Chunrong Wei, Wen Dai, Zhiqiang Dou, Qingyu Sun, Cheng-Te Lin, et al. An ultrasensitive contact lens sensor based on self-assembly graphene for continuous intraocular pressure monitoring. *Advanced Functional Materials*, 31(29):2010991, 2021.

[37] Yuxuan Liu, Michael Zheng, Brendan O'Connor, Jingyan Dong, and Yong Zhu. Curvilinear soft electronics by micromolding of metal nanowires in capillaries. *Science Advances*, 8(46):eadd6996, 2022.

[38] Jin-Chern Chiou, Yu-Chieh Huang, and Guan-Ting Yeh. A capacitor-based sensor and a contact lens sensing system for intraocular pressure monitoring. *Journal of Micromechanics and Microengineering*, 26(1):015001, 2015.

[39] Ismail E Araci, Baolong Su, Stephen R Quake, and Yossi Mandel. An implantable microfluidic device for self-monitoring of intraocular pressure. *Nature medicine*, 20(9):1074–1078, 2014.

[40] John Yan. An unpowered, wireless contact lens pressure sensor for point-of-care glaucoma diagnosis. In *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 2522–2525. IEEE, 2011.

[41] Bohee Maeng, Hyung-kwan Chang, and Jungyul Park. Photonic crystal-based smart contact lens for continuous intraocular pressure monitoring. *Lab on a Chip*, 20(10):1740–1750, 2020.

[42] Hongbin An, Liangzhou Chen, Xiaojun Liu, Bin Zhao, Hong Zhang, and Zhigang Wu. Microfluidic contact lenses for unpowered, continuous and non-invasive intraocular pressure monitoring. *Sensors and Actuators A: Physical*, 295:177–187, 2019.

[43] Sevda Agaoglu, Priscilla Diep, Matthew Martini, KT Samudhyatha, Murat Baday, and I Emre Araci. Ultra-sensitive microfluidic wearable strain sensor for intraocular pressure monitoring. *Lab on a Chip*, 18(22):3471–3483, 2018.

[44] Google lens. https://blog.google/alphabet/introducing-our-smart-contact-lens/.

[45] Nicholas M Farandos, Ali K Yetisen, Michael J Monteiro, Christopher R Lowe, and Seok Hyun Yun. Contact lens sensors in ocular diagnostics. *Advanced healthcare materials*, 4(6):792–810, 2015.

[46] Huanfen Yao, Angela J Shum, Melissa Cowan, Ilkka Lähdesmäki, and Babak A Parviz. A contact lens with embedded sensor for monitoring tear glucose level. *Biosensors and Bioelectronics*, 26(7):3290–3296, 2011.

[47] Mojo. https://www.mojo.vision/technology.

[48] Inwith. https://inwithcorp.com/envisioning-reality/.

[49] Jihun Park, Joohee Kim, So-Yun Kim, Woon Hyung Cheong, Jiuk Jang, Young-Geun Park, Kyungmin Na, Yun-Tae Kim, Jun Hyuk Heo, Chang Young Lee, et al. Soft, smart contact lenses with integrations of wireless circuits, glucose sensors, and displays. *Science advances*, 4(1):eaap9841, 2018.

[50] Andrew R Lingley, Muhammad Ali, Y Liao, Ramin Mirjalili, Maria Klonner, M Sopanen, Sami Suihkonen, Tueng Shen, Brian P Otis, H Lipsanen, et al. A single-pixel wireless contact lens display. *Journal of Micromechanics and Microengineering*, 21(12):125014, 2011.

[51] Jagdish Pandey, Yu-Te Liao, Andrew Lingley, Ramin Mirjalili, Babak Parviz, and Brian P Otis. A fully integrated rf-powered contact lens with a single element display. *IEEE Transactions on Biomedical Circuits and Systems*, 4(6):454–461, 2010.

[52] Liyao Li, Yaxiong Xie, Jie Xiong, Ziyu Hou, Yingchun Zhang, Qing We, Fuwei Wang, Dingyi Fang, and Xiaojiang Chen. Smartlens: sensing eye activities using zero-power contact lens. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, pages 473–486, 2022.

[53] Do Hee Keum, Su-Kyoung Kim, Jahyun Koo, Geon-Hui Lee, Cheonhoo Jeon, Jee Won Mok, Beom Ho Mun, Keon Jae Lee, Ehsan Kamrani, Choun-Ki Joo, et al. Wireless smart contact lens for diabetic diagnosis and therapy. *Science advances*, 6(17):eaba3252, 2020.

[54] Carolin Tetyczka, Kira Brisberger, Martin Reiser, Manuel Zettl, Ramona Jeitler, Christina Winter, Dagmar Kolb, Gerd Leitinger, Martin Spoerk, and Eva Roblegg. Itraconazole nanocrystals on hydrogel contact lenses via inkjet printing: implications for ophthalmic drug delivery. *ACS Applied Nano Materials*, 5(7):9435–9446, 2022.

[55] Meysam Habibi, Saleh Mobasseri, Azam Zare, and Vahid Souriaee. Drug delivery with therapeutic lens for the glaucoma treatment in the anterior eye chamber: a numerical simulation. *Biomedical Engineering Advances*, 3:100032, 2022.

[56] Woohyun Park, Van Phuc Nguyen, Yale Jeon, Bongjoong Kim, Yanxiu Li, Jonghun Yi, Hyungjun Kim, Jung Woo Leem, Young L Kim, Dong Rip Kim, et al. Biodegradable silicon nanoneedles for ocular drug delivery. *Science Advances*, 8(13):eabn1772, 2022.

[57] Sijin Li, Andreea Pasc, Vanessa Fierro, and Alain Celzard. Hollow carbon spheres, synthesis and applications–a review. *Journal of Materials Chemistry A*, 4(33):12686–12713, 2016.

[58] Zhiguang Qiu, Yongbiao Wan, Wohua Zhou, Jingyi Yang, Junlong Yang, Jun Huang, Jianming Zhang, Qingxian Liu, Siya Huang, and Ningning Bai. Ionic skin with biomimetic dielectric layer templated from calathea zebrine leaf. *Advanced Functional Materials*, 28(37):1802343.1–1802343.9, 2018.

[59] Magnus s3. https://www.axzon.com/packagedQFN.html.

[60] Jin-Woong Kim and Kyung-Do Suh. Monodisperse micron-sized polystyrene particles by seeded polymerization: effect of seed crosslinking on monomer swelling and particle morphology. *Polymer*, 41(16):6181–6188, 2000.

[61] teato 510i. https://www.testo.com/en-US/testo-510i/p/0560-1510.

[62] Hioki im7587. https://www.hioki.com/in-en/products/lcr-meters/3-ghz/id_6618.

[63] R&s znb8 vector network analyzer. https://www.rohde-schwarz.com/us/products/test-and-measurement/network-analyzers/rs-znb-vector-network-analyzer_63493-11648.html.

[64] Thingmagic mercury6 rfid reader. https://www.barcodediscount.com/catalog/thingmagic/part-m6-na-poe.htm.

[65] Portable ophthalmic equipment. https://sh-fuan.en.made-in-china.com/product/FOQtBMexCNYn/China-Portable-Ophthalmic-Equipment-Veterinary-Tonometer-Cat-Dog-Fa-800vet-Pet-Hospital-Animal-Tonometer.html.

[66] icare ic200. https://www.icare-world.com/us/product/icare-ic200-tonometer/.

[67] Edward Hughes, Paul Spry, and Jeremy Diamond. 24-hour monitoring of intraocular pressure in glaucoma management: a retrospective review. *Journal of glaucoma*, 12(3):232–236, 2003.

[68] Shubin Ma, Toni Björninen, Lauri Sydänheimo, Merja H Voutilainen, and Leena Ukkonen. Double split rings as extremely small and tuneable antennas for brain implantable wireless medical microsystems. *IEEE Transactions on Antennas and Propagation*, 69(2):760–768, 2020.

A Tag fabcication workflow

The manufacturing process of our tag mainly consists of three steps: synthesis of UHCS, *Calathea zebrine* mold casting, and tag production. The specific details are as follows:

A.1 Calathea Zebrine Template.

Step 1: Preparing Leaves. Take the fresh *Calathea zebrine* leaves and cut them into suitable rectangular shapes (5 cm × 5 cm) from the center of the whole leaves, removing the leaf edges to ensure a consistent microstructure on the surface. Then washed them at least five times with deionized water to remove surface impurities. Afterward, place the leaves in a drying oven to remove surface moisture.

Step 2: Casting the Template. Multiple vent holes are manually created on the back of the prepared *Calathea zebrine* leaves, and the leaves are fixed with the front side up on a glass substrate. We prepared a PDMS solution by mixing PDMS base and curing agent in a ratio of 5:1 and poured onto the *Calathea zebrine* leaves. The assembly is placed in a vacuum chamber at 20 $^{\circ}C$ until the bubbles disappear. After the bubbles disappear, the temperature is raised to 80 $^{\circ}C$ for curing for two hours.

Step 3: Plasma Oxygen Treatment. After peeling off the *Calathea zebrine* leaves, we obtain a template with the sunken structure. Then we expose the formed template to air plasma treatment for one minute. This step involves a hydrophobic treatment process that prevents materials from sticking to the template. The sunken structure template can be reused multiple times.

A.2 Synthesis of UHCS.

Step 1: Synthesis the Polystyrene (PS) Nanoparticle. We use the seeded swelling polymerization method to synthesis the polystyrene nanoparticle. Specifically, we added 20 g of styrene, 0.2 g of 2,2-azo-bisisobutyronitrile, and 1.8 g of polyvinyl pyrrolidone into a mixture of 60.4 g of ethanol and 7.6 g of deionized water. The mixture is then stirred under a nitrogen atmosphere at 900 rpm for one hour. Then heated to 70 $^{\circ}C$ for 24 hours. After vacuum filtration and drying, we obtained the powdered PS nanoparticles.

Step 2: Synthesis the PS Polyaniline Core-shell Spheres. We take 0.3 g of the PS nanospheres and wash them using deionized for five times. Then, we disperse them into 20 mL of deionized water. Subsequently, we add 0.6519 g of aniline and stir it at 100 rpm for 5 hours to initiate the polymerization of aniline. Following this step, we add 84 mL of 0.5 M $Fe(NO_3)_3$ aqueous solution into the reaction and stir it at 300 rpm for 24 hours at room temperature. The 0.5 M $Fe(NO_3)_3$ can help to form the aniline spines on the PS surface.

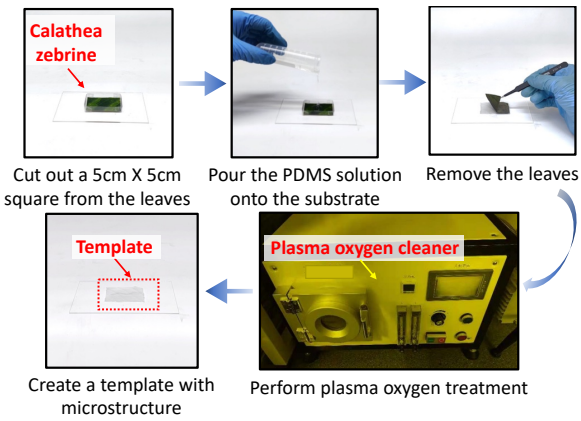


Figure 30. The template fabrication workflow.

Step 3: Synthesis the UHCS. During the last step, the PS polyaniline core-shell spheres are obtained. These spheres need to be washed at least five times to remove the $Fe(NO_3)_3$. Subsequently, the core-shell spheres are placed in a vacuum oven at 40 $^{\circ}C$ for 48 hours to facilitate drying and remove any remaining moisture. Then, the core-shell spheres are placed into the tube furnace and heated at 350 $^{\circ}C$ under an Ar atmosphere for one hour to eliminate the PS cores. In this step, we need to control the heating speed, gradually heating to 350 $^{\circ}C$ at a rate of 1 $^{\circ}C/min$. If the temperature increase too fast, the gaseous PS cores will suddenly explode from inside the carbon sphere. The carbon sphere will broken due to the explosion, just as shown in Figure 32. After remove the PS core, the temperature is then increased to 900 $^{\circ}C$ at a rate of 2 $^{\circ}C$ per minute for carbonization in order to obtain UHCS.

A.3 UHCS-PDMS Film

Take 3 g of PDMS solution and 53.5 mg of UHCS powder, string in a 20 $^{\circ}C$ for one hour to mix them thoroughly. Then pour the mixture onto the *Calathea zebrine* template and spin-coated the solution to obtain a UHCS-PDMS film with a thickness of 60 μm. Next, heat it to 80 $^{\circ}C$ for one hour to thermal curing. After peeling off the film from the template, the UHCS-PDMS film is formed with array of micocones.

We stack the top layer of the antenna and the UHCS-PDMS film with the bottom layer of the antenna. The chip is connected to the top layer of the antenna using the conductive silver paste. Then we coated the structure with the PDMS and heated at 80 $^{\circ}C$ for 10 min, allowing the PDMS to partially solidify. While in this semi-solidified state, the PDMS

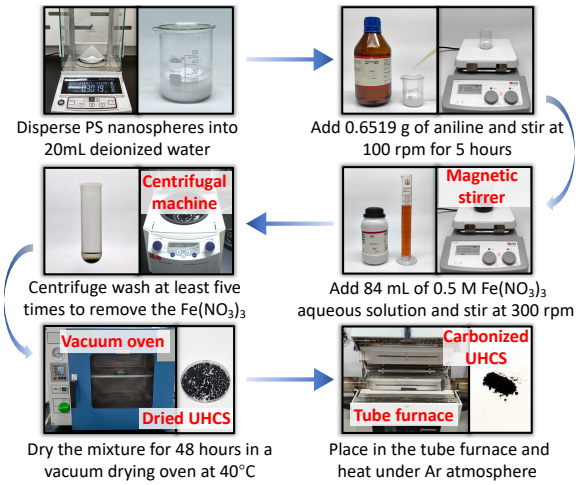


Figure 31. The UHCS synthesis workflow.

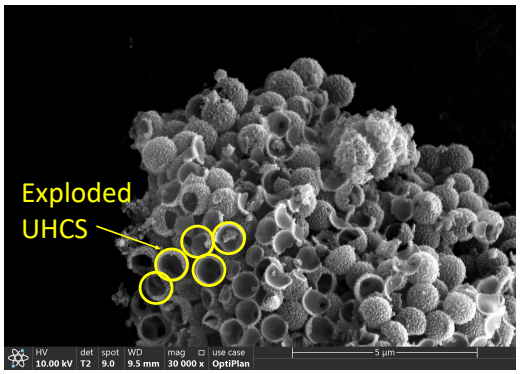


Figure 32. The scanning electron microscopy (SEM) image of the exploded UHCS.

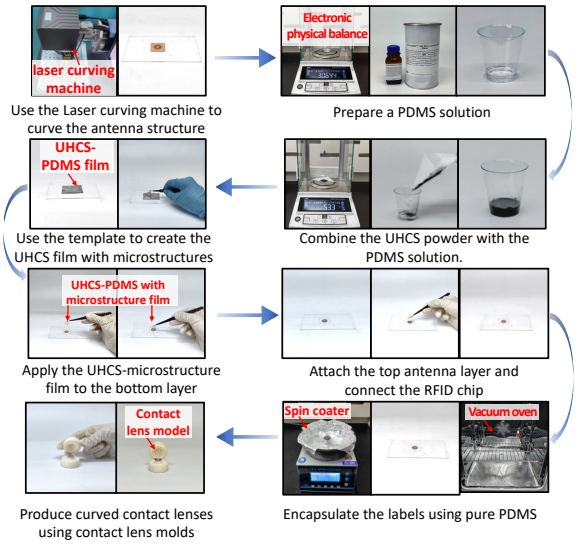


Figure 33. The Cyclops contact lens fabrication workflow.

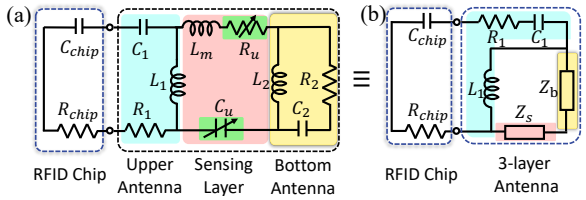


Figure 34. The equivalent circuit of the Cyclops lens antenna.

is transferred to a contact lens mold for thermal curing at 80 $^{\circ}C$ for one hour. Finally, we meticulously detach the cured antenna from the mold, remove the excess edges, and obtain the contact lens tag.

B Impedance of the Cyclops lens Antenna

The equivalent circuit of the Cyclops lens antenna is shown in Figure 34(a). Where R_1 and R_2 denote the parasitic resistance of the upper layer and the bottom layer antenna. L_1 and L_2 represent parasitic capacitance, and C_1 and C_2 represent parasitic inductance of two antenna layers. L_m is the mutual-inductance between two antenna layers. C_u and R_u are variable capacitor and resistor, changing as the applied pressure varies.

From the equivalent circuit, we can observe that the C_1, L_1 and R_1 are connected in series. L_m and R_u are in series and they are parallel connected with C_u. On the right side of the equivalent circuit, R_2 and C_2 are connected in series and the L_2 is connected parallel to them. In order to simplify the entire circuit, we use Z_b to represent the impedance of the bottom layer, which can be represented as:

$$\begin{aligned} Z_b &= (-\frac{1}{j\omega C_2} + R_2)//j\omega L_2 \\ &= (\frac{R_2 \cdot j\omega C_2 - 1}{j\omega C_2})//j\omega L_2, \end{aligned} \tag{9}$$

where // represents the parallel relationship between the left-hand and right-hand sides of the formula. To calculate the total impedance of the bottom layer, we need to calculate the reciprocal of each individual branch impedance. Then we sum the reciprocals together. Finally, the total impedance of the parallel circuit Z_b is determine by taking the reciprocal of the resulting sum. The derivation process is as follows:

$$\begin{aligned} Z_b &= \frac{1}{\frac{j\omega C_2}{R_2 \cdot j\omega C_2 - 1} + \frac{1}{j\omega L_2}} \\ &= \frac{1}{\frac{j\omega C_2 \cdot j\omega L_2 + (R_2 \cdot j\omega C_2 - 1)}{(R_2 \cdot j\omega C_2 - 1)(j\omega L_2)}} \\ &= \frac{j^2\omega^2 R_2 L_2 C_2 - j\omega L_2}{j^2\omega^2 L_2 C_2 + j\omega R_2 C_2 - 1} \\ &= \frac{\omega^2 R_2 L_2 C_2 + j\omega L_2}{1 + \omega^2 C_2 L_2 - j\omega R_2 C_2}. \end{aligned} \tag{10}$$

As shown in Figure 34(b), we use Z_b represent R_2, L_2 and C_2 in the equivalent circuit. So the Z_b is connected in series with

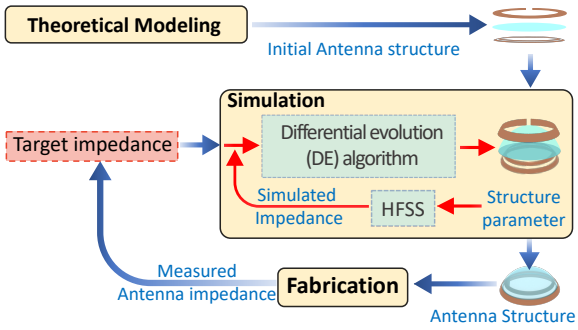


Figure 35. The workflow of fine-tuning the antenna impedance to match the target impedance.

L_m, R_u and C_u. Then we utilize Z_s represent the impedance of the sensing layer, which can be expressed as follows:

$$\mathrm{Z_s} = -\frac{1}{\mathrm{j\omega C_u}} + \mathrm{R_u} + \mathrm{j\omega L_m}. \tag{11}$$

We substitute Z_s and Z_b into the circuit, resulting in an equivalent circuit as shown in Figure 34(b). Z_s and Z_b are connected in series. Both of them are connected parallel with L_1. Then the parallel part is connected series with R_1 and C_1. The impedance of the tag antenna can be represented as:

$$\begin{aligned} \mathrm{Z_{ant}} &= -\frac{1}{\mathrm{j\omega C_1}} + \mathrm{R_1} + \mathrm{j\omega L_1} // (\mathrm{Z_s} + \mathrm{Z_b}) \\ &= -\frac{1}{\mathrm{j\omega C_1}} + \mathrm{R_1} + \frac{1}{\frac{1}{\mathrm{j\omega L_1}} + \frac{1}{\mathrm{Z_s}+\mathrm{Z_b}}} \\ &= -\frac{1}{\mathrm{j\omega C_1}} + \mathrm{R_1} + \frac{\mathrm{j\omega L_1}(\mathrm{Z_s}+\mathrm{Z_b})}{\mathrm{j\omega L_1} + \mathrm{Z_s} + \mathrm{Z_b}}. \end{aligned} \tag{12}$$

C Fine-Tuning Antenna Impedance

Theoretical Modeling. Considering the limited space of the human eyes, we first determine the basic ring shape for our antenna. Taking inspiration from inductively coupled loops [68], we design the antenna as a double rings with a split for connecting the RFID chip. Based on the basic antenna structure, we establish a theoretical model using microstrip transmission line theory. Then we estimate the antenna geometry by building an equivalent transmission line model. Calculate the transmission line resistance, inductance and capacitance based on the estimated antenna dimensions. Through conjugate impedance matching with the chip, we obtain the initial antenna structure just as shown in Fig.35.

Simulation. In practical implementation, our tag antenna requires curvature to fit the contours of the human eye. However, assessing the impact of this curving operation on the antenna's impedance is a challenge. Therefore, we rely on simulation software to replicate the effects of 3D curving and refine our design. We input the initial antenna structure into the simulation software to obtain a simulated antenna impedance. Subsequently, we employ the differential evolution algorithm to guide our parameter search process. After completing the simulation with a series of structure parameters, we obtain the simulated antenna impedance. This impedance data is subsequently utilized within our optimization algorithms, with the objective of achieving the target impedance.

Iterative Fabrication and Simulation. The final step involves fabricating the tag antenna based on the refined antenna structure and evaluate its actual impedance. Real-world manufacturing often introduces engineering errors leading to impedance deviations from the target value. To effectively minimize these deviations, we employ an iterative approach. We first update the simulation objective based on the difference between simulated and measured impedance values. This process iterates until we identify the antenna structure that optimally matches the target impedance, ensuring the best performance.

Habitus: Boosting Mobile Immersive Content Delivery through Full-body Pose Tracking and Multipath Networking

Anlan Zhang[†*], Chendong Wang[◇], Yuming Hu[††], Ahmad Hassan[†*], Zejun Zhang[†*], Bo Han[‡], Feng Qian[†*], Shichang Xu[°]
[†]*University of Southern California,* [††]*University of Minnesota – Twin Cities,*
[‡]*George Mason University,* [◇]*University of Wisconsin – Madison,* [°]*Google*

Abstract

Delivering immersive content such as volumetric videos and virtual/mixed reality requires tremendous network bandwidth. Millimeter Wave (mmWave) radios such as 802.11ad/ay and mmWave 5G can provide multi-Gbps peak bandwidth, making them good candidates. However, mmWave is vulnerable to blockage/mobility and its signal attenuates very fast, posing a major challenge to mobile immersive content delivery systems where viewers are in constant motion and the human body may easily block the line-of-sight.

To overcome this challenge, in this paper, we investigate two under-explored dimensions. First, we use the combination of a viewer's full-body pose and the network information to predict mmWave performance as the viewer exercises six-degree-of-freedom (6-DoF) motion. We apply both offline and online transfer learning to enable the prediction models to react to unseen changes. Second, we jointly use the omnidirectional radio and mmWave radio available on commodity mobile devices to deliver immersive data. We integrate the above two features into a user-space software framework called Habitus, and demonstrate how it can be easily integrated into existing immersive content delivery systems to boost their network performance, which leads to up to 72% of quality-of-experience (QoE) improvement.

1 Introduction

Immersive content, such as virtual/mixed reality (VR/MR) and volumetric videos, allows viewers wearing VR/MR headsets to exercise six-degree-of-freedom (6-DoF) motion (yaw, pitch, roll, X, Y, Z), offering a truly engaging experience [12, 47, 56, 62]. Networked immersive content delivery systems require tremendous network resources (*e.g.,* hundreds Mbps or even Gpbs for high-quality volumetric videos [47, 55, 88]). This poses a major challenge for mobile immersive content delivery systems, which use wireless radios instead of HDMI/USB cables [20, 21] for content delivery.

Recent advances in millimeter wave (mmWave) radio technologies make it feasible to transmit immersive content at a multi-Gbps data rate. mmWave protocols such as 802.11ad [65] (802.11ay [43] in the future) and mmWave 5G [49] have been commercialized on commodity mobile devices. Despite its high data rate, compared to omnidirectional radio, mmWave radio is much more vulnerable to blockage/mobility and its signal attenuates much faster [65]. This creates a major issue for immersive applications where viewers are in constant motion and the human body may easily block the line-of-sight. To overcome this issue, existing systems take three categories of approaches.

- *Improving the PHY layer*. Numerous studies have been conducted on the mmWave radio in general, such as improving MIMO [42] and beamforming [79, 81].
- *Enhancing line-of-sight (LoS)*. Some off-the-shelf commercial products [23] mount the mmWave radio on top of a VR headset to avoid blockages.
- *Using specialized equipment*. Some prior research [27] proposes to deploy multiple reflectors paired with a custom PHY protocol design to improve VR performance over mmWave.

The above approaches help but all have limitations. Engineering the PHY layer alone is inadequate to handle, *e.g.,* the throughput fluctuation incurred by viewers' fast motion. Mounting the radio overhead may still incur frequent non-line-of-sight (NLoS) blockages (*e.g.,* when the viewer looks up/down, raises arms, or passes through an obstacle). Adding reflectors to combat NLoS raises the deployment bar and is incompatible with commodity mmWave protocols. Even in the absence of blockage, highly dynamic user mobility can still cause significant performance drops of mmWave [28, 71, 74].

In this paper, we investigate two complementary, under-explored dimensions to improve the performance of mmWave-based immersive content delivery systems: (1) *full-body-pose guided mmWave throughput prediction* and (2) *joint use of mmWave and omnidirectional radios*. We then integrate them into a holistic middleware framework called Habitus. At a high level, Habitus features a judicious cross-layer design that considers the interplay among viewers' motion, wireless networks, and immersive applications. It creatively leverages features on cheap commodity mobile devices (*e.g.,* dual 802.11ac/ad radios and multi-lens cameras capable of producing stereo images) for affordable high-quality immersive content delivery. Habitus is readily deployable without requiring any change to the existing wireless protocol stack, hardware, or driver. It is orthogonal to and can co-exist with the three categories of solutions described above. The **key challenges** we face include: (1) the dynamics of viewers' motion and mmWave channel incur complex interplay, making accurate throughput prediction difficult; (2) diverse locations and human viewers add more complexity in developing a robust prediction model; even at the same location, the environment

* Work done while at University of Minnesota – Twin Cities.

may change (*e.g.,* a moved chair or a walking spectator); (3) the heterogeneous characteristics of mmWave and omnidirectional radios make their duet difficult.

Full-body-pose Guided mmWave Throughput Prediction (§4). Over a mmWave link, although the throughput fluctuations cannot be completely avoided, they can potentially be predicted to improve the quality-of-experience (QoE) of immersive applications. Habitus utilizes not only the network information, but also viewers' motion to predict mmWave performance. The rationale is that by continuously tracking the 6-DoF motion, an immersive content delivery system can estimate the viewer's future motion trajectory [47, 66, 86], which can then be mapped to the future mmWave performance given the sensitivity of mmWave signal to the physical environment. In particular, we make a new discovery that using the viewer's *full-body pose* (how a person stands, sits, or moves as represented by a set of key points associated with body parts/joints) as features can significantly improve the throughput prediction accuracy, due to the *spatial correlation among body parts* during typical human motion [30]. Motivated by the above, we develop a first-of-its-kind framework that predicts mmWave throughput by jointly leveraging a headset's 6-DoF motion, the viewer's body pose, and network information, through a unified machine learning model. The full-body pose can be captured by a commodity stereo camera conveniently placed, *e.g.,* next to the WiFi AP.

Reacting to Unseen Changes (§5). Using a pre-trained model to predict throughput suffers from a key limitation: it cannot adapt to changes deviating from the training data. We systematically investigate how various types of changes in the immersive streaming context impact the prediction accuracy of a pre-trained model. Based on the insights, we design three orthogonal mechanisms for reacting to different types of unseen changes: (1) *offline transfer learning* handles large changes such as switching to a new location/user; (2) *online transfer learning* updates the model at runtime to tackle smaller changes such as new motion patterns and environmental perturbations; (3) we also leverage the stereo camera to *proactively detect/respond to moving objects* (*e.g.,* a passing person) that affect the mmWave performance.

Joint Use of mmWave and Omnidirectional Radios (§6). Multi-band radio access is a common feature on both mobile devices and WiFi APs. For example, the Asus ROG Phone Series [16] support both 802.11ac and ad. Strategically combining them can boost the network performance for metaverse. We design a lightweight yet effective multipath scheduler for immersive content delivery over mmWave and omnidirectional radios (802.11ad and ac in our prototype). It employs two core design ideas. First, it prioritizes the (low-bandwidth but stable) ac path to guarantee the basic user experience, and opportunistically leverages (high-bandwidth but fluctuating) ad whenever possible. Second, it enhances the mmWave throughput prediction through robust statistical trend analysis [39] to facilitate longer-term throughput forecast.

Implementation (§7). Instead of building a monolithic application, we develop the above features as a generic, user-space middleware framework called Habitus. It offers simple interfaces and data-handling paradigms that are compatible with a wide range of existing immersive applications. It also addresses practical system-level challenges, such as accurate throughput measurement of the highly bursty traffic of immersive content delivery. Habitus consists of 3,541 lines of code (LoC). To demonstrate its efficacy, we develop two immersive apps using its API: one is built from scratch in 5.2K LoC; the other is adapted from a state-of-the-art volumetric streaming system [47] by only modifying 47 LoC.

Datasets (§4, §5) and Evaluation (§8). We thoroughly evaluate Habitus through real-world data and deployment.

- We conduct IRB-approved data collection involving 10 representative motion patterns at 4 representative indoor locations from 3 users. This results in a 21-hour dataset that was used to evaluate Habitus's prediction framework.
- We enhance the above dataset with both static and dynamic environmental changes in a reproducible manner (*e.g.,* using a robotic arm to programmatically inject NLoS, see our demo video [2]), to evaluate Habitus's reaction to changes.
- Using full-body poses reduces 802.11ad throughput prediction error by up to 29% (25%) in MAE (RMSE), compared to using only 6-DoF head motions. This translates to an average QoE improvement of 29% for volumetric content delivery.
- Habitus effectively responds to unseen changes. The offline transfer learning reduces the model training time by 36% to 55% compared to building the model from scratch when switching to a new location or user. The online transfer learning can adapt to a new motion pattern or a typical static environmental change in 32 secs and 15 secs, respectively. By proactively detecting and responding to moving objects, Habitus reduces the volumetric streaming stall by 7%.
- Our multipath solution boosts the average volumetric video quality by 67%, reduces the stall by 64%, and improves the QoE by 72%, compared to using 802.11ad alone. Compared to a recent multipath solution for 802.11ac/ad [71], Habitus reduces the stall by 58% and boosts the quality by 19%.
- We conduct another IRB-approved user trial where we collect 12 viewers' subjective feedback when watching volumetric content. The average ratings for 802.11ad only (basic prediction), 802.11ac+ad (no ad prediction), 802.11ad+ac (basic ad performance prediction), and the full Habitus system (with multipath and full-fledged ad prediction) are 2.67, 2.75, 3.08, and 3.50, respectively (in a 1–5 scale).

Habitus represents to our knowledge a first software framework aiming at optimizing the upper-layer network protocol stack for immersive content delivery (and metaverse applications in general). This paper makes three-fold contributions: the design of the Habitus framework; its implementation, evaluation, and integration into two volumetric content delivery systems; and the release of data [6] (802.11ac/ad performance correlated with full-body motion) and source code [5].

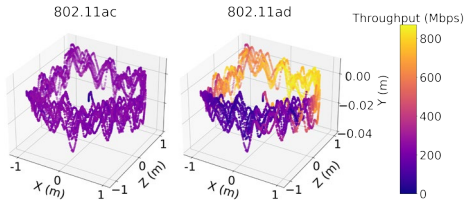


Figure 1: Spatial heatmap of 802.11ac/ad throughput in a room.

2 Background and Motivation

Despite their great potentials on boosting immersive content delivery [47, 55], mmWave signals suffer from increased attenuation, mobility, and blockages [65]. In contrast, 802.11ac operates at the 5GHz band with omnidirectional signal propagation, providing lower but more stable throughput than 802.11ad. In a case study conducted in a personal office (Figure 3), we investigate how the PHY properties of ac and ad affect the QoE of immersive content delivery. Using a smartphone [16] mounted on the user's head and a volumetric video streaming application [47], we measure the QoE while the user walks around the room. Results show that using 802.11ad greatly improves the content quality by 113%, but also hugely increases the video stall by 502% due to its fluctuating throughput under user mobility (Figure 1). Our case study reveals *that ac and ad have distinct network performance due to their complementary PHY properties, and motivates us to strategically combine them to enhance the QoE of immersive content delivery.*

Many studies focus on improving the communication quality of mmWave on the PHY layer [42, 79, 81], while ignoring contextual information for immersive content delivery where a viewer's full body is constantly in motion. On the other side, although solutions with application domain knowledge (*e.g.,* [23, 27]) have shown some effectiveness, they are either incompatible with existing PHY-layer protocols [27] or still suffer from significant LoS blockages [23]. Such a gap motivates us to propose solutions that *judiciously leverage viewer's full-body motion to facilitate mmWave performance forecast, while being compatible to commercial mmWave protocols (802.11ad, mmWave 5G/6G,* etc.*) and easily integrable into diverse immersive applications.*

3 Habitus Overview

Habitus is a software framework enabling immersive content delivery applications to better interact with heterogeneous off-the-shelf wireless networks. It offers two essential features: *accurate runtime mmWave throughput prediction* and *multipath networking* over mmWave (*e.g.,* 802.11 ad) and omnidirectional radio (*e.g.,* 802.11 ac). We assume that the bottleneck is the last-mile radio link(s). Figure 2 shows the workflow of Habitus. As the viewer is watching immersive content, Habitus collects various features in real-time and sends them to an edge node. The edge employs a machine learning model to perform accurate mmWave throughput predictions. The prediction results are then utilized by the application and Habitus's multipath scheduler to determine the appropriate content quality level and how to distribute the content over diverse radio links, respectively. Habitus works on top of the transport layer as a middleware. It is ready for deployment without requiring any changes to the existing wireless protocol stack, hardware, or drivers.

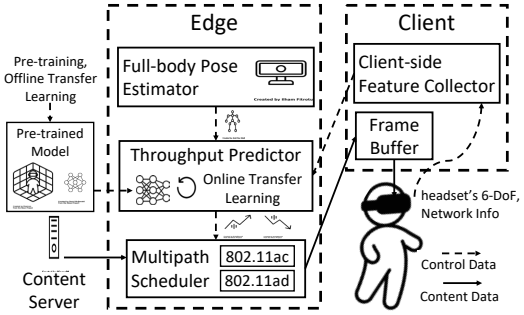


Figure 2: The system architecture of Habitus.

Developing Habitus brings us several challenges: (1) **How to accurately predict mmWave throughput at runtime?** Due to its sensitivity to blockage and mobility, mmWave throughput is difficult to predict, especially under user mobility. (2) **How to ensure the robustness of the prediction models?** A pre-trained model may not be able to handle various unseen changes such as moving to a new location, switching to a different user, or even smaller environmental perturbations. (3) **How to make multipath scheduling efficient and intelligent?** mmWave and omnidirectional radios have their unique natures. How to strategically use these properties to facilitate multipath scheduling is another critical problem.

Habitus tackles the above challenges with 3 core designs.

• **Full-body-pose Guided mmWave Throughput Prediction (§4).** Habitus enhances mmWave throughput prediction by exploiting the viewer's full-body pose, along with the headset's 6-DoF motion and network information, as the features. It employs a cheap stereo camera with off-the-shelf computer vision techniques to capture viewers' poses. We systematically demonstrate the benefit of leveraging body pose through a 21-hour dataset collected at 4 diverse locations.

• **Reacting to Unseen Changes (§5).** Habitus reacts to unseen changes in training data via three orthogonal approaches: offline transfer learning for location/user changes, online transfer learning for new motion patterns and small environmental changes, and proactively detecting/responding to moving objects (*e.g.,* a passing person) affecting mmWave performance.

• **Joint Use of mmWave and Omnidirectional Radio (§6).** Habitus employs the omnidirectional radio that provides stable throughput as a basis. It then opportunistically takes advantage of the fluctuating mmWave radio. It exposes simple, generic interfaces to a wide range of immersive applications.

4 Full-body-pose Guided mmWave Throughput Prediction

Habitus utilizes not only the network information, but also viewers' motion as features for mmWave throughput predic-

tion. It is based on our two observations. (1) The fast signal attenuation and vulnerability to LoS blockages make mmWave throughput highly correlated with the headset's physical position and orientation [65, 83]. Both of them can be predicted from the viewer's historical 6-DoF motion trajectory [47]. (2) Various body parts exhibit spatial correlation [30]. It provides opportunities to enhance the headset motion prediction, which can facilitate mmWave throughput prediction.

4.1 Full-body Pose Estimation

Full-body Pose Representation and Retrieval. Typically, a full-body pose can be represented by a set of key points where each key point corresponds to a joint/part of the human body. We represent the viewer's full-body pose with 15 key points, covering the nose, neck, shoulders, elbows, wrists, hips, knees, and ankles. More details can be found in Appendix A.1.

Some commercial products (*e.g.,* smart suit [17] and body-mounted sensors [22]) allow tracking the full-body pose, but they are expensive and uncomfortable to wear. Habitus instead employs a cheap and easy-to-deploy approach to capture/track the viewer's full-body pose through a stereo camera. It first applies a machine learning model [29, 31, 33, 34, 60, 68] to the RGB frame to estimate the 2D key points of the full-body pose, each comprised of a 2D coordinate and a confidence value w ($0 \le w \le 1$). Habitus then maps the 2D key points to 3D space using the depth map [45] that is generated from stereo images, and keeps their confidence values unchanged.

Estimating Missing Key Points. In some cases, for example, when some parts of the body are outside the stereo camera's viewport, we are not able to capture their key points. We observe from our dataset in §4.2 that for 86% of time, the ML model can retrieve at least 10 (out of 15) key points. The 90-th percentile, mean, and median duration of a key point's missing time are 1s, 0.43s, and 0.07s, respectively. This indicates that in most cases, a key point misses for a very short duration. We estimate a missing key point's 3D coordinate on the fly using a combination two approaches: reusing its most recently captured 3D coordinate (when the missing period is short), and linearly extrapolating its coordinate using its historical trajectory (when the missing period is long). The detailed design and evaluation can be found in Appendix A.2.

4.2 Data Collection

We perform a first-of-its-kind study on full-body-pose assisted throughput prediction in real-world settings. We conduct an IRB-approved data collection involving 10 motion patterns at 4 indoor locations from 3 users, resulting in a 21-hour dataset consisting of both the network data and viewers' motion data. This unique dataset is used to evaluate mmWave throughput prediction (§4.3), techniques for tackling unseen changes (§5), and the Habitus system (§8).

4 Indoor Locations. We investigate 4 representative indoor locations with diverse environments (Figure 3). More details can be found in Appendix A.3. **3 Users.** We recruit 3 users with different heights (1.6m, 1.7m, and 1.8m) and genders (1 female and 2 males) to collect data in all the four locations.

Patterns	Description
S1	The user stands in the center of the room, turning around in a clockwise direction.
S2	The user stands in the center of the room, turning around in a counterclockwise direction.
S3	The user walks around in a clockwise direction.
S4	The user walks around in a counterclockwise direction in a normal speed.
S5	The same as S4, but in a slow speed.
S6	The same as S4, but in a fast speed.
S7	A chair occupies the front place of the access point. The user walks around in a counterclockwise direction.
S8	The same as S3, but the user does not change the orientation of his/her head.
S9	The same as S4, but the user does not change the orientation of his/her head.
S10	The user walks around following the walking trace in S7, but there is no chair.

Table 1: User motion patterns.

10 Motion Patterns. We consider 10 representative motion patterns when watching immersive contents [47] and summarize them in Table 1. For each motion pattern, we repeat data collection three times.

Dataset Overview. Our dataset consists of 12 {Location, User}-specific sub-datasets, each having 30 (10 motion patterns×3 repeats) data traces. The duration of each data trace is 120 secs and the time granularity of each data point is 1/60 secs. Our dataset consists of not only the network information (throughput and signal strength of both 802.11ac and 802.11ad), but also users' motion information (*i.e.,* 6-DoF motion of users' headsets and users' full-body pose, see §4.1). Across all data traces, the average 802.11ad throughput varies from 275 to 886 Mbps, with the standard deviation ranging from 85 to 358 Mbps. Meanwhile, the average 802.11ac throughput varies from 175 to 378 Mbps, with the standard deviation ranging from 26 to 86 Mbps. The highest throughput of 802.11ad only achieves 2.34× of 802.11ac due to the limitation of our hardware setup.

Hardware Setup. We take *Personal Office* in Figure 3 as an example. **For the edge**, we set up a desktop PC with two network interfaces (NICs) at the corner of the room. It has an Intel Core i9-10900X CPU @ 3.70GHz, an NVIDIA 2080Ti GPU, and 32GB memory. Each NIC is connected to an access point by a 1-Gbps Ethernet cable, one [19] for 802.11ac and the other [13] for 802.11ad. The two[1] APs reside on the floor side-by-side. A stereo camera [24] is installed on the wall and connected to the PC via a USB 3.0 cable. It captures users' motion as RGB-D videos at up to 100 FPS. **For the client device**, users wear a headset [11] for collecting 6-DoF motion of their heads. We mount a smartphone [16] that supports both 802.11ac/ad on the headset to collect network information.

[1] Ideally one access point is able to handle both ac and ad. We use two access points due to the 1-Gbps speed limitation of our Ethernet cables.

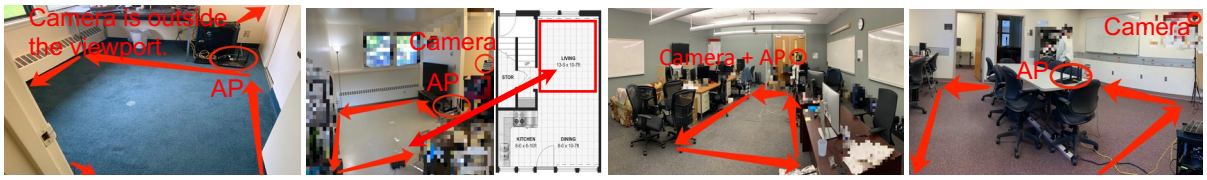


Figure 3: Data collection locations (from left to right: *Personal Office*, *Living Room*, *University Office*, and *Meeting Room*).

We use the same hardware setup for all four locations.

Data Collection Methodology. On the client side, the smartphone establishes two TCP connections with the edge over 802.11ac and 802.11ad, respectively. It performs bulk download from the edge through both paths, and measures the throughput and signal strength (in terms of RSSI). The headset keeps sending the 6-DoF to the smartphone by UDP over 802.11ac. Meanwhile, **on the edge side**, the desktop PC keeps sending the most recent RGB-D frame ID captured by the stereo camera to the smartphone through UDP over 802.11ac for data synchronization purpose (see next).

Data Synchronization and Post-processing. During data collection, the smartphone logs all the information (generated by itself or received from the headset/edge) except the RGB-D video that is recorded at the edge. Since all the devices are in close proximity, the UDP one-way delay is negligible ($\leq$ 2ms) so different pieces of data are properly synchronized. We use zed-openpose [26] (which consumes the RGB-D video) to estimate the user's body pose offline. The pose is synchronized with other information through the RGB-D frame ID, which is recorded by the client at runtime. For key point extraction, we set the input resolution to 320×240 and keep it consistent in our implementation (§7).

4.3 Prediction Methodology and Evaluation

We first formulate our prediction task. Let x_t denote the feature vector at time t, y_t denote the predicted throughput at time t, Δt denote the time granularity, and $\mathbb{M}$ denote the prediction model. Assuming that we perform prediction at time t_0, we have $Y_{t_0,m} = \mathbb{M}(X_{t_0,n})$ where where $X_{t_0,n} = [x_{t_0-(n-1)\times\Delta t}, x_{t_0-(n-2)\times\Delta t}, ..., x_{t_0-1\times\Delta t}, x_{t_0}]$ is the feature sequence within a history window (*hw*) n, and

$$Y_{t_0,m} = \begin{cases} [y_{t_0+m\times\Delta t}] & \text{or} \\ [y_{t_0+1\times\Delta t}, y_{t_0+2\times\Delta t}, ..., y_{t_0+(m-1)\times\Delta t}, y_{t_0+m\times\Delta t}] \end{cases}$$

can be either a single predicted value after a prediction window (*pw*) m or a predicted sequence within the *pw* m where $y_{t_0+i\times\Delta t} (1 \leq i \leq m)$ corresponds to a future timestamp $t_0 + i \times \Delta t$. The *hw* (*pw*) in secs is computed as $n \times \Delta t$ ($m \times \Delta t$).

Habitus uses the full-body pose by taking the coordinates and confidence values (§4.1) of its key points as important features to the mmWave throughput prediction models. We investigate 5 different models from recent studies on mmWave throughput prediction [28, 63]. We customize them to our prediction task by tuning the model architecture and the parameters. We list them as follows. **(1) Gradient Boosting Decision Tree (GBDT) [63].** Our *GBDT* model has 100 estimators, bounded by a depth of size 3. It takes $X_{t_0,1}$ as the input and predicts a single throughput value $Y_{t_0,m}$. **(2) Fully-connected Neural Network (BP) and Recurrent Neural Network (RNN) [28].** *BP8* is a fully-connected neural network with 3 hidden layers, each with 40 neurons. It takes $X_{t_0,8}$ as the input and predicts a single throughput value $Y_{t_0,1}$. *RNN8* and *RNN20* have the same network architecture, *i.e.,* a recurrent neural network with 3 hidden layers, each with 8 or 20 neurons. They take $X_{t_0,8}$ and $X_{t_0,20}$ as the input, respectively, and both predict a single throughput value $Y_{t_0,1}$. **(3) Sequence-to-sequence Learning (Seq2Seq) [37, 63, 75, 77].** Our *Seq2Seq* model has a single-layer LSTM encoder-decoder architecture with 128 hidden units. It takes $X_{t_0,n}$ as the input and predicts future throughput sequence $Y_{t_0,m}$ where $m = 2 \times n$.

We use our dataset (§4.2) to train/evaluate the above models {*w/*, *w/o*} the full-body pose as extra features. We perform 10-fold cross-validation for each model on each {Location, User}'s dataset, and quantify their prediction errors by mean absolute error (MAE) and root mean square error (RMSE). For *Seq2Seq*, to fairly compare it with the other models, we only use the value y_{t_0+pw} in its predicted sequence. We use three prediction windows (*pw*) of {0.5, 1, 2} secs.

Figure 4 shows the average MSE and RMSE for different models across all {Location, User}'s datasets when *pw*=1 sec. The model trained with (without) full-body pose is denoted as *Model w/ Pose* (*Model*) in Figure 4. We also normalize the prediction error by the average 802.11ad throughput (493 Mbps) of our dataset. We have four observations here. (1) *Seq2Seq w/ Pose* achieves the lowest prediction error: 31 (47) Mbps in MAE (RMSE). (2) Leveraging full-body pose as extra features effectively reduces the prediction error for all the models. The reduction ranges from 5% (*GBDT*) to 29% (*RNN20*) in MAE and 5% (*GBDT*) to 25% (*RNN20*) in RMSE. This quantitatively confirms that leveraging spatial correlation among body parts helps boost the throughput prediction accuracy under viewers' constant motion. (3) Although not shown, the benefit of leveraging full-body pose is similar in both simple (*i.e., Personal Office* and *Living Room*) and complex (*i.e., University Office* and *Meeting Room*) environments. (4) Deep learning models of prior work [28] (*RNN20*, *RNN8*, *BP8*) do not necessarily outperform the non-deep-learning model (*GBDT*). This is likely because those in [28] are designed for limited motion (2-DoF) as opposed to the complex 6-DoF motion in real-world settings. We confirm that the above findings also hold for *pw*=0.5s and *pw*=2.0s.

5 Reacting to Unseen Changes

In this section, we investigate how the throughput prediction

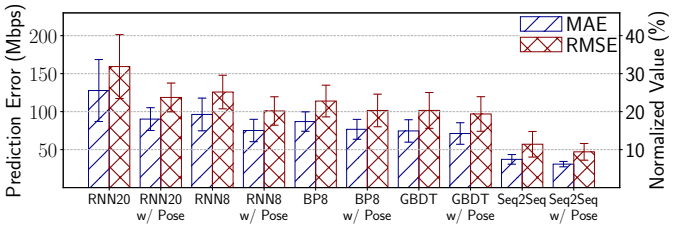


Figure 4: Prediction error of different models {*w/, w/o*} full-body pose as extra features, $pw = 1$ sec.

models developed in §4 react to changes unseen in the training data. This is an important aspect we must consider since the changes are common in practice (*e.g.,* new motion patterns, a chair being moved in the room, another person passing by). We then develop solutions to tackle these changes.

5.1 Measurement Methodology

Recall from §4.3 that we train a model for a particular user's motion patterns in the environment of a particular location. Therefore, a change to any of the above factors may degrade the model's prediction accuracy. We first define these changes. **C1: New Location.** The dimensions, interior design, and furnishings of locations differ vastly, as do the locations of the AP and camera. These variations can significantly impact the behavior of mmWave signal propagation [67, 72, 81, 91]. **C2: New User.** Users differ in their shape and height, resulting in variations of their body poses used by the model. **C3: New Motion Patterns.** Even for the same user at the same location, a new motion pattern may lead to unseen trajectories of the head's motion or the body's pose. **C4: Static Environmental Changes** such as furniture being moved and even small objects being manipulated can affect the mmWave signal propagation [81, 91]. **C5: Dynamic Environmental Changes** are similar to **C4** except that the object(s) that perturb the environment are in motion. A representative scenario is that, people as passerby(s) or spectator(s) can temporarily block the LoS and henceforth cause a throughput drop.

We next describe how to measure the impact of the above changes. For a given change C, we construct three datasets $\mathbb{T}_B$, $\mathbb{T}_A$, and $\mathbb{E}^A$. $\mathbb{T}_B$ and $\mathbb{T}_A$ contain the training data *before* and *after* C, respectively; $\mathbb{E}^A$ contains the testing data collected *after* C for evaluating the impact. We use $\mathbb{T}_B$ and $\mathbb{T}_A$ to train two models $\mathbb{M}_B$ and $\mathbb{M}_A$, respectively, using *Seq2seq w/ Pose* with the same hyper-parameters. Next, we test $\mathbb{M}_B$ and $\mathbb{M}_A$ using $\mathbb{E}^A$, and calculate the corresponding MAE as MAE_B^A and MAE_A^A respectively. Then the impact of C on the throughput prediction accuracy is calculated as $\text{MAE}_B^A - \text{MAE}_A^A$. The tradeoff here is accuracy vs. training overhead: MAE_A^A gives the best accuracy but requires retraining the model; MAE_B^A reuses the old model at the cost of degraded accuracy.

We now describe how to construct $\mathbb{T}_B$, $\mathbb{T}_A$, and $\mathbb{E}^A$ for **C1** to **C5**. Recall from §4.2 that our dataset is divided into (3 users) × (4 locations) = 12 groups (*i.e.,* sub-datasets), and each group contains traces of 10 motion patterns. Also, each group's traces are randomly split into training (70%) and testing (30%). Since a model is created for a given (user u, location l) pair, we measure the impact on a per-group basis, and then average the impact across all groups. For **C1**, for a given group (u,l), $\mathbb{T}_B$ consists of the training data of (u,l); $\mathbb{T}_A$ and $\mathbb{E}^A$ contain the training and testing data of $\cup_{l' \neq l}(u,l')$, respectively. For **C2**, it is similar to **C1** except that $\mathbb{T}_A$ and $\mathbb{E}^A$ belonging to $\cup_{u' \neq u}(u',l)$. For **C3**, the three sets all belong to the same (u,l) pair but they contain different motion patterns: $\mathbb{T}_A$ contains all 10 motion patterns; we remove one motion pattern e from $\mathbb{T}_B$, and only keep e in $\mathbb{E}^A$. We repeat the above measurement 10 times, each time using one of the 10 motion patterns as e. For **C4** and **C5**, the three sets all belong to the same (u,l) pair, but we physically introduce the environmental changes and then recollect data for $\mathbb{T}_A$ and $\mathbb{E}^A$, as elaborated next.

We inject two static environmental changes (**C4**) and study them separately: (1) move four chairs in the room to fixed places (Figure 6 Left), and (2) put four large packages on designated spots. Then we ask the same users to exercise the same motion patterns (as those in $\mathbb{T}_B$) as if there were no environmental change (we ensure that the changes do not block any motion pattern). Injecting a dynamic change (**C5**) in a reproducible manner requires a more sophisticated setup. We use a robotic arm to move a box covered by aluminum foil back and forth between two designated spots to periodically introduce NLoS (Figure 6 Right). The box size and aluminum foil thickness are determined through a separate controlled experiment (details in Appendix §B) to mimic real humans in terms of NLoS-incurred throughput degradation. In this way, we programmatically emulate a passerby intermittently causing NLoS (demo video [2]). We study **C4** and **C5** only at *University Office* due to setup complexity.

5.2 Measurement Results and Insights

Figure 5 plots the impact of **C1** to **C5** on the mmWave throughput prediction accuracy. The two Y axes show both the absolute MAE growth ($\text{MAE}_B^A - \text{MAE}_A^A$) and the relative growth (normalized by the average 802.11ad throughput in our dataset). We highlight our findings next.

A new location incurs the highest impact. C1 degrades the model's accuracy by 19% (93 Mbps) on average. It reveals that the physical property of mmWave and its throughput distribution in a location is the fundamental knowledge learned by our model. Since the indoor location, characterized by its room layout, surface materials, furniture arrangement, *etc.* plays a dominant role in determining the mmWave propagation, changing the location will reshape the throughput distribution at different 3D coordinates.

A new user incurs a moderate impact. C2 increases the average MAE by 8% (38 Mbps). The impact is lower than that of **C1**, but still non-negligible. This suggests that our model also learns how a user's motion affects the throughput received by the headset. Changing the user alters the relative

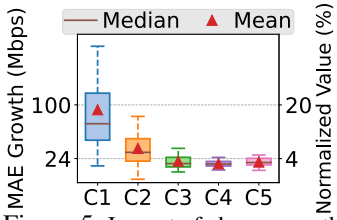


Figure 5: Impact of changes on the model prediction accuracy (*Seq2seq w/ Pose*, $pw = 1$ sec).

Figure 6: Static (left) and dynamic (right) environmental changes, from AP's view.

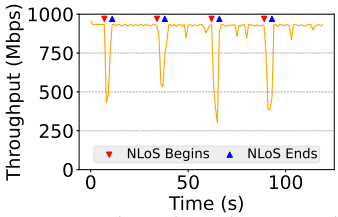


Figure 7: mmWave throughput drop caused by dynamic changes (**C5**) in our demo [2].

positions among the key points used by our model; it also changes the trajectory that the headset can reach even with the same motion pattern. Both degrade the model's accuracy. Nevertheless, the impact is lower than **C1** because the throughput distribution, mostly shaped by the surrounding environment, largely remains consistent.

New motion patterns and static/dynamic environmental changes may incur a small impact. Upon **C3**, **C4**, and **C5**, our model's prediction accuracy only drops marginally, by 4%, 3%, 4% (20, 17, 19 Mbps) on average, respectively. The primary reason is that the changes are spatially small (**C3** and **C4**) or temporarily short (**C5**). For **C3**, a new motion pattern oftentimes has positions overlapped with old ones, making the previously learned knowledge relevant. For our **C4** instances, the moved chairs and packages usually do not incur additional NLoS. Our **C5** instance does incur NLoS, but the overall impact is small due to its short duration (Figure 7). Note that, however, an environmental change may cause a big mmWave performance impact (*e.g.,* there are apparent blockages near the mmWave AP). Users of Habitus (and any mmWave system in general) should avoid such scenarios.

5.3 Methods of Handling Changes

The results in §5.2 suggest that besides taking the time-consuming approach of retraining a model from scratch, Habitus may adopt different strategies to tackle changes. We next introduce three orthogonal mechanisms. The first two (offline/online transfer learning) adopt the concept of *homogeneous transfer learning* [51,59,93] that transfers the knowledge learned from a past experience to a new setting. The two properties below make transfer learning a desirable solution. (1) Before and after the change, the feature space (*i.e.,* mmWave signal strength and throughput, 6-DoF motion, full-body pose) remains the same but their distributions (or domains) may differ [93]; (2) before and after a change, there is invariant knowledge (*e.g.,* the physical property of mmWave and the throughput distribution in certain positions) that can be reused. The third mechanism promptly handles **C5** by fusing real-time computer vision into Habitus.

Offline Transfer Learning. For **C1** and **C2**, given their infrequency and non-negligible impact on the model's performance, collecting data under the new setting to update old the model *before using it* helps mitigate large prediction accuracy drops. Specifically, when switching to a new location or a new user, Habitus asks the (new) user to exercise motion patterns (*e.g.,* those in Table 1) in the (new) location while measuring the mmWave bandwidth and collecting input features. A typical data collection only needs 1 to 2 minutes (see §8.5). Habitus then uses the collected training data to update the old model before starting streaming for the new user or location.

Online Transfer Learning. To handle **C3** to **C5** (and also **C1**, **C2**), since they occur much more frequently with usually a much smaller accuracy impact, Habitus can collect data under the new setting and update the model *on-the-fly*. Unlike offline transfer learning, online transfer learning does not incur additional data collection overhead and is transparent to users. Specifically, during a streaming session, Habitus updates the model in consecutive epochs. Epoch i produces a new model M_i and a set of training data samples D_i (input features and the measured ground truth mmWave throughput). Habitus also maintains a global training dataset D_G. At the beginning of Epoch i, Habitus (1) sets the current model for throughput prediction to M_{i-1}; (2) appends D_{i-1} to D_G, and start using D_G to update M_{i-1}; (3) start collecting D_i that will be used to update M_i in Epoch $i+1$. To bootstrap the above process, Epoch 0 only collects D_0 for a fixed period of 10 secs. To avoid D_G becoming too large, Habitus limits D_G to contain only data collected in the recent 5 minutes. We tune the batch size (64) to balance each epoch's convergence speed and the total model copy overhead after epochs. We also apply a small learning rate (0.001) given that we are fine-tuning the model rather than training it from scratch.

Vision-based Dynamic Change Handling. We find that even online transfer learning is too slow to react to **C5**. We thus devise a heuristic-based design to improve the responsiveness. The idea is to leverage the stereo camera, which already belongs to Habitus's infrastructure, to visually capture dynamic changes and penalize the predicted mmWave throughput accordingly. Specifically, we focus on the most common dynamic change: a person temporarily blocks the LoS between the viewer and mmWave AP. During a streaming session, the edge performs continuous human detection [25, 32, 78]. If a passerby is detected (we know the viewer's position so the viewer will not be confused with the passerby), Habitus uses the detected 3D bounding box, the known position of the mmWave AP, and 6-DoF motion reported by the headset, to determine if the passerby is causing NLoS or may cause NLoS in the near future by examining the distance from the bounding box center to the LoS between the viewer and AP. If so, Habitus adds an empirical

penalty to the mmWave throughput B_{ad} predicted by the model: $B'_{ad} = -\frac{s_{max}-s}{s_{max}-s_{min}} \times B_{ad}$ where s is the observed signal strength; s_{min} and s_{max} denote the typical indoor signal strength range (empirically set to -70 and -30 dBm, respectively). We evaluate the above three methods in §8.6.

6 System Design of Habitus

We now detail the system design of Habitus that leverages the functionalities introduced in §4 and §5 as building blocks.

As shown in Figure 2, except the client-side feature collector, all the other components of Habitus reside on the edge. This helps minimize the energy consumption and heat dissipation on client devices.[2] The choice of the edge is flexible. It can be either a user's own desktop PC, or an edge node co-located with a mmWave 5G base station (*e.g.,* AWS wavelength [1]). The edge also acts as a proxy by forwarding the client's requests to the server and the server's streamed content to the client. Habitus supports both client request and server push. Our prototype uses the former.

6.1 Application Interface

Habitus jointly utilizes mmWave and omnidirectional radios (802.11ad and 802.11ac in our prototype) to deliver immersive content. It exposes simple interfaces to applications.

• Through a callback, Habitus keeps informing the application of the two radio links' *aggregated* bandwidth. The application should ensure that its actual streaming bitrate does not overshoot the aggregated bandwidth. The bandwidth update is at a fast pace (*e.g.,* 30 FPS) to match the viewer's fast motion.

• The application streams immersive contents on a per *data block* basis, which can be flexibly defined by the application based on its semantics. Each block can be independently decoded. We use examples in 6.4 to show that our block-based paradigm is aligned with the design of many existing immersive apps (*e.g.,* 360° videos, volumetric videos, and VR). When the client requests for (or the server pushes) a block, it uses Habitus API to attach two parameters: the block's *priority* and *playback deadline*. Habitus forwards the blocks based on their playback deadline in a FIFO manner, and distributes high-priority and low-priority blocks over the omnidirectional and mmWave radios, respectively. The rationale is that the omnidirectional radio is more reliable, so high-priority blocks get a higher chance of being delivered (and hence decoded and rendered) before its deadline than low-priority blocks.

6.2 Utilizing mmWave Throughput Prediction

Habitus exercises multipath content delivery in two steps. It first estimates the aggregated network bandwidth by treating the multipath ac/ad connections as one logical connection. Second, it splits the block stream over ac and ad paths. We now detail the first step, and describe the second step in §6.3.

Recall that the predicted throughput of 802.11ad, denoted as B_{ad}, can be obtained from our mmWave throughput prediction model (§4, §5). The 802.11ac throughput, B_{ac}, is much more stable and largely not affected by the environment. We therefore simply use the harmonic mean of a past window of 5 secs to predict B_{ac}. Then Habitus predicts the aggregated capacity as $B_{ac} + c \times B_{ad}$. We use c, which we call the *trend-aware coefficient*, to further enhance the 802.11ad throughput prediction by considering the trend of a finite horizon in the future as produced by our model (§4.3). The idea is to analyze the monotonic trend of the predicted throughput sequence. If the trend is increasing (decreasing), we can use the 802.11ad link aggressively (conservatively).

To derive c, Habitus first uses Cox-Stuart Test [39, 70], a lightweight, non-parametric approach, to determine the trend (increase, decrease, or neither). Its details are in Appendix C.

$$c = \begin{cases} 1 + \text{avg}\left(\sum_{i=1}^{\frac{n}{2}} \mathbb{I}\left(z_i < z_{i+\frac{n}{2}}\right) \times \left|\frac{z_{i+\frac{n}{2}} - z_i}{z_i}\right|\right), & \text{if increase trend} \\ 1 - \text{avg}\left(\sum_{i=1}^{\frac{n}{2}} \mathbb{I}\left(z_i > z_{i+\frac{n}{2}}\right) \times \left|\frac{z_{i+\frac{n}{2}} - z_i}{z_i}\right|\right), & \text{if decrease trend} \\ 1, & \text{otherwise} \end{cases}$$

Next, Habitus uses the above formula to compute c by averaging the future changes in the predicted throughput sequence $\{z_i\}$. $\mathbb{I}(x) = 1$ iff x is true (otherwise 0), and n is the length of the predicted throughput sequence. The formula splits the sequence into two sub-sequences in the middle, and computes the normalized increase (decrease) of the second subsequence compared to the first one, on a per-element basis. The normalized changes are then averaged. We empirically set the lower and upper bound of c to 0.5 and 1.25, respectively.

6.3 Multipath Scheduling

Upon receiving the block stream from the server, Habitus splits it over 802.11ac and ad paths. Regarding the splitting mechanism, a straightforward solution is MPTCP [41] or its variants for wireless networks [48, 71, 82]. We reject this design due to three reasons. First, MPTCP exposes to the upper layer a single logical connection whose byte stream is delivered in-order; Habitus instead decouples the two paths that independently deliver blocks. Second, adapting MPTCP's scheduler to Habitus requires kernel modifications. Third, MPTCP is known to cause issues on ad/ac (*e.g.,* throughput drops over ad due to periodical network scans on ac [71]).

To avoid the above issues, Habitus establishes two single-path connections[3] and performs scheduling in the user space. The scheduling logic is straightforward: transmitting high-priority and low-priority blocks over 802.11ac and ad, respectively, with the reason explained in §6.1. Specifically, the server sends to Habitus's edge node the metadata (headers)

[2] For example, in our experiment, running our *Seq2Seq* model (§4.3) on the ROG phone II [16] for only two minutes will trigger an overheating issue.

[3] Our prototype uses TCP. A better design would be using QUIC [54] to avoid head-of-line blocking across blocks within a path under packet losses.

of multiple blocks with the same playback deadline in a single bundle, followed by parallel streams of individual blocks' content. As the blocks' content arrives, the edge distributes them over the two paths according to their priority fields in the metadata and each path's estimated bandwidth. A block only usually uses one path, but a small number of blocks may be split over both paths if one path's bandwidth budget is insufficient. Once a block arrives at the client, it will be immediately passed to the application for decoding and rendering. The server-side transmission, edge-side forwarding, and client-side reception are pipelined.

6.4 Example Use Cases of Immersive Apps

Habitus can be easily integrated with a wide range of immersive applications and content formats as exemplified below.

360° Videos. State-of-the-art 360° video systems [46, 80, 90] spatially segment each panoramic video chunk into tiles. Tiles are selectively transmitted based on the viewport. Each tile naturally maps to a block in Habitus, and its priority can be set to the probability that it will appear in the viewport. Many existing systems already have this metric calculated [35, 66].

Volumetric Videos. The above viewport adaptation technique and henceforth the block/priority assignment also applies to volumetric videos, where a 2D tile becomes a 3D cube consisting of 3D points. Alternatively, since each volumetric frame consists of unstructured points, it can be arbitrarily split into multiple layers each constituting a block in Habitus. A "base layer" with a low-density point cloud can be assigned a high priority; one or more "enhancement layers" each encompassing additional details can be assigned lower priorities.[4]

Generic VR. Networked VR systems either stream raw 3D models [55, 57, 88, 89] or rendered 2D scenes [36, 58]. Depending on the content format, a block can be either a 3D model (or part of it) or a rendered 2D patch. The are several studies/systems on determining the priority of VR content, such as those based on foreground/background [53, 57, 85], the viewing distance [44, 61], and user gaze behaviors [36].

7 Implementation

Our implementation consists of three parts: (1) the main Habitus middleware in 3.5K LoC; (2) a 802.11 throughput measurement module plugged into Habitus; (3) two sample applications using the Habitus API (5.2K LoC, 4.4K LoC).

The Main Habitus System is implemented in C++ and Python. We use ROG Phone II [16] and plug it into a low-end VR headset [7] (costs $26) as the client-side device and the same server used in §4.2 as the edge node. On the client side, we use Linux `iw` [10] to monitor 802.11 signal strength; we use ARCore [3] for 6-DoF motion tracking (based on IMU and camera data [4]). On the edge side, we use PyTorch-1.10.0 [15] for training and transferring our models. For inference, we save the models in TorchScript [18] for C++ execution. We implement the body pose estimator over zed-openpose [26], using a pre-trained model [14, 34] to detect 2D poses. We pipeline the body pose estimation stages (capture, 2D detection, 2D-to-3D mapping). We use the object detection module in ZED SDK 3.8.2 [25] to detect passersby.

802.11 Throughput Measurement Module. Compared to traditional 2D video traffic, immersive content traffic is highly bursty [36, 47, 55]. This poses several challenges for 802.11 (in particular, mmWave) throughput measurement. We thus implement an 802.11 throughput measurement module using Libpcap-1.10.1 [8]. We detail its design in Appendix D.1.

Two Volumetric Streaming Applications using Habitus. To demonstrate how Habitus can benefit real immersive applications, we build two volumetric (point cloud) streaming systems with different logic and complexity using the Habitus API. The first app (**App1**) employs layered encoding of point clouds (§6.4) so each data block corresponds to a (frame, layer) pair. The second app (**App2**) performs viewport adaptation (§6.4) by spatially segmenting each frame (*i.e.,* point cloud) into cubical cells, so each data block constitutes a (frame, cell) pair. We build the first app from scratch in 5.2K LoC, and the second app replicating ViVo [47], a state-of-the-art, visibility-aware volumetric streaming system. For ViVo, we only change 47 LoC for Habitus integration. Both apps are equipped with the same bitrate adaptation algorithm whose details can be found in Appendix D.2.

8 Evaluation

8.1 Experimental Setup

Dataset, Devices, and Models. We use the dataset collected in §4.2 for our controlled experiments. The devices are the same as those used in §7 and §4.2. In §8.2, we use {*GBDT, BP8, RNN8, RNN20, Seq2Seq*} {*w/, w/o*} *Pose* models, and three prediction windows (*pw*): {0.5, 1, 2} secs. Experiments in other sections use *Seq2Seq* {*w/, w/o*} *Pose* with *pw*=1 sec.

Controlled Experiments. To ensure the reproducibility, during our controlled experiments, we replay the headset's 6-DoF motion traces and the signal strength traces on the smartphone, which is connected to the edge via real 802.11ac/ad links. On the edge side, we replay the RGB-D videos for online full-body pose estimation. We emulate ac/ad throughput traces by Linux `tc` [9]. We fix the smartphone static in LoS to the 802.11ad AP to keep a good mmWave signal for throughput emulation. We do not add additional RTT since the client already connects to the edge via real wireless links.

Volumetric Videos. We use three point-cloud-based volumetric videos (denoted as *V1*, *V2*, and *V3*, respectively) throughout our evaluation. {*V1*, *V2*, *V3*} has {2612, 2700, 3000} frames ({~87, 90, 100} secs), respectively. Each frame of them is split into 64 data blocks (§6.1) and details can be found in §7 and Appendix D.2. All the videos are at 30 FPS, encoded into ten quality levels. The highest bitrates are {570, 687, 738} Mbps for {*V1*, *V2*, *V3*}, respectively. Unless otherwise mentioned, the results reported in the remainder of

[4] A similar concept called Scalable Video Encoding (SVC [73]) can be applied to 2D content, albeit at a higher encoding overhead.

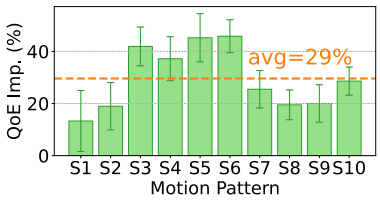


Figure 8: Per motion pattern QoE improvement of *Seq2Seq w/ Pose* over *Seq2Seq w/o Pose*.

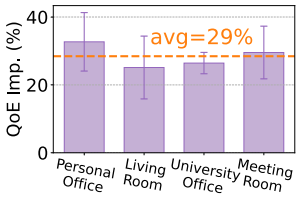


Figure 9: Per location QoE improvement of *Seq2Seq w/ Pose* over *Seq2Seq w/o Pose*.

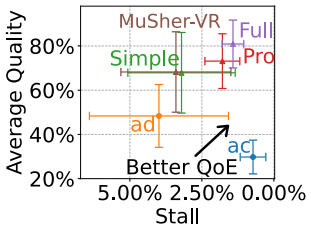


Figure 10: Quality vs. Stall of Habitus variants.

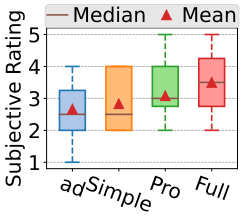


Figure 11: Subjective ratings of 12 users.

this section are generated using all three videos.

Roadmap and Metrics. §8.2 evaluates 802.11ad throughput prediction error reduction brought by full-body pose. §8.3 focuses on the QoE improvement brought by full-body pose. We assess the QoE using the QoE model for point cloud from [88]. It is a linear combination of frame quality, inter-frame & intra-frame quality switch, and stall. §8.4 and §8.7 evaluate the end-to-end performance (quality and stall) of Habitus. §8.5 conducts a user study to examine real users' QoE. §8.6 evaluates how our design handles unseen changes. §8.8 provides additional micro benchmarks. Except for §8.7, we use **App1** in §7 (details in Appendix D.2) for evaluation.

8.2 802.11ad Throughput Prediction Error

Recall that in §4.3, we perform 10-fold cross validation for {*GBDT, BP8, RNN8, RNN20, Seq2Seq*} {*w/, w/o*} *Pose* models on each {Location, User}'s dataset, with three *pw*s {0.5, 1, 2} secs. As Figure 4 shows, leveraging full-body pose effectively reduces mmWave throughput prediction error for all these models, ranging from 5% (*GBDT*) to 29% (*RNN20*) in MAE and 5% (*GBDT*) to 25% (*RNN20*) in RMSE, respectively.

8.3 QoE over 802.11ad Network

We evaluate how full-body pose guided mmWave throughput prediction improves **App1**'s QoE through controlled experiments. First, for each {Location, User}, we train a *Seq2Seq w/ Pose* and a *Seq2Seq w/o Pose* model, respectively. We then run **App1** over a single-path 802.11ad network. For each data trace, we run the experiment twice, using *Seq2Seq* {*w/, w/o*} *Pose* model, respectively. We log the quality for each data block (§8.1) and the stall for each frame to assess the QoE. Figure 8 shows the QoE improvement by leveraging full-body pose for each motion pattern across all data traces. We have two findings. First, leveraging full-body pose effectively improves the QoE by 29% on average for all our motion patterns. Second, the QoE improvement varies across different motion patterns, from 13.30% (S1) to 45.82% (S6). The full-body pose does not help much for S1, S2, S8, and S9. This is due to two reasons. First, in S1 and S2, the user does not make translational movement; this reduces the effectiveness of the pose. Second, in S8 and S9, the LoS between the smartphone and the 802.11ad AP is well maintained; this makes the throughput prediction easier compared to other motion patterns. Figure 9 presents the QoE improvement for each location in Figure 3. The QoE improvement remains similar between simple (*Personal Office* and *Living Room*) and complex locations (*University Office* and *Meeting Room*).

Variant	Predict ad?	*trend-aware*?	Scheduler
ac	No	N/A	single-path ac
ad	No	N/A	single-path ad
Simple	No	N/A	ours in §6.3
Pro	*Seq2Seq w/o Pose*	Yes	ours in §6.3
Full	*Seq2Seq w/ Pose*	Yes	ours in §6.3

Table 2: Habitus variants.

8.4 End-to-end Performance of Habitus

We evaluate the end-to-end performance, including the content quality and stall, of diverse Habitus variants using all {Location, User}'s data and **App1** in §7.

Habitus Variants. Table 2 summarizes 5 Habitus variants. We consider two *single-path* variants, *ac* and *ad*, that only schedule data to ac and ad, respectively, without ad throughput prediction. We also consider three *multipath* variants, all using the multipath scheduler from §6.3: the *Simple* variant does not utilize 802.11ad throughput prediction; the *Pro* and *Full* variant apply the *Seq2Seq w/o Pose* and *Seq2Seq w/ Pose* model, respectively, for ad throughput prediction. Both *Pro* and *Full* enable the *trend-aware* feature (§6.2). As shown in Figure 10, compared to *ac* and *ad*, *Simple* boosts the quality (normalized by the highest quality level) by 127.88% and 40.36%, respectively. *Simple* incurs a much higher stall compared to *ac* because the ad network is highly fluctuating and *Simple* blindly uses it without predicting its future condition. Compared to *Simple*, *Pro* boosts the quality by 7.75% and reduces the stall by 44.25%, thanks to the ad throughput prediction and the *trend-aware* multipath scheduler. Compared to *Pro*, *Full* enhances the ad throughput prediction accuracy by using full-body poses, leading to a further stall reduction of 20.55% and video quality improvement of 10.58%.

Habitus vs. Existing Approaches. We compare Habitus with MuSher [71], a recently proposed MPTCP scheduler for ac/ad networks. Musher periodically probes the ratio between the current ad and ac throughput, and splits the traffic accordingly. In each probe, it tries to increase and decrease the radio, and greedily selects the direction to move based on the aggregated ac/ad throughput measurement. It also has a SCAN component to mitigate the negative impact of network scanning and a BLOCKAGE component to accelerate TCP congestion window recovery after an ad blockage event.

We implement MuSher's scheduling algorithm in the appli-

cation layer. We plug it into Habitus and denote it as *MuSher-VR*. We do not implement the SCAN component because we use establish separate TCP connections over ac/ad links so network scans on one interface do not affect the other one. We repeat the same experiment on *MuSher-VR*.As shown in Figure 10, compared to *MuSher-VR*, Habitus (*Full*) significantly reduces the stall by 58.24% and boosts the quality by 18.52%. Habitus outperforms *MuSher-VR* due to two reasons. First, *MuSher-VR* incurs stalls when it aggressively probes the scheduling ratio by scheduling more data to one path than its actual capacity. Habitus instead takes a prediction-based approach to avoid the stall caused by aggressive probing. Second, Habitus prioritizes using the ac path and opportunistically uses the ad path if possible. In contrast, *MuSher-VR* lacks such prioritization. It schedules the data to the ac/ad paths based on a calculated ratio that ideally should converge to the ratio between ac/ad throughput. However, under the constant movement of the viewer, the actual instantaneous ratio may significantly deviate from the calculated ratio, leading to stalls or under-utilizing the ad path.

8.5 User Study

We conduct an IRB-approved user study at *University Office* (Figure 3) to assess real users' QoE when using **App1** (§7). We recruit 12 users with various demographics.[5] We let each user watch a video randomly selected from our test videos and then subjectively rate the watching experience through 5 choices {1=very bad, 2=bad, 3=fair, 4=good, 5=very good}. Each user performs the above assessment four times. Each time, we randomly plug a Habitus variant into **App1**. We consider four variants: *ad*, *Simple*, *Pro*, and *Full* as listed in Table 2. Before each user's trial begins, we collect 2 minutes' worth of data from the user to transfer a pre-trained model to the user. We let the user wear a low-end VR headset [7] with a ROG Phone II plugged into it. The user can freely make 6-DoF motions in the room during the study. As shown in Figure 11, compared to {*ad*, *Simple*, *Pro*}, *Full* improves the average subjective rating by {0.83, 0.67, 0.42} (in the scale of 1 to 5), respectively. Note that the best scheme (*Full*) has an average rating of 3.50 (between fair and good), likely because of the hardware limitation of the VR headset (costs $26) compared to a full-fledged VR headset.

8.6 Handling Unseen Changes

We evaluate the three techniques introduced in §5.3 for handling unseen changes. We reuse the datasets $\{\mathbb{T}_B, \mathbb{T}_A, \mathbb{E}^A\}$ and models $\{\mathbb{M}_B, \mathbb{M}_A\}$ introduced in §5.1. The experiments use the *Seq2Seq w/ Pose* model on an NVIDIA 1660Ti GPU.

Offline Transfer Learning. For **C1** and **C2**, we compare the training time between (1) transferring from $\mathbb{M}_B$ to $\widetilde{\mathbb{M}}_{B\to A}$ and (2) training a new model $\widetilde{\mathbb{M}}_A$ from scratch after the change. $\widetilde{\mathbb{M}}_{B\to A}$ and $\widetilde{\mathbb{M}}_A$ denote the transferred model and the built-from-scratch model, respectively. For both $\widetilde{\mathbb{M}}_{B\to A}$ and $\widetilde{\mathbb{M}}_A$, we use $p\% \in \{100, 80, 60, 40, 20\}\%$ of the samples in $\mathbb{T}_A$ to transfer (train) them. Their training stops when the prediction accuracy evaluated on $\mathbb{E}^A$ reaches MAE_A^A (*i.e.,* $\mathbb{M}_A$'s prediction accuracy on $\mathbb{E}^A$). We find that the training always converges even when p is as low as 20%. We show the measured training time in Figure 12, where the dashed red line marks the training time of $\widetilde{\mathbb{M}}_A$ with $p = 100\%$. As shown, to achieve the same evaluation accuracy, $\widetilde{\mathbb{M}}_{B\to A}$ significantly reduces the training time by 36% to 41% (48% to 55%) for **C1** (**C2**) across all five p values, compared to $\widetilde{\mathbb{M}}_A$. In particular, training $\widetilde{\mathbb{M}}_{B\to A}$ using only 40% (20%) of the samples in $\mathbb{T}_A$ is still faster than training $\widetilde{\mathbb{M}}_A$ using all the samples in $\mathbb{T}_A$ for **C1** (**C2**). The reason, as explained in §5.3, is that $\widetilde{\mathbb{M}}_{B\to A}$ effectively reuses the invariant knowledge (*e.g.,* the physical property of mmWave and the throughput distribution in certain positions) that is already present in $\mathbb{M}_B$.

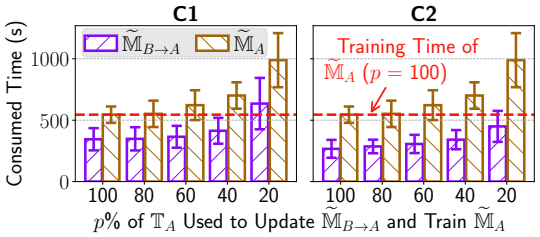


Figure 12: Time consumption for (1) transferring from $\mathbb{M}_B$ to $\widetilde{\mathbb{M}}_{B\to A}$ and (2) training a new model $\widetilde{\mathbb{M}}_A$ from scratch after the change, using p% of $\mathbb{T}_A$. (Left: **C1**; Right: **C2**).

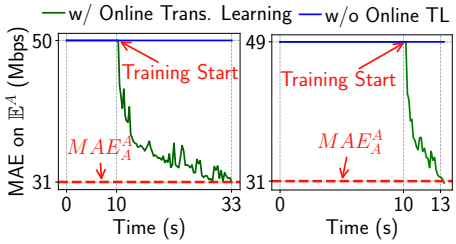


Figure 13: Online training cases (Left: **C3**; Right: **C4**).

Online Transfer Learning. For **C3** and **C4**, we measure the time consumption when $\widetilde{\mathbb{M}}_{B\to A}$ first converges to the target prediction accuracy MAE_A^A on $\mathbb{E}^A$. To accurately emulate the online setting in a reproducible manner, when training $\widetilde{\mathbb{M}}_{B\to A}$, we feed $\mathbb{T}_A$'s data at the same pace as the real-world training data collection rate. The results indicate that it takes on average 32 (15) secs for the training (*i.e.,* online transfer learning) to converge on **C3** (**C4**), with a standard deviation of 11 (12) secs. The convergence time includes the initial 10-sec bootstrapping (§5.3). Figure 13 shows case studies for **C3** and **C4**. Note that without online transfer learning, the prediction error on $\mathbb{E}^A$ will never decrease.

Dynamic Change Handling. For **C5**, we evaluate the end-to-end performance of the Habitus-supported volumetric streaming app (**App1**) *with* and *without* vision-based dynamic change handling (§5.3). The controlled experiment is conducted over a single-path 802.11ad network at *University*

[5] *Gender*: Male: 7, Female: 5; *Height*: <1.65m: 3, 1.65-1.75m: 5, >1.75m: 4. The subjects' ages vary between 20 and 30. 8 out of them do not have prior experience on watching volumetric videos.

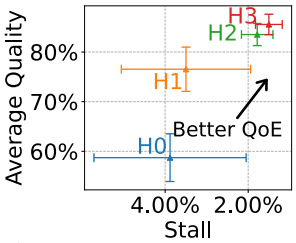


Figure 14: Quality vs. Stall of *H0* to *H3*.

H0	Vanilla ViVo [47] w/ single-path 802.11ad
H1	*H0* and multipath 802.11ac and 802.11ad
H2	*H1* and ad throughput prediction *w/o Pose*
H3	*H2* and ad throughput prediction *w/ Pose*

Table 3: Apply our solution to ViVo [47] (cumulative).

Office with the robotic arm. We pre-train the vision-based object detection model in a separate experiment so the model can reliably detect the aluminum-foil-covered box manipulated by the robotic arm (§5.1). We use $\mathbb{M}_B$ as the throughput prediction model. The results indicate that vision-based dynamic change handling reduces the stall by 7% with a video quality reduction of only 2.2%.

8.7 Applying Habitus to Existing Systems

We integrate Habitus into **App2** (ViVo [47], an existing volumetric streaming system) by changing only 47 LoC (details in Appendix D.2). ViVo adopts visibility-aware streaming by only fetching content that will appear in the future viewport. As listed in Table 3, *H0* is our comparison baseline: the vanilla ViVo over single-path ad. *H1* to *H3* involve key components of Habitus. Figure 14 shows the quality and stall of *H0* to *H3* across our dataset. As shown, by cumulatively enabling Habitus's components from *H1* to *H3*, both the average video quality and stall improve accordingly. Compared to *H0*, *H3* reduces the stall by 61% and improves the average quality by 46%. In addition, compared to not using pose (*H2*), full-fledged Habitus (*H3*) reduces the stall by 15.75% while slightly boosting the quality by 2.44%. The absolute stall rate of *H3* is 1.67%, meaning that the user encounters less than 0.9 secs of stall per minute on average.

8.8 Micro Benchmarks and Resource Usage

We run experiments to show: (1) The GPU memory usage (~4.7G out of 11G on 2080Ti) of Habitus is acceptable. The average processing time of pose estimation and throughput prediction is 27ms and 3.5ms on 2080Ti, respectively. (2) Compared to single-path ac, the additional energy usage and heat increase of **App1** using Habitus are only 1% and 1.3°*C* respectively. The details can be found in Appendix E.

9 Related Work

Immersive Content Delivery. We elaborate on some immersive content delivery systems mentioned in §6.4 (more can be found in [76]). Flare [66] and ViVo [47] apply viewport adaptation to optimize mobile 360° and volumetric videos streaming, respectively. M5 [89] investigates volumetric video streaming using adaptive mmWave beamforming. InstantReality [36] introduces a perceptual-aware approach to enhance VR media streaming. As a middleware framework, Habitus can be integrated into most of the above systems to enhance the application QoE (we have conducted a case study for ViVo in §8.7). Also note that Habitus is orthogonal to some VR systems (*e.g.,* MoVR [27]) that enhances the PHY layer (§1).

mmWave Throughput Prediction. Recent measurement studies have explored the feasibility of predicting mmWave throughput for various radios, such as commercial mmWave 5G [63], 802.11ad [28], and 802.11ay [83]. Lumos5G [63] establishes a composable machine learning framework to predict mmWave 5G throughput. Wu *et al.* uses Markov chain to predict the link quality of 802.11ay, based on the headset's motion data [83]. Aggarwal *et al.* conducts a measurement study on using a smartphone's motion sensor data to predict 802.11ad throughput [28]. They only consider 2-DoF (with a radio mounted on a guided rail) and LoS scenarios. None of the above studies employs the full-body pose, which we found to be an important feature for improving the prediction accuracy. Also, none of them conducts in-depth investigations on how to handle unseen changes as we do.

Multipath TCP Support for 802.11ac/ad. Despite a plethora of works on WiFi/cellular multipath [48,82,92], there are only a few studies on dual-band 802.11ac/ad multipath networking. MUST [74] predicts the best 60GHz beam and PHY rate setting, and switches between ac/ad links accordingly. We discussed MuSher [71], an MPTCP scheduler for 802.11ac/ad and compared it with Habitus in §8.4. Compared to the above works, Habitus is an application-layer solution designed specifically for immersive content delivery.

Improving mmWave Network Performance at PHY layer. There is rich literature on improving mmWave performance at the PHY layer, such as efficient beam selection [79], LiDAR-assisted beam management [81], and beam relay through smart metasurface [38], to name a few. Unlike the above, Habitus aims at optimizing the upper-layer network protocol stack for immersive content delivery without requiring modifying PHY-layer protocols.

10 Limitations and Concluding Remarks

Limitations. First, our prototype and experiments only use 802.11ac+ad. We expect Habitus's high-level design to also work with other radio technologies such as 4G + mmWave 5G, but field tests are needed to verify this claim. Second, Habitus's reaction to unseen changes could be further improved. We plan to employ more advanced techniques such as parameter sharing [93] to speed up transfer learning. Third, we focus on the single-user use case. Extending Habitus to multiple viewers will involve additional challenges such as dealing with the interplay among the viewers.

Despite the limitations, we have demonstrated through a working system and rich real-world data that, full-body-pose guided throughput prediction and joint use of omnidirectional+mmWave radio can significantly improve the QoE (up to 72%) for immersive applications. Furthermore, by fusing transfer learning and vision-based object recognition, Habitus can smoothly adapt to unseen changes.

Acknowledgments

We thank the anonymous reviewers and our shepherd Ravi Netravali for their insightful comments. This work was supported in part by NSF CNS-2212296 and CNS-2235049.

References

[1] AWS Wavelength. https://aws.amazon.com/wavelength/.

[2] Demo Video for Dynamic Change Injection. https://docs.google.com/presentation/d/e/2PACX-1vTokmujVFURIX6YBgyYz9aAjHUV9Ajr1a6dKqYHLmRbrcnI92flVA0O4TLOD338YXXUwyZOjIsSc0Hh/pub?start=false&loop=false&delayms=3000&slide=id.p.

[3] Google ARCore. https://developers.google.com/ar.

[4] Google ARCore Motion Tracking. https://developers.google.com/ar/develop/fundamentals.

[5] Habitus Code. https://github.com/zhan6841/Habitus-open-sourced-code.git.

[6] Habitus Data. https://drive.google.com/drive/folders/1B7rlIQG6ycg2OFML3CHxoPNge56xPhq4.

[7] KCXGHYI VR Headset. https://bit.ly/41UMh19.

[8] Libpcap. https://www.tcpdump.org.

[9] Linux TC Man Page. https://linux.die.net/man/8/tc.

[10] Linux Wireless. https://wireless.wiki.kernel.org/en/users/documentation/iw.

[11] Magic Leap. https://www.magicleap.com/device.

[12] Metaverse. https://about.facebook.com/metaverse/.

[13] Netgear Nighthawk X10. https://www.netgear.com/support/product/r9000.aspx.

[14] OpenPose Codebase. https://github.com/CMU-Perceptual-Computing-Lab/openpose.

[15] Pytorch. https://pytorch.org.

[16] ROG Phone II. https://rog.asus.com/phones/rog-phone-ii-model.

[17] Rokoko Smartsuit Pro II. https://www.rokoko.com/products/smartsuit-pro.

[18] TorchScript. https://pytorch.org/docs/stable/jit.html.

[19] Tp-link Archer A7. https://www.tp-link.com/us/home-networking/wifi-router/archer-a7.

[20] VIVE Cosmos Series. https://www.vive.com/us/product/#cosmos%20series.

[21] VIVE Pro Series. https://www.vive.com/us/product/#pro%20series.

[22] VIVE Tracker 3.0. https://www.vive.com/us/accessory/tracker3.

[23] VIVE Wireless Adapter. https://www.vive.com/us/accessory/wireless-adapter.

[24] ZED 2i Camera. https://www.stereolabs.com/zed-2i.

[25] ZED 3D Object Detection. https://www.stereolabs.com/docs/object-detection/.

[26] zed-openpose Codebase. https://github.com/stereolabs/zed-openpose.

[27] O. Abari, D. Bharadia, A. Duffield, and D. Katabi. Enabling {High-Quality} untethered virtual reality. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 531–544, 2017.

[28] S. Aggarwal, Z. Kong, M. Ghoshal, Y. C. Hu, and D. Koutsonikolas. Throughput prediction on 60 ghz mobile devices for high-bandwidth, latency-sensitive applications. In *International Conference on Passive and Active Network Measurement*, pages 513–528. Springer, 2021.

[29] B. Artacho and A. Savakis. Unipose: Unified human pose estimation in single images and videos. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 7035–7044, 2020.

[30] S. Bak, E. Corvee, F. Bremond, and M. Thonnat. Person re-identification using spatial covariance regions of human body parts. In *2010 7th IEEE International Conference on Advanced Video and Signal Based Surveillance*, pages 435–440. IEEE, 2010.

[31] V. Bazarevsky and I. Grishchenko. On-device, real-time body pose tracking with mediapipe blazepose. *Google AI Blog*, 2020.

[32] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[33] J. Butepage, M. J. Black, D. Kragic, and H. Kjellstrom. Deep representation learning for human motion prediction and classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6158–6166, 2017.

[34] Z. Cao, T. Simon, S.-E. Wei, and Y. Sheikh. Realtime multi-person 2d pose estimation using part affinity fields. In *CVPR*, 2017.

[35] J. Chen, X. Qin, G. Zhu, B. Ji, and B. Li. Motion-prediction-based wireless scheduling for multi-user panoramic video streaming. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[36] S. Chen, B. Duinkharjav, X. Sun, L.-Y. Wei, S. Petrangeli, J. Echevarria, C. Silva, and Q. Sun. Instant reality: Gaze-contingent perceptual optimization for 3d virtual reality streaming. *IEEE Transactions on Visualization and Computer Graphics*, 28(5):2157–2167, 2022.

[37] C.-C. Chiu, T. N. Sainath, Y. Wu, R. Prabhavalkar, P. Nguyen, Z. Chen, A. Kannan, R. J. Weiss, K. Rao, E. Gonina, et al. State-of-the-art speech recognition with sequence-to-sequence models. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4774–4778. IEEE, 2018.

[38] K. W. Cho, M. H. Mazaheri, J. Gummeson, O. Abari, and K. Jamieson. mmwall: A reconfigurable metamaterial surface for mmwave networks. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, pages 119–125, 2021.

[39] D. R. Cox and A. Stuart. Some quick sign tests for trend in location and dispersion. *Biometrika*, 42(1/2):80–95, 1955.

[40] Q. Cui, H. Sun, Y. Kong, and X. Sun. Deep human dynamics prior. In *Proceedings of the 29th ACM International Conference on Multimedia*, pages 4371–4379, 2021.

[41] A. Ford, C. Raiciu, M. J. Handley, O. Bonaventure, and C. Paasch. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 8684, Mar. 2020.

[42] X. Gao, L. Dai, S. Han, I. Chih-Lin, and R. W. Heath. Energy-efficient hybrid analog and digital precoding for mmwave mimo systems with large antenna arrays. *IEEE Journal on Selected Areas in Communications*, 34(4):998–1009, 2016.

[43] Y. Ghasempour, C. R. Da Silva, C. Cordeiro, and E. W. Knightly. Ieee 802.11 ay: Next-generation 60 ghz communication for 100 gb/s wi-fi. *IEEE Communications Magazine*, 55(12):186–192, 2017.

[44] E. Gobbetti and F. Marton. Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. In *ACM SIGGRAPH 2005 Papers*, pages 878–885. 2005.

[45] Y. Guan, X. Hou, N. Wu, B. Han, and T. Han. Deepmix: mobility-aware, lightweight, and hybrid 3d object detection for headsets. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 28–41, 2022.

[46] Y. Guan, C. Zheng, X. Zhang, Z. Guo, and J. Jiang. Pano: Optimizing 360 video streaming with a better understanding of quality perception. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 394–407. 2019.

[47] B. Han, Y. Liu, and F. Qian. Vivo: Visibility-aware mobile volumetric video streaming. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–13, 2020.

[48] B. Han, F. Qian, L. Ji, and V. Gopalakrishnan. Mp-dash: Adaptive video streaming over preference-aware multipath. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 129–143, 2016.

[49] A. Hassan, A. Narayanan, A. Zhang, W. Ye, R. Zhu, S. Jin, J. Carpenter, Z. M. Mao, F. Qian, and Z.-L. Zhang. Vivisecting mobility management in 5g cellular networks. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 86–100, 2022.

[50] J. Jiang, V. Sekar, and H. Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 97–108, 2012.

[51] W. M. Kouw and M. Loog. An introduction to domain adaptation and transfer learning. *arXiv preprint arXiv:1812.11806*, 2018.

[52] T. Kucherenko, J. Beskow, and H. Kjellström. A neural network approach to missing marker reconstruction in human motion capture. *arXiv preprint arXiv:1803.02665*, 2018.

[53] Z. Lai, Y. C. Hu, Y. Cui, L. Sun, N. Dai, and H.-S. Lee. Furion: Engineering high-quality immersive virtual reality on today's mobile devices. *IEEE Transactions on Mobile Computing*, 19(7):1586–1602, 2019.

[54] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the conference of*

the ACM special interest group on data communication, pages 183–196, 2017.

[55] K. Lee, J. Yi, Y. Lee, S. Choi, and Y. M. Kim. Groot: a real-time streaming system of high-fidelity volumetric videos. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.

[56] L.-H. Lee, T. Braud, P. Zhou, L. Wang, D. Xu, Z. Lin, A. Kumar, C. Bermejo, and P. Hui. All one needs to know about metaverse: A complete survey on technological singularity, virtual ecosystem, and research agenda. *arXiv preprint arXiv:2110.05352*, 2021.

[57] X. Liu, C. Vlachou, F. Qian, C. Wang, and K.-H. Kim. Firefly: Untethered multi-user vr for commodity mobile devices. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pages 943–657, 2020.

[58] Y. Liu, B. Han, F. Qian, A. Narayanan, and Z.-L. Zhang. Vues: Practical mobile volumetric video streaming through multiview transcoding. *ACM MobiCom 2022*, 2022.

[59] Y. Ma, H. Tian, X. Liao, J. Zhang, W. Wang, K. Chen, and X. Jin. Multi-objective congestion control. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 218–235, 2022.

[60] J. Martinez, M. J. Black, and J. Romero. On human motion prediction using recurrent neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2891–2900, 2017.

[61] S. R. Musse, C. Babski, T. Capin, and D. Thalmann. Crowd modelling in collaborative virtual environments. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 115–123, 1998.

[62] S. Mystakidis. Metaverse. *Encyclopedia*, 2(1):486–497, 2022.

[63] A. Narayanan, E. Ramadan, R. Mehta, X. Hu, Q. Liu, R. A. Fezeu, U. K. Dayalan, S. Verma, P. Ji, T. Li, et al. Lumos5g: Mapping and predicting commercial mmwave 5g throughput. In *Proceedings of the ACM Internet Measurement Conference*, pages 176–193, 2020.

[64] J. Navratil and R. L. Cottrell. Abwe: A practical approach to available bandwidth estimation. In *Proceedings of the 4th Passive and Active Measurement Workshop PAM 2003*. Citeseer, 2003.

[65] T. Nitsche, C. Cordeiro, A. B. Flores, E. W. Knightly, E. Perahia, and J. C. Widmer. Ieee 802.11 ad: directional 60 ghz communication for multi-gigabit-per-second wi-fi. *IEEE Communications Magazine*, 52(12):132–141, 2014.

[66] F. Qian, B. Han, Q. Xiao, and V. Gopalakrishnan. Flare: Practical viewport-adaptive 360-degree video streaming for mobile devices. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 99–114, 2018.

[67] V. Raghavan, A. Partyka, L. Akhoondzadeh-Asl, M. A. Tassoudji, O. H. Koymen, and J. Sanelli. Millimeter wave channel measurements and implications for phy layer design. *IEEE Transactions on Antennas and Propagation*, 65(12):6521–6533, 2017.

[68] D. Rempe, T. Birdal, A. Hertzmann, J. Yang, S. Sridhar, and L. J. Guibas. Humor: 3d human motion model for robust pose estimation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 11488–11499, 2021.

[69] V. J. Ribeiro, R. H. Riedi, R. G. Baraniuk, J. Navratil, and L. Cottrell. pathchirp: Efficient available bandwidth estimation for network paths. In *Passive and active measurement workshop*, 2003.

[70] A. Rutkowska. Properties of the cox–stuart test for trend in application to hydrological series: the simulation study. *Communications in Statistics-Simulation and Computation*, 44(3):565–579, 2015.

[71] S. K. Saha, S. Aggarwal, R. Pathak, D. Koutsonikolas, and J. Widmer. Musher: An agile multipath-tcp scheduler for dual-band 802.11 ad/ac wireless lans. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[72] K. Sato, T. Manabe, T. Ihara, H. Saito, S. Ito, T. Tanaka, K. Sugai, N. Ohmi, Y. Murakami, M. Shibayama, et al. Measurements of reflection and transmission characteristics of interior structures of office building in the 60-ghz band. *IEEE transactions on antennas and propagation*, 45(12):1783–1792, 1997.

[73] H. Schwarz, D. Marpe, and T. Wiegand. Overview of the scalable video coding extension of the h. 264/avc standard. *IEEE Transactions on circuits and systems for video technology*, 17(9):1103–1120, 2007.

[74] S. Sur, I. Pefkianakis, X. Zhang, and K.-H. Kim. Wifi-assisted 60 ghz wireless networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, pages 28–41, 2017.

[75] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[76] J. van der Hooft, H. Amirpour, M. T. Vega, Y. Sanchez, R. Schatz, T. Schierl, and C. Timmerer. A tutorial on immersive video delivery: From omnidirectional video to holography. *IEEE Communications Surveys & Tutorials*, 2023.

[77] S. Venugopalan, M. Rohrbach, J. Donahue, R. Mooney, T. Darrell, and K. Saenko. Sequence to sequence-video to text. In *Proceedings of the IEEE international conference on computer vision*, pages 4534–4542, 2015.

[78] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao. Scaled-YOLOv4: Scaling cross stage partial network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 13029–13038, June 2021.

[79] S. Wang, J. Huang, and X. Zhang. Demystifying millimeter-wave v2x: Towards robust and efficient directional connectivity under high mobility. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, 2020.

[80] S. Wang, S. Yang, H. Li, X. Zhang, C. Zhou, C. Xu, F. Qian, N. Wang, and Z. Xu. Salientvr: saliency-driven mobile 360-degree video streaming with gaze information. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, pages 542–555, 2022.

[81] T. Woodford, X. Zhang, E. Chai, K. Sundaresan, and A. Khojastepour. Spacebeam: Lidar-driven one-shot mmwave beam management. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 389–401, 2021.

[82] J. Wu, C. Yuen, B. Cheng, M. Wang, and J. Chen. Streaming high-quality mobile video with multipath tcp in heterogeneous wireless networks. *IEEE Transactions on Mobile Computing*, 15(9):2345–2361, 2015.

[83] Z. Wu, C.-Y. Huang, and P. Ramanathan. Measuring millimeter wave based link bandwidth fluctuations during indoor immersive experience. *IEEE Networking Letters*, 2022.

[84] G. Xia, H. Sun, B. Chen, Q. Liu, L. Feng, G. Zhang, and R. Hang. Nonlinear low-rank matrix completion for human motion recovery. *IEEE Transactions on Image Processing*, 27(6):3011–3024, 2018.

[85] C. Xie, X. Li, Y. Hu, H. Peng, M. Taylor, and S. L. Song. Q-vr: system-level design for future mobile collaborative virtual reality. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 587–599, 2021.

[86] T. Xu, B. Han, and F. Qian. Analyzing viewport prediction under different vr interactions. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 165–171, 2019.

[87] Z. Yuan, H. Venkataraman, and G.-M. Muntean. ibe: A novel bandwidth estimation algorithm for multimedia services over ieee 802.11 wireless networks. In *IFIP/IEEE International Conference on Management of Multimedia Networks and Services*, pages 69–80. Springer, 2009.

[88] A. Zhang, C. Wang, B. Han, and F. Qian. {YuZu}:{Neural-Enhanced} volumetric video streaming. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 137–154, 2022.

[89] D. Zhang, P. Zhou, B. Han, and P. Pathak. M5: Facilitating multi-user volumetric content delivery with multi-lobe multicast over mmwave. 2022.

[90] W. Zhang, F. Qian, B. Han, and P. Hui. Deepvista: 16k panoramic cinema on your mobile device. In *Proceedings of the Web Conference 2021*, pages 2232–2244, 2021.

[91] P. Zhao, C. X. Lu, J. Wang, C. Chen, W. Wang, N. Trigoni, and A. Markham. mid: Tracking and identifying people with millimeter wave radar. In *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 33–40. IEEE, 2019.

[92] X. Zhu, J. Sun, X. Zhang, Y. E. Guo, F. Qian, and Z. M. Mao. Mpbond: efficient network-level collaboration among personal mobile devices. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 364–376, 2020.

[93] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He. A comprehensive survey on transfer learning. *Proceedings of the IEEE*, 109(1):43–76, 2020.

Appendices

A Additional Details of mmWave Throughput Prediction Study

A.1 The OpenPose Format

Typically, a full-body pose can be represented by a set of key points where each key point corresponds to a joint/part of the human body. We customize the OpenPose [34] BODY_25 format – a popular format used in the computer vision community – to represent the full-body pose, as shown in Figure 15.

Location	Moving Area (m^2)	Room Space (m^3)
Personal Office	2.5×2.5	3.3×2.5×2.5
Living Room	3.0×2.7	6.6×5.4×2.5
University Office	4.0×2.0	9.0×7.0×3.0
Meeting Room	4.0×3.0	6.5×6.5×2.8

Table 4: The moving area and room space of the four locations.

We discard some key points (*i.e.,* eyes, ears, toes, and heels) that have little contribution to the full-body pose.

A.2 Missing Key Point Estimation

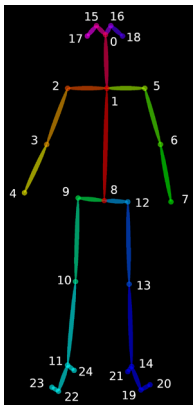
To estimate a missing key point's 3D coordinate on the fly, we consider two baseline approaches: simply reusing its most recently captured 3D coordinate (Method 1), and linearly extrapolating its coordinate using its historical trajectory (Method 2 [40,52,84]). To assess them, we select a subset of our dataset with no missing key point (as the ground truth, referred to as D_g). From D_g, we create a dataset D_a where key points are removed for k consecutive frames where k is exercised from 1 to 60. We apply the above two approaches to D_a, and find that when the missing duration is short (long), Method 1 (Method 2) gives a lower average estimation error (RMSE). This finding leads to our solution where we switch between the two baselines based on the missing duration. The switching threshold is empirically set to 7 or 14 frames for 30 and 60 FPS respectively, based on the data.

To evaluate our solution, we construct another dataset D_b from D_g. In D_b, key points are removed in such a way that their missing time follows the same distribution as that in our entire dataset. Compared to using the two baselines alone, our solution reduces the average RMSE by 21% and 15%, respectively.

Recall from §4.1 that a key point contains a confidence value w. We gradually decay w as a key point remains absent, because as the missing time t increases, its 3D coordinate estimation becomes less reliable. We let $w(t) = w_0 \times max(0, 1 - \frac{t}{T})$ where w_0 is the most recently captured confidence value of this key point and T is a threshold controlling the decay speed. We empirically set T to 1 sec, *i.e.,* the 90-th percentile missing time for a key point in our dataset. The confidence value will be used in §4.3 as an input to the prediction model.

A.3 Details of Data collection Locations

As shown in Figure 3, the four data collection locations we select (*Personal Office*, *Living Room*, *University Office*, *Meeting Room*) have diverse environments in terms of the layout, floor materials, furniture types, and spatial openness, *etc.* The data collection area of *Personal Office* covers almost the entire room. While *Living Room* also has a similarly simple setup, its data collection area only covers one-third of the room and appears more open, as shown in the floor plan. *University Office* is a large room with a complex layout. *Meeting Room* has a long table in the center of the data collection area. The



Used	Discarded
0: Nose	15: Right Eye
1: Neck	16: Left Eye
2: Right Shoulder	17: Right Ear
3: Right Elbow	18: Left Ear
4: Right Wrist	19: Left Big Toe
5: Left Shoulder	20: Left Small Toe
6: Left Elbow	21: Left Heel
7: Left Wrist	22: Right Big Toe
8: Mid Hip	23: Right Small Toe
9: Right Hip	24: Right Heel
10: Right Knee	
11: Right Ankle	
12: Left Hip	
13: Left Knee	
14: Left Ankle	

Figure 15: OpenPose BODY_25 Format.

relative positions between the camera and the WiFi AP also differ across the four locations. Table 4 summarizes the moving area and room space of the four locations. The moving area is the area for data collection. The room space refers to the total space of the entire room.

B Details of Injecting Dynamic Change

We provide details on how to use a robotic arm to mimic real humans in terms of NLoS-incurred throughput degradation (§5.1). The idea is to find a material/object that can incur a similar throughput drop to that caused by a real human and is also lightweight enough for the robotic arm to carry.

To achieve the above, we perform the following experiment consisting of three steps. (1) We install the smartphone on a tripod and fix it in LoS to the 802.11ad access point. We measure the mmWave throughput when there is no blockage between the smartphone and AP. (2) We ask a real human (height: 1.75 m) to stand between the smartphone and the AP to introduce NLoS and measure the mmWave throughput, which now drops. (3) We ask our volunteer to walk away, and use the robotic arm to hold an object at the same position where the real human stands. We make sure the object blocks the LoS between the AP and the smartphone. We then measure the mmWave throughput and compare it with the measured throughput in Step (2). We want their difference to be small. We try three objects: a paper box, a box covered by an outwear, and a box covered by aluminum foil. We find that a box covered by aluminum foil has the most similar impact on mmWave throughput as a real human (with an average throughput difference of 5%, or 28 Mbps). We therefore use it in our dynamic change experiments in §5.1.

C Trend Test For Throughput Sequence

The Cox-Stuart test starts with two statistical hypotheses: (1) H_0: No monotonic trend exists in the series, and (2) H_A: The series is characterized by a monotonic trend, which is further considered as three cases, *i.e.,* (a) an increase or decrease trend exists, (b) an increase trend exists, and (c) a decrease trend exists. Mathematically, in the testing procedure, a throughput

sequence $Z = \{z_1, z_2, ..., z_n\}$ (we suppose n is an even number for simplicity) is divided into two parts $\{z_1, z_2, ..., z_{\frac{n}{2}}\}$ and $\{z_{\frac{n}{2}+1}, z_{\frac{n}{2}+2}, ..., z_n\}$. The test statistic $T(+)$ and $T(-)$ is then calculated as $T(+) = \sum_{i=1}^{\frac{n}{2}} \mathbb{I}(z_i < z_{i+\frac{n}{2}})$ and $T(-) = \sum_{i=1}^{\frac{n}{2}} \mathbb{I}(z_i > z_{i+\frac{n}{2}})$, respectively, where $\mathbb{I} \in \{0, 1\}$ is an indicator function. If the null hypothesis H_0 is true, the statistic $T(+)$ and $T(-)$ should obey the binomial distribution with parameters $\frac{n}{2}$ and $\frac{1}{2}$, *i.e.,* $T(+), T(-) \sim B(\frac{n}{2}, \frac{1}{2})$. Otherwise if $T(+) > T(-)$ (or $T(-) > T(+)$) and the p-value is less than a threshold (*e.g.,* 0.05 in our case), the hypothesis H_A case (b) (or H_A case (c)) is true.

D Additional Implementation Details

D.1 802.11ad Throughput Measurement

Compared to traditional 2D video traffic, immersive content traffic is highly bursty. Take volumetric content as an example. First, different from the traditional 2D videos that are encoded at a group of pictures (GOP) level, volumetric videos are typically encoded on a per-frame basis due to the difficulty of inter-frame encoding. Second, volumetric content players often apply visibility-aware techniques [36, 47, 55, 66, 88] per frame to only download the content inside the viewer's predicted viewport. To maintain accurate viewport prediction results, the client player has to maintain a shallow buffer (*e.g.,* 5 frames in ViVo [47]). Both factors above lead to an extremely frequent request/reply pattern, which renders traditional throughput measurement methods used by 2D video players (simply calculating the ratio between the video chunk size and the chunk download time) very inaccurate. Over mmWave that offers Gbps throughput, the inaccuracy is further deteriorated.

To address the above challenge, we adopt a cross-layer design to measure the throughput by passively examining incoming packets containing immersive content on the client side. Our approach works for both single-path and multipath cases. Specifically, at the application layer, the edge explicitly informs the client how much data will be transmitted over each path before sending data blocks belonging to each frame back-to-back. At the transport layer, the client tracks the arrival time and TCP sequence numbers of the incoming packets. The TCP sequence numbers indicate how much data has been received. Utilizing these information, the client-side throughput measurement module is able to group the *back-to-back* packets in each "burst" as a packet train [64, 69, 83, 87] and use their sizes and timing for throughput measurement. Our approach disregards the ordering and duplicate of packets, and is therefore robust to packet out-of-order and retransmission.

D.2 Development of Two Sample Volumetric Streaming Applications

To demonstrate how Habitus can benefit real immersive applications, we implement two sample volumetric content delivery applications with different logic and complexity.

App 1: Simple Volumetric Streaming. We build a simple volumetric streaming system using the Habitus API from scratch in 5,208 LoC. It delivers the volumetric content stored on a Linux server to an Android client over the Internet. The client player uses a shallow buffer of 5 frames (consistent with App 2) for streaming. The content format uses the layered encoding scheme described in §6.4: each volumetric frame (point cloud) is split into 64 layers each consisting of non-overlapped points through uniform sampling. Each (frame, layer) pair corresponds to a data block in Habitus's term. The priority of each block is inverse proportional to the number of points in the block. The intuition is to prioritize streaming blocks with sparse points so that the viewer can see the partial content as early as possible.

App 2: Visibility-aware Volumetric Streaming. We also replicate ViVo [47], a state-of-the-art networked volumetric video streaming system. ViVo performs visibility-aware streaming where it only fetches content falling into the viewer's predicted viewport. In ViVo, each volumetric frame (point cloud) is spatially segmented into 64 cubical cells. Each (frame, cell) thus constitutes to a data block in Habitus. The priority of a block is calculated at runtime, *i.e.,* inverse proportional to the Euclidean distance from the center of its cubical cell to the center of the predicted viewport. To integrate Habitus into ViVo, we only change 47 LoC that is mainly for library initialization and blocks transmission/reception.

For both applications, each data block is encoded into 10 quality levels with different point density levels. Both applications use the same throughput-based adaptive bitrate (ABR) algorithm [50] to determine the quality level of each data block. The bitrate selection logic works as follows. Initially, all the to-be-fetched blocks are set to the highest quality level. The ABR algorithm then greedily picks the block with the lowest priority and reduces its quality level by 1. The above process is repeated until the total calculated bandwidth usage does not exceed the aggregated network capacity reported by Habitus, or all the blocks reach the lowest quality level.

E Micro Benchmarks and Resource Usage

Resource Usage and Processing Time For a Habitus-enhanced volumetric content delivery system, the average CPU utilization is 36% on the client side (*i.e.,* ROG Phone II) and 169% (*i.e.,* equivalent to 1.69 cores being fully utilized) on the edge side. The peak GPU memory usage on the edge side is 4721MiB (out of 11GB on 2080Ti) in total, including 2101MiB for video capturing and pose estimation, 1017MiB for 802.11ad throughput prediction, and 1603MiB for object detection. The average processing time on an NVIDIA 2080Ti GPU is 27ms and 3.5ms for pose estimation and throughput prediction using a *Seq2Seq with Pose* model, respectively. The processing time meets the system's requirements.

Energy and Heat To profile the energy consumption and heat increase of the client device, we run our control experiment using *V2* and {*Personal Office*, User 1}'s data traces

repeatedly on a ROG Phone II for 30 minutes. We use **App1** (§7) and three Habitus variants {*ac*, *Full*, *MuSher-VR*} in §8.4. We start each experiment on a fully-charged phone. After 30-minute running, the battery level drops from 100% to 93% for *ac*, 92% for *Full*, and 92% for *MuSher-VR*, while the device temperature rises from 30.0°*C* to 36.2°*C* for *ac*, from 30.5°*C* to 38.0°*C* for *Full*, and 30.2°*C* to 38.0°*C* for *MuSher-VR*. Compared to *ac*, the additional energy consumption and heat increase of *Full* is 1% and 1.3°*C*, respectively. Overall, we believe the resource usage of Habitus is acceptable.

BFMSense: WiFi Sensing Using Beamforming Feedback Matrix

Enze Yi [1], Dan Wu [1], Jie Xiong [2], Fusang Zhang [3,4], Kai Niu [5], Wenwei Li [1], Daqing Zhang [1,6]
[1] *Peking University,* [2] *University of Massachusetts Amherst,*
[3] *Institute of Software, Chinese Academy of Sciences,* [4] *University of Chinese Academy of Sciences,*
[5] *Beijing Xiaomi Mobile Software Company Ltd.,* [6] *Institut Polytechnique de Paris*

Abstract

WiFi-based contactless sensing has attracted a tremendous amount of attention due to its pervasiveness, low-cost, and non-intrusiveness to users. Existing systems mainly leverage channel state information (CSI) for sensing. However, CSI can only be extracted from very few commodity WiFi devices through driver hacking, severely limiting the adoption of WiFi sensing in real life. We observe a new opportunity that a large range of new-generation WiFi cards can report another piece of information, i.e., beamforming feedback matrix (BFM). In this paper, we propose to leverage this new BFM information for WiFi sensing. Through establishing the relationship between BFM and CSI, we lay the theoretical foundations for BFM-based WiFi sensing for the first time. We show that through careful signal processing, BFM can be utilized for fine-grained sensing. We showcase the sensing capability of BFM using two representative sensing applications, i.e., respiration sensing and human trajectory tracking. Comprehensive experiments show that BFM-based WiFi sensing can achieve highly accurate sensing performance on a large range of new-generation WiFi devices from various manufacturers, moving WiFi sensing one big step towards real-life adoption.

1 Introduction

In recent years, WiFi-based contactless sensing has attracted lots of attention from both academia and industry owing to the pervasive deployment of WiFi infrastructure in indoor environments. A large number of sensing applications have been successfully demonstrated with WiFi sensing such as passive localization [25, 36], fall detection [42, 46], gesture recognition [38, 40], activity recognition [33, 45, 56], and vital sign monitoring [61, 67]. The basic principle of WiFi sensing is that WiFi signals vary with target movement and we can thus extract target movement information through analyzing the induced signal variations.

Early research on wireless sensing was mostly based on software-defined radio (SDR) platforms such as WARP [31] and USRP [4, 35]. On these platforms, raw WiFi signal samples can be obtained at very high rates for sensing purposes. Though promising, there is actually a huge gap between SDR platform and commodity hardware. Commodity WiFi hardware can only report the processed low-frequency received-signal-strength-indicator (RSSI) samples. While RSSI can still be utilized for sensing [48, 59], it is relatively coarse and only contains signal amplitude information. Researchers further proposed to use channel state information (CSI) which contains both signal amplitude and phase information for sensing [44, 50, 57, 61]. Although CSI contains finer-grained information, one critical issue hindering the wide adoption of CSI-based sensing is that CSI can only be extracted from few commodity WiFi cards such as Intel 5300 [17] and Atheros WiFi cards [55] through driver hacking.

In the last few years, we observed an exciting trend which may be leveraged to address the above issue, i.e., with the popularity of IEEE 802.11ac standard [47], more and more commodity WiFi devices now adopt MU-MIMO (multi-user multiple input and multiple output) technology [8, 23]. New generation WiFi cards from Qualcomm, Broadcom, MediaTek, and others supporting MU-MIMO have become increasingly dominant on the market. For example, Broadcom BCM4366 and BCM43684 chips are used in ASUS RT-AC86U [22] and TP-LINK XDR6060 WiFi routers [9]. Qualcomm QCA9886 and QCA9984 are used in Linksys EA8300 and Netgear X4S R7800 [6].

The key feature of MU-MIMO is enabling a single access point (AP) to simultaneously transmit to multiple stations. To enable MU-MIMO, each WiFi station needs to measure the channel and send the channel measurement, i.e., beamforming feedback matrix (BFM) to the access point (AP). As BFM also contains channel information, BFM may also be utilized for sensing like CSI. There are two obvious advantages of leveraging BFM for sensing.

- BFM is transmitted without encryption.
- BFM is protocol-compliant, and it can be extracted from all new-generation MU-MIMO-enabled WiFi devices without a need of special firmware or driver.

Though promising, we quickly realize the key challenge of

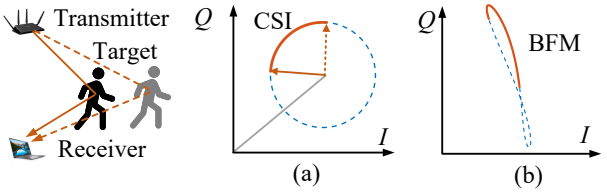


Figure 1: (a) The trajectory (i.e., a circle) of ideal CSI on the complex plane. (b) For BFM, the signal variation does not form a circle on the complex plane.

utilizing BFM for sensing. To reduce the transmission overhead, BFM does not contain raw channel amplitude/phase information like CSI. Instead, BFM only contains the SVD-decomposed component of the channel required for beamforming. That is to say, BFM only contains partial information of CSI. Hence, to utilize BFM for sensing, the key questions we need to answer are:

- As BFM only contains partial information of the channel (i.e., CSI), can BFM still be used for sensing?
- Even if BFM can be utilized for sensing, will BFM be able to achieve comparable performance to that achieved with CSI?
- A large number of sensing models based on CSI have been developed. Are those sensing models and algorithms designed for CSI still applicable to BFM sensing?

This paper aims to answer these questions. Specifically, this paper targets to reveal the relationship between CSI and BFM and understand the basic principle of BFM sensing. These theoretical analyses lay a foundation for WiFi sensing on new-generation cards. Also, we aim to explore the sensing capability of BFM in real-world environments.

After some initial studies, we find that BFM does change with target movement. However, different from CSI, the quantitative relationship between BFM and target motions does not exist. One key property utilized to obtain quantitative measurement (e.g., target movement distance) in CSI-based sensing is that for small displacements, target motions only induce a signal phase change and the signal amplitude remains a constant [41, 44]. Thus, the signal variation on the complex I-Q plane forms a circle as shown in Figure 1(a). We can therefore use the amount of phase rotation to calculate fine-grained target displacement. However, for BFM, the signal variation does not form a circle anymore as shown in Figure 1(b), and that quantitative relationship disappears.

Upon deeper investigation, we find that this is because each time when BFM is calculated, the BFM is scaled by a time-varying coefficient. To address this issue, we leverage one key observation, i.e., although the BFM is scaled by a time-varying coefficient, the same coefficient is applied to all the BFM elements. Based on this observation, we define a new measurement, i.e., BFM-ratio by taking the ratio between two BFM elements[1] for sensing. We demonstrate that theoretically, the sensing performance using BFM-ratio can approach that using CSI. We show that BFM-ratio can be used to not just detect target motions but also quantitatively sense the target movement distance. The main contributions of this paper are summarized below.

- Through theoretically deriving the mathematical relationship between BFM and CSI, we propose a BFM-ratio model to make fine-grained BFM sensing possible for the first time. We believe the widely available BFM data can move WiFi sensing one big step forward towards wider adoption compared to traditional CSI sensing.
- We explore the sensing capability of BFM both theoretically and experimentally to lay a foundation for WiFi sensing with new-generation cards.
- We showcase BFM sensing using two representative sensing applications, i.e., respiration monitoring and human trajectory tracking. The extensive evaluation demonstrates the feasibility of BFM sensing on a large range of new-generation WiFi cards.

2 Background

Before introducing BFM-based sensing, we briefly introduce the background of WiFi CSI and BFM.

2.1 CSI Primer

In a communication system, the channel state describes how a signal propagates through a wireless channel. For a signal X to be transmitted and Y to be received, the signal propagation characteristics are represented by CSI h as $Y = hX + n$, where n is the channel-induced noise. In an indoor environment as shown in Figure 2(a), WiFi signals travel from a transmitter (Tx) to a receiver (Rx) through multiple paths. As the signal received by the receiver is a superposition of signals propagated along multiple paths, mathematically the CSI with frequency f for a pair of transmitter-receiver at time t can be expressed as:

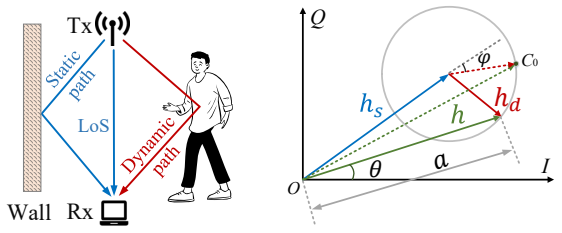
$$h(f,t) = \sum_{i=1}^{K} A_i e^{-j\frac{2\pi f d_i(t)}{c}}, \quad (1)$$

where A_i and $d_i(t)$ are the signal amplitude attenuation and the i-th signal path length, respectively. K is the total number of propagation paths and c is the speed of light.

Among the propagation paths, some are static and do not change within a short period of time, such as the paths reflected from static objects in the environment. Other paths affected by moving objects (e.g., human motions) are dynamic, and the lengths of these paths change with human motions. CSI can then be denoted as the summation of the static component and the dynamic component:

$$h(f,t) = h_s(f) + h_d(f,t) = h_s(f) + A_d e^{-j\frac{2\pi f d(t)}{c}}, \quad (2)$$

[1]The number of elements equals to the number of antennas at the WiFi AP. For example, when there are four antennas at the AP side, there are four BFM elements.



(a) (b)

Figure 2: (a) The human motion causes variation of the reflection path. (b) The ideal CSI on the complex I-Q plane. C_0 indicates the initial point of CSI.

where $h_s(f)$ and $h_d(f,t)$ are the static component and the dynamic component, respectively. $d(t)$ is the length of the dynamic path. According to Equation (2), when the change of dynamic path length is one wavelength, the dynamic component vector $h_d(f,t)$ rotates 360 degrees on the I-Q plane as shown in Figure 2(b).

We denote $a(f,t)$ and $\theta(f,t)$ as the amplitude and phase of CSI $h(f,t)$, i.e., $h(f,t) = a(f,t)e^{j\theta(f,t)}$. For an ideal CSI, its amplitude and phase can be calculated as [41, 61]:

$$a(f,t)^2 = |h_s(f)|^2 + A_d^2 + 2A_d|h_s(f)|\cos\left(\frac{2\pi f \Delta d(t)}{c} + \varphi(f)\right), \tag{3}$$

$$\theta(f,t) \approx \angle h_s(f) - \frac{A_d \sin\left(\frac{2\pi f \Delta d(t)}{c} + \varphi(f)\right)}{a(f,t)}, \tag{4}$$

respectively, where $\Delta d(t)$ is the path length change. $\varphi(f) = \pi + \angle h_s(f) + \frac{2\pi f d_0}{c}$ represents the supplementary angle of the initial phase difference between the dynamic component and the static component, where d_0 is the initial dynamic path length. Equation (3) and Equation (4) show that for one wavelength change of Δd, both signal amplitude and phase change for one cycle.

Given M antennas at the transmitter and N antennas at the receiver, CSI is a $N \times M$ dimensional complex-valued matrix $H(f,t)$ for one subcarrier at time t. $h_{p,q}(f,t)$ denotes the channel state from the q-th antenna of the transmitter to the p-th antenna of the receiver. Specifically, $h_{p,q}(f,t)$ can be expressed as $a_{p,q}(f,t)e^{j\theta_{p,q}(f,t)}$, where the expression of $a_{p,q}$ and $\theta_{p,q}(f,t)$ are presented in Equation (3) and Equation (4) respectively.

Due to unsynchronized clocks between the WiFi transmitter and receiver, the phase of actual CSI (denoted as $\widehat{H}(f,t)$) contains a time-varying random phase offset $\varepsilon(f,t)$. The (p,q) element of $\widehat{H}(f,t)$ can then be expressed as:

$$[\widehat{H}(f,t)]_{p,q} = a_{p,q}(f,t)e^{j\left(\theta_{p,q}(f,t)+\varepsilon(f,t)\right)}, \tag{5}$$

where $[\cdot]_{p,q}$ represents the element of the p-th row and q-th column of a matrix. For the sake of brevity, f and t will be omitted in the rest of formulas.

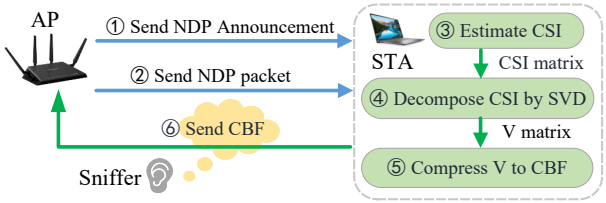


Figure 3: The sounding procedure for MU-MIMO transmissions (IEEE 802.11ac).

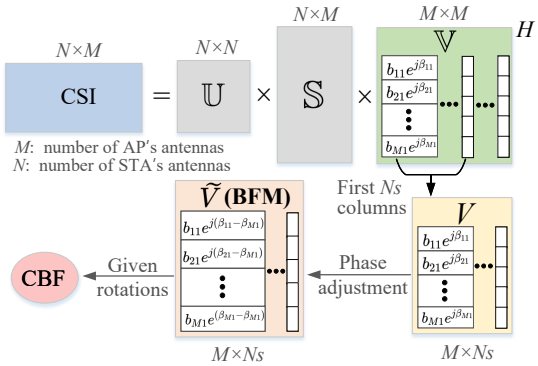


Figure 4: The data flow from CSI to CBF.

2.2 BFM primer

Beamforming feedback matrix (BFM) is another type of information in WiFi that is capable of depicting wireless channels. BFM is mainly used to enable MU-MIMO transmission, which is one key feature of the 802.11ac WiFi standard.

To support MU-MIMO, WiFi AP needs to know the channel information between AP and all the stations (STAs). The channel information (BFM) is sent from STAs to AP in a compressed form, i.e., compressed beamforming feedback (CBF). With the BFM information, AP can then steer the beams towards the desired directions by adjusting the weight of each antenna's transmitted signal.

Although BFM also depicts wireless channel, it is different from CSI. This is because the AP only requires information to calculate the antenna weights for beamforming, as opposed to the entire channel state information. In order to reduce the high transmission overhead of sending the entire channel state, STA decomposes the measured CSI using singular value decomposition (SVD). BFM is then extracted from the right singular matrix and further compressed as CBF, which is subsequently sent back to AP.

Taking an AP and a STA as an example, the detailed procedure is shown in Figure 3. It contains the following steps:

- Step 1: AP broadcasts a Null Data Packet Announcement (NDPA) frame to inform STAs in the same WLAN that the sounding procedure starts.
- Step 2: AP broadcasts a NDP packet to STAs for CSI estimation.
- Step 3: STA estimates CSI as a $N \times M$ matrix ($\widehat{H}$) for each subcarrier, where M and N are the numbers of transmitting and receiving antennas, respectively.

- Step 4: STA decomposes $\widehat{H}$ using SVD operation:

$$\widehat{H} = \mathbb{U}\mathbb{S}\mathbb{V}^H, \tag{6}$$

where $\mathbb{S}$ is a diagonal matrix whose diagonal elements σ_k ($k = 1, \cdots, \min(N,M)$) are real singular values. $\mathbb{U}$ and $\mathbb{V}$ are left and right singular matrices which are unitary matrices containing complex-valued elements with a size of $N \times N$ and $M \times M$, respectively. $\mathbb{V}^H$ indicates the Hermitian transpose of $\mathbb{V}$. Mathematically, we have

$$\sum_{p=1}^{M} b_{p,q}^2 = 1, \tag{7}$$

where $b_{p,q}$ is the amplitude of the (p,q) element in $\mathbb{V}$, and $q = 1, \cdots, M$.
- Step 5: STA extracts the first N_s singular vectors out of $\mathbb{V}$ to construct a new matrix V for beamforming, where N_s is the number of spatial streams which is no more than $\min(N,M)$. Before compression, a phase adjustment step is applied to each column. Specifically, the phase of each column element of V is subtracted by the phase of the last element in the column of V, and a phase-adjusted matrix $\widetilde{V}$ is obtained, which is equivalent to V for beamforming [23, 34]. $\widetilde{V}$ is known as beamforming feedback matrix (BFM). Then, $\widetilde{V}$ is further compressed by applying Givens Rotation to obtain CBF. Figure 4 shows the detailed conversion procedure from CSI to CBF. For the detailed compression algorithm, please refer to IEEE 802.11-2016 [14].
- Step 6: STA sends the CBF back to AP. Based on IEEE standard [14], CBF is transmitted without encryption.

When there are multiple STAs, they send CBF to AP one by one. CBF is transmitted without encryption and can be overheard by any third party in the environment.

3 Understanding the relationship between BFM and human motion

As BFM is partial information of CSI, we proceed in Subsection 3.1 to establish the mathematical connection between BFM and CSI. Based on the established connection, we derive the relationship between BFM and human motion, and analyze the properties of BFM for sensing in Subsection 3.2.

3.1 The mathematical connection between CSI and BFM

The data processing procedure from CSI to BFM involves the decomposition of CSI using SVD to obtain the matrix V. Subsequently, BFM is derived by applying simple phase adjustment to the column vector of V. In light of this, our effort is directed towards understanding the mathematical relationship between V and CSI, following which we deduce the relationship between BFM and CSI, utilizing V as an intermediary.

3.1.1 Relationship between CSI and V

In this section, we study the mathematical relationship between V and CSI. We construct the following equation:

$$\widehat{H}^H\widehat{H} = \mathbb{V}\mathbb{S}^H\mathbb{U}^H\mathbb{U}\mathbb{S}\mathbb{V}^H = \mathbb{V}\mathbb{S}^H\mathbb{S}\mathbb{V}^H. \tag{8}$$

Due to the fact that $\mathbb{U}$ is a unitary matrix, we have $\mathbb{U}^H\mathbb{U} = E$, where E is an identity matrix. Thus the impact of $\mathbb{U}$ can be eliminated by Equation (8).

We assume that AP has M antennas, STA has N antennas, and $M \geq N$. We first substitute Equation (5) into the left side of Equation (8) and obtain a $M \times M$ matrix and the (p,q) element can be written as:

$$[\widehat{H}^H\widehat{H}]_{p,q} = \sum_{k=1}^{N} a_{k,p}a_{k,q}e^{j(\theta_{k,q}-\theta_{k,p})}, \tag{9}$$

where $p, q = 1, 2, \cdots, M$. p and q are the antenna index at the AP while k is the antenna index at the STA. As two antennas at the AP share the same phase offset, the offset is canceled out in $\theta_{k,q} - \theta_{k,p}$.

For the right side of Equation (8), if the diagonal matrix S is marked as $\text{diag}(\sigma_1, \cdots, \sigma_N)$, and the amplitude and phase of (p,q) element in $\mathbb{V}$ are denoted as $b_{p,q}$ and $\beta_{p,q}$ respectively, the (p,q) element of $\mathbb{V}\mathbb{S}^H\mathbb{S}\mathbb{V}^H$ can be expressed as:

$$[\mathbb{V}\mathbb{S}^H\mathbb{S}\mathbb{V}^H]_{p,q} = \sum_{k=1}^{N} \sigma_k^2 b_{p,k}b_{q,k}e^{j(\beta_{p,k}-\beta_{q,k})}. \tag{10}$$

According to Equation (8), the items represented by Equation (9) and Equation (10) are equal. Without loss of generality, we assume there is only one antenna at the STA, that is, $N = N_s = 1$, and V is generated using the first singular vector of the $\mathbb{V}$ matrix. We thus have:

$$\sigma_1^2 b_{p,1}b_{q,1}e^{j(\beta_{p,1}-\beta_{q,1})} = a_{1,p}a_{1,q}e^{j(\theta_{1,q}-\theta_{1,p})}. \tag{11}$$

Next, we discuss the relationship between the amplitude and phase of V and the CSI amplitude and phase.

Amplitude of V. We let $p = q$ and Equation (11) is simplified as $\sigma_1^2 b_{p,1}^2 = a_{1,p}^2$. Then, the amplitude of elements in V can be obtained as:

$$b_{p,1} = \frac{a_{1,p}}{\sigma_1}. \tag{12}$$

The amplitude of the p-th element in V is equal to a scaled amplitude of CSI between the p-th antenna of AP and STA, where the scaling factor is the singular value. Based on Equation (7), we can further derive the singular value σ_1^2 as follows:

$$\sigma_1^2 = \sum_{p=1}^{M} a_{1,p}^2. \tag{13}$$

Equation (13) indicates that σ_1^2 is the sum of squares of the CSI amplitudes between all the AP's antennas and STA's

antenna. It should be noted that, under target motions, CSI amplitude varies and σ_1 is thus time varying.

Phase of V. If $p \neq q$ in Equation (11), we can obtain the relationship between the phase difference of V and phase difference of CSI as follows:

$$\beta_{p,1} - \beta_{q,1} = \theta_{1,q} - \theta_{1,p}. \tag{14}$$

Equation (14) shows that the phase difference between the p-th and q-th elements in V is equal to the phase difference between the CSIs of AP's q-th, p-th antenna and STA.

3.1.2 From V to BFM $\widetilde{V}$

As AP adjusts the phase of multiple transmitted signals on multiple antennas simultaneously based on the feedback, the phase difference matters rather than the absolute phase. To effectively compress V and reduce the amount of feedback, a phase adjustment is made first. Specifically, the phase of each element in V is subtracted by the phase of the last element in V to obtain BFM $\widetilde{V}$. The element in $\widetilde{V}$ can be expressed as follows:

$$[\widetilde{V}]_{p,1} = b_{p,1} e^{j(\beta_{p,1} - \beta_{M,1})} = \frac{a_{1,p}}{\sigma_1} e^{j(\theta_{1,M} - \theta_{1,p})}. \tag{15}$$

The amplitude of $\widetilde{V}$ is the same as that of V. Since the amount of phase adjustment is not fed back to AP, we are unable to reconstruct V but can only reconstruct BFM $\widetilde{V}$.

To sum up, we establish the mathematical relationship between BFM and CSI as follows:

- The amplitude of the p-th element in BFM is equal to the amplitude of CSI between the p-th antenna of AP and STA scaled by the singular value.
- The phase of the p-th element in the BFM is equal to the CSI phase difference between AP-last antenna-STA and AP-p-th antenna-STA.

Noted that in the above derivation, the AP is considered to have multiple antennas, and the STA is considered to have one antenna. So the BFM in this case is actually a 1-D vector but not a 2-D matrix. Therefore, the "element" in the derivation indicates a single element of a vector.

3.2 Sensing Target Motion using BFM

In the previous section, we establish the mathematical expression of BFM amplitude and phase. Based on it, we can analyze the characteristics when we use BFM amplitude and phase for sensing. We take respiration sensing as an example to illustrate the concept.

3.2.1 Sensing motion using BFM amplitude

Based on the mathematical relationship between the amplitude ($|\widetilde{V}|$) of BFM and that of CSI presented in Equation (12) and the relationship between the CSI amplitude and length change of the dynamic path (Δd) given in Equation (3), the mathematical relationship between $|\widetilde{V}|$ and Δd can be obtained as follows:

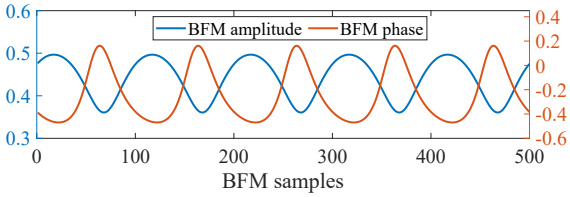


Figure 5: The simulated variation of BFM amplitude ($|\widetilde{V}|$) and phase ($\angle\widetilde{V}$) when dynamic path length changes by 5 wavelengths.

$$|[\widetilde{V}]_{p,1}|^2 = \frac{a_{1,p}^2}{\sigma_1^2} = \frac{\mathcal{F}_p + \mathcal{G}_p \cos\left(\frac{2\pi f \Delta d_{1,p}}{c} + \varphi_{1,p}\right)}{\sum_{m=1}^{M}\left(\mathcal{F}_m + \mathcal{G}_m \cos\left(\frac{2\pi f \Delta d_{1,m}}{c} + \varphi_{1,m}\right)\right)}, \tag{16}$$

where $\mathcal{F}_p = |h_{s_{1,p}}|^2 + A_{d_{1,p}}^2$, and $\mathcal{G}_p = 2A_{d_{1,p}}|h_{s_{1,p}}|$. Since the target is usually far away from the transceiver, we can assume that the dynamic path changes (Δd) at different antennas are equal, i.e., $\Delta d_{1,1} = \Delta d_{1,2} = \cdots = \Delta d_{1,M}$. In this case, we can regard $|[\widetilde{V}]_{p,1}|$ as a function $|[\widetilde{V}]_{p,1}|(\Delta d_{1,p})$ with $\Delta d_{1,p}$ as the variable because other parameters in Equation (16) are constants. Then we can obtain $|[\widetilde{V}]_{p,1}|(\Delta d_{1,p}) = |[\widetilde{V}]_{p,1}|(\Delta d_{1,p} + n\lambda)$, where n is an integer. Therefore, each time the dynamic path length changes by one wavelength, the BFM amplitude varies by one cycle as shown in Figure 5.

3.2.2 Sensing motion using BFM phase

Based on the mathematical relationship between the phase ($\angle\widetilde{V}$) of BFM and that of CSI given in Equation (15) and the relationship between CSI phase and length change of dynamic path Δd given in Equation (4), the mathematical expression between $\angle\widetilde{V}$ and Δd can be obtained as follows:

$$\begin{aligned}\angle[\widetilde{V}]_{p,1} =& \theta_{1,M} - \theta_{1,p} = \angle h_{s_{1,M}} - \angle h_{s_{1p}} \\ &+ \frac{A_{d_{1,p}} \sin\left(\frac{2\pi\Delta d_{1,p}}{\lambda} + \varphi_{1,p}(f)\right)}{a_{1,p}} \\ &- \frac{A_{d_{1,M}} \sin\left(\frac{2\pi\Delta d_{1,M}}{\lambda} + \varphi_{1,M}(f)\right)}{a_{1,M}}.\end{aligned} \tag{17}$$

Similar to BFM amplitude, BFM phase $\angle\widetilde{V}_{p,1}$ can be regarded as a function $\angle\widetilde{V}_{p,1}(\Delta d_{1,p})$ with $\Delta d_{1,p}$ as the variable. Then we can obtain $\angle[\widetilde{V}]_{p,1}(\Delta d_{1,p}) = \angle[\widetilde{V}]_{p,1}(\Delta d_{1,p} + n\lambda)$, where n is an integer. Therefore, each time the dynamic path length changes by one wavelength, the BFM phase varies by one cycle as shown in Figure 5.

3.2.3 Sensing using both BFM amplitude and phase

We compare the differences between BFM and CSI patterns on the complex plane using simulation. By setting the change of dynamic reflection path length to one wavelength in all the

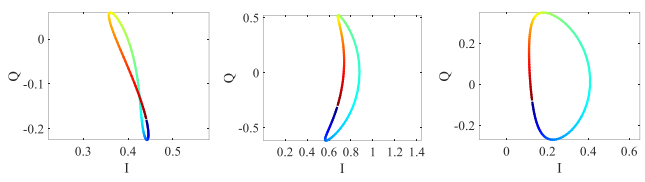


Figure 6: Three examples of one single element of BFM on the complex plane when the dynamic path length increases by a wavelength (the blue color indicates the starting point).

simulations, we obtain various BFM patterns on the complex plane under different settings as shown in Figure 6. Different from the trajectory of CSI which is a circle, the trajectory of BFM is not a circle but there exist a variety of patterns. This difference is due to varying BFM amplitude and phase detailed as below:

- Compared to CSI, the amplitude of BFM is multiplied by a scaling factor. Since this factor is affected by the signal amplitude of all antennas, it is a time-varying variable, causing the BFM amplitude to vary.
- The phase of BFM element is the phase difference of two antennas. It varies with target movement and the coupling between phase difference and signal amplitude is much weaker than that between CSI phase and signal amplitude.

Due to the clear difference between BFM and CSI patterns on the complex plane, many existing CSI-based sensing models and methods cannot be directly applied for BFM sensing.

3.2.4 Case study

We take respiration monitoring as an example to analyze the characteristics of BFM sensing. Respiration induces repetitive small motions. During natural breath, the movement of the chest is about 5mm [29], and the corresponding dynamic path length varies by about one-sixth of a wavelength for 5 GHz WiFi signals. When the reflection path length changes by one wavelength, the BFM amplitude varies for one complete cycle. As respiration causes a change smaller than a complete wavelength, the BFM amplitude change induced by respiration is a fragment of the complete curve. The curve fragments differ when a subject is at different positions, resulting in different amounts of amplitude variations as shown in Figure 7(b). At some positions, the BFM amplitude variations caused by respiration are large and can be easily detected. We call such positions 'good positions' for sensing. Those positions with small amplitude variations can be buried by noise and they are called 'bad positions'.

Through the above example, we can see that the sensing performance using BFM amplitude varies across locations. Similarly, using the BFM phase alone also faces the same problem. This problem was also studied in a recent work [61]. In this work, the complementary property of CSI amplitude and phase on sensing capability is utilized to remove those

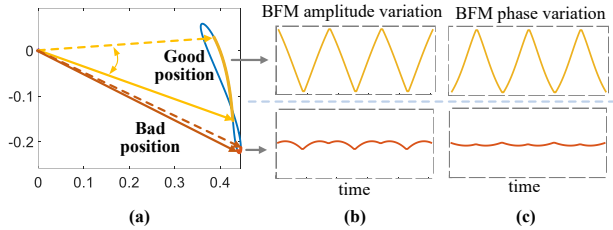


Figure 7: (a) At different initial positions, signal variations induced by respiration correspond to different BFM fragments.

'bad positions'. This is because a 'bad position' for CSI amplitude is a 'good position' for CSI phase, and vice versa.

Unfortunately, such an idea is not applicable in BFM-based sensing. Taking the BFM trajectory on the complex plane in Figure 7 as an example, we can see that at the 'bad' positions, both BFM amplitude and phase perform badly with small signal fluctuation. This is because the complementary property relies on the circular trajectory. As the BFM trajectory on the complex plane is no longer a circle, its amplitude and phase lose the complementary sensing capability.

In summary, the 'bad position' problem still exists when we employ BFM for sensing. Furthermore, conventional solutions such as employing the complementary property of phase and amplitude can not be applied on BFM sensing. In addition, because the quantitative relationship between BFM and target displacement is lost, BFM cannot be used to sense target displacement.

4 Motion Sensing with BFM-Ratio Metric

To address the 'bad position' issue and enable quantitative target displacement sensing, we propose a new metric named BFM-ratio and validate its benefits for sensing in this section.

4.1 BFM-Ratio Metric

As demonstrated in Section 3.2, the time-varying scaling factor in BFM amplitude and uncertain phase offset are the key reasons corrupting the mapping between BFM and target motion. To obtain the mapping, we need to eliminate the effect of the scaling factor on BFM amplitude and the phase offset on BFM phase. Fortunately, although the scaling factors keep changing over time, they are identical at different elements of the BFM matrix, as shown in Equation (12). We thus propose to leverage the ratio of two BFM elements to eliminate the effect of the scaling factor. Specifically, we call it BFM-ratio which can be expressed as:

$$\frac{\widetilde{V}_{1,p}}{\widetilde{V}_{1,q}} = \frac{\frac{a_{1,p}}{\sigma_1} e^{j(\theta_{1,M}-\theta_{1,p})}}{\frac{a_{1,q}}{\sigma_1} e^{j(\theta_{1,M}-\theta_{1,q})}} = \frac{a_{1,p}e^{-j\theta_{1,p}}}{a_{1,q}e^{-j\theta_{1,q}}} \tag{18}$$
$$= \frac{\overline{h_{s_{1,p}}} + A_{1,p}e^{j(\frac{2\pi\Delta d_{1,p}}{\lambda}+\varphi_{1,p})}}{\overline{h_{s_{1,q}}} + A_{1,q}e^{j(\frac{2\pi\Delta d_{1,q}}{\lambda}+\varphi_{1,q})}},$$

where $\overline{h_{s_{1,p}}}$ is the conjugate of $h_{s_{1,p}}$. For simplicity, we let $\mathcal{A}_p = \overline{h_{s_{1,p}}}$, $\mathcal{A}_q = \overline{h_{s_{1,q}}}$, $\mathcal{B}_p = A_{1,p}e^{j\varphi_{1,p}}$, and $\mathcal{B}_q = A_{1,q}e^{j\varphi_{1,q}}$.

When the person is far away from the transceivers, the difference between the two reflection path length variations can be regarded to be the same at two close-by antennas, that is $\Delta d_{1,p} \approx \Delta d_{1,q}$. Therefore, we let $\mathcal{Z}$ present $e^{j\frac{2\pi\Delta d_{1,p}}{\lambda}}$ and $e^{j\frac{2\pi\Delta d_{1,q}}{\lambda}}$. Then Equation (18) can be simplified as:

$$\frac{\widetilde{V}_{1,p}}{\widetilde{V}_{1,q}} = \frac{\mathcal{A}_p + \mathcal{B}_p\mathcal{Z}}{\mathcal{A}_q + \mathcal{B}_q\mathcal{Z}}, \tag{19}$$

where $\mathcal{A}_p$, $\mathcal{A}_q$, $\mathcal{B}_p$ and $\mathcal{B}_q$ can be regarded as complex constants when the target moves a small distance (e.g., chest displacement during respiration). $\mathcal{Z}$ represents a unit circle rotating counterclockwise when $\Delta d_{1,p}$ is increased by one wavelength. From Equation 19, we can see that the ratio of two BFM elements is in the form of the Möbius transformation [37] as long as $\mathcal{A}_p\mathcal{B}_q \neq \mathcal{A}_q\mathcal{B}_p$. One property of Möbius transformation is that it maps circles to circles [24]. For the unit circle of $\mathcal{Z}$, the BFM ratio keeps the shape of a circle on the complex plane [7]. That is to say, when the path length of the dynamic signal is changed by one wavelength, the BFM-ratio vector also experiences a phase rotation of 360 degrees on the I-Q plane. Based on this property, we can obtain the following properties of the BFM ratio for sensing:

1. BFM-ratio rotates a circle on the complex plane if the reflection path length is changed by one wavelength. The BFM-ratio rotates an arc on the complex plane if the reflection path length is changed by less than a wavelength.
2. When the magnitude of the static component of CSI is larger than that of the dynamic component of CSI ($|\mathcal{A}_q| > |\mathcal{B}_q|$), BFM ratio rotates counterclockwise as the length of the reflection path increases. Otherwise, it rotates clockwise. In most cases, the static path component (i.e., LoS path and reflection from static objects) is stronger than the reflection from the human body.

With the above properties, we successfully construct a mapping between BFM-ratio and the change in signal path length induced by motion, laying the foundation for BFM sensing.

4.2 Experiment Verification

We verify the above properties via benchmark experiments. We utilize a Netgear Nighthawk X4S R7800 WiFi AP as the transmitter and a Tenda U10 WiFi Dongle as the receiver. They are placed with a LoS distance of 2.4 m. The central frequency is set as 5.765 GHz, corresponding to a signal wavelength of 5.2 cm. To mimic a moving target, an iron plate was placed on a sliding track in the direction of the perpendicular bisector of the LoS path, as shown in Figure 8. We moved the iron plate to vary the reflection path length. When the dynamic path length changes by one wavelength, while the trajectory of BFM on the I-Q plane is not circular, the BFM ratio trajectory is a complete circle as shown in Figure 9(a). Notably, the BFM ratio rotated counterclockwise when the plate was moved away from the LoS path, and

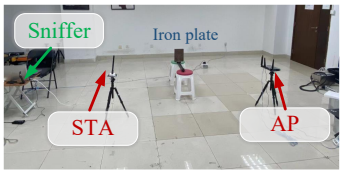


Figure 8: Experiment setting for model verification.

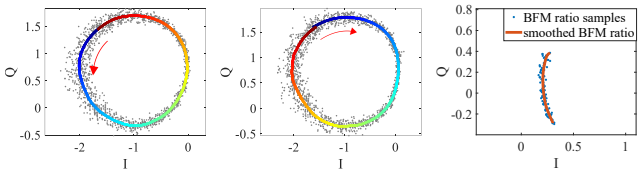


Figure 9: Trajectory of smoothed BFM ratio when the dynamic path length (a) increases by one wavelength (the blue color indicates the starting point), (b) decreases by one wavelength, and (c) changes by one-sixth of a wavelength.

clockwise when it approached the LoS path. We also control the starting position of the iron plate to make the length of the dynamic reflection path change by one-sixth of a wavelength, and the trajectory of the BFM ratio is shown in Figure 9(c). The radian of the BFM ratio trajectory is roughly 60 degrees, which well matches the theoretical value.

5 Case Study

As BFM ratio restores the quantitative mapping between signal variation and target motion, various sensing applications can be realized based on the BFM ratio. In this section, we employ two applications, i.e., micro-motion breath monitoring and macro-motion human walking tracking to demonstrate the sensing capability of BFM ratio.

5.1 Respiration Monitoring

BFM ratio can be used for reliable respiration monitoring without incurring the 'bad position' problem. We calculate the BFM ratio of each pair of elements in BFM and perform this operation on all the subcarriers. For example, when AP and STA have 4 antennas and 1 antenna respectively with 234 subcarriers, $234 \times C_4^2$ streams of BFM ratios can be obtained. Each BFM ratio stream can be further divided into amplitude and phase streams, and we can obtain a total of 2808 streams. We can now extract the respiration information by applying the principal component analysis (PCA) method. We further apply the Savitzky-Golay filter on the respiration waveform to smoothen it. The respiration rate is finally calculated by performing an auto-correlation operation on the respiration waveform with a window size of 30 s.

5.2 Human Trajectory Tracking

The proposed BFM ratio can also be used to sense macro-motions, such as the trajectory of human walking. The key

(a) Asus RT-AC86U (b) Netgear R7800 (c) Linksys EA8100

(d) Linksys MR9000X (e) TP-LINK WDR7660 (f) TP-LINK WDR7661

Figure 10: All tested routers support MU-MIMO technology.

information we use for human trajectory tracking is the movement speed of the target. Recall that the BFM ratio model has established a quantitative relationship between the BFM-ratio phase change and the dynamic path length change in Section 4.1. By leveraging the geometric relationship between the target and transceivers, we can convert the path length change speed into the radial speed of the target with respect to the transceivers [32]. With another pair of transceivers, we obtain a second radial speed. By fusing the two speeds, we obtain the true target velocity. Note that we assume the initial location of the target is known. Within a short period of time window such as 0.2 s, we can safely assume the target velocity is a constant. We can then calculate the target location after the small time window and iteratively compute the following locations to enable continuous tracking.

6 Evaluation

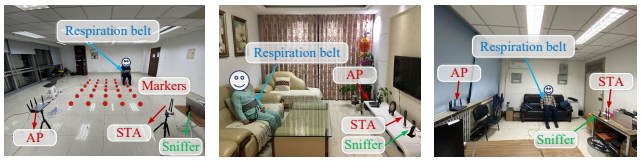
In this section, we conduct experiments to evaluate the performance of applying the proposed BFM ratio metric to realize two typical sensing applications, i.e., micro-motion respiration monitoring and macro-motion human tracking.

6.1 BFM collection.

We first test the feasibility of BFM data collection on diverse devices and evaluate the performance of sensing using real communication traffic.

BFM data collection on various devices. To demonstrate the generality of our approach, we test six different WiFi routers including Asus RT-AC86U, Netgear R7800, Linksys EA8100, Linksys MR9000X, TP-LINK WDR7660, and TP-LINK WDR7661 as shown in Figure 10. We denote the above routers as R1 $\sim$ R6. Although these routers are of various brands and equipped with different WiFi chips, we are able to successfully collect BFM during the communication process of these routers without changing any firmware or driver. This is because BFM is protocol-compliant feedback information that is available on all the devices that support MU-MIMO.

Sensing under real communication traffic. We also evaluate the performance of respiration monitoring under real



(a) empty room (b) living room (c) office

Figure 11: Three typical environments with different levels of multipath and experiment setup for human respiration sensing.

communication traffic between WiFi AP and STAs. Specifically, we watch live webcasts, listen to music, and browse the web to generate different real-world traffic patterns. In this experiment, we use Asus RT-AC86U as AP and two Tenda U10 cards as the STAs. For different activities, the packet rate varies significantly. When watching live webcasts, the mean sample rate of BFM always exceeds 10 Hz, which is high enough for respiration monitoring. When listening to music and browsing the Web, the BFM comes intermittently and the mean sample rate of BFM is 3 $\sim$ 6 Hz and 3 $\sim$ 4 Hz respectively. For human trajectory tracking, usually a sample rate higher than 20 Hz is required. To trigger more frequent beamforming feedback packets, we use the iperf3 tool [30] to generate UDP traffic from AP to STAs.

6.2 Respiration Monitoring using BFM Ratio

Devices. We employ a Netgear R7800 router equipped with Qualcomm WiFi QCA9984 chipset as the AP and two Tenda U10 as the STAs to collect BFM samples. Our system operates at a central frequency of 5.75 GHz, and the channel bandwidth is 80 MHz. We use a Netgear A6210 as a sniffer to collect feedback packets from the STAs.

Environment. We conduct experiments in an empty room (9.5 m $\times$ 9.8 m), a living room (3.4 m $\times$ 6.1 m) and an office (4.4 m $\times$ 4 m), as shown in Figure 11. In the living room and the office, there are a lot of furniture and electrical appliances which create rich multipath. In the living room, our transceivers are placed at one side of the room, as shown in Figure 11(b). In the office, our transceivers are placed at two sides of the room as shown in Figure 11(c).

Participants. We recruit six volunteers including four males and two females for our experiment, aged between 23 and 57. Throughout the experiment, the user wears a commercial sensor (Neulog Respiration Monitor Belt logger sensor NUL-236 [19]) to obtain ground truth. Their respiration rates range from 12 to 18 bpm (beats per minute).

6.2.1 BFM vs. BFM ratio.

In this section, we compare the performance of respiration monitoring based on BFM and BFM ratio. We let the target breathe naturally at different positions. The positions in the environment are marked with red dots in Figure 11(a).

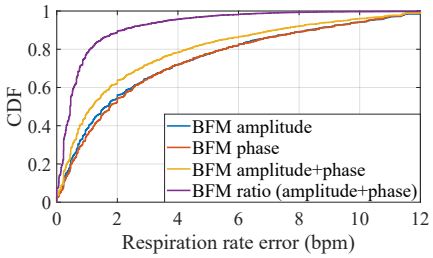


Figure 12: The CDF of respiration sensing error with different signals as input.

Experiment Results. We show the overall respiration rate error and also snapshots of the signal patterns.

- **Respiration rate error.** Figure 12 presents the overall respiration rate errors of different signals. The median errors are 1.61 bpm, 1.76 bpm, 1.17 bpm, and 0.44 bpm for BFM amplitude, BFM phase, BFM amplitude+phase, and BFM ratio, respectively. The errors of using only BFM amplitude or phase are the largest, while using both can only slightly improve the accuracy since the complementary property does not exist. Due to the complementarity of BFM ratio amplitude and phase in respiration monitoring, significantly better performance can be achieved.
- **Breathing induced signal pattern.** Figure 13 presents the detailed patterns of the amplitude and phase of BFM and those of BFM ratio. These signals are filtered by a Savitzky-Golay filter. We can see that while BFM amplitude or phase can be used for respiration monitoring at position 1 and position 2, both of them are too weak to be used at position 3. In comparison, at all three positions, at least one of the BFM ratio amplitude and phase can be utilized for respiration monitoring, outperforming BFM-based method.

6.2.2 Ideal CSI vs. BFM ratio.

To compare the performance with ideal CSI, we use WARP V3 platform [39] to collect CSI data. We deploy the AP and STA with a LoS distance of 2.4 m as shown in Figure 14. We ask a target to sit in the chair located on the perpendicular bisector of the LoS path of the transceivers. We record CSI and BFM data simultaneously when a target breathes naturally. We vary the distance between the target and the LoS path from 2 m to 5 m at a step size of 1 m. Since the sample rates of WARP CSI and BFM are different, we first align the sample rates of these two signals. In addition, the number of subcarriers provided by BFM (234 subcarriers) is larger than that of CSI (56 subcarriers). Here we select the same number of subcarriers (56) for BFM with equal intervals. Then, we feed the amplitude and phase of CSI, and the amplitude and phase of BFM ratio into the method described in Section 5.1 while keeping the other parameters the same.

We plot the mean absolute error (MAE) of respiration rate

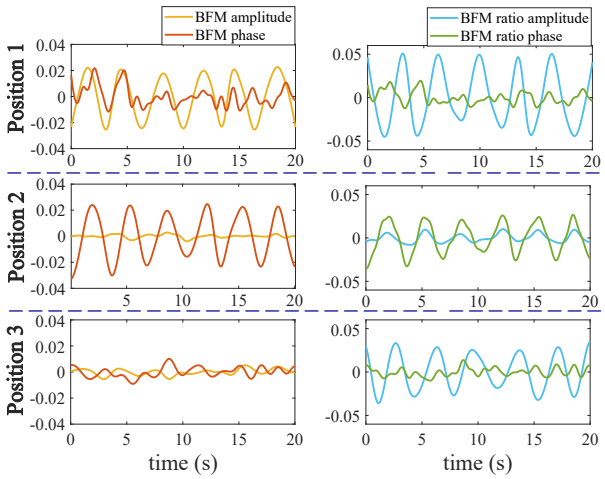


Figure 13: Phase and amplitude patterns of the BFM and BFM ratio when the subject breathes six cycles at three different positions. While BFM amplitude or phase can be used for sensing at Position 1 and 2, neither can be used for sensing at Position 3. In contrast, at least one of the BFM ratio's phase and amplitude can be used for sensing at all locations.

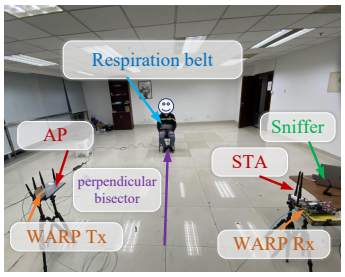


Figure 14: The experiment setup for respiration monitoring. A WARP SDR is used to collect CSI readings for comparison with the BFM-ratio-based method.

with respect to the distance from 2 m to 5 m in Figure 15. The experiment results show that the MAE of respiration monitoring based on BFM ratio is slightly higher than that of CSI. We believe this is mainly because WARP is a high-end software-defined radio platform which allows us to extract cleaner CSI signal. On the other hand, BFM samples extracted from commodity hardware have more noise. Furthermore, since BFM is obtained from CSI through a series of steps including singular value decomposition and compression, there can be a precision loss during the process. Despite that, BFM ratio can still achieve reliable human respiration sensing. We believe that the sensing accuracy can be further improved through signal enhancement algorithms leveraging data from multiple subcarriers [27].

6.2.3 Impact of device and environment diversity

Device diversity. We keep the same environment setting and collect BFM with different routers placed at the same location one by one as shown in Figure 10. We compare the archived MAE for respiration rate monitoring. Despite differ-

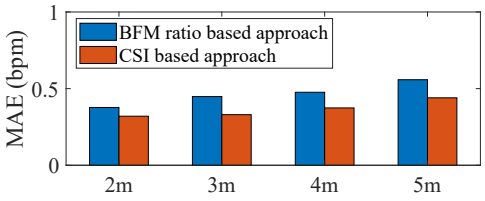


Figure 15: The respiration rate error of CSI-based and BFM-ratio-based approaches at different distances.

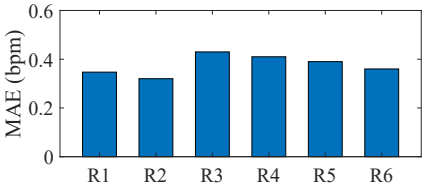


Figure 16: The respiration rate error of different devices.

ent numbers of antennas equipped, similar MAEs are achieved at different devices as illustrated in Figure 16. The results demonstrate that the proposed BFM sensing can be realized on a large range of WiFi devices.

Environment diversity. We deploy AP and STAs in two other rooms with rich multipath as shown in Figure 11(b) and Figure 11(c). The achieved MAEs for respiration rate sensing are 0.43 bpm and 0.46 bpm respectively, which are comparable to that in the empty room.

Through-wall deployment. We further conduct experiments under the following through-wall deployments. AP and sniffer are in the same room while STAs and the target are in another room. The MAE of respiration rate is 0.76 bpm. Compared to the previous experiment where all devices are in the same room, the dynamic path is now attenuated, which leads to a decrease of the signal fluctuation caused by respiration. The error is now larger but still below 1 bpm. Note that the location of the sniffer does not matter as long as it can overhear the CBF packets containing BFM information.

Multi-target scenario. In this subsection, we test the capability of multi-target respiration monitoring. We consider three typical cases as shown in Figure 18. For MU-MIMO transmission, at least two WiFi stations need to be involved so we consider two WiFi stations in these cases.

- Case 1: Two targets are in the same room and the two targets are close to each other.
- Case 2: Two targets are in the same room but are separated from each other with an 8 m distance in between.
- Case 3: Two targets are in different rooms.

In multi-target scenarios, signals reflected from multiple targets are superimposed. When the distance between targets is close (Case 1), signals from multiple targets can not be easily separated in the time domain or spatial domain. We exploit the independence of breathing patterns between targets and apply the blind source separation method [60] to obtain

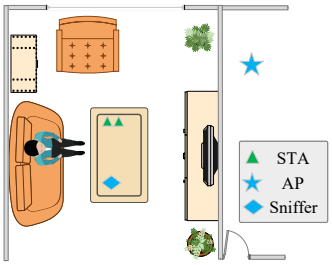
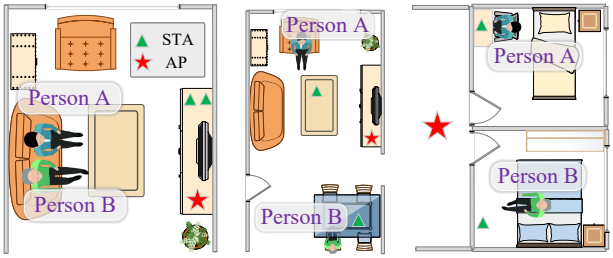


Figure 17: A through-wall deployment for respiration sensing.



(a) Case 1 (b) Case 2 (c) Case 3

Figure 18: Experiment environments and setup for multi-target respiration sensing.

each target's breathing information. In our experiment, with BFM ratio as the input, a low MAE of 0.53 bpm can be achieved. For Cases 2 and 3, the targets are separated by a large distance or a wall. Each STA's signal variation is more affected by the nearer target. The respiration rate of each target can thus be directly estimated from the BFM ratio variation of each STA. The MAEs of the estimated respiration rates in the two cases are 0.44 bpm and 0.38 bpm, respectively. These achieved low MAEs show that BFM ratio can support accurate multi-target respiration monitoring.

6.3 Human Tracking using BFM ratio

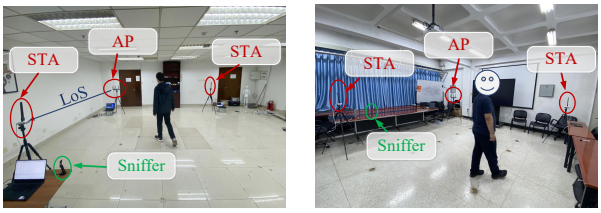
In this section, we evaluate the performance of applying BFM-ratio for velocity estimation to support human tracking.

6.3.1 Experiment setup

We conduct experiments in two typical indoor environments, i.e., an empty room (9.5 m $\times$ 9.8 m), as shown in Figure 19(a), and a meeting room (6.4 m $\times$ 6.4 m), as shown in Figure 19(b). One WiFi AP (Netgear R7800) and two STAs (Tenda U10) are mounted on tripods at a height of one meter. We place two STAs at different locations to make the two AP-STA lines perpendicular to each other. The LoS distance between the transmitter and receivers is set as 4.2 m. Three volunteers (2 men and 1 woman) are recruited to participate in the experiments. The participants are asked to walk naturally along ten trajectories of various shapes, including straight line, diamond, rectangle, triangle and 'N' path. Each volunteer walks along each trajectory 20 times.

6.3.2 Performance of human tracking

We plot the recovered human walking trajectories and the ground truth trajectories (GTT) in Figure 20. We can see



(a) empty room (b) meeting room

Figure 19: The experiment setup for human trajectory tracking in two different environments.

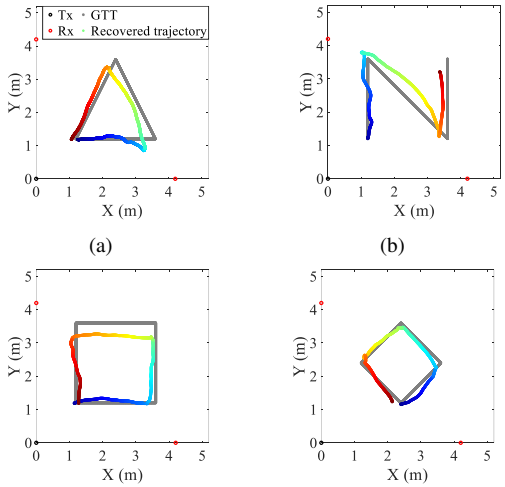


Figure 20: Human trajectory tracking results with BFM ratio.

that with just two STAs, we can achieve accurate human trajectory tracking with commodity WiFi. We present the tracking error in Figure 21. The median localization error is 0.43 m and 0.72 m in the empty room and in the meeting room, respectively. This result shows that BFM ratio can be used to not just detect motion but also quantitatively measure the motion displacement.

6.3.3 Performance in the challenging real-life scenario

In order to demonstrate the performance of BFM ratio tracking in real environments with rich multipath and NLoS, we conduct additional experiments in a living room as shown in Figure 22(a). The size of the room is 3.4 $m \times 6.1$ m, and most of the space is filled with furniture, including sofas, coffee table, and TV cabinet. AP and one STA are placed near the wall. The other STA is placed in the adjacent room with a wall of 30 cm thickness. The straight-line distances between two STAs and AP are 4 m. The target starts walking from a corner near the coffee table and stops at the TV cabinet. We ask the participant to walk along this trajectory 20 times. An example of the trajectory estimate is shown in Figure 22(b) which matches the ground truth. The median localization error is 0.98 m as shown in Figure 21. Due to richer multipaths, the tracking error in the living room is larger than those in the empty room and meeting room.

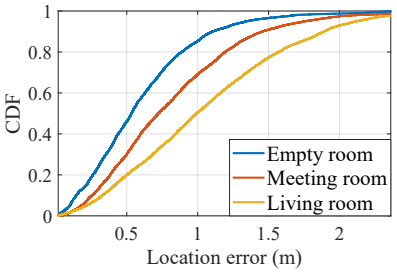
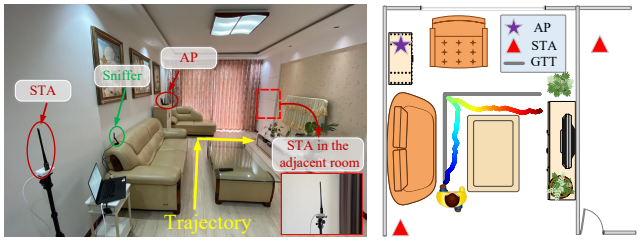


Figure 21: CDF plot of the localization error.



(a) Living room with rich multipath (b) The estimated trajectory

Figure 22: Human trajectory tracking in a multipath-rich and NLoS environment.

7 Related work

In this section, we briefly review the most related work.

7.1 WiFi based contactless sensing

Due to easy access of RSSI at most WiFi devices, in the early years, human sensing was mainly based on RSSI readings. RSSI was used for human respiration monitoring [1] and also for fingerprinting-based localization [59]. However, as RSSI only contains signal strength information, the performance is relatively coarse.

In 2010, some open source CSI tools [17] were developed to support extracting CSI from several 802.11n WiFi chips. Since then, CSI-based sensing has drawn widespread attention in both academia and industry. Promising progress has been achieved from coarse-grained fall detection [42, 46], gesture recognition [15, 38, 40], activity recognition [33, 56], indoor tracking [25, 36, 51] to fine-grained vital sign monitoring [61, 67]. Compared to RSSI, CSI contains both amplitude and phase information. Richer target context such as speed and angle can be obtained from CSI data. A lot of models were developed based on CSI input including CSI-speed model [44], Fresnel zone model [41, 52], Angle-of-Arrival model [26] and CSI-ratio model [62]. CSI-based sensing requires specially tailored WiFi firmware and CSI tools to extract CSI samples. It thus only works with very few WiFi cards such as Intel 5300 and Atheros 9x series cards.

On the other hand, BFM can be extracted from a large range of WiFi devices without the need of hacking WiFi firmware or drivers. It is thus promising to utilize BFM for WiFi sensing. There has been some initial exploration of BFM sensing. In a recent work [20], the authors observed that BFM amplitude is

affected by human respiration and respiration rate can be monitored using BFM. Another recent work [21] further showed that the number of repetitive activities can be counted using BFM phase information. However, these initial studies are empirical research based on experiment observations. There is no theoretical analysis to help people fully understand the underlying mechanism and also the relationship between CSI and BFM. In this work, we lay the theoretical foundation for BFM sensing and analyze the sensing capabilities of BFM. A recent work (BeamSense [49]) proposed to recover CSI from BFM for sensing. To recover BFM from CSI, BeamSense required to extract both downlink BFM and uplink BFM. While downlink BFM can be easily extracted, uplink BFM can only be extracted from 1% WiFi stations, which is acknowledged in BeamSense. In contrast, downlink CBF alone is enough for our proposed system to work.

7.2 Other RF-based sensing

In addition to WiFi signals, a wide range of wireless technologies such as RFID, LoRa, FMCW radar, and IR-UWB radar, have been used for sensing. RFID is a technology that locates and identifies tags attached to items using radio waves. RFID has been utilized to realize sensing applications including localization [28], authentication [12], human activity recognition [58], gesture recognition [70] and material sensing [43, 53]. LoRa is a low-power, long-distance wireless technology designed for connections between IoT devices. LoRa is also utilized for contactless sensing applications such as localization [18], human detection [11], respiration monitoring [63] and human tracking [54, 63]. Recent works further applied LoRa signals for multi-target sensing [65] and soil moisture monitoring [10]. FMCW and IR-UWB radars are two commonly used radars for contactless sensing. They have been used for vital sign monitoring [5, 16, 64, 66], human tracking [2, 3, 13, 69], and even emotion recognition [68]. While the above technologies showed promising results in various sensing applications, they still require dedicated devices that are not yet widely used in our daily life.

8 Discussion

Privacy concern. Since BFM is not encrypted, anyone including attackers can eavesdrop on the BFM transmissions and use the captured BFM readings to sniff human contexts such as location and activities. One potential solution is to apply random offsets (both amplitude and phase offsets) to each element of the BFM readings to corrupt the quantitative relationship between BFM ratio and target motion. However, this method does affect the communication function of BFM. We thus need to embed the random offsets in the encrypted WiFi packets to compensate the applied offsets at legitimate nodes. This will unavoidably lead to more complicated protocol design and higher computational load which eventually may degrade the throughput performance. We believe protecting targets from being sensed by BFM readings is an exciting topic worth more effort.

AoA and ToF estimation from BFM. The key information to enable AoA estimation is the signal phase readings, more precisely, the phase difference between antennas. Since BFM preserves the phase difference between antennas during the compression process, the steering vector can still be constructed for AoA estimation. BFM may have difficulties being used for ToF estimation. This is because the phase relationship between sub-carriers is the key information for ToF estimates. During the compression process, the phase readings subtracted at different subcarriers are different, corrupting the phase relationship between subcarriers which is critical for ToF estimation.

CSI vs. BFM. We believe few manufacturers are willing to release the raw CSI readings to the public and we do not expect a change in the near future. There are two main reasons for this. The first reason is that due to the high sampling rate and detailed information on each subcarrier, the high transmission cost of CSI (i.e., a large amount of data) can degrade the communication throughput. Second, detailed CSI information can also leak confidential chip performance information to the public. This is why BFM adopts a lower sampling rate and only transmits compressed partial data. Therefore, compared to CSI, we believe BFM is the choice for real-world adoption of WiFi sensing. We believe building the theoretical foundation for BFM-based sensing moves a big step towards real-world adoption of WiFi sensing.

9 Conclusion

While WiFi CSI sensing has been popular in the past decade, CSI can only be extracted from very few WiFi cards, greatly hindering the adoption of CSI-based WiFi sensing. We observe an exciting opportunity brought by the new-generation WiFi devices supporting 802.11ac MU-MIMO technology, i.e., a new channel information called BFM can be extracted from a large range of WiFi cards for sensing. In this paper, we studied the underlying principle of BFM sensing by revealing the relationship between BFM and CSI. We propose to take the ratio of two BFM elements to avoid the inherent issues associated with BFM for sensing. The theoretical analysis lays a foundation for WiFi sensing on new-generation WiFi devices. Comprehensive experiments demonstrated the effectiveness of applying BFM for fine-grained sensing.

Acknowledgments

This research is supported by NSFC A3 Project (No. 62061146001), PKU-NTU Collaboration Project, China Postdoctoral Science Foundation (No. 2021TQ0048), Beijing Nova Program (20220484138), Beijing Natural Science Foundation (L223034), National Natural Science Foundation of China (No. 62172394), and Youth Innovation Promotion Association, Chinese Academy of Sciences (No. 2020109).

References

[1] Heba Abdelnasser, Khaled A Harras, and Moustafa Youssef. UbiBreathe: A ubiquitous non-invasive WiFi-based breathing estimator. In *Proceedings of the 16th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 277–286, 2015.

[2] Fadel Adib, Zach Kabelac, Dina Katabi, and Robert C Miller. 3d tracking via body radio reflections. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 317–329, 2014.

[3] Fadel Adib, Zachary Kabelac, and Dina Katabi. Multi-person localization via rf body reflections. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 279–292, 2015.

[4] Fadel Adib and Dina Katabi. See through walls with WiFi! In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 75–86, 2013.

[5] Fadel Adib, Hongzi Mao, Zachary Kabelac, Dina Katabi, and Robert C Miller. Smart homes that monitor breathing and heart rate. In *Proceedings of the 33rd annual ACM conference on human factors in computing systems*, pages 837–846, 2015.

[6] Andreas Anderhub and Zac Wilson. Centrifi: A centralizedwireless access point management platform. 2020.

[7] Douglas N Arnold and Jonathan P Rogness. Möbius transformations revealed. *Notices of the American Mathematical Society*, 55(10):1226–1231, 2008.

[8] Apurv Bhartia, Bo Chen, Feng Wang, Derrick Pallas, Raluca Musaloiu-E, Ted Tsung-Te Lai, and Hao Ma. Measurement-based, practical techniques to improve 802.11ac performance. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, page 205–219, New York, NY, USA, 2017. Association for Computing Machinery.

[9] Wireless CAT. "TP-LINK TL-XDR6060", [online]. https://wikidevi.wi-cat.ru/TP-LINK_TL-XDR6060.

[10] Zhaoxin Chang, Fusang Zhang, Jie Xiong, Junqi Ma, Beihong Jin, and Daqing Zhang. Sensor-free soil moisture sensing using lora signals. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 6(2):1–27, 2022.

[11] Lili Chen, Jie Xiong, Xiaojiang Chen, Sunghoon Ivan Lee, Kai Chen, Dianhe Han, Dingyi Fang, Zhanyong Tang, and Zheng Wang. Widesee: Towards wide-area contactless wireless sensing. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, pages 258–270, 2019.

[12] Xingyu Chen, Jia Liu, Xia Wang, Haisong Liu, Dong Jiang, and Lijun Chen. Eingerprint: Robust energy-related fingerprinting for passive RFID tags. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1101–1113, Santa Clara, CA, February 2020. USENIX Association.

[13] Zhe Chen, Chao Cai, Tianyue Zheng, Jun Luo, Jie Xiong, and Xin Wang. Rf-based human activity recognition using signal adapted convolutional neural network. *IEEE Transactions on Mobile Computing*, 22(1):487–499, 2021.

[14] IEEE Computer Society LAN/MAN Standards Committee et al. Ieee standard for information technology—telecommunications and information exchange between systems local and metropolitan area networks—specific requirements - part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)*, pages 1–3534, 2016.

[15] Ruiyang Gao, Wenwei Li, Yaxiong Xie, Enze Yi, Leye Wang, Dan Wu, and Daqing Zhang. Towards robust gesture recognition by characterizing the sensing quality of wifi signals. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 6(1):1–26, 2022.

[16] Unsoo Ha, Salah Assana, and Fadel Adib. Contactless seismocardiography via deep learning radars. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.

[17] Daniel Halperin, Wenjun Hu, Anmol Sheth, and David Wetherall. Tool release: Gathering 802.11n traces with channel state information. *ACM SIGCOMM CCR*, 41(1):53, Jan. 2011.

[18] Bashima Islam, Md Tamzeed Islam, and Shahriar Nirjon. Feasibility of lora for indoor localization. *Appl. Sci*, 2017:8565550, 2017.

[19] Wern Kam, Waleed Soliman Mohammed, Gabriel Leen, Mary O'Keeffe, Kieran O'Sullivan, Sinead O'Keeffe, and Elfed Lewis. Compact and low-cost optical fiber respiratory monitoring sensor based on intensity interrogation. *Journal of Lightwave Technology*, 35(20):4567–4573, 2017.

[20] Takamochi Kanda, Takashi Sato, Hiromitsu Awano, Sota Kondo, and Koji Yamamoto. Respiratory rate estimation based on wifi frame capture. In *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)*, pages 881–884. IEEE, 2022.

[21] Sorachi Kato, Tomoki Murakami, Takuya Fujihashi, Takashi Watanabe, and Shunsuke Saruwatari. Cbr-ace: Counting human exercise using wi-fi beamforming reports. *Journal of Information Processing*, 30:66–74, 2022.

[22] Simon Kaufmann. Sharing is caring: Throughput fairness in virtual wireless lans.

[23] Joonsuk Kim and Inkyu Lee. 802.11 wlan: history and new enabling mimo techniques for next generation standards. *IEEE Communications Magazine*, 53(3):134–140, 2015.

[24] Vladimir V Kisil. *Geometry of Möbius Transformations*. IMPERIAL COLLEGE PRESS, 2012.

[25] Manikanta Kotaru, Kiran Joshi, Dinesh Bharadia, and Sachin Katti. Spotfi: Decimeter level localization using wifi. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 269–282, 2015.

[26] Xiang Li, Shengjie Li, Daqing Zhang, Jie Xiong, Yasha Wang, and Hong Mei. Dynamic-music: accurate device-free indoor localization. In *Proceedings of the 2016 ACM international joint conference on pervasive and ubiquitous computing*, pages 196–207, 2016.

[27] Yang Li, Dan Wu, Jie Zhang, Xuhai Xu, Yaxiong Xie, Tao Gu, and Daqing Zhang. Diversense: Maximizing wi-fi sensing range leveraging signal diversity. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 6(2):1–28, 2022.

[28] Bo Liang, Purui Wang, Renjie Zhao, Heyu Guo, Pengyu Zhang, Junchen Guo, Shunmin Zhu, Hongqiang Harry Liu, Xinyu Zhang, and Chenren Xu. RF-Chord: Towards deployable RFID localization system for logistic networks. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1783–1799, Boston, MA, April 2023. USENIX Association.

[29] C Lowanichkiattikul, M Dhanachai, C Sitathanee, S Khachonkham, and P Khaothong. Impact of chest wall motion caused by respiration in adjuvant radiotherapy for postoperative breast cancer patients. *SpringerPlus*, 5:1–8, 2016.

[30] Bruce Mah, Jon Dugan, Brian Tierney, Jef Poskanzer, and Seth Elliot. iperf, version 3, [online]. https://iperf.fr.

[31] Pedro Melgarejo, Xinyu Zhang, Parameswaran Ramanathan, and David Chu. Leveraging directional antenna capabilities for fine-grained gesture recognition. In *Proceedings of the 2014 ACM International Joint Conference on pervasive and ubiquitous computing*, pages 541–551, 2014.

[32] Kai Niu, Xuanzhi Wang, Fusang Zhang, Rong Zheng, Zhiyun Yao, and Daqing Zhang. Rethinking doppler effect for accurate velocity estimation with commodity wifi devices. *IEEE Journal on Selected Areas in Communications*, 2022.

[33] Sameera Palipana, David Rojas, Piyush Agrawal, and Dirk Pesch. FallDeFi: Ubiquitous fall detection using commodity Wi-Fi devices. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(4):1–25, 2018.

[34] Eldad Perahia and Robert Stacey. *Next Generation Wireless LANs: Throughput, Robustness, and Reliability in 802*. Cambridge Univ. Press, 2008.

[35] Qifan Pu, Sidhant Gupta, Shyamnath Gollakota, and Shwetak Patel. Whole-home gesture recognition using wireless signals. In *Proceedings of the 19th annual international conference on Mobile computing & networking*, pages 27–38, 2013.

[36] Kun Qian, Chenshu Wu, Yi Zhang, Guidong Zhang, Zheng Yang, and Yunhao Liu. Widar2. 0: Passive human tracking with a single wi-fi link. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 350–361, 2018.

[37] H Schwerdtfeger. Geometry of complex numbers: Circle geometry moebius transformation. 1979.

[38] Sheng Tan and Jie Yang. Wifinger: Leveraging commodity wifi for fine-grained finger gesture recognition. In *Proceedings of the 17th ACM international symposium on mobile ad hoc networking and computing*, pages 201–210, 2016.

[39] Rice University. "warp project", [Online]. http://warp.rice.edu.

[40] Raghav H Venkatnarayan, Griffin Page, and Muhammad Shahzad. Multi-user gesture recognition using wifi. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 401–413, 2018.

[41] Hao Wang, Daqing Zhang, Junyi Ma, Yasha Wang, Yuxiang Wang, Dan Wu, Tao Gu, and Bing Xie. Human respiration detection with commodity wifi devices: do user location and body orientation matter? In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 25–36, 2016.

[42] Hao Wang, Daqing Zhang, Yasha Wang, Junyi Ma, Yuxiang Wang, and Shengjie Li. Rt-fall: A real-time and contactless fall detection system with commodity wifi devices. *IEEE Transactions on Mobile Computing*, 16(2):511–526, 2017.

[43] Ju Wang, Jie Xiong, Xiaojiang Chen, Hongbo Jiang, Rajesh Krishna Balan, and Dingyi Fang. Tagscan: Simultaneous target imaging and material identification with commodity rfid devices. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, pages 288–300, 2017.

[44] Wei Wang, Alex X Liu, Muhammad Shahzad, Kang Ling, and Sanglu Lu. Understanding and modeling of wifi signal based human activity recognition. In *Proceedings of the 21st annual international conference on mobile computing and networking*, pages 65–76, 2015.

[45] Yan Wang, Jian Liu, Yingying Chen, Marco Gruteser, Jie Yang, and Hongbo Liu. E-eyes: device-free location-oriented activity identification using fine-grained wifi signatures. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 617–628, 2014.

[46] Yuxi Wang, Kaishun Wu, and Lionel M. Ni. Wifall: Device-free fall detection by wireless networks. *IEEE Transactions on Mobile Computing*, 16(2):581–594, 2017.

[47] Wikipedia. "ieee_802.11ac-2013", [online]. https://en.wikipedia.org/wiki/IEEE_802.11ac-2013.

[48] Joey Wilson and Neal Patwari. See-through walls: Motion tracking using variance-based radio tomography networks. *IEEE Transactions on Mobile Computing*, 10(5):612–621, 2010.

[49] Chenhao Wu, Xuan Huang, Jun Huang, and Guoliang Xing. Enabling ubiquitous wifi sensing with beamforming reports. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 20–32, New York, NY, USA, 2023. Association for Computing Machinery.

[50] Dan Wu, Ruiyang Gao, Youwei Zeng, Jinyi Liu, Leye Wang, Tao Gu, and Daqing Zhang. Fingerdraw: Sub-wavelength level finger motion tracking with wifi signals. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(1):1–27, 2020.

[51] Dan Wu, Youwei Zeng, Ruiyang Gao, Shengjie Li, Yang Li, Rahul C Shah, Hong Lu, and Daqing Zhang. Witraj: robust indoor motion tracking with wifi signals. *IEEE Transactions on Mobile Computing*, 2021.

[52] Dan Wu, Daqing Zhang, Chenren Xu, Yasha Wang, and Hao Wang. Widir: Walking direction estimation using wireless signals. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '16, page 351–362, New York, NY, USA, 2016. Association for Computing Machinery.

[53] Binbin Xie, Jie Xiong, Xiaojiang Chen, Eugene Chai, Liyao Li, Zhanyong Tang, and Dingyi Fang. Tagtag: material sensing with commodity rfid. In *Proceedings of the 17th conference on embedded networked sensor systems*, pages 338–350, 2019.

[54] Binbin Xie, Yuqing Yin, and Jie Xiong. Pushing the limits of long range wireless sensing with lora. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 5(3), sep 2021.

[55] Yaxiong Xie, Zhenjiang Li, and Mo Li. Precise power delay profiling with commodity wifi. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, page 53–64, New York, NY, USA, 2015. ACM.

[56] Yang Xu, Wei Yang, Jianxin Wang, Xing Zhou, Hong Li, and Liusheng Huang. Wistep: Device-free step counting with wifi signals. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(4):1–23, 2018.

[57] Zheng Yang, Zimu Zhou, and Yunhao Liu. From RSSI to CSI: Indoor localization via channel response. *ACM Computing Surveys (CSUR)*, 46(2):1–32, 2013.

[58] Lina Yao, Quan Z. Sheng, Xue Li, Tao Gu, Mingkui Tan, Xianzhi Wang, Sen Wang, and Wenjie Ruan. Compressive representation for device-free activity recognition with passive rfid signal strength. *IEEE Transactions on Mobile Computing*, 17(2):293–306, 2018.

[59] Moustafa Youssef, Matthew Mah, and Ashok Agrawala. Challenges: device-free passive localization for wireless environments. In *Proceedings of the 13th annual ACM international conference on Mobile computing and networking*, pages 222–229, 2007.

[60] Vicente Zarzoso and Pierre Comon. Robust independent component analysis by iterative maximization of the kurtosis contrast with algebraic optimal step size. *IEEE Transactions on neural networks*, 21(2):248–261, 2009.

[61] Youwei Zeng, Dan Wu, Ruiyang Gao, Tao Gu, and Daqing Zhang. FullBreathe: Full human respiration detection exploiting complementarity of CSI phase and amplitude of WiFi signals. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(3):1–19, 2018.

[62] Youwei Zeng, Dan Wu, Jie Xiong, Enze Yi, Ruiyang Gao, and Daqing Zhang. Farsense: Pushing the range limit of wifi-based respiration sensing with csi ratio of two antennas. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 3(3):1–26, 2019.

[63] Fusang Zhang, Zhaoxin Chang, Kai Niu, Jie Xiong, Beihong Jin, Qin Lv, and Daqing Zhang. Exploring lora for long-range through-wall sensing. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 4(2), jun 2020.

[64] Fusang Zhang, Zhaoxin Chang, Jie Xiong, Junqi Ma, Jiazhi Ni, Wenbo Zhang, Beihong Jin, and Daqing Zhang. Embracing consumer-level uwb-equipped devices for fine-grained wireless sensing. 6(4), jan 2023.

[65] Fusang Zhang, Zhaoxin Chang, Jie Xiong, Rong Zheng, Junqi Ma, Kai Niu, Beihong Jin, and Daqing Zhang. Unlocking the beamforming potential of lora for long-range multi-target respiration sensing. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 5(2), jun 2021.

[66] Fusang Zhang, Jie Xiong, Zhaoxin Chang, Junqi Ma, and Daqing Zhang. Mobi2sense: Empowering wireless sensing with mobility. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, MobiCom '22, page 268–281, New York, NY, USA, 2022. Association for Computing Machinery.

[67] Fusang Zhang, Daqing Zhang, Jie Xiong, Hao Wang, Kai Niu, Beihong Jin, and Yuxiang Wang. From Fresnel Diffraction Model to Fine-grained Human Respiration Sensing with Commodity Wi-Fi Devices. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(1):1–23, 2018.

[68] Mingmin Zhao, Fadel Adib, and Dina Katabi. Emotion recognition using wireless signals. In *Proceedings of the 22nd annual international conference on mobile computing and networking*, pages 95–108, 2016.

[69] Mingmin Zhao, Yonglong Tian, Hang Zhao, Mohammad Abu Alsheikh, Tianhong Li, Rumen Hristov, Zachary Kabelac, Dina Katabi, and Antonio Torralba. Rf-based 3d skeletons. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 267–281, 2018.

[70] Yongpan Zou, Jiang Xiao, Jinsong Han, Kaishun Wu, Yun Li, and Lionel M. Ni. Grfid: A device-free rfid-based gesture recognition system. *IEEE Transactions on Mobile Computing*, 16(2):381–393, 2017.

mmComb: High-speed mmWave Commodity WiFi Backscatter

Yoon Chae[†], Zhenzhe Lin[†], Kang Min Bae[§], Song Min Kim[§], and Parth Pathak[†]
[†]*George Mason University,* [§]*Korea Advanced Institute of Science and Technology*

Abstract

High-speed connectivity is key to enabling a range of novel IoT applications. Millimeter-wave (mmWave) backscatter has emerged as a possible solution to create high-speed, low-power IoT networks. However, state-of-the-art mmWave backscatter systems are costly due to the need for dedicated mmWave reader devices. This paper presents mmComb, a mmWave backscatter system that is built to operate on commodity mmWave WiFi. mmComb is developed with the aim that mmWave backscatter tags can be directly integrated into 802.11ad/ay mmWave WiFi networks. mmComb makes two key contributions. First, We propose a technique to communicate with backscatter tags using existing beamforming protocol frames from mmWave WiFi devices, without any protocol modification. Second, we develop a self-interference suppression solution that intelligently uses receive beamforming to extract weak mmWave backscatter signal even in indoor multipath-rich channels. We implement our solution with a tag prototype and 60 GHz commodity WiFi devices. Our results show that mmComb can achieve a maximum data rate of 55 Mbps just by leveraging 802.11ad/ay control frames while consuming 87.3 μW with BER lower than 10^{-3} up to 5.5 m range.

1 Introduction

The number of Internet-of-Things (IoT) devices is anticipated to grow close to 30 billion by 2030 [1], creating a wide range of novel applications enabled through high-speed edge and cloud connectivity. Examples of such applications include immersive computing and mixed reality, AI-assisted cyber-physical systems like autonomous vehicles, smart homes, and many more. Millimeter-wave (mmWave) wireless is at the forefront of designing 6G and beyond networks that can enable high-speed connectivity to IoT devices. However, today's mmWave devices consume a significant amount of energy (several watts in 802.11ad and 5G NR devices [39]). Hence, the majority of IoT devices (e.g., RFID, WiFi backscatter) operate in the sub-6 GHz spectrum. While the power consumption of such devices can be in tens of microwatts, their data rates are limited to a few kilobits per second [3, 17, 25] or even bits per second [8, 46, 56] in some cases. mmWave backscatter has emerged as a potential solution to bridge this gap and to enable high-speed, ultra-low-power IoT connectivity using the large bandwidths available in mmWave bands. A key limitation of prior work on mmWave backscatter [6, 23, 28, 41] is that they are not compliant with existing mmWave networks. This means that their adaptation requires either deploying dedicated readers in the network or non-trivial hardware and protocol modifications, increasing the overall cost and reducing their adaptability. For example, Millimetro [41] and OmniScatter [6] use mmWave FMCW radars as mmWave backscatter readers. While the objectives of these works are high-precision localization and massive-scale IoT deployment, deployment of dedicated radars not only increases the cost but also creates non-trivial interference to existing mmWave networks [50]. mmTag [28], on the other hand, aims at high-speed mmWave backscatter but cannot achieve truly commodity operations. mmTag requires additional dedicated hardware (polarized antenna) on the transceiver to receive the polarized backscatter signal. It also requires a custom antenna with separate polarization on the AP and tags for self-interference suppression. Furthermore, the impact of integrating mmTag in mmWave networks is not clear given that it is not designed to be protocol compliant where specific frames are used for piggybacking the backscatter data. These limitations call for a new mmWave backscatter solution that is high-speed, ultra-low-power, and truly compatible with mmWave commodity networks by design.

In this paper, we present mmComb, a high-speed mmWave commodity WiFi backscatter system. mmComb is developed with the aim that mmWave backscatter tags can be seamlessly integrated into 802.11ad/ay mmWave WiFi networks. mmComb tags embed backscatter data bits by exploiting beamforming training frames that are frequently exchanged between 802.11ad/ay APs and clients. mmComb does not require any changes to mmWave WiFi APs or clients in terms of their hardware (no additional RF chains or custom antenna) or protocol stack, making it possible to directly accommodate mmWave backscatter tags into existing mmWave WiFi networks. mmComb can achieve a maximum data rate of 55 Mbps when leveraging 802.11ad beamforming frames for backscattering which is orders of magnitude higher than state-of-the-art sub-6 GHz WiFi backscatter systems [3, 17, 25, 53–55]. Our observed BER is lower than 10^{-3} for a range up to 5.5 m, enabling many practical applications within WLANs. Furthermore, the tag consumes 87.3 μW (10.5 μW only for modulation) including frame detection and modulation. We demonstrate an end-to-end backscatter system using a tag prototype and commercial off-the-shelf 802.11ad AP and clients acting as readers. mmComb addresses the following important challenges:

	Systems	Frequency band	Commodity WiFi compatible?	Data rate	Power consumption (μW)
sub-6 GHz	WiFi backscatter [17]	2.4 GHz	Yes	1 Kbps	24
	WiTAG [3]	2.4, 5 GHz	Yes	4 Kbps	10
	MOXcatter [55]	2.4 GHz	Yes	50 Kbps	33
	Freerider [54]	2.4 GHz	Yes	60 Kbps	30
	Hitchhike [53]	2.4 GHz	Yes	300 Kbps	33
mmWave	MilliMetro [41]	24 GHz	No (FMCW)	300 bps	2.36
	OmniScatter [6]	24, 60 GHz	No (FMCW)	150 Kbps	7
	mmX [27]	24 GHz	No (Dedicated reader)	100 Mbps	1.1×10^4
	mmTag [28]	24 GHz	No (Dedicated reader)	100 Mbps	0.2×10^4
	mmComb	**60 GHz**	**Yes**	**55 Mbps**	**87.3**

Table 1: Comparison with state-of-the-art systems.

(1) Embedding backscatter bits in mmWave WiFi frames. There are three key questions here: (i) which mmWave WiFi frames to use for backscatter communication?, (ii) how to detect the ongoing transmission of these frames?, and (iii) how to modulate and demodulate the backscatter data onto the frame? To answer the first question, mmComb selects 802.11ad/ay control frames, specifically beamforming frames, for backscatter communication. These frames are regularly transmitted (beacons, sector-level sweep, beam refinement, etc.) for beamforming and their reuse for backscattering creates no additional network overhead. Furthermore, they have relatively static structures and are transmitted in all directions (i.e., sectors) which improves demodulation and tag coverage. mmComb introduces an ultra-light detection technique for beamforming frames in tags by intelligently utilizing the unique differences between data and beamforming frames. The tags then perform symbol inversion on incoming symbols using phase shifting to modulate the backscatter data. The backscatter bits are then extracted by determining the change in the channel (estimated vs. measured) at the receiver.

(2) Suppressing self-interference. Another key challenge in mmWave backscatter is that due to high attenuation at mmWave frequencies, the backscattered mmWave signal is weak, making it very difficult to isolate the backscatter symbols from self-interference (i.e., the signal traveling directly from a transmitter to a receiver over LoS or reflected paths). Prior WiFi backscatter systems operating at sub-6 GHz such as HitchHike [53–55] address this issue by shifting the frequency of the backscattered signal to the adjacent WiFi channel. However, mmWave WiFi channels are 2.16 GHz wide, and switching such a channel would require a high-speed oscillator consuming as much as 0.5 mW which is clearly infeasible for a low-power tag [21]. Changing the polarization of the incoming and outgoing signal is another approach presented in [28]. However, such a solution requires non-trivial modifications to WiFi APs to equip them with differently polarized antennas and multiple switches on the tag (increased power consumption). Such solutions cannot be readily adapted to work with commodity mmWave WiFi networks.

mmComb addresses this challenge by exploiting the directionality of mmWave beams to create high-gain receive beams at the receiver towards the desired tag while creating nulls in the direction of self-interference. Using the unique characteristics of beamforming antennas, mmComb eliminates the self-interference without shifting the center frequency. A salient benefit of this approach is that it requires no specialized hardware or protocol modifications in mmWave WiFi devices. Table 1 compares mmComb with state-of-the-art sub-6 GHz WiFi and mmWave backscatter systems.

(3) Tag prototyping, implementation and evaluation. We develop a custom prototype of mmComb tag with high switching speed (up to 100 MHz), capable of modulating the 802.11ad/ay frames at 55 MHz speed. We use off-the-shelf 802.11ad devices and software radios equipped with phased arrays as readers. We evaluate mmComb in diverse scenarios with extensive experiments. Our experiments show that mmComb can achieve a data rate of 55 Mbps using 802.11ad control frames with BER lower than 10^{-3} up to the 5.5m range. We also demonstrate that the control frames with backscatter data embedded in them can be received on an unmodified 802.11ad receiver. We find that our self-interference suppression technique, which beamforms towards a tag while creating nulls towards self-interference directions, provides over 19 dB increase in backscatter SINR compared to beams used in commodity 802.11ad/ay devices. mmComb's self-interference suppression is extensively evaluated (i.e., the impact of phase resolution, spatial smoothing, multiple self-interference paths, and different types of receiver devices). Our results show that even in environments with severe multi-path (3-5 reflected mmWave paths), the average BER remains low (6.5×10^{-3}) due to our self-interference suppression solution. We demonstrate that practical deployments of mmComb are feasible as they can support not only Line-of-Sight (LoS) but also Non-Line-of-Sight (NLoS) paths (tag behind a cardboard box or in an office cubicle). mmComb can scale to multiple tags densely deployed in the environment with very low inter-tag interference. Lastly, we verify that backscattering from the tag does not negatively impact standards-compliant beamforming.

Contributions. Our main technical contributions are:

- mmComb is the first mmWave commodity WiFi backscatter system where the tags can seamlessly integrate into mmWave WiFi networks without any hardware or protocol

modifications.

- We present a self-interference suppression technique that can use beamforming and nulling to significantly improve SINR otherwise weak mmWave backscatter signal reception.
- mmComb is implemented with a custom tag prototype and commodity off-the-shelf 802.11ad devices as readers. We extensively evaluate mmComb in a diverse set of scenarios demonstrating the feasibility of high-speed, low-power mmWave commodity WiFi backscatter.

2 Background

A typical backscatter system consists of a reader and a tag where the reader sends an excitation query signal which is then reflected back by a tag. A tag can switch between different impedances to modulate the amplitude, phase, or frequency [9, 11] of the backscattered signal. In the case of commodity WiFi backscatter, an existing commodity device (e.g., a WiFi AP or client [53, 54]) in the wireless network operates as the reader, thus eliminating the need to deploy dedicated readers and reducing cost. Existing protocol frames transmitted from the commodity readers can also be retrofitted to create the query signal. In mmComb, mmWave WiFi 802.11 ad/ay devices and protocol frames are used for enabling high-speed mmWave backscatter communication. To this end, we now provide a brief background on mmWave WiFi.

802.11ad/ay DMG beamforming. The 60 GHz Directional Multi-Gigabit (DMG) mmWave 802.11ad/ay WiFi provides multi-gigabit-per-second data rate. Due to the high attenuation experienced at mmWave frequencies, directional beams created through phased antenna arrays are commonly utilized. Fig. 1 shows the beamforming training process [32, 35] that is used to determine the best transmit sector (i.e., beam pattern) pair between an AP and a station (STA). The procedure

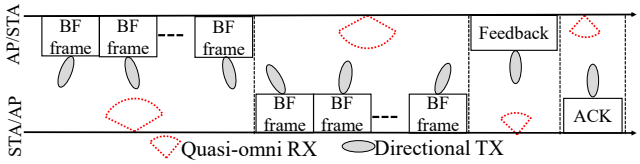


Figure 1: 802.11ad/ay beamforming training process.

involves sending beamforming (BF) frames from the AP in different sectors, and the station measures the received SNR for each. Then, the station sends the BF frames back to the AP, which measures the SNR. The feedback conveys the Tx sector with the highest SNR measurement from both AP to station and vice versa, followed by an acknowledgment.

BF frames. The beamforming training is carried out using BF frames which is a type of control frame. The BF frame consists of a preamble, a header, and a payload as shown in Fig. 2. Both beacons and sector level sweep (SLS) frames are referred to as BF frames because they are sent out in all sectors by the AP, and can be used for backscatter in mmComb. The preamble consists of 59 repeated Golay sequences (Ga_{128}, Gb_{128}, $-Ga_{128}$, and $-Gb_{128}$), while the header and

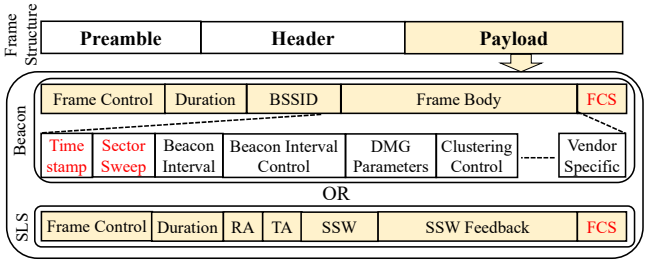


Figure 2: 802.11ad beamforming (BF) frame.

payload are spread with Ga_{32} where the subscript represents the length of sequence. Most of the fields in the BF frame are predetermined and remain constant, except for the timestamp, sector sweep field, and FCS (frame check sequence) marked in red in Fig. 2. The BF frames are transmitted in different sectors during the BF training. The BF frame's data rate is 27.5 megabits per second, coded with an LDPC code at a 1/2 coding rate, and each coded bit is then spread using a 32-point length Golay sequence, producing symbols at a rate of 55 MSym/s at a chip rate of 1760 MHz.

3 mmWave WiFi Backscatter

This section explains how the mmComb tag inserts backscatter bits into ongoing mmWave WiFi frames and how a commodity mmWave WiFi device retrieves these bits through demodulation.

3.1 Modulation and demodulation

Modulation. Our objective in mmComb is to reuse the existing mmWave WiFi frames as a query signal for the backscatter tag. However, reusing any arbitrary frame makes it difficult to isolate backscatter bits from the original frame's bits, as both are unknown to the receiver. Prior work on sub-6 GHz commodity WiFi backscatter [53, 54] relies on separating the two in the frequency domain by shifting the backscatter signal to an adjacent WiFi channel. Albeit effective, this process is not feasible in mmWave WiFi where each 802.11ad/ay channel is 2.16 GHz wide, and shifting to adjacent frequency will require a power-hungry high-speed oscillator on the tag.

In order to address this challenge, mmComb utilizes 802.11 ad/ay BF frames. As described in Sec. 2, most bits of 802.11 ad/ay BF frame are fixed. Since 802.11ad/ay BF frames adopt DBPSK modulation, mmComb embeds backscatter bits onto the BF frames by changing the symbol phase according to them. This is demonstrated in Fig. 3. A mmComb tag

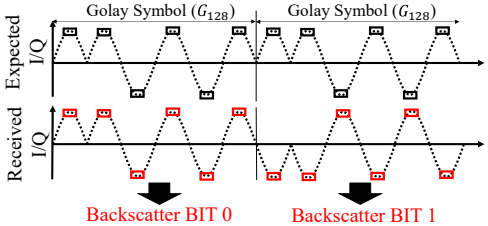


Figure 3: mmComb backscatter modulation.

backscatters the incoming WiFi signal with an offset of either

0° (backscatter bit 0) or 180° (backscatter bit 1). Here, the reflected symbol $S_{\text{reflected}} = S_{\text{original}} \times e^{j\pi b_{\text{back}}}$ where S_{original} is the original incoming symbol and b_{back} is the backscatter bit to be embedded. This means that if the backscatter bit is 0, the signal has no change, and if the backscatter bit is 1, the original signal is phase-shifted by π before backscattering. Our backscatter data rate of 55 Mbps is achievable due to a symbol rate of BF frames (55 MSym/s).

To create a phase shift, we develop a tag that uses a Single Pole, Double Throw (SPDT) switch with one input port and two output ports as shown in Fig. 4. The WiFi signal enters

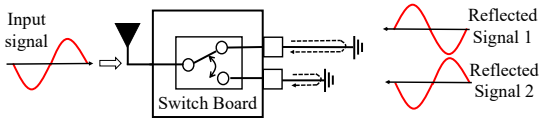


Figure 4: mmComb tag schematic: by switching between the output ports, the tag can reflect with different phases.

the input port and bounces back through one of the output ports. The output ports create different phase shifts due to the different lengths of lines, resulting in a quarter wavelength discrepancy between them. This leads to a half wavelength difference, or a phase shift of π, during a complete round trip. By switching between the output ports, the reflected signal can be modulated with two different phases. mmComb uses a high-speed SPDT switch with 100 MHz of switching speed and $1ns$ rise and fall times (more details of the prototype in Sec. 5).

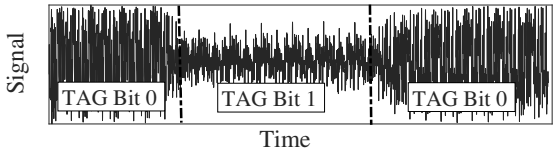
Demodulation. An 802.11ad/ay receiver can now demodulate the backscatter bits by comparing expected and received signals. The received signal $y(t)$ is a linear combination of the transmitted signals from K different paths as

$$y(t) = A(\theta_b) \cdot h_b(\theta_b) \cdot e^{j\pi b(t)} \cdot s(t) + \sum_{i=1, i\neq b}^{K-1} A(\theta_i) \cdot h_i(\theta_i) \cdot s(t) \quad (1)$$

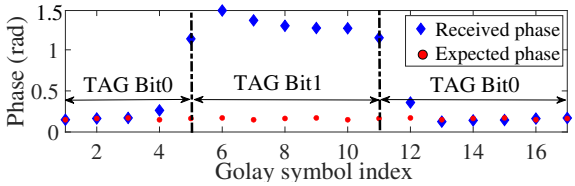
where $A(\theta_i)$ and $h(\theta_i)$ are the complex antenna weight vector and channel gain in direction θ_i, respectively. $s(t)$ is the transmitted signal, and θ_b and $b(t)$ represent the angle of the backscatter path and backscatter bits, respectively.

Typically when a frame is transmitted, the receiver first estimates the channel using the channel estimation field (CEF) and uses the estimated channel to demodulate the transmitted signal. In the case of mmComb, with the use of BF frames, $s(t)$ is already known to the receiver, enabling us to solve for $b(t)$ to extract the backscatter bits.

To achieve this, mmComb tag creates no phase change (i.e., backscatter bit $b(t) = 0$) during the CEF of the BF frame, resulting in the receiver being able to estimate $H = A(\theta_b) \cdot h_b(\theta_b) + \sum_{i=1, i\neq b}^{K-1} A(\theta_i) \cdot h_i(\theta_i)$. This channel estimation and known $s(t)$ can be used to estimate the expected received signal $y'_e(t)$ at the receiver. That is, $y'_e(t)$ represents the case of a signal received by the receiver without being changed by the tag. The backscatter bit $b(t)$ can then be extracted by comparing the received signal $y(t)$ and the estimated signal $y'_e(t)$. If the different $y - y'_e$ is zero, the backscatter bit $b(t) = 0$. On the other hand, if the difference becomes $2 \cdot A(\theta_b) \cdot h_b(\theta_b) \cdot s(t)$, the backscatter bit $b(t) = 1$. Fig. 5 shows an example of expected and received signals and corresponding backscatter bits. In practice, the difference might vary from $2 \cdot$



(a) Raw signal of 802.11 ad golay symbol.



(b) Phase shift due to tag operation.

Figure 5: Embedded backscatter bit on the Golay symbol.

$A(\theta_b) \cdot h_b(\theta_b) \cdot s(t)$ and we use a threshold which is the mean difference over the frame for determining the backscatter bit.

Embedding backscatter frame. Fig. 6 shows our mmComb backscatter frame is embedded inside the 802.11ad/ay frame. Since the length of the BF frame is fixed, the backscatter frame size can also be predetermined including the size of the preamble and data. The *B-preamble* provides synchroniza-

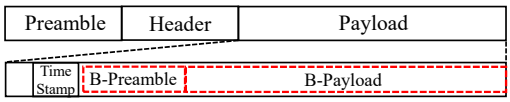


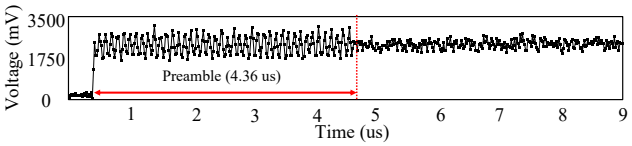
Figure 6: The backscatter frame (B-preamble & B-payload) can be embedded inside the 802.11ad/ay frame.

tion to indicate the start of the backscatter frame. We note that the *B-preamble* must be inserted after the timestamp of the BF frame body at 11.58 μs (4.30 μs of preamble, 4.67 μs of header, 1.45 μs of frame control, duration, and BSSID, and 1.16 μs for timestamp). Since our tag demodulation is based on comparing the original bit sequence with the changed one, it is essential that the tag bit flips occur only after the timestamp. Although it is possible to embed backscatter data even before the timestamp, 802.11ad/ay frames scramble the bits, and any modification to the timestamp because of backscatter data can result in incorrect descrambling. Since mmComb relies on fixed data of the BF frame, proper descrambling is essential to extract the backscatter bits. The backscatter frame can be of length $(L - 11.58)\,\mu s$ where L is a length of the original BF frame, leading to $\lfloor \frac{(L-11.58)*1000}{55} \rfloor$ backscatter bits per BF frame.

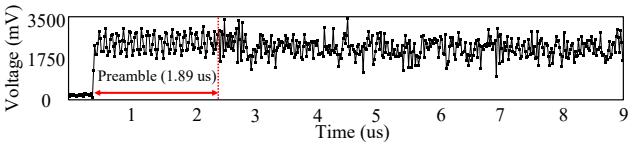
3.2 Ultralight DMG control frame detection

A natural challenge that arises in using a DMG BF control frame is that a tag needs to respond only to a control frame.

In order to determine the type of 802.11ad packet, it is necessary to demodulate the packet from the tag. However, this is not possible under a limited power budget, as demodulation requires a mixer and a high-speed oscillator. Instead, there are two distinctive aspects to the control frame: 1) the preamble sequence (repeated Gb_{128} symbols) remains highly correlated after passing through a narrow bandwidth filter; 2) the control frame (48 Gb_{128} symbols) has a longer preamble than the data frame (16 Gb_{128} symbols), and this observation allows the tag to detect the type of frame without adding complex hardware. The type of frame is detected using a power detector. A power detector is a powerless device that simply converts received RF signals into voltages without consuming power.



(a) Control frame.

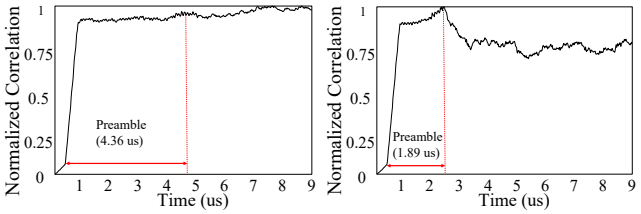


(b) Data frame.

Figure 7: A control frame and a data frame captured by a power detector with a sampling rate of 55 MHz.

Interestingly, a narrow 10MHz bandwidth (about 0.4% of 2.17 GHz) still shows a high correlation for the power detector's output voltage during the preamble due to the redundancy of the repeated Gb_{128} symbols. Although the detector's narrow bandwidth causes signal distortion, we observe high similarity between successive distorted Golay symbols. Using this, it is possible to detect a control frame in a tag efficiently without a complicated demodulation process and hardware. Fig. 7(a) and Fig. 7(b) show the output voltage of the power detector for a control frame and a data frame, separately. It's worth noting that there is a clear difference between control frames and data frames. This is due to the fact that a control frame utilizes a Golay sequence that spreads both the preamble and the payload, whereas only the preamble field in a data frame uses a Golay sequence. Leveraging its unique property, mmComb tag calculates the auto-correlation of output voltage on distorted Golay symbols using a sliding window. The correlation output can detect both the start and type of a frame.

As shown in Fig. 8, a high correlation is observed throughout the entire control frame, while the correlation drops after the preamble in the data frame. Using a threshold-based cutoff, we can detect the type of frame with 98.5% sensitivity ($\frac{TP}{TP+FN}$) and 99.1% accuracy. The detection can be completed in $\approx 2\mu s$ as the length of the data frame preamble is only $1.89\mu s$.



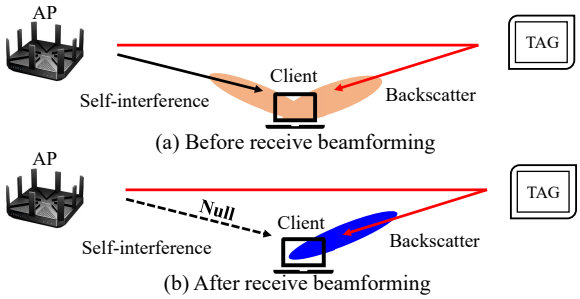
(a) Control frame. (b) Data frame.

Figure 8: The correlation outputs differ between a control frame and a data frame.

4 Suppressing Self-interference

While the mmWave commodity backscatter techniques described in the previous section ensure proper modulation, demodulation, and embedding of the backscatter bits, high self-interference can result in low backscatter SINR. Here, the self-interference refers to all incoming signals received directly from the AP to the client, including LoS and NLoS paths, but not the backscatter path.

As mentioned earlier, shifting the center frequency of the backscattered signal is not feasible for wide 802.11ad/ay WLAN channels due to a limited power budget. Instead, we exploit the directionality of mmWave communication to separate the backscatter and self-interference signals in the spatial domain through careful beamforming and nulling. Our main objective here is to identify a *receiver beam antenna weight vector* that maximizes the gain towards the backscatter tag while creating the nulls in the direction of self-interference. This is demonstrated in Fig. 9. Apart from improving SINR, a



(a) Before receive beamforming

(b) After receive beamforming

Figure 9: Our receive beamforming creates a custom beam with high gain towards the tag and nulls towards the self-interference to improve backscatter SINR.

key advantage of this design is that it offloads the responsibility of self-interference mitigation on the receiver (a commodity device) without adding any complexity to the tag design.

Such beamforming and nulling require us to address the following questions: (i) how can we determine the backscatter path in a composite signal received at the receiver, which also includes self-interference signals? (ii) how can we estimate the angle of arrival (AoA) for the backscatter and self-interference paths? (iii) how can we create a beam towards the tag direction and nulls towards self-interference directions using the AoA information when both signals are highly correlated? and lastly, (iv) how can we develop a backscat-

ter protocol that is compatible with commodity mmWave devices?

4.1 Backscatter beamforming with nulling

In this section, we first address how to create a high-gain beam towards the tag while nulling the self-interference.

Beamforming primer. Beamforming is achieved through a weight vector applied to antenna elements in mmWave's phased antenna array. This creates varying phases and amplitudes in incoming or outgoing signals. Fig. 10 shows the receive beamforming. Each antenna element receives the signal with a different phase value, also known as the steering vector, depending on the angle of arrival (AoA). This can result in a decrease in signal strength due to destructive interference between received signals from each antenna element. To address this concern, each element is assigned a weight based on the AoA to enhance signal strength and improve performance, ensuring that signals are received with high SNR. If the transmitted signal $s(t)$ arrives at the receiver with an incident angle of θ_k for the kth path, the received signal becomes $x(t) = s(t) * a(\theta_k)$ where $a(\theta_k)$ is the steering vector for p antenna elements. $a(\theta_k)$ is essentially a vector of phase shifts τ_k introduced by different travel lengths for different antenna elements and can be written as $a(\theta_k) = \{1, e^{\tau_k}, \cdots, e^{(p-1)\tau_k}\}$, $\tau_k = \frac{-j2\pi f_c d cos\theta_k}{c}$ where d, c, and f_c are the spacing between adjacent antenna elements, the speed of the light, and the center frequency, respectively. When there is only a single path ($k = 1$), the weight vector can be calculated by compensating for the additional phase shifts for that path, i.e., $w(\theta_k) = \{1, e^{-\tau_k}, \cdots, e^{-(p-1)\tau_k}\}$. When there is more than one path, the weight vector should depend on the steering vector matrix $A = [\sum_{j=1}^{k} a_1(\theta_j), \ldots, \sum_{j=1}^{k} a_p(\theta_j)]$ for all incoming paths, i.e., $w = [\sum_{j=1}^{k} w_1(\theta_j), \ldots, \sum_{j=1}^{k} w_p(\theta_j)]$. However, when the incoming signal contains both the desired signal and the self-interference as in mmComb, it is non-trivial to find the weight vector that maximizes only the backscatter signal while nulling the self-interference.

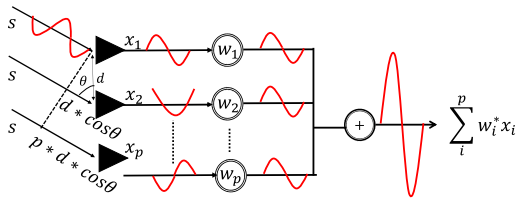


Figure 10: Illustration of beamforming.

Beamforming for backscatter only. mmComb creates the optimal weight vector by calculating a covariance matrix using the AoA information. This method provides an approach that can be readily implemented without any protocol modifications or additional measurements. Existing works such as [13, 26] propose to use evolutionary algorithms or neural networks for estimating the optimal weight vector. They then use multiple SNR measurements to evaluate the fitness of the estimated solution. However, in our case, both the backscatter and self-interference signals are strongly correlated (they are simply a phase-shifted copy of each other) which makes it difficult to use the SNR measurements for the fitness evaluation. mmComb utilizes spatial smoothing technique to decorrelate incoming signals to find the optimal weight vector. We now describe the optimal weight vector calculation process adopted for our scenario.

For ease of exploration, let us consider a linear antenna array with p elements receiving signals. The received signal y is then represented by $y = \sum_{i=1}^{p} w_i^* x_i = w^* x$ where x and w are the received signals and weight vectors, respectively, and * indicates complex conjugate transpose of vectors like in Fig. 10. As we described in Sec. 4.2, we can differentiate between backscatter and self-interference paths and calculate their AoA. This can enable us to calculate the optimal weight vector. Specifically, if the ideal received signal (i.e., only the backscatter signal) from the backscatter path is y_b, we can nullify the self-interference by minimizing the error $\varepsilon = y - y_b = w^* x - y_b$ between the received signal and the ideal backscatter signal. The mean squared error $E[\varepsilon\varepsilon^*]$ can be calculated as

$$\begin{aligned} E[\varepsilon\varepsilon^*] &= E[(w^* x - y_b)(w^* x - y_b)^*] \\ &= w^* R w - 2w^* r + y_b * y_b \end{aligned} \tag{2}$$

where $R = E[xx^*]$ is the auto-correlation of the signal arriving at each antenna element and $r = E[y_b x^*]$ is the cross-correlation between the ideal backscatter signal and arrived signal. By finding the minimum using differentiation $\frac{dE[\varepsilon\varepsilon^*]}{dw} = 2Rw - 2r$, we can get the optimal weight vector $w = R^{-1} r$. Intuitively, R^{-1} cancels the self-interference signals and r creates a high-gain beam towards the backscatter direction. We calculate the y_b and x by replacing the desired signal and interference signal with the steering vectors from their AoA. Therefore, y_b can be the steering vector of a tag direction while x is the steering vector of the combined signal. We then use them to calculate the optimal weight vector.

There is, however, a critical challenge in simply adapting the correlation matrix to determine the optimal vector. The matrix calculation assumes that the incoming signals of backscatter and self-interference are mutually not correlated. If the signals are coherent, there will be a rank loss in the covariance matrix [10], resulting in the creation of wrong weight vectors.

Spatial smoothing for phased arrays. To address the challenge, we leverage a unique characteristic of mmWave WiFi that has a large number of antenna elements to decorrelate the incoming signals. Using this redundancy of these antenna elements, mmComb applies the spatial smoothing technique by splitting the array into several subarrays [40]. This is shown in Fig. 11. Spatial smoothing has been used for decorrelating signals in the context of AoA estimation [20, 48]. In order to accurately estimate the Angle of Arrival (AoA), it is important to identify the Eigenvector of the steering vectors for incoming signals. These vectors need to be orthogonal to each

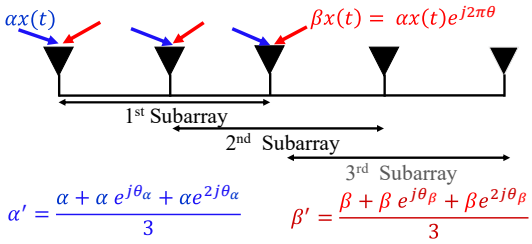


Figure 11: Spatial smoothing: The correlated incoming signal α and β become decorrelated signal α' and β'.

other. However, if the incoming signals are correlated, they lose their orthogonality even if they have different AoAs. To overcome the issue of correlated signals, as shown in Fig. 11, the incoming signals are averaged over the subarrays to decorrelate them.

Let us assume that L subarrays are created from the p antenna elements where each subarray size is M. This yields $L = p - M + 1$. Let $x_i(t) = AD^{i-1}s(t)$ denote the received signal at the ith subarray where A is the steering vector for the subarray. A is $M \times K$ Vandermonde matrix with rank K (number of paths). D is K dimensional diagonal matrix of K paths' AoA ($D = diag\{e^{-j\omega_0\tau_1}, \ldots, e^{-jf_c\tau_k}\}$). Then the correlation matrix of the ith subarray is represented by $X_i = E[xx^*] = E[(A_M D^{i-1})S)(A_M D^{i-1})S)^*] = A_M D^{i-1} S D^{*(i-1)} A_M^*$ where S is the correlation matrix of the transmitted signal ($S = E[s(t)s(t)^*]$). If there are L subarrays, the smoothed covariance matrix becomes the mean of the subarray covariances which can be presented as

$$\overline{X} = \frac{1}{L}\sum_{i=1}^{L} X_i = A(\frac{1}{L}\sum_{i=1}^{L} D^{(i-1)} S D^{*(i-1)})A^* = A\overline{S}A^* \quad (3)$$

where $\overline{S}$ is the modified signal source and is non-singular even with coherence signals given that $L \geq K$. As an example, with our commercial mmWave WiFi hardware, the 6×6 phased array can be configured as 4 subarrays of 5×5 as shown in Fig. 12(a). We then average the measured power delay profile over the subarrays of the antenna so that the signals from different directions become decorrelated signals. Figs. 12(c)-12(e) show the impact of spatial smoothing on an example beam pattern with and without nulling.

4.2 Extracting the backscatter path

To achieve the above-mentioned beamforming with nulls, we need to identify the backscatter path(s) and self-interference path(s) along with their AoA. Although mmWave channel sparsity (3-4 paths in typical indoor environment [49]) reduces the complexity of the problem, It is difficult to distinguish between the backscatter signal and the self-interference signal as they are the same signal but with different channels. We adopt a power delay profile (PDP) based solution for estimating AoA. Specifically, we leverage the approach proposed in [43] for beam alignment and [34] for localization. A detailed description of the AoA estimation method can be found in Appendix A1.

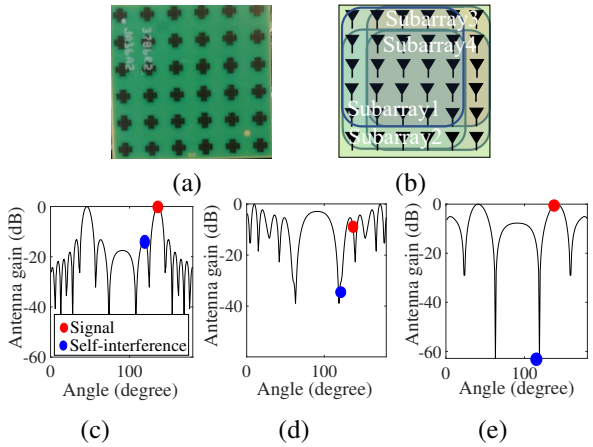


Figure 12: (a) 6×6 phased array used in 802.11ad devices and (b) spatial smoothing with L=4; Example beam patterns of (c) without both nulling and smoothing, (d) with nulling and without smoothing, (e) with both nulling and smoothing

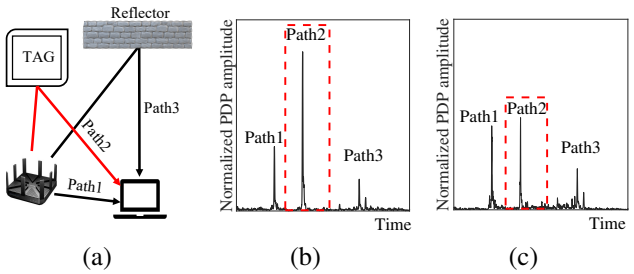


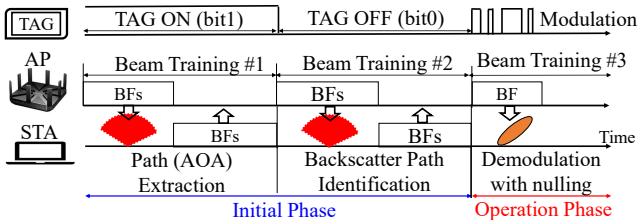
Figure 13: The tag operation changes the PDP peak of Path2.

Identifying the backscatter path. After determining the AoA of different paths, we need to classify them as either backscatter paths or self-interference paths. We leverage a simple observation: during the tag operation, only the amplitude of the backscatter path changes, not the self-interference (LoS and ambient reflections) paths. Fig. 13 shows measured PDP for one backscatter path and two self-interference paths.

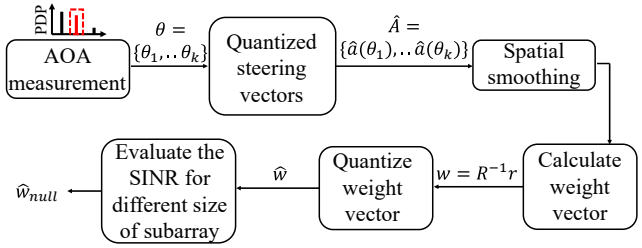
4.3 mmComb backscatter protocol

We now discuss how mmComb can operate with 802.11ad/ay protocol without any protocol modification. Our system operates as shown in Fig. 14(a).

The AP periodically exchanges BF frames with a client (STA) to maintain the link between the AP and the STA. Note that while we refer to this as beam training here which is typically accomplished through SLS frames, a round of beacons sent out by the AP can also be used for the purpose as mentioned in Sec. 2. During the initial phase, the tag operates as ON (bit 1) for the first beam training and as OFF (bit 0) for the second beam training. The initial phase helps the client receiver extract all paths, identify the backscatter path, and estimate the AoA for the backscatter and self-interference paths. Although AoA estimation requires 2 beamforming training, only one measurement is needed for starting backscattering communication. This is followed by a procedure at the client where quantized steering vectors are calculated, spatial



(a) mmComb over 802.11ad/ay.



(b) Steps in beamforming vector calculation with nulling

Figure 14: mmComb framework.

smoothing is applied, and the weight vector that provides the best SINR is selected, as shown in Fig. 14(b). Through this process, the client receiver determines a custom beam that can provide high SINR for backscatter communication. The AP sends out BF frames in different Tx sectors, modulating them with tags. The receiver then demodulates the backscatter data using a custom receive beam. In case of a channel change (for example, human mobility nearby), the AoAs can be recalculated. Fortunately, the channel change is also likely to trigger beamforming for the STAs in the network. This can be leveraged for quickly recalculating the AoAs to maintain a continuous backscatter connection. Therefore, there is a high chance that more than two beamforming opportunities are available within one beacon interval under frequent beamforming. This results in the initial phase being completed within one beacon interval, including path extraction which takes less than $300\mu s$ [43, 45]. We note that mmComb can leverage any control frame for piggybacking the backscatter data. Also, beamforming can occur multiple times within one beacon interval especially when multiple clients are connected to the AP. Both these factors greatly increase the opportunity to perform backscatter communications. Furthermore, the idea of mmComb is to integrate mmWave backscatter tags in the mmWave WLANs (just like conventional mmWave clients). So, it is possible that the AP uses additional beamforming frames to enable communication with the tags.

This protocol also holds for multiple tag cases. Due to the channel sparsity of mmWave, only a few tags respond in each sector (i.e., have high gain for incoming signal). Additionally, due to the different traveling distances, different tags can be differentiated as separate peaks in PDP (Fig. 13), allowing the receiver to determine the AoA for each tag. With this given AoA information, a receiver can perform Rx beamforming for a specific tag while nulling other tags that can respond to the same Tx sector. We further demonstrate this in Sec. 6.4.

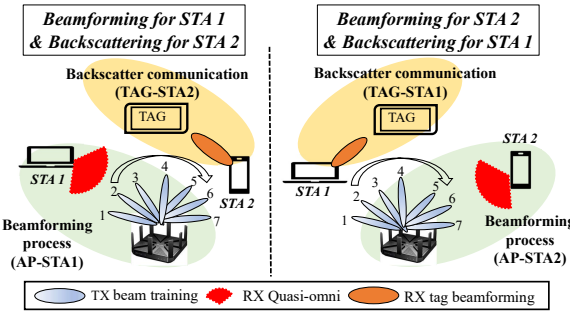


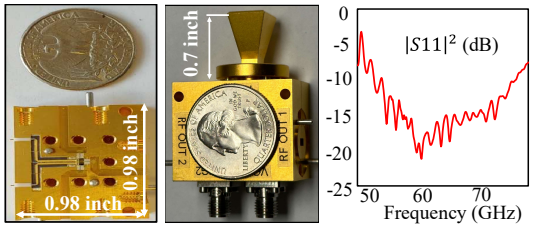
Figure 15: Illustration of mmComb protocol

One potential issue is that when a station (STA) acts as a receiver (Rx) for backscatter and changes its receive beam from quasi-omni to a new customized beam, it may provide the AP with an inaccurate sector (the highest SNR sector from the AP to the STA) during the process of beamforming (Fig. 1). We have observed that this problem does not arise in our system. The reason behind this is that when an STA performs beamforming with the AP, it doesn't allow backscatter communication to prevent any interference with its beamforming process. Instead, the STA acts as the Rx for a backscatter when the AP is performing beamforming with other STAs. This is demonstrated in Fig. 15 using one AP and two STAs. During beamforming training with STA1, the AP can utilize STA2 as a receiver for a backscatter tag. In this process, the tag uses BF frames from the AP to backscatter data to STA2, received over a custom beam with self-interference nulling. The STA1 receives AP's BF frames in quasi-omni mode as per protocol while maintaining unaffected communication with the AP. The beamforming training for different stations is conducted by the AP in separate time slots. Hence, the same is true when the AP performs beamforming with STA2 and STA1 receives the backscatter signals.

5 Implementation

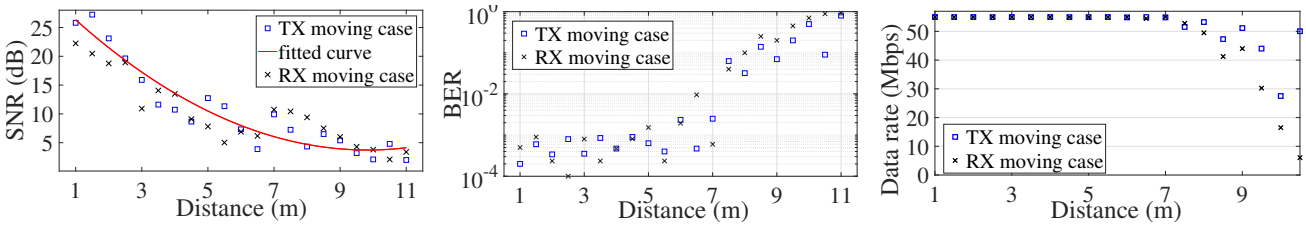
5.1 mmComb Tag Prototype

Our tag prototype is shown in Fig. 16. We fabricated four prototype tags with a GotMIC gSSD0011 SPDT switch that is attached to the housing using a conductive epoxy and packaged with gold-plated aluminum. The switch can support switching speeds up to 100 MHz with rise and fall times of up to $1\,ns$. The return loss (S_{11}) of the tag is measured and shown in Fig. 16(c).



(a) (b) (c)

Figure 16: (a-b) Our mmWave tag prototype, (c) S(11) return loss measurements of our tag.



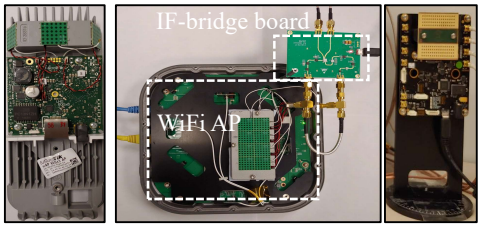
(a) SNR (b) BER (c) Data rate

Figure 17: Backscatter SNR, BER, and data rate at different distances.

mmComb tag attaches a tag antenna (15 dBi V-band antenna SAR-1532-15-S2 with a half-power beamwidth of 41 degrees) to absorb or reradiate a backscatter signal. The tag output ports are attached to reflective ends where one of them reradiates the received signal without any change and the other introduces a phase shift of π using an additional round-trip of quarter wavelength. The switch is controlled through a TerasIC FPGA board with a microcontroller (SAM4SD32C).

5.2 mmComb Commodity WiFi Readers

Commodity 802.11ad devices from Airfide [4] and Mikrotik [2] are used as commodity readers in mmComb, as they use the same 802.11ad Qualcomm QCA6310 chipset and 6×6 antenna array. To enable desired modifications, we implement various changes to the 802.11ad Wil6210 driver and firmware to (i) send BF frames to selected sectors as desired, (ii) set selected Tx and Rx beams, (iii) extract per-element CSI (amplitude and phase), and (iv) create and apply new codebook to firmware with desired weight vectors.



(a) (b) (c)

Figure 18: mmComb uses 802.11ad APs as Tx with two types of Rx: (a) 802.11ad MikroTik AP TX, (b) Airfide AP RX with IF bridge board, and (c) SiversIMA 60 GHz RF SDR RX.

We consider two types of setup: (i) both Tx and Rx are commodity 802.11ad devices, and (ii) the Tx is a commodity 802.11ad device while the Rx is a 60 GHz SDR. In both setups, the Tx transmits 802.11ad BF frames. The first setup includes two types of commodity receivers: an unmodified commodity 802.11ad device and a commodity 802.11ad device equipped with an IF bridge board [57] to extract raw I/Q data as shown in Fig.18. The unmodified 802.11ad receiver allows for commodity compatibility, demonstrating that backscattered beamforming frames can be received without any modifications to hardware, software, or protocol. In contrast to the prior work such as [53] where the checksum errors are ignored (through driver/firmware modification in sub-6 GHz WiFi), we note that the current and only publicly available 802.11ad firmware (wil6210) internally drops the frames with a checksum error. To avoid it, mmComb tags further modulate the checksum (Sec. 2) based on the modified data so that the frame can be correctly received on an unmodified 802.11ad Rx. However, it only evaluates frames correctly received at the Rx and does not allow accurate calculation of BER. In order to evaluate it even on low SNR scenarios where the frame cannot be correctly received by the firmware, we use the IF bridge board on 802.11ad Rx to extract the raw I/Q data. The raw bits are decoded and analyzed with Keysight's 81199A Wideband Waveform Analyzer. *We note that the IF bridge board is only used on the Rx for detailed bit-level analysis of backscatter BF frames. Our proposed design works with an unmodified 802.11ad Rx without the IF bridge board as we show in Fig. 21.*

In the second setup, we use SiversIMA 60 GHz RF frontend with a 16-element phased array on the Rx. *Compared to the phased array found in commodity 802.11ad devices which provides 2-bit phase control, the SiversIMA array provides 6-bit phase control, enabling us to better analyze the impact of beamforming and nulling on backscatter BER.* Since most antenna arrays used COTS 802.11ad devices do not provide per-element amplitude control, we also focus only on phase control in this work.

6 Evaluation

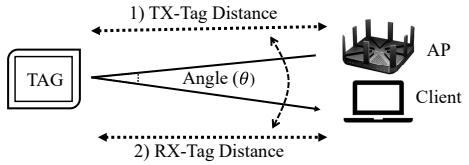
We perform exhaustive experiments to evaluate the performance of mmComb under diverse scenarios. They include (i) benchmarking read range and angles, (ii) commercial device FRR, (iii) nulling performance and impact on BER, (iv) NLoS situations in practical deployments, (v) impact of tag on mmWave WiFi, (vi) multiple simultaneous tags, and (vii) power consumption.

6.1 mmComb read range and angle

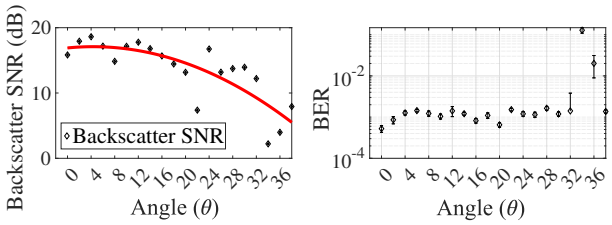
Backscatter range. We first perform a micro-benchmark for mmComb read range in a $12m \times 12m$ office room. The setup is shown in Fig. 19(a) where 802.11ad devices are used as Tx and Rx. The tag and one endpoint (either Tx or Rx) are stationary while the other endpoint gradually moves to increase distance. This is a common scenario in mmWave WLANs, where an AP (Tx) is fixed and a client (Rx) is located at varying distances from the tag. The Rx captures the modulated 802.11ad BF frames sent by the Tx AP. These

frames are then analyzed to determine the backscatter SNR and BER. Figs. 17(a) and 17(b) show the mean SNR and BER for different distances. We find that even up to 5.5m distance, the BER remains below 10^{-3}. Even for a range of 7m, the BER remains lower than 10^{-2}. Fig. 17(c) shows that mmComb achieves a data rate of 54.4 Mbps up to 7m (out of the maximum possible data rate of 55 Mbps).

Impact of angle. Unlike legacy WiFi, mmWave WiFi is highly susceptible to angular changes due to directionality. As shown in Fig. 19(a), we set a tag at 2m distance from the



(a) Setting for tag angle & distance evaluation.



(b) SNR (c) BER

Figure 19: Backscatter SNR and BER over angle.

802.11ad devices and increase the angle from 0° to 38°. Here, the Rx position is fixed as it utilizes a beam pattern to receive the signal from the tag, while the Tx position is changed to vary the angle. As observed in Fig. 19(b), the backscatter SNR remains higher than 7.9 dB even for 38° which is close to the antenna beamwidth of 41°. Fig. 19(c) shows that the mean BER remains lower than 1.3×10^{-2} over all angles within the tag antenna beamwidth.

Tag at different locations. We conduct an experiment by varying the incident angle of the backscattered signal to Rx. Fig. 20 shows a 10m × 10m room where we perform over 200 measurements at 20 different locations. These locations

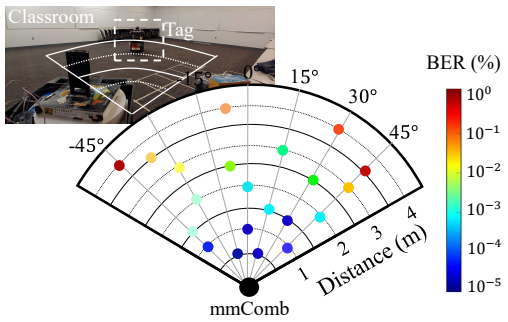


Figure 20: Backscatter BER for 20 different tag positions

are randomly chosen to create different incident angles and distances. The position of Tx and Rx devices are fixed while the Rx performs beamforming towards the tag (as well as nulling for self-interference) depending on its position. The incident angle is varied from −45° to +45°. As we can see in Fig. 20, we can achieve consistently low BER over the 90° span and up to 4m distance.

Frame reception ratio. We now use the 802.11ad MikroTik devices as both Tx (AP) and Rx (client). We deploy the two in a 12m × 12m classroom along with a tag that backscatters by modulating the BF frames sent by the AP. As mentioned earlier in Sec. 5.2, the tag not only modulates the BF frame body part but also the checksum, making it possible for the unmodified commodity Rx to receive the frame. Since the received frame is processed through the (closed source, proprietary Wil6210) firmware, we directly measure the frame reception ratio (FRR) at the Rx along with SNR. To collect the SNR from commercial devices, we utilize the SNR report from feedback frames. In the BF process, the SSW feedback frames report the client's observed SNR back to the AP. The measured SNR is reported as 8-bit two's complement value of $4 \times (SNR - 19)$). The Rx extracts backscatter data by comparing known/expected BF frame bits with received ones. Fig. 21

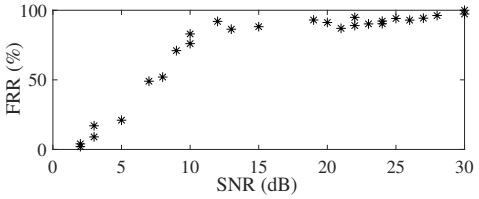


Figure 21: Frame Reception Ratio

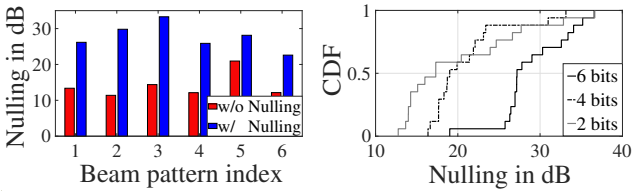
shows the FRR for different SNR values. Similar to previous results, when Rx performs beamforming toward the tag to achieve a high gain (SNR ≥ 10 dB), the average FRR is observed to be 91.4%.

6.2 Nulling backscatter self-interference

In this section, we evaluate how mmComb's beamforming with self-interference nulling performs. To do so, we perform extensive experiments to analyze: (i) the creation of custom beams with nulls, (ii) the impact of phase resolution, (iii) the subarray size in spatial smoothing, (iv) nulling with multiple nulls, (v) performance comparison with state-of-the-art nulling techniques, and (vi) different mmWave WiFi devices.

Creating beam patterns with nulls. To understand the nulling performance, we place the tag at arbitrary locations. The identification of the path and estimation of AoA are performed for backscatter and self-interference paths. We then compare two types of beams - one that is chosen directly from the codebook without any self-interference nulling and another one that uses mmComb nulling. Fig. 30 in Appendix A2 shows examples of six beam patterns before and after nulling with 4-bit phase resolution. The receiver was mechanically rotated while the transmitter sent signals over the LoS path to measure beam patterns. As shown in Fig. 22(a), the nulling performance can be calculated by subtracting the gain of the main lobe in the backscatter direction from the gain of the

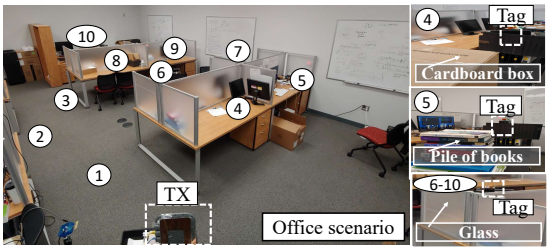
null in the self-interference direction. The results indicate an average nulling performance of 27.6 dB, with a maximum of 33.4 dB. Compared to the default beam pattern in the codebook, our nulling results in an average gain of 19 dB, but it reduces the gain of the main lobe to an average of 1.95 dB.



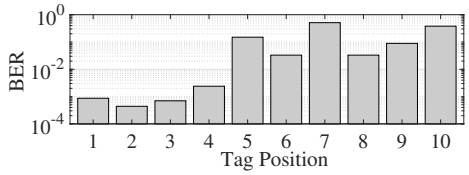
(a) (b)

Figure 22: Nulling performance for (a) 6 beam patterns shown in Fig. 30 and (b) 25 beams with different phase resolution.

Impact of phase resolution. Given that today's mmWave WiFi antenna arrays offer low phase resolution, we compare nulling performance at 25 different tag locations (different directions of backscatter and self-interference) for 2-bit, 4-bit, and 6-bit phase resolution. The results are shown in Fig. 22(b). We find that over 50% of measurements achieve nulling performance of 27.2 dB or higher with a 6-bit resolution. The performance degrades with a coarser resolution of phase control as we expected. However, the mean nulling performance is still 20.1 dB and 20.9 dB for 2-bit and 4-bit resolution, respectively. Additional evaluations for nulling (different devices, number and size of subarray in spatial smoothing, and multiple self-interference paths) can be found in Appendix A2.



(a) Evaluation for LoS (1-3) and NLoS (4-10).



(b) BER at different positions

Figure 23: mmComb performance in LOS and NLOS cases

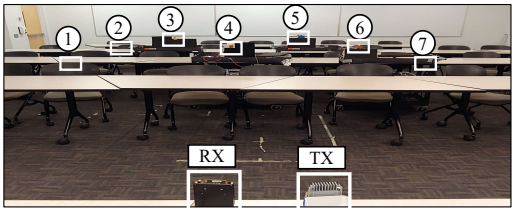
6.3 Practical deployment

We now evaluate a practical scenario where the tags are deployed in an office environment (a $10m \times 10m$ room) as shown in Fig. 23(a). We deploy an Airfide 802.11ad AP in the corner of the room. The AP transmits BF frames across 36 different sectors. The 802.11ad clients are receivers placed in various cubicles. The Fig 23(a) shows ten different tag positions. Positions (1-3) have a LoS path with the AP. The remaining positions (4-10) have blockage (cardboard boxes, books, etc.) and the signal either is received after penetration or reflection (NLoS) from nearby objects.

Fig. 23(b) shows the BER for 10 tag locations. We find that BER for positions 1-3 is less than 10^{-3} due to LoS (similar to Sec. 6.1). For NLoS cases, position 4 achieves low BER due to low penetration loss from the cardboard. The BER increases in positions 5-10 with books and glass blockages. Given that glass has a relative permittivity (ε) of 3.7 to 10 [19], the backscatter SNR drops. However, the corresponding average BER is still 2.1×10^{-1}. This is primarily due to the received beamforming towards the tag and self-interference nulling for other reflections in the room which significantly improves the SINR even when the incoming signal to the tag is weak.

6.4 Scalability of mmComb

We evaluate the scalability of mmComb with multiple tags and simultaneous operations. We create a scenario where we place tags at 7 positions in a $10m \times 10m$ classroom as shown in Fig. 24(a). We first operate the tags individually (with no other tag interference). Fig. 24(b) shows 10 Tx sectors (5 each



(a) Experiment setting for multiple tags.

Single tag response				Two tags response			
TX sector	Tag Response	RX beamforming	Null for other tags	TX sector	Tag Response	RX beamforming	Null for other tags
TX1	Tag3	Tag3	-	TX3	Tag1 & Tag4	Tag 1/4	Tag 4/1
TX2	Tag2	Tag2	-	TX6	Tag2 & Tag3	Tag 2/3	Tag3/2
TX4	Tag5	Tag5	-	TX8	Tag3 & Tag5	Tag 3/5	Tag 5/3
TX5	Tag2	Tag2	-	TX9	Tag6 & Tag7	Tag 6/7	Tag 7/6
TX7	Tag1	Tag1	-	TX10	Tag1 & Tag3	Tag 1/3	Tag 3/1

(b) Tag response corresponding to a TX sector.

Figure 24: Multitag experiment.

for one and two tag responses) and the corresponding tags that can respond when a BF frame is sent in that Tx sector. We note that this setup uses a COTS 802.11ad AP as the Tx where the phased array antenna has non-uniform beam patterns with non-trivial sidelobes. This represents a real-world scenario and we anticipate the performance to be better (i.e., less inter-tag interference) when antenna arrays achieve a better directionality in the future. We observe that due to the directionality of Tx sectors, at most one tag responds in the majority (82.8%) of Tx sectors (not all Tx sectors are shown in Fig. 24(b)) even when they are densely deployed. In such cases, the Rx can beamform to the responding tag in each Tx sector to receive data from it in a time-divided manner (i.e., TDMA). We also observe that in a few Tx sec-

tors (17.2%), two tags respond. This is primarily attributed to the non-uniformity of Tx beams in today's phased arrays. As shown in Fig. 24(b), the Rx can beamform to one of the two tags and null to the other to reduce interference. Even with various randomly generated deployments in the presented scenario, we find that at most two tags respond in one Tx sector, requiring nulling interference from at most one tag at any time. *We note that interference nulling through Rx beamforming is required only when two tags respond in the same Tx sector. It is not required if only one tag responds in a given Tx sector.* When two tags respond in the same sector, the Rx can beamform to one and null to the other one by one in different beamforming rounds to receive data from both of them in a TDMA manner. To further validate this, we

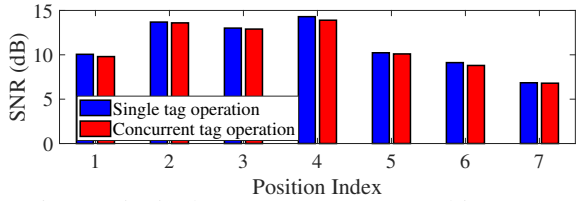


Figure 25: Single tag vs. concurrent multi-tag SNR.

operate all seven tags in Fig. 24(a) simultaneously and use the TDMA-based tag communication for different Tx sectors and corresponding Rx sectors as shown in Fig. 24(b). Fig. 25 shows the difference between tags operating separately versus all tags operating concurrently. We find that even when the Rx does not perform nulling for other tags (e.g., TX1 or TX2) because only one tag responds in the Tx sector, concurrent operation of the other tags does not affect the SNR (an average SNR difference of only 0.2 dB). Similarly, when the Rx performs nulling (e.g., TX3 or TX6) in the case when two tags respond, the SNR difference between separate and concurrent operations is still very small (average 0.2 dB). Overall, these results show that mmComb can scale to multiple tags with backscatter communication happening in a time-divided manner (different tags readable in different AP sectors).

6.5 Power consumption

mmComb tag consists of four main components: clock, modulator, frame detector, and SPDT switch. We analyze the power consumption of the components using Libero SoC SmartPower [24]. Serial inverters are used to generate a 55 MHz clock which is then input to the modulator and frame detector. The clock consumes $7.5\mu W$. The modulator provides control input to the switch for embedding backscatter bits in the signal and consumes only $2\mu W$. The frame detector module consumes $76.8\mu W$ and our GotMIC gSSD0011 SPDT switch only needs less than $1\mu W$ [14]. This results in total power consumption of $87.3\mu W$. This means that the tag can operate with a 1000mAh coin cell battery for over 4 years. Furthermore, it can operate battery-free using a solar cell which can harvest $100\mu W$ in a typically illuminated office environment [33].

7 Related Work

Conventional and sub-6 GHz WiFi backscatter systems. Conventional backscatter systems such as UHF RFID have been studied for over two decades [11, 16, 30]. Compared to mmComb which can provide megabits per second data rates, these systems are primarily designed for low-power, low-rate reader-tag communication. High cost [12] of deploying dedicated readers has given rise to commodity WiFi backscatter systems. Such systems include WiFi backscatter [17], BackFi [7], Passive WiFi [18], Hitchhike [53] and WiTAG [3]. These systems are primarily designed for WiFi in sub-6 GHz bands.

mmWave WiFi and backscatter. mmWave backscatter systems have been recently proposed in [6, 28, 41]. Both [41] and [6] do not use WiFi commodity devices as readers but instead use FMCW radars. Apart from that, both works achieve limited data rates compared to mmComb. A high-speed mmWave backscatter system was proposed in [28] achieving a data rate of 100 Mbps. However, the proposed system requires additional hardware for a reader and a specialized antenna that can isolate backscatter signals using polarization. Extensive measurement studies have been conducted to understand 60 GHz links [36–38, 44, 49, 58] and interference [5, 31, 42, 52]. Creating nulls in the direction of interference has been studied in prior works including [15, 26, 29, 51]. Various novel types of algorithms including genetic algorithm [15, 26], BAT algorithm [47], Neural Network [13, 22] have been used to create antenna patterns with desired null behavior. The application of these algorithms poses several limitations in our case as the backscatter and self-interference signals are strongly correlated.

8 Conclusion and Discussion

In this paper, we introduced a 60 GHz mmWave WiFi commodity backscatter system. We believe that our work can be improved in three following aspects: (1) Tx beamforming: mmComb currently uses Rx beamforming at the client to receive data from the tag, while the AP utilizes unmodified codebook beams for BF frames. Incorporating TX beamforming along with RX beamforming can further reduce the self-interference and improve backscatter SINR and BER. The Tx beamforming will also improve multi-tag operations by reducing inter-tag interference (i.e., improve TDMA) as Tx beamforming naturally assigns specific tags to different TX sectors under densely deployed multi-tag cases. Guaranteeing protocol compatibility with AP beamforming to the tags is a challenging issue that needs to be addressed in this context. (2) Tag antenna: An improved antenna design on the tag with a higher gain and wider field of view can help us further improve the backscatter SNR and BER. (3) Data rates: We note that our work only scratches the surface in terms of mmWave commodity backscatter. With an ample amount of available bandwidth, our work can be extended even further to achieve higher data rates while being commodity-compliant to realize the full potential of mmWave backscatter.

References

[1] GLOBAL IOT CONNECTIONS TO HIT 29.4 BILLION IN 2030. https://transformainsights.com/news/global-iot-connections-294.

[2] Mikrotik 60 GHz 802.11ad AP. https://mikrotik.com/product/wap_60g_ap.

[3] Ali Abedi, Farzan Dehbashi, Mohammad Hossein Mazaheri, Omid Abari, and Tim Brecht. Witag: Seamless wifi backscatter communication. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 240–252, 2020.

[4] Airfide AFN2200. https://airfidenet.com/.

[5] H. Assasa, S. Kumar Saha, A. Loch, D. Koutsonikolas, and J. Widmer. Medium access and transport protocol aspects in practical 802.11 ad networks. In *2018 IEEE 19th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, pages 1–11, 2018.

[6] Kang Min Bae, Namjo Ahn, Yoon Chae, Parth Pathak, Sung-Min Sohn, and Song Min Kim. Omniscatter: extreme sensitivity mmwave backscattering using commodity fmcw radar. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 316–329, 2022.

[7] Dinesh Bharadia, Kiran Raj Joshi, Manikanta Kotaru, and Sachin Katti. Backfi: High throughput wifi backscatter. *ACM SIGCOMM Computer Communication Review*, 45(4):283–296, 2015.

[8] Spyridon-Nektarios Daskalakis, John Kimionis, Ana Collado, Manos M Tentzeris, and Apostolos Georgiadis. Ambient fm backscattering for smart agricultural monitoring. In *2017 IEEE MTT-S International Microwave Symposium (IMS)*, pages 1339–1341. IEEE, 2017.

[9] Daniel Dobkin. *The rf in RFID: uhf RFID in practice*. Newnes, 2012.

[10] Weixiu Du and Rodney Lynn Kirlin. Improved spatial smoothing techniques for doa estimation of coherent signals. *IEEE Transactions on signal processing*, 39(5):1208–1210, 1991.

[11] Klaus Finkenzeller. *RFID handbook: fundamentals and applications in contactless smart cards, radio frequency identification and near-field communication*. John wiley & sons, 2010.

[12] Gary M Gaukler and Ralf W Seifert. Applications of rfid in supply chains. *Trends in supply chain design and management*, pages 29–48, 2007.

[13] R Ghayoula, N Fadlallah, A Gharsallah, and M Rammal. Phase-only adaptive nulling with neural networks for antenna array synthesis. *IET microwaves, antennas & propagation*, 3(1):154–163, 2009.

[14] Gotmic AB. gssd0011. https://gotmic.se/switches.html.

[15] Randy L Haupt. Phase-only adaptive nulling with a genetic algorithm. *IEEE Transactions on Antennas and Propagation*, 45(6):1009–1015, 1997.

[16] Elisabeth Ilie-Zudor, Zsolt Kemény, Fred Van Blommestein, László Monostori, and André Van Der Meulen. A survey of applications and requirements of unique identification systems and rfid techniques. *Computers in Industry*, 62(3):227–252, 2011.

[17] Bryce Kellogg, Aaron Parks, Shyamnath Gollakota, Joshua R Smith, and David Wetherall. Wi-fi backscatter: Internet connectivity for rf-powered devices. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 607–618, 2014.

[18] Bryce Kellogg, Vamsi Talla, Shyamnath Gollakota, and Joshua R Smith. Passive {Wi-Fi}: Bringing low power to {Wi-Fi} transmissions. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 151–164, 2016.

[19] Ghenadii Korotcenkov. *Handbook of humidity measurement, volume 2: Electronic and electrical humidity sensors*. CRC Press, 2019.

[20] Manikanta Kotaru, Kiran Joshi, Dinesh Bharadia, and Sachin Katti. Spotfi: Decimeter level localization using wifi. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 269–282, 2015.

[21] Han-il Lee, Tae-young Choi, Saeed Mohammadi, and Linda PB Katehi. An extremely low power 2 ghz cmos lc vco for wireless communication applications. In *The European Conference on Wireless Technology, 2005.*, pages 31–34. IEEE, 2005.

[22] Bo Li, Tara N Sainath, Ron J Weiss, Kevin W Wilson, and Michiel Bacchiani. Neural network adaptive beamforming for robust multichannel speech recognition. 2016.

[23] Zhengxiong Li, Baicheng Chen, Zhuolin Yang, Huining Li, Chenhan Xu, Xingyu Chen, Kun Wang, and Wenyao Xu. Ferrotag: A paper-based mmwave-scannable tagging infrastructure. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, pages 324–337, 2019.

[24] Libero. Soc v11.8 archive. https://www.microsemi.com/product-directory/root/5485-libero-soc-v11-8-archive.

[25] Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Gollakota, David Wetherall, and Joshua R Smith. Ambient backscatter: Wireless communication out of thin air. *ACM SIGCOMM computer communication review*, 43(4):39–50, 2013.

[26] Sohrab Madani, Suraj Jog, Jesus O Lacruz, Joerg Widmer, and Haitham Hassanieh. Practical null steering in millimeter wave networks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 903–921, 2021.

[27] Mohammad H. Mazaheri, Soroush Ameli, Ali Abedi, and Omid Abari. A millimeter wave network for billions of things. SIGCOMM '19, 2019.

[28] Mohammad Hossein Mazaheri, Alex Chen, and Omid Abari. mmtag: a millimeter wave backscatter network. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 463–474, 2021.

[29] Moctar Mouhamadou, Patrick Vaudon, and Mohammed Rammal. Smart antenna array patterns synthesis: Null steering and multi-user beamforming by phase control. *Progress In Electromagnetics Research*, 60:95–106, 2006.

[30] Pavel V Nikitin and KV Seshagiri Rao. Theory and measurement of backscattering from rfid tags. *IEEE Antennas and Propagation Magazine*, 48(6):212–218, 2006.

[31] Thomas Nitsche, Guillermo Bielsa, Irene Tejado, Adrian Loch, and Joerg Widmer. Boon and bane of 60 ghz networks: Practical insights into beamforming, interference, and frame level operation. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery.

[32] Thomas Nitsche, Carlos Cordeiro, Adriana B Flores, Edward W Knightly, Eldad Perahia, and Joerg C Widmer. Ieee 802.11 ad: directional 60 ghz communication for multi-gigabit-per-second wi-fi. *IEEE Communications Magazine*, 52(12):132–141, 2014.

[33] Joseph A Paradiso and Thad Starner. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive computing*, 4(1):18–27, 2005.

[34] Ioannis Pefkianakis and Kyu-Han Kim. Accurate 3d localization for 60 ghz networks. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 120–131, 2018.

[35] Eldad Perahia, Carlos Cordeiro, Minyoung Park, and L Lily Yang. Ieee 802.11 ad: Defining the next generation multi-gbps wi-fi. In *2010 7th IEEE consumer communications and networking conference*, pages 1–5. IEEE, 2010.

[36] S. K. Saha, H. Assasa, A. Loch, N. M. Prakash, R. Shyamsunder, S. Aggarwal, D. Steinmetzer, D. Koutsonikolas, J. Widmer, and M. Hollick. Fast and infuriating: Performance and pitfalls of 60 ghz wlans based on consumer-grade hardware. In *2018 15th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, pages 1–9, 2018.

[37] S. K. Saha, V. V. Vira, A. Garg, and D. Koutsonikolas. A feasibility study of 60 ghz indoor wlans. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, Aug 2016.

[38] Swetank Kumar Saha, Yasaman Ghasempour, Muhammad Kumail Haider, Tariq Siddiqui, Paulo De Melo, Neerad Somanchi, Luke Zakrajsek, Arjun Singh, Owen Torres, Daniel Uvaydov, Josep Miquel Jornet, Edward Knightly, Dimitrios Koutsonikolas, Dimitris Pados, and Zhi Sun. X60: A programmable testbed for wideband 60 ghz wlans with phased arrays. In *Proceedings of the 11th Workshop on Wireless Network Testbeds, Experimental Evaluation and CHaracterization*, WiNTECH '17, page 75–82, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Swetank Kumar Saha, Tariq Siddiqui, Dimitrios Koutsonikolas, Adrian Loch, Joerg Widmer, and Ramalingam Sridhar. A detailed look into power consumption of commodity 60 ghz devices. In *2017 IEEE 18th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 1–10. IEEE, 2017.

[40] Tie-Jun Shan and Thomas Kailath. Adaptive beamforming for coherent signals and interference. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 33(3):527–536, 1985.

[41] Elahe Soltanaghaei, Akarsh Prabhakara, Artur Balanuta, Matthew Anderson, Jan M Rabaey, Swarun Kumar, and Anthony Rowe. Millimetro: mmwave retro-reflective tags for accurate, long range localization. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 69–82, 2021.

[42] Daniel Steinmetzer, Daniel Wegemer, Matthias Schulz, Joerg Widmer, and Matthias Hollick. Compressive millimeter-wave sector selection in off-the-shelf ieee 802.11ad devices. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, page 414–425,

New York, NY, USA, 2017. Association for Computing Machinery.

[43] Sanjib Sur, Ioannis Pefkianakis, Xinyu Zhang, and Kyu-Han Kim. Towards scalable and ubiquitous millimeter-wave wireless networks. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 257–271, 2018.

[44] Sanjib Sur, Vignesh Venkateswaran, Xinyu Zhang, and Parmesh Ramanathan. 60 ghz indoor networking through flexible beams: A link-level profiling. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '15, page 71–84, New York, NY, USA, 2015. Association for Computing Machinery.

[45] Sanjib Sur, Xinyu Zhang, Parmesh Ramanathan, and Ranveer Chandra. Beamspy: enabling robust 60 ghz links under blockage. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 193–206. USENIX Association, 2016.

[46] Vamsi Talla, Mehrdad Hessar, Bryce Kellogg, Ali Najafi, Joshua R Smith, and Shyamnath Gollakota. Lora backscatter: Enabling the vision of ubiquitous connectivity. *Proceedings of the ACM on interactive, mobile, wearable and ubiquitous technologies*, 1(3):1–24, 2017.

[47] Tong Van Luyen and Truong Vu Bang Giang. Interference suppression of ula antennas by phase-only control using bat algorithm. *IEEE Antennas and Wireless Propagation Letters*, 16:3038–3042, 2017.

[48] Jie Xiong and Kyle Jamieson. {ArrayTrack}: A {Fine-Grained} indoor location system. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 71–84, 2013.

[49] Hao Xu, Vikas Kukshya, and Theodore S Rappaport. Spatial and temporal characteristics of 60-ghz indoor channels. *IEEE Journal on selected areas in communications*, 20(3):620–630, 2002.

[50] Y Xu and Q Zhaojun. Research on interference test of 24ghz millimeter wave radar to 5g equipment. In *Journal of Physics: Conference Series*, volume 1584, page 012028. IOP Publishing, 2020.

[51] Shiwen Yang, Yeow Beng Gan, and Anyong Qing. Antenna-array pattern nulling using a differential evolution algorithm. *International Journal of RF and Microwave Computer-Aided Engineering: Co-sponsored by the Center for Advanced Manufacturing and Packaging of Microwave, Optical, and Digital Electronics (CAMPmode) at the University of Colorado at Boulder*, 14(1):57–63, 2004.

[52] D. Zhang, P. S. Santhalingam, P. Pathak, and Z. Zheng. Characterizing interference mitigation techniques in dense 60 ghz mmwave wlans. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, 2019.

[53] Pengyu Zhang, Dinesh Bharadia, Kiran Joshi, and Sachin Katti. Hitchhike: Practical backscatter using commodity wifi. SenSys '16, 2016.

[54] Pengyu Zhang, Colleen Josephson, Dinesh Bharadia, and Sachin Katti. Freerider: Backscatter communication using commodity radios. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 389–401, 2017.

[55] Jia Zhao, Wei Gong, and Jiangchuan Liu. Spatial stream backscatter using commodity wifi. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 191–203, 2018.

[56] Jia Zhao, Wei Gong, and Jiangchuan Liu. X-tandem: Towards multi-hop backscatter communication with commodity wifi. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 497–511, 2018.

[57] Renjie Zhao, Timothy Woodford, Teng Wei, Kun Qian, and Xinyu Zhang. M-cube: A millimeter-wave massive mimo software radio. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, MobiCom '20, New York, NY, USA, 2020. Association for Computing Machinery.

[58] Yibo Zhu, Zengbin Zhang, Zhinus Marzi, Chris Nelson, Upamanyu Madhow, Ben Y. Zhao, and Haitao Zheng. Demystifying 60ghz outdoor picocells. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, MobiCom '14, page 5–16, New York, NY, USA, 2014. Association for Computing Machinery.

Appendix A1

Estimating the AoA of incoming signals. We adopt a power delay profile-based solution (used for beam alignment in [43] and localization in [34]) for the purpose. As shown in Fig. 26, each peak in power delay profile corresponds to a path taken by the transmitted signal and arriving at different times at the receiver. The amplitude of each peak depends on

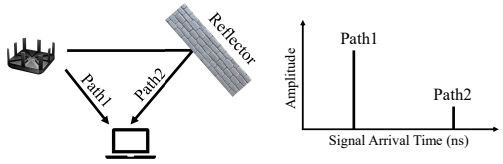


Figure 26: A power delay profile of incoming signals.

the gain of the antenna beam pattern toward the path direction.

Even for different beam patterns, the relative time difference between the peaks remains unchanged and only the amplitude is affected. With K paths and different time delay Δt_d for each path and angle of arrival θ_i, the received signal with the mth beam pattern can be expressed as:

$$y(t) = \sum_{i=1}^{K} \sum_{\theta} A_m(\theta_i^{az}, \theta_i^{el}) \cdot h_i(\theta_i^{az}, \theta_i^{el}) \cdot x(t - \Delta t_{d_i}) \quad (4)$$

where $A_m(\theta_i^{az}, \theta_i^{el})$ and $h_i(\theta_i^{az}, \theta_i^{el})$ is the gain of mth beam and the complex channel gain, respectively. If we measure the power delay profile with different receive beams $m = 1, ..., M$, each measurement provides a set of peaks $Amp_m = \{a_{m,1}, a_{m,2}, ..., a_{m,k}\}$. Hence, we can collect the amplitude set $Amp_{m,k} = \{a_{1,k}, a_{2,k}, ..., a_{M,k}\}$ with M beam patterns for k^{th} path. Due to the different gain of mth beam for different AoA, the peaks will have different amplitudes for the same physical paths. Hence, we can utilize this to estimate the AoA by comparing the normalized amplitude set $Amp'_{m,k} = \frac{Amp_{m,k}}{min(Amp_{m,k})}$ and beam pattern gain $G'_{m,\theta} = \frac{G_{m,\theta}}{min(G_{m,\theta})}$ where $G_{m,\theta}$ denotes the gain of mth beam pattern toward θ. Hence, the angle of arrival θ_k of kth path can be determined by solving following optimization:

$$\theta_k = \arg\min_{\theta} |Amp'_{m,k} - G'_{m,\theta}|^2 \quad (5)$$

Equ. 5 searches for a discrete angle θ_k that provides a minimum error between the measured CIR and beam pattern. Given this is a non-linear error curve fitting problem, we use the least mean square algorithm to find an approximate solution.

Appendix A2

Different Rx devices. As discussed in Sec. 5, we use two types of Rx devices: commodity 802.11ad AP from Airfide and the one with SiversIMA RF frontend. Since both have different types of antenna arrays, we compare their performance in terms of SINR when nulling is performed. We create custom beam patterns based on different tag positions (different backscatter and self-interference angles). For a fair comparison, we use 2-bit phase resolution (maximum for Airfide) on both. Fig. 27 shows a comparison of the two devices for an example beam pattern as well as SINR for three cases: (i) beam without nulling, (ii) custom beam on Airfide with nulling, and (iii) custom beam on SiversIMA with nulling. We find that our nulling and backscatter beamforming improves the SINR on both Rx devices. SiversIMA achieves a higher SINR due to the greater number of elements compared to Airfide.

Number and size of subarrays in spatial smoothing. Figure 28(a) illustrates the impact of different numbers and sizes of subarrays used for spatial smoothing in decorrelating incoming signals. We use the SiversIMA RF frontend with 16 antenna elements and form different combinations of subarrays. Initially, we evaluate the performance without spatial smoothing, i.e., using correlated signals for beam creation. We find that the average SINR nulling performance for a given

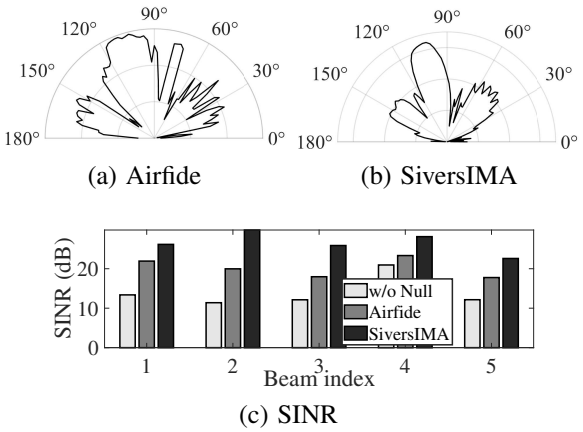


Figure 27: Backscatter SINR for different Rx devices.

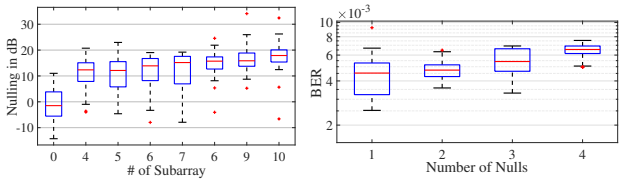


Figure 28: Nulling performance and backscatter BER for different (a) number of subarrays and (b) number of nulls.

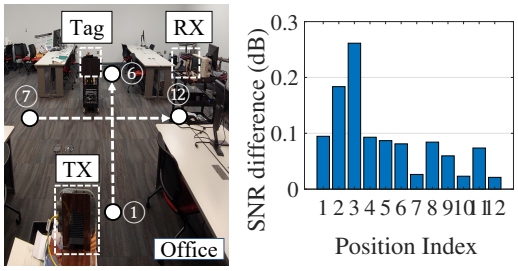
tag direction and self-interference is -1.4 dB. Next, we create different numbers and sizes of subarrays where the size of each subarray is determined as $L - N + 1$, where L and N are the total numbers of antenna elements and the number of subarrays, respectively. For instance, four subarrays imply that the size of each subarray is 13 elements. As demonstrated in Figure 28(a), average nulling performance improves by 6.5 dB as the number of subarrays increases from 4 to 10. Theoretically, n antenna elements provide a gain of $10, log, n$ dB. Therefore, the gain reduction from using 7 elements instead of 16 elements is 3.5 dB. However, spatial smoothing enhances the nulling performance by 18.5 dB due to better beam design after decorrelating the signals.

Multiple self-interference paths. In indoor scenarios, there can be multiple paths between a Tx and Rx position, even though not all paths are active simultaneously due to the directionality of Tx and Rx. To handle this, mmComb needs to generate Rx beams that can create nulls in multiple directions while still maintaining high gain towards the tag. To evaluate this, we set up an indoor environment where the tag's direction can be one of nine possible angles (depending on tag position), while the self-interference comes from 1, 2, 3, or 4 paths simultaneously out of 8 possible angles. Based on the number and angle of paths, we calculate the Rx beam pattern with multiple nulls. Fig. 28(b) shows the backscatter BER for different numbers of nulls. As expected, the BER increases as the number of nulls increases, but even with four nulls, the average BER remains 6.5×10^{-3}. This indicates that mmComb

can improve the SINR by suppressing self-interference, even in situations with severe multipath propagation.

Appendix A3

Impact on existing mmWave WiFi. As mmComb is designed to integrate tags into existing mmWave WiFi networks by reusing existing signals, we need to evaluate the potential impact of a tag's operation on other devices in the network. Since the tag responds to beamforming frames from the AP, a potential issue could arise where the tag's operations affect the beamforming between the AP and a client device (a receiver not intended to receive from the tag). As shown in Fig. 1 earlier, such a client device operates in quasi-omni mode when the AP sends out the beamforming frames, making it more susceptible to receiving backscatter data from surrounding tags.



(a) (b)

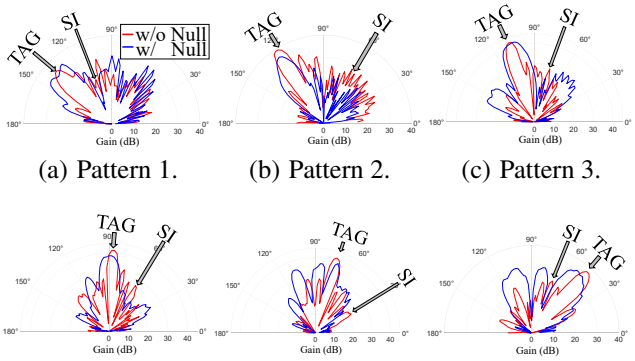
Figure 29: Impact of mmComb on mmWave WiFi: (a) experiment setting and (b) SNR different in beamforming

To evaluate the impact of tag operation on other devices in the mmWave WiFi network, we created a scenario where the AP and a client (not a tag receiver) were interested in performing beamforming, and the tag's operation could result in incorrect SNR measurements at the client. The client uses quasi-omni patterns and faces the tag to evaluate the worst-case impact of the tag operation. As shown in Fig. 29(a), the AP (Tx) is located close to position 1. The tag is located at a 2.5m distance from the Tx. The Rx is moved from positions 1 through 12.

Fig. 29(b) shows the difference in SNR observed by the Rx from the AP with and without tag operation. We find that the SNR difference introduced by the tag operation is on average 0.18 dB for locations 1-3 and 0.06 dB for locations 4-10. The mean difference is no more than 0.19 dB. Therefore, we concluded that the tag operations do not negatively impact the beamforming between WLAN devices because when the client operates in the quasi-omni beam, the backscattered signal from the tag to the client is much weaker compared to the direct signal from the AP. Consequently, tags can coexist in a mmWave WiFi network without causing significant interference.

Appendix A4

Fig. 30 shows examples of six beam patterns before and after nulling with 4-bit phase resolution. The tag directions are $\{-45°, -30°, -15°, 0°, 15°, 45°\}$ for each patterns, while having a self-interference direction at $\{-15°, 30°, 15°, 15°, 60°, 20°\}$ accordingly.



(a) Pattern 1. (b) Pattern 2. (c) Pattern 3.

(d) Pattern 4. (e) Pattern 5. (f) Pattern 6.

Figure 30: Beam patterns with and without nulling.

Where The Wild Things Are: Brute-Force SSH Attacks In The Wild And How To Stop Them

Sachin Kumar Singh
University of Utah

Shreeman Gautam
University of Utah

Cameron Cartier
University of Utah
Black Hills Information Security

Sameer Patil
University of Utah

Robert Ricci
University of Utah

Abstract

SSH (Secure Shell) is widely used for remote access to systems and cloud services. This access comes with the persistent threat of SSH password-guessing brute-force attacks (BFAs) directed at `sshd`-enabled devices connected to the Internet. In this work, we present a comprehensive study of such attacks on a production facility (CloudLab), offering previously unreported insight. Our study provides a detailed analysis of SSH BFAs occurring on the Internet today through an in-depth analysis of `sshd` logs collected over a period of four years from over 500 servers. We report several patterns in attacker behavior, present insight on the targets of the attacks, and devise a method for tracking individual attacks over time across sources. Leveraging our insight, we develop a defense mechanism against SSH BFAs that blocks 99.5% of such attacks, significantly outperforming the 66.1% coverage of current state-of-the-art rate-based blocking while also cutting false positives by 83%. We have deployed our defense in production on CloudLab, where it catches four-fifths of SSH BFAs missed by other defense strategies.

1 Introduction

The Secure Shell [72] is widely used for remote administration and command execution. Due to this popularity, it is common for SSH servers to be targeted for password-guessing Brute-Force Attacks (BFAs). In such attacks, a malicious party attempts to connect to an SSH server using one or more `{username,password}` pairs, guessing values for both fields. Attempting to brute-force SSH may seem like an "outdated" attack: best practices recommend key-based authentication [29], many IPv4 devices are behind NAT [28], and scanning for hosts on IPv6 is notoriously difficult [36]. Yet, our experience shows that BFAs are still prevalent– in fact, increasing—on the IPv4 Internet. Such attacks are leveraged to exploit poorly configured and poorly secured SSH servers to build botnets [56]. If the attack against a machine succeeds, the attackers can use the machine to carry out further attacks as part of the botnet [11, 13, 19, 43, 46] or steal its computing resources (e.g., for cryptomining [24, 59]).

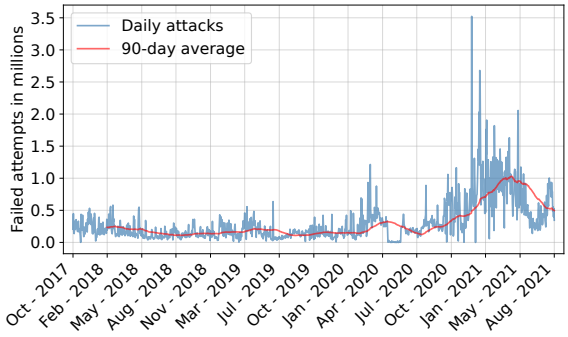


Figure 1: Failed SSH attempts on CloudLab (in millions).

While others have studied SSH attacks, existing studies involve smaller timescales [10, 30, 58, 71], examine fewer hosts [10, 58], use honeypots that do not include legitimate users [10, 30, 58, 71], or cover periods that are not recent [10, 58]. In contrast, we captured real-world BFAs mixed with the activities of legitimate users in a live production facility by capturing `sshd` logs for four years across ≈500 servers in CloudLab [27]. Due to this unique characteristic, our dataset provides a rare opportunity to understand changing attacker behavior over a longitudinal span and to compare it with the practices of legitimate users. Our analysis shows that SSH BFAs are evolving. As Figure 1 illustrates, BFA attackers are becoming more aggressive, with daily attempts increasing, particularly in recent years.

By "fingerprinting" usernames, we were able to track many attackers over time and across IP addresses. Our analysis revealed a wide range of attacker behaviors. Many attack quickly and then disappear, while others persist in their attacks for months or years; some focus on one username, while others attempt thousands. We found that attackers target a wide variety of devices (e.g., servers, Internet of Things [IoT] devices, routers) and software (e.g., databases, games, chat servers), shifting from typical administrator usernames

(root and admin) to those associated with cloud images, network and IoT devices, and specific software. Based on observing that username fingerprints are a strong differentiator between attackers and legitimate users, we designed Dictionary-Based Blocking (DBB)—a novel technique for blocking SSH BFAs with high effectiveness. DBB blocks traffic based on *dictionaries* of usernames attempted by attackers. The low false positive rates of DBB ensure that legitimate users retain reliable access.

With our research, we make the following contributions:

- We present an analysis of the properties of the SSH brute-force attacks going on today, including insights about their methodology, inferences about their targets, investigations into their network sources, and analysis of attackers' persistence. (Sections 4, 5, 6)
- We develop a method for creating username dictionaries that allow us to track attacks across IP addresses and over time, even when data is incomplete or attackers make small adjustments to their username lists. (Section 7)
- We present a new method, Dictionary Based Blocking (DBB), for blocking SSH brute-force attacks using these dictionaries. (Section 8)
- We evaluate DBB using real-world data, showing it to be highly effective at blocking attackers while allowing legitimate users access. (Section 8.3)
- We present results from a production deployment of DBB, showing that it performs exceptionally well in practice. (Section 8.5)

We begin with an overview of related literature in Section 2 and data collection setting and method in Section 3.

2 Related Work

The literature most relevant to our work analyzes SSH BFAs and examines approaches to block them.

Analyzing SSH BFAs: An analysis of 103K login attempts from 271 IPs on three honeypots over 11 weeks revealed that the attacks target administrator as well as user accounts and can be thwarted with strong passwords and key-based logins [58]. More recently, researchers found that such attacks employ root and admin as popular usernames (95% and 3% of attempts, respectively) [10]. Complimentary research has focused on the use of stolen credentials and botnets when carrying out SSH BFAs [17, 71].

SSH BFAs have been captured by honeypots deployed via IoT hardware and software as well and demonstrated that attackers who gain access engage in diverse activities, such as bitcoin mining, UDP/TCP flooding, SSH scans, and SSH port forwarding [24]. In a larger-scale deployment via 102 medium-interaction honeypots across three continents, researchers monitored 12 million connections originating from 38K unique IPs and examined how attacker behavior is influenced by the location of the honeypot, the difficulty of compromise, and the variety of files available on the honeypot [15]. Use of cipher suites and SSH version strings to fingerprint the mechanisms used in the attacks has identified that attackers use popular tools off-the-shelf software, such as Ncrack [6] and Hydra [4], as well as custom tools [30].

A recent deployment of medium-interaction honeypots for five protocols captured 73K IP addresses, noting an increasing frequency of SSH attacks throughout the period. More longitudinal honeypot deployments have observed a significant number of IP addresses engaged in multiple activities over 15 months [47] and two years [40]. The data covered various attacks, including SSH BFAs and non-standard port accesses, and the researchers observed that attacker preferences were relatively stable [40]. The studies mentioned above mostly use honeypots that collect only attacks. In contrast, we examined data from a production system, enabling us to compare attackers and legitimate users accessing the *same* system.

Blocking SSH BFAs: Researchers have proposed various approaches to detect and block BFAs, including network flow analysis [26, 32–35, 42, 67] and machine learning/deep learning techniques [31, 37, 38, 45, 48, 49, 62]. However, employing network flow data to detect SSH BFAs can result in a high number of false positives.

Alternately, defenses against SSH BFAs can employ host-based approaches to track user/IP characteristics, such as failed attempts and interarrival time. Tools such as Fail2ban [3], denyhosts [1], and sshguard [9] use host-based blocking of suspicious traffic. They analyze authentication logs to compute relevant features, such as the number of failed login attempts, and block corresponding IP addresses with host-based firewalls such as iptables [7]. Fail2ban is one of the most widely used tools for stopping BFAs: it blocks IP addresses exceeding a threshold number of failed attempts within a specified period. Tuned time- and rate-based blocking mechanisms have been used to improve blocking strategies [63]. A simulation with synthetic data showed that a distributed active-response architecture can enable the sharing of relevant information—particularly attacking IP addresses—among trusted hosts [44]. However, such an approach is not privacy-preserving and requires a set of trusted hosts. By analyzing data from a production system, we were able to design and deploy a novel defense and compare its effectiveness (including false positives) with rate-based designs.

3 Data Collection

Our research is based on an analysis of sshd logs from CloudLab [27], a public facility used by academic researchers at institutions around the world. CloudLab has a cloud-like user model: the users are "tenants" who access servers *temporarily* assigned to them. Although CloudLab has some policy control, such as initial sshd and logging configurations, the users are not under direct control of CloudLab. Once users acquire control of their assigned nodes, they may, and occa-

sionally do, alter the SSH settings without any supervision or regulation from CloudLab. Therefore, an additional layer of SSH security, beyond the defaults permitting only key-based authentication and prohibiting `username,password` authentication, is needed.

We used two sets of log files collected by CloudLab. The first (*Log1*) was collected on a single cluster over four years (October 2017 – August 2021) and contains a large number of attacks that enabled us to study general trends in SSH BFAs. The second (*Log2*) was collected at three different CloudLab sites over ten weeks (November 2022 – January 2023) to evaluate our proposed blocking mechanism. The three CloudLab sites are geographically dispersed and use unrelated IP addresses owned by different networks.

SSH Logs: Each host (also called a "node" or "server") in CloudLab has a persistent public IPv4 address and runs `sshd` for remote access—the primary way legitimate users interact with the host. By default, CloudLab hosts do not run other public-facing services, though users are permitted to start their own services if they wish. All CloudLab hosts are configured to log SSH login attempts to a central `syslog` server from which we obtained our datasets. We parsed the logs to extract various relevant features, such as source IP addresses, attempted usernames, authentication responses, etc. The log files did not contain passwords. To get a view approximating the network boundary (e.g., firewall or bastion host), we first removed any SSH attempts originating from within CloudLab itself. After removing the internal attempts, *Log1* included $\approx$ 840K unique source IP addresses attempting $\approx$ 277K unique usernames and making $\approx$ 427M login attempts *Log2* contained $\approx$ 91K unique source IPs attempting $\approx$ 98K unique usernames and $\approx$ 213M login attempts.

When analyzing the logs, we took into account that log messages can be lost due to central server overload, network congestion, misconfigurations on the hosts or the server, or intentional configuration changes by users. We further considered that CloudLab hosts are responsive to SSH requests only when in use, with usage periods of varying lengths distributed unevenly. As a result, log data pertaining to any given host can exhibit short-term gaps corresponding to periods in which it is not in use. To avoid incorrect conclusions on account of such short-term gaps, we used methods that support "fuzzy" matching and looked at long-term trends. Of the hosts included in *Log1*, 352 logged SSH connections every day of the logged period, and the three sites in *Log2* averaged connections from 1,322 hosts daily (616, 384, and 322 each).

Distinguishing Legitimate Users From Attackers: We considered an IP address as belonging to a legitimate user if it had at least one *successful* login. In other words, we assumed that IP addresses associated only with *failed* logins belonged to attackers. CloudLab's default `sshd` configuration allows only public-key authentication, and the CloudLab staff know of no attacks using stolen private keys. Therefore, we assumed that only legitimate users logged in successfully. On the other hand, attackers typically attempt to log in using passwords. Even though CloudLab is set to accept only key-based authentication, password-based login attempts are logged despite being disallowed. Although legitimate users may attempt (and fail) to log in with passwords, nearly all failed login attempts in our logs appear to be part of attacks rather than erroneous password-based attempts from legitimate users.

Advantages of Production Facility over Honeypots: Unlike honeypot logs that exclusively contain attacks, the CloudLab logs we analyzed contain actions of attackers along with those of legitimate users. As a result, analyzing the data allowed us to explore the practices of legitimate users in addition to those of attackers, thus facilitating comparisons between the two. One of the main challenges of honeypot logs is the inability to test blocking strategies because of the absence of legitimate users, resulting in a lack of false positives for the assessment. In contrast, our data can serve the dual purpose of aiding the development of blocking strategies *and* providing a means to assess their effectiveness.

Metadata Limitations: In parts of our analysis, we used data regarding network owners and geographic locations of the attacker IP addresses [5, 14]. Since we fetched this data in Summer 2022, after collecting *Log1*, it is possible that some IP addresses changed ownership after we had logged attacks from them. In addition, the use of NAT and dynamic address assignment on source networks may obscure the true number of attacking devices captured in the logs.

Ethical Considerations: Collecting logs is a routine operation in facilities like CloudLab. We have reported results using large aggregates without identifying individual users. In cases where mentioning specific users can provide illustrative value, we have anonymized the usernames. At the same time, we have assumed that attackers do not have a legitimate expectation of privacy. Still, we have not mentioned IP addresses since the devices may be compromised unbeknownst to their owners, who may have no malicious intent.

4 The Anatomy of SSH BFAs

In this section, we cover the basic features of SSH BFAs, attaching specific numbers and concrete behaviors from *Log1* to illustrate the concepts in the general descriptions.

Source IP Addresses: An SSH BFA originates from a *source* to a set of *targets* using a *guessing vector* of credentials. Such attacks can come from a large number of IP addresses. For instance, *Log1* contains attacks from over 800,000 IP addresses, with at least one attack from 90% of the 249 countries with ISO country codes [39]. However, few conclusions about the individual or entity controlling an attack can be drawn from source IP addresses alone. In many cases, the attacking devices do not belong to malicious actors themselves but to botnets composed of compromised machines [13, 46, 56] infected with self-replicating worms [25] or used to mask the locations of the actual attackers.

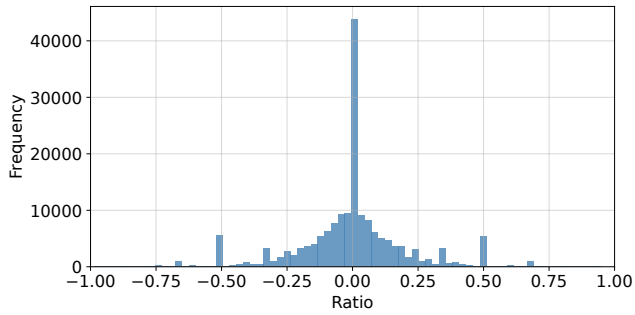


Figure 2: Histogram of sequence bias.

Target Host(s): Attackers may select hosts to target with several methods: scanning sequentially or randomly, checking for specific vulnerabilities on a host, etc. In practice, nearly all BFAs in *Log1* seemed to be based on random scanning of the IPv4 address space as seen in Figure 2, which is a histogram of the *sequence bias* for the sources in *Log1* that attacked at least 50 hosts. For each source IP address, we calculated the sequence bias as the fraction of successive target IP addresses higher than the one previously attacked. The sequence bias for a given sequence of IP addresses, $S_1, \ldots S_N$ is:

$$SB = \frac{1}{N-1}\sum_{i=1}^{N-2}\begin{cases}0 & \text{if } S_{i+1} = S_i\\ 1 & \text{if } S_{i+1} > S_i\\ -1 & \text{if } S_{i+1} < S_i\end{cases}$$

A sequence bias of 1 indicates that the source moves from lower-numbered addresses to higher ones 100% of the time, with -1 indicating the reverse movement order. A sequence bias of 0 means that each successive target is equally likely to be 'up' or 'down' in the IP space. The density of sources around zero—82% of the sources depicted in Figure 2 are in the range [-0.25,0.25]—indicates that most traverse the IP address space in random order.

Guessing Vectors: A BFA attacker attempts SSH login with a set of usernames and passwords. A username can be associated with multiple passwords and vice-versa. Each `{username, password}` combination is called a *credential vector*. Multiple such credential vectors are combined to construct a *guessing vector*. A study published in 2015 found that over 98% of BFAs contain the usernames `root` or `admin` [10]. Our more recent data shows that these usernames have become less dominant over time (see Figure 3). As seen in Figure 3, `root` has fallen dramatically, and `admin` is used only in a small fraction of attack attempts. Instead, as we present in more detail in Section 5, attackers are switching to usernames associated with cloud services and network devices.

5 Properties of SSH BFAs In Practice

After looking at the general structure of SSH BFAs, we used *Log1* to analyze specific aspects in more depth. We guided our analysis with a series of questions that arose from the general observations in the preceding section.

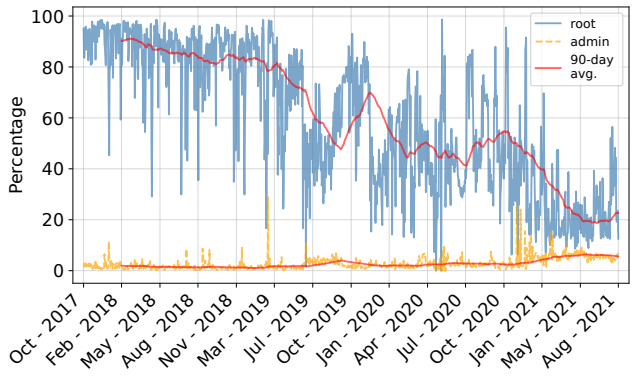


Figure 3: Percentage of BFAs with usernames `root` and `admin`.

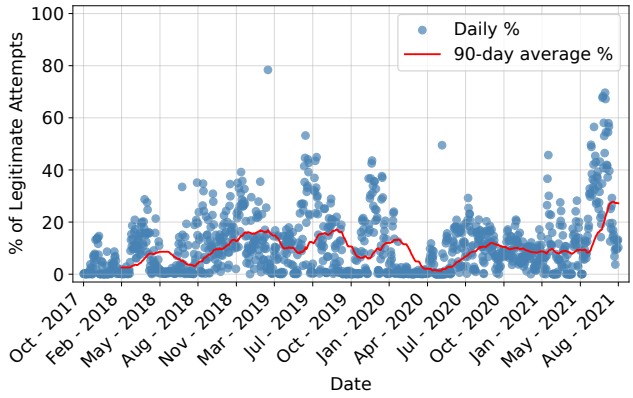


Figure 4: Percentage of accepted attempts at CloudLab.

How aggressive are individual attackers? A core reason behind blocking attacking IP addresses is that attackers attempt far more logins than real users. *Log1* shows that CloudLab is no exception, with only 10% of the login attempts being successful. On average, CloudLab experiences 25 successful SSH logins per minute as opposed to 211 failed ones.

Our data contains exceptions where the number of failed logins on a particular day did not outnumber successful ones. For instance, on a specific day, accepted attempts reached 621K—2.2 times the number of failed attempts. This case was mostly because of a single legitimate user, `river`,[1] who allocated numerous nodes and established thousands of SSH connections to each allocated node. Figure 4 shows the percentage of accepted attempts in the data: accepted SSH attempts were greater than failed ones on only 13 days in the four-year period covered by *Log1*. We speculate that the discrepant days may have resulted from the use of DevOps tools, such as Chef or Ansible [60], that make heavy use of SSH.

A majority of the failed attempts belonged to a small fraction of attacking IP addresses, with 1% accounting for 78%,

[1] Username changed for privacy.

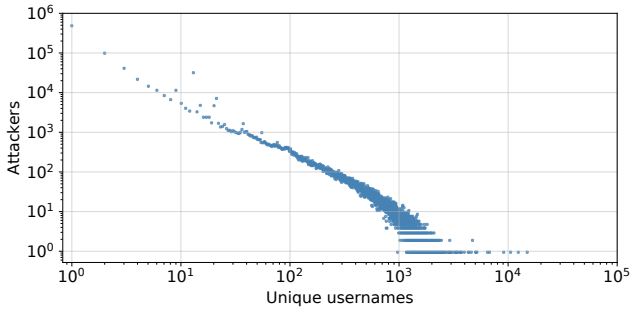


Figure 5: The number of usernames attempted vs. the number of attacking IP addresses using that many usernames. Both axes use a log scale to depict the extreme ranges.

2% for 85%, 4% for 90%, 8% for 95%, and 22% for 99% of all failed attempts. Over the four years represented in *Log1*, each attacking IP address performed 458 attempts on average. On the one hand, these numbers demonstrate that blocking by IP address is an attractive approach since blocking only a modest fraction of active attackers can greatly reduce attack volume. On the other hand, the numbers show that achieving perfect coverage by blocking specific IP addresses is difficult—moving from 99% to 100% coverage requires identifying and blocking nearly five times as many offending IP addresses.

Can we assume that all login failures are attacks? Of the ≈277K unique usernames in *Log1*, ≈99.995% were associated only with failed login attempts, but the remaining ≈0.005% had at least one accepted connection. For every legitimate username in *Log1*, there were about 199 associated only with failed attempts. Importantly, we found that *every* legitimate username logged at least one failure. We suspect failures associated with legitimate users are because of errors such as forgetting to add SSH keys to their agent, typos in hostnames, configuration errors on the servers, etc. The upshot is that administrators cannot deem an IP address to be malicious solely because of one, or even a few, failed login attempts. Since all legitimate users are likely to make the occasional error, effective identification of SSH attacks requires strategies more complex than simple failure-based blocking.

Do attackers try many usernames or focus on a small, high-value set? We uncovered numerous strategies regarding the number of usernames attackers tried. The number of usernames employed by each source IP address varied from one to ≈14K. More than half (54%) of the attacking IP addresses attempted only a single username in their guessing vectors. On average, attackers attempted 28 usernames, with 75% of the attackers using fewer than seven usernames and 90% using fewer than 39. A high variance (≈15170) in the distribution of usernames per source IP address shows that the number of usernames attempted by attackers is highly dispersed.

Figure 5 shows the number of usernames attempted against the number of attackers who attempted that many usernames. It can be seen in Figure 5 that a large number of attackers tried relatively few usernames (from one up to a few hundred). Only a small number of attaackers attempted a large number of usernames (in thousands). Those who tried the smallest number of usernames (i.e., 1 or 2) tended to go after administrator access with the usernames `root` and `admin`. The sets of usernames attempted in BFAs form the basis for our novel blocking strategy (described in Sections 7 and 8).

Are attackers successful at guessing legitimate usernames? We found a significant overlap between the usernames in guessing vectors used by attackers and those of CloudLabusers, indicating that attackers are somewhat successful at guessing legitimate usernames. Overall, 609 usernames belonging to legitimate CloudLab users appeared in the attacks. Although small, this is a non-trivial fraction (3%) of the ≈20,000 CloudLab users.[2] However, we saw no evidence that the guessing vectors targeted CloudLab specifically (e.g., via a leak of CloudLab's user database) since guessing vectors of substantial size were composed mostly of usernames *not* used by CloudLab's userbase. Instead, the presence of real usernames in the guessing vectors seems to be because of attackers attempting common names that are likely to be present in any sizable user base. Regardless, the overlap with legitimate usernames underscores the need to enforce good security practices (e.g., key-based login and/or strong passwords or passphrases).

What types of devices or software are targeted for attacks? Some Internet-connected devices have specific, known usernames, allowing us to infer that attackers trying those usernames were targeting devices of that type (regardless of whether such devices are present in CloudLab). Similarly, some usernames are commonly associated with specific software or operating system images intended for cloud platforms. Therefore, we split the usernames in *Log1* into three groups: non-administrator usernames, *generic* administrator usernames, and *device- or software-specific* usernames. We then manually classified the top 100 usernames, which collectively represented 83% of the failed logins. Of the top 100 usernames, 17% were non-administrator usernames, 67% were generic administrator usernames, and 16% targeted specific devices or software. Appendix E provides the full detail of these usernames and their classifications.

Non-administrator usernames consisted mostly of generic roles or common personal names containing no information about a specific targeted device or software. Most non-administrator usernames (66% of attacks in this username category) were based on specific user *roles*, such as `support`, `user1`, `profile1`, and `demo`. The rest were either personal names (e.g., `john`, `vivek`, `zhang`) or unattributable to particular attacker intentions (e.g., `default`, `ts`, etc.).

Generic administrator usernames were dominated by the username `root`, which accounted for 86% of all attacks in

[2]Not all CloudLab users logged in during the logged period.

this category of usernames, with `admin` being a distant second with 7%. Other generic administrator usernames included `administrator`, `admin1`, and `sysadmin`. Such generic administrator usernames are used broadly by UNIX-like systems, including servers, desktops, and even IoT devices, thus providing little information about specific targets.

Device- or software-specific usernames can be used for devices and software with default usernames (and sometimes default passwords). The largest set of such usernames (37%) targeted devices or software associated with network vendors Ubiquiti, Mikrotik, and Huawei. The nature of these attacks suggests that attackers attempt to gain access to entire networks by compromising routers and switches. Such an approach is particularly concerning because these vendors run the gamut from home to enterprise and from telecommunications to backbone networks. Another large set of attacks (30% of attacks in the category of device- or software-specific usernames) seemed to be leveraging usernames that match well-known software packages to target hosts with that software installed. While most of these usernames corresponded to typical server software such as `oracle`, `mysql`, `hadoop`, and `nagios`, there was a significant subset targeting gaming-related software, such as `minecraft`, `csgoserver` (Counter-Strike), and `teamspeak3` (used for in-game voice chat). Additionally, 23% of attacks employing device- or software-specific usernames tried usernames associated with common server services (but not necessarily specific implementations of those services), such as `web`, `ftp`, and `git`.

Finally, 10% of attacks with device- or software-specific usernames appeared to target specific Linux distributions with usernames such as `ubuntu`, `debian`, and `centos`. Such usernames are often used by default in the disk images produced by these distributions for cloud use. Interestingly, the second-most-popular distribution-based username was `pi`, the default username for Raspberry Pi OS (formerly known as Raspbian), likely because, until 2022, this account had a well-known default username/password pair if `sshd` was enabled [61].

Do usernames become more popular when vulnerabilities are publicized? We noted that certain usernames in guessing vectors rose in volume with the release of corresponding Common Vulnerabilities and Exposures (CVEs) or breaking news about specific vulnerabilities. For instance, after a public disclosure that particular devices are affected by vulnerabilities, we found that a set of usernames associated with those devices surged among the attacks. Specifically, in mid and late 2019, it became known that vulnerabilities in products from video-centric IoT manufacturer Dahua [20] could be exploited for unauthorized remote access, device restart, or arbitrary code execution [21, 22, 52]. As Figure 6 shows, right after the public disclosure, there was a sharp spike in SSH attempts with two of Dahua's default usernames, `888888` and `666666` [23]. The two usernames follow a similar pattern, suggesting they belong to the same guessing vector. Notably, attacker interest in these usernames dwindled after the initial

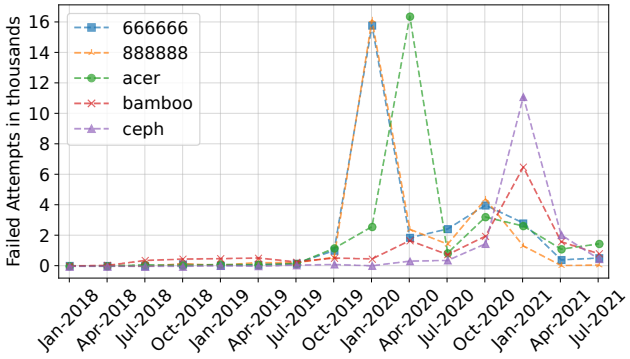


Figure 6: Spikes in usernames associated with CVEs.

spike, possibly because of lower-than- expected exploitability or the ban on these devices imposed by the US government due to security concerns [66]. Figure 6 also shows similar spikes in attacks for three other usernames `acer`, `bamboo` and `ceph`, when corresponding CVEs [51, 54, 55] were announced. Interestingly, none of these CVEs are directly related to SSH.

Sometimes, there was a direct or potential indirect connection between vulnerabilities and SSH access. In other cases, no such connection was obvious. It does appear that CVEs increase attacker use of related usernames, but it is not always clear *why*. Overall, however, attacker behavior shows the need to react relatively quickly when new usernames become more prevalent. An effective blocking strategy must be able to discover and block new usernames quickly.

6 Attacker Distribution and Persistence

SSH BFAs are highly distributed, but the persistence of attackers varies greatly. We examined trends in attacker distribution across countries and networks, along with their persistence across time. It is important to note that attackers often use compromised machines [24], so we cannot draw strong conclusions that the entity *controlling* an attack is based in the same country/network as the device *launching* the attack.

How widely are the attackers and legitimate users distributed? Only 1% of the IP addresses in *Log1* were used by legitimate users, with the other 99% (831K) used exclusively by attackers. These addresses were associated with 223 country codes and 18.5K network providers.

Most attacking IP addresses were from China (23%), followed by the United States (14.2%), Russia (5.8%), and Brazil (5.2%). Conversely, most legitimate IP addresses were from the United States (64%), followed by Indonesia (7.3%), Pakistan (5.2%), China (2.8%), and Brazil (2.8%). A majority of attacking IP addresses captured by honeypots have similarly been reported as originating from the United States and China [47, 64]. IP addresses from 21% of the country codes had at least one legitimate user, while those from the remaining 79% of the country codes were only sources of

attacks. The number of IP addresses from a country was not always correlated with the number of *attacks* from that country. For instance, IP addresses from China were responsible for 44.7% of failed SSH login attempts, followed by those from the United States with 8.4%. IP addresses from China and the United States made up more than half of all failed login attempts. The distribution fell off quickly, with 99% of the failed logins coming from just 5.2% of all countries.

The number of attacks per network provider was similarly skewed. About half of the failed attempts were associated with just six network providers: China Telecom (26.8%), China Unicom (6.1%), DigitalOcean Cloud (6.0%), Tencent Cloud (5.5%), OVH Cloud (2.9%), and the Vietnam Posts and Telecommunications Group (2.7%). IP addresses from the top 100 network providers contributed 79.9% of the failed login attempts, with the top 13% contributing 99% of the failed attempts. Among the network providers seen in our data, 99 were used only by legitimate users, 431 by legitimate users as well as attackers, and the remaining 18,067 only by attackers. Such a distribution indicates that flagging entire networks as malicious is a poor strategy because most legitimate users are on networks that are also the sources of attacks.

Similar to prior research [16, 64], we observed a significant number of attacking IP addresses (at least 15.3%, responsible for 20% of the failed attempts) hosted by cloud providers. Of the top ten network providers based on the number of failed attempts, three were cloud providers. Six among the top 15 networks—and ten among the top 25—were cloud providers.

Further detail on the network providers and countries for IP addresses in *Log1* is included in Appendix F.

Are there instances of concentrated attacks from specific networks? The growth in the attacking IP addresses in CloudLab was generally steady over time. About 1–2% of the attacking IP addresses were new each month. At the high end, CloudLab experienced 4–6% new attackers in some months.

However; there were period during which there were large spikes in new addresses from specific networks or countries. To quantify this aspect, we calculated the month-to-month *growth ratio* for IP addresses from each country and network provider. We defined the growth ratio for a month as the number of new IP addresses seen that month divided by the average number of new IP addresses per month over the four-year logging period. The growth ratio enabled us to examine those network providers whose IP addresses exhibited high growth ratios at some point. Figure 7 shows growth ratios for four network service providers with Comcast (a large US ISP) as the baseline compared with Lumen/CenturyLink [69] (another large US ISP), Selectel [70] (a Russian cloud company that others have identified as an attack source [18]), and LG Uplus [68] (a South Korean mobile network operator). While there were many attackers from Comcast, they appeared at a fairly constant rate, with the growth ratio staying close to 1.0 as seen in Figure 7 In stark contrast, the IP addresses from the other three network providers exhibited large spikes, i.e.,

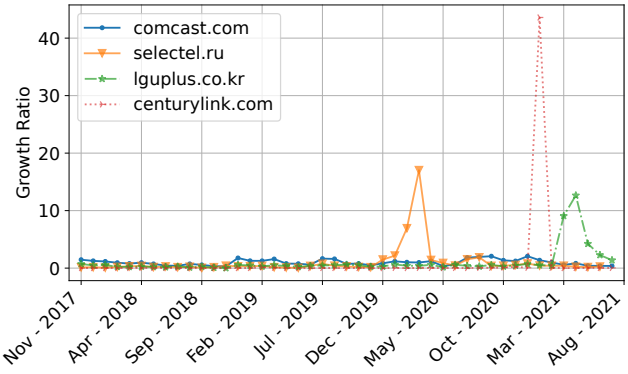


Figure 7: Growth ratios for four networks with Comcast as the baseline.

months during which there were far more new attackers from these networks than typical. We typically observed 82 new attacking IP addresses from Comcast per month, with a maximum of 170 new IP addresses in a month. In comparison, there were an average of 657 new attacking IP addresses per month from Lumen/CenturyLink, but the spike in January 2021 was composed of 28,640 new Lumen/CenturyLink IP addresses involved in attacks. Such concentrated attacks from a single network can have diverse causes, such as attackers forming botnets from vulnerable routers or cable modems specific to an ISP [50, 53], or gaining legal or illicit access to cloud or hosting services [12, 57]. Regardless of the reason, detecting spikes can help uncover and investigate anomalies.

How long do individual IP addresses persist? The IP addresses of most attackers and legitimate users had an active duration of less than one day, vanishing on the same day they first appeared. Figure 8 compares the IP addresses of attackers and legitimate users in terms of the active durations, i.e., the differences between the last and the first days on which the respective IP addresses were seen, in days. Although there were more IP addresses of legitimate users with short druations, the overall shapes of the distribution curves for the two groups are remarkably similar.

What metrics distinguish attackers and legitimate users? The ability to block attacks based on metrics, such as the number of attempts, relies mainly on finding metrics that clearly distinguish attackers from legitimate users. Figures 9 and 10 are scatter plots for three pairs of quantitative metrics, with each point in the plots representing one source IP address. Since there are far more attacking IP addresses than legitimate ones, the plots depicting the latter are sparser.

In all cases, legitimate users were "embedded" within the portions of the space covered by attackers. This can be clearly seen in Figure 9, which is a plot of the length of time between observing an IP address for the first and last times vs. the fraction of days within that span that the IP address was active. While legitimate users are clustered more around the axes of the plot, there are enough outliers such that few regions

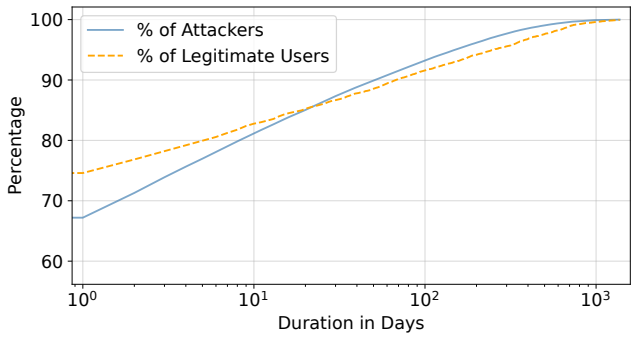


Figure 8: Cumulative Distribution Function (CDF) of the active durations of IP addresses used by attackers and legitimate users. The X axis is on a log scale.

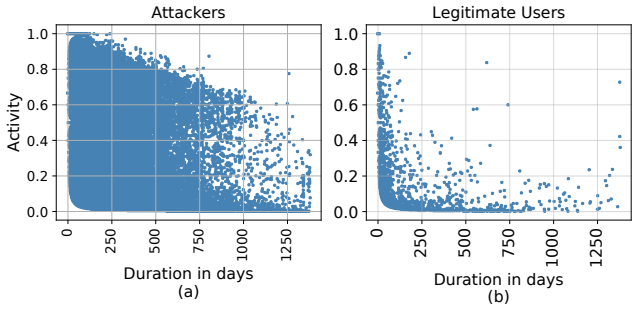


Figure 9: Duration (i.e., the span between the first and last connection) vs. activity ratio (i.e., the fraction of days in the span with connections) for IP addresses used by attackers and legitimate users.

of the plot are occupied exclusively by attackers. Practically speaking, using these metrics to mark IP addresses as attackers can catch a significant number of them, but at the expense of falsely flagging a number of legitimate users as attackers.

Plots (a) and (b) in Figure 10 show the total number of login attempts from each IP address vs. the number of usernames attempted. There is a clear region in each plot (above 100 usernames) containing a substantial number of attackers but only a single legitimate user. Still, these plots highlight the difficulty of trying to identify attackers with either metric because, like most attackers, a substantial number of IP addresses of legitimate users appear to use tens of usernames (likely due to NAT). In terms of the number of attempts, there is complete overlap on the X axis. Plots (c) and (d) in Figure 10 show similar observations for the number of hosts attacked by each source IP address vs. the number of usernames attempted.

7 Tracking Attacks With Dictionaries

As we saw in previous sections, BFAs come from a variety of locations, with attackers exhibiting a wide range of behavior. In addition, we showed that quantitative metrics alone cannot

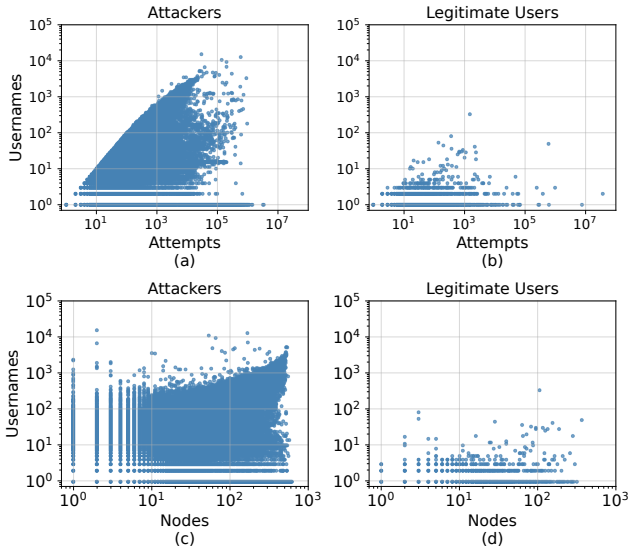


Figure 10: Plots (a) and (b) show the number of attempts vs. the number of usernames. Plots (c) and (d) show the number of hosts vs. the number of usernames for IP addresses used by attackers and legitimate users.

easily differentiate attackers from legitimate users. Therefore, we turned to identifying attacks by using the *sets of usernames* attempted by each source IP address as its *fingerprint*.

Constructing dictionaries and dictionary groups: We found that many individual source IP addresses had the same—or similar—fingerprints. We therefore compiled *dictionaries* of usernames, with each dictionary being a set of usernames used by a set of (more than one) source IP addresses. These dictionaries served two purposes. First, they enabled us to find the sets of usernames prevalent among attackers, forming the basis of DBB. Second, they helped us infer potential relationships between attacking IP addresses.

For an accurate count, we needed to find fingerprints that were not only identical but also *similar*. Such matching was required for two reasons. First, the data from production systems can be incomplete (as discussed in Section 3) so we needed a mechanism that can tolerate some amounts of missing data. Second, we hypothesized that attacker dictionaries would drift over time, with small additions of new usernames and/or removal of unfruitful ones, similar to dynamic credential fetching seen in recent malware [13]. Therefore, we developed a method for grouping similar dictionaries by determining the Jaccard Similarity (JS) [41] between username sets. The JS for two sets `X` and `Y` is defined as the ratio of $length|X \cap Y|$ to $length|X \cup Y|$. JS ranges from 0 to 1, being 0 when there is a null intersection between sets and 1 when the intersection and the union are the same.

Our method captures the transitive similarity between different dictionaries in two steps. First, we constructed an undirected graph `G = (V, E)` where each dictionary is repre-

sented as a vertex in V. An edge is added to E for every pair of vertices (v, v′) if $JS(v,v') \geq \hat{j}$, where JS(...) is the Jaccard similarity of the pair of dictionaries and $\hat{j}$ is a selected threshold. Second, we determined all *connected components* in G. i.e., groups of vertices $\tilde{V} \subset V$ for which every $v \in \tilde{V}$ can be reached from every other $v' \in \tilde{V}$ via the (undirected) edges. Note that individual dictionaries may form their own connected singleton components if they have no edges (i.e., they are not similar to any other dictionaries), and groups of dictionaries can become transitively included in larger components (e.g., if a dictionary A is similar to B, and B is in turn similar to C, the three dictionaries form one connected component). We selected a threshold value for $\hat{j}$ of 0.88 by considering a range of values and selecting the one that maximized the number of non-trivial (i.e., larger than size one) connected components. Intuitively, our choice of the threshold value is reasonable because it is large enough to group together only those dictionaries that are similar yet small enough for modest-sized dictionaries (e.g., with ten usernames) to pass the bar despite differing by one or two usernames.

Attacker Use of Dictionaries: Our construction of individual dictionaries from *Log1* resulted in 829 dictionaries covering 64% of the attacking IP addresses. After applying the similarity determination method above, we ended up with 48 non-trivial connected components, reducing the initial set of 829 dictionaries to a set of 682 Dictionaries *Groups* (DGs). As the numbers indicate, most DGs are singletons. A list of the most frequently encountered usernames in the DGs is included in Appendix D. Across the 682 DGs, the number of usernames per DG varied from one to ≈ 4.6K, with a median of four, and the number of attackers using the same DG varied from two[3] to ≈ 158K, with a median of 27.5. The numbers show that several DGs were large enough to make it highly unlikely that multiple sets of attackers independently stumbled upon them by coincidence. For dictionaries of non-trivial size, the use of the same dictionary intuitively suggests a potential connection between the attacking IPs—those employing the same set of usernames may be controlled by the same entity, using the same tool, following the same strategies, or sourcing the username lists from a common origin.

The DGs with a smaller number of usernames included mainly generic administrator or device/software-specific usernames as described in Section 5, while those with a larger number of usernames tended to contain guesses for usernames of regular users. Figure 11 depicts the number of IP addresses using a DG and the number of usernames in the DG. The pattern in Figure 11 shows that a large number of attackers use small DGs and vice versa. For instance, only two attackers used the largest DG with 4.6K usernames.

Attackers differed in the sequences in which they try usernames. The username sequences were random in some DGs, such as one with 18 usernames used by 3,447 attackers attempted in 3,342 unique sequences. The sequences were mostly the same in some other cases, such as a DG with just 41 unique sequences of 16 usernames used by 31,249 attackers. Such sequencing information can be used to fingerprint specific tools or scripts used by the attackers.

[3]The definition of a dictionary precludes the creation of dictionaries used by a single source.

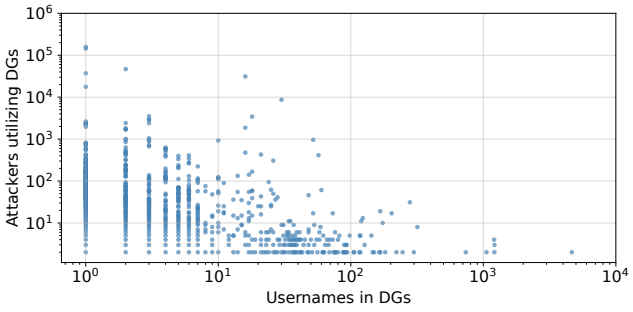


Figure 11: Usernames in a DG vs. attacking IP addresses using that DG. Both axes are on log scales.

Can dictionary-based blocking stop attacks? As we noted earlier, a small set of usernames dominated the BFAs in *Log1*. Most attackers used *at least one* username from a dictionary. We identified 9,819 unique usernames in the dictionaries. Blocking source IPs that used one of these usernames caught 100% of the attackers who used a complete dictionary and 84% of those who did not. We noted that a blocklist of around *10K usernames* (fewer than 4% of all usernames) effectively captured 94% of the attackers (the 64% of using a dictionary, plus 84% of the remaining 36%). Based on this observation, we designed a blocking approach called Dictionary-Based Blocking (DBB). As we describe in Section 8, we found the actual effectiveness of DBB to be even higher than expected: at timescales smaller than four years, only a few hundred usernames are needed at a time, and attackers employing these username lists are more active than average.

8 Dictionary-Based Blocking (DBB)

As we have shown in the previous sections, the usernames attempted by attackers are fundamental to BFAs. We leveraged this observation to devise DBB as an easy-to-apply technique to identify and block attackers and prevent SSH BFAs. We first evaluated DBB on *Log2*, which has no intersection with *Log1*, and later deployed it in production on a distinct cluster. By demonstrating the effectiveness of DBB over various sets of nodes, we highlight its general-purpose applicability for deployment on diverse machines over the Internet.

8.1 Threat Model

DBB is intended to counter attacks such as BFAs that target a large number of hosts and/or username/password sets. The threat model further assumes that an attacker may be aware

of the presence of DBB and the dictionaries shared among the cooperating parties. Although we have described the use of DBB to protect individual hosts, it can be applied network-wide by using log files from individual hosts to generate rules for a border firewall. That said, DBB is not intended to defend against targeted attacks in which only a single facility is attacked for a short period. Further, DBB cannot defend against stolen credentials, which allow attackers to succeed at logging in on the first attempt. Defending against such attacks would require additional protective mechanisms [65, 71, 73].

8.2 Design of DBB

DBB involves a set of *collectors* that observe SSH login attempts. In the simplest cases, the collectors can be production machines that harvest their logs for usernames and source IP addresses associated with failed logins. Alternately, SSH honeypots or machines can be set up specifically as collectors. As described in Section 7, the collectors create a set of dictionaries by formulating a union of all usernames in their dictionaries. A Username Block List (*UBL*) is created by removing any locally valid usernames from the union. Each collector sends its *UBL* to a central *coordinator* that combines them with those received from other collections and distributes the union to any *defending hosts*, i.e., hosts that wish to defend themselves against SSH BFAs. DBB involves employing *UBL* to identify attacker IP addresses and blocking them at the defending host and/or at a network-wide firewall. When a defending host receives a failed login attempt for any username in the *UBL* that is not a local user[4], it blocks all further traffic from that source (e.g., by adding a local firewall rule [7]). We envision a deployment of DBB in which volunteers worldwide contribute *UBL*s from collectors on their networks to a trusted central coordinator that publishes a global *UBL* that any host can use for defense.

It is crucial to note a couple of important properties of DBB. First, no IP addresses are exchanged. As a result, unlike IP-based blocklists, DBB preserves the privacy of users and facilities and prevents "false accusations" against specific IP addresses. Second, DBB removes legitimate usernames when creating and applying *UBL*s since legitimate usernames can end up in dictionaries (as we showed in our analysis). Removal of legitimate usernames avoids information leakage from the collectors and prevents defending hosts from locking out unlucky users whose usernames end up in the *UBL*.

8.3 Effectiveness of DBB

We evaluated DBB using *Log2*, which was collected for nearly ten weeks—from November 2022 to January 2023—on three CloudLab sites: *Site-A*, *Site-B*, and *Site-C*. While *Site-B* and *Site-C* have characteristics similar to *Site-A*, they are geographically distributed and use distinct IP addresses on different networks. We simulated DBB on the data from these sites to derive and distribute new *UBL*s once a day using their log files as traces. In addition, we deployed DBB in production for three weeks at the *Emulab* [2] cluster in CloudLab.

Evaluating DBB: We simulated DBB at each site independently to determine its effectiveness by measuring the fraction of attack attempts blocked and the number of false positives (i.e., the number of blocked IP addresses of legitimate users).[5] We have reported false positives using absolute numbers rather than percentages because each false positive corresponds roughly to a single blocked user and/or one support ticket for the staff to resolve. For the same reason, we have reported false positives with respect to the number of blocked source IP addresses belonging to legitimate users.

We identified legitimate user IP addresses in *Log2* with the same method used for *Log1*, finding that *Site-A*, *Site-B* and *Site-C* had 2,952, 1,733, and 1,504 IP addresses of legitimate users, respectively. At *Site-A*, *Site-B* and *Site-C* DBB blocked 99.58%, 99.59%, and 99.39% of the BFAs with 17, 18, and 5 false positives, respectively. As we showed earlier, most attacks use a dictionary. Therefore, DBB achieved uniformly high block rates, with only 0.5% of BFAs going unblocked. The false positive rates for all sites were remarkably low, with an average of one false positive every five days.

Comparing DBB to Fail2ban: We ran simulations using *Log2* to compare DBB with Fail2ban [3], a widely deployed state-of-the-art tool for blocking SSH BFAs at the host. Fail2ban has three adjustable parameters: *Maxretry*—the number of failed attempts from the same IP address that activate blocking; *Findtime*—the period during which failed attempts are counted; and *Bantime*—the duration of the block. In comparison, DBB employs a single parameter equivalent to *maxretry*. Since we recommend blocking with DBB at the first failed login attempt, i.e. *maxretry* = 1, this can serve as the default configuration.

Trying every one of the numerous possible combinations of Fail2ban parameters is practically infeasible. Therefore, we tuned the parameters to conduct a fair comparison between Fail2ban and DBB without favoring either technique.We compared the performance of Fail2ban with DBB with three different settings variations: S_1**:** Default settings for DBB (*maxretry* = 1) and Fail2ban (*maxretry* = 5, *findtime* = 10 minutes, *bantime* = 10 minutes); S_2**:** Default settings for DBB (*maxretry* = 1) with Fail2ban adjusted to use the same *maxretry* value (*maxretry* = 1, *findtime* = 10 minutes *bantime* = 10 minutes); and S_3**:** No unblocking in Fail2ban (*bantime* = ∞) and varying values of *maxretry* from $[1, 45]$ for both techniques. All variations used the default Fail2ban *findtime* of 10 minutes.

[4] We recommend `root` be deemed an invalid username even if a facility permits its use.

[5] Even though defending hosts remove local usernames from the *UBL*, false positives can occur if a legitimate user tries to log in using a wrong username that happens to be in the *UBL* or shares an IP address with an attacker (e.g.. due to NAT).

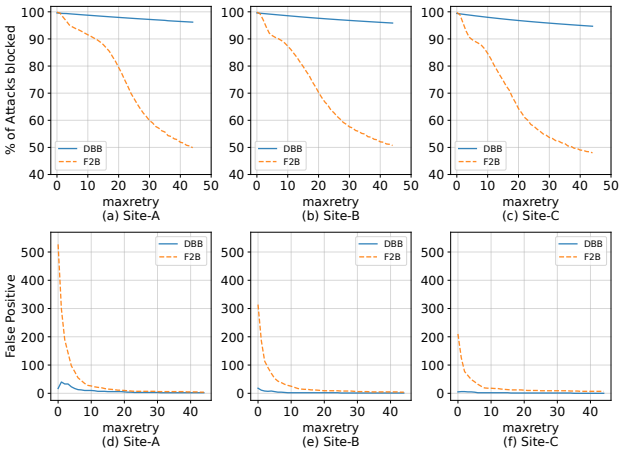


Figure 12: S_3: DBB and Fail2ban for *maxretry* $\in [1, 45]$.

With default settings, DBB outperformed Fail2ban by a large margin. Across all sites, DBB blocked an average of 99.5% of the attacks, while Fail2ban blocked only 66.1%. One reason for this difference was the more aggressive default setting of DBB, with a lower value for *maxretry* compared to Fail2ban. Moreover, DBB generated only one-fifth as many false positives as Fail2ban (DBB_{FP}=40, $F2B_{FP}$=217). In other words, DBB locked far fewer legitimate users out of their accounts. When *maxretry* for Fail2ban was set to the DBB default of 1, the attacks blocked by Fail2ban increased from 66.1% to 85.20%. Appendix B illustrates site performance for expirement S_1 and S_2 . The tradeoff is in the number of false positives: the increase in blocked attacks was accompanied by a corresponding increase in false positives for Fail2ban from 217 to 1051, which was 26 times the rate for DBB.

To check if there is *any* setting for Fail2ban in which it meets or exceeds the performance of DBB, we varied *maxretry* for both from 1 to 45. In addition, we configured Fail2ban to mimic DBB by permanently blocking IP addresses. As Figure 12 shows, DBB achieved a better blocking rate than Fail2ban for all values of *maxretry* except 1, i.e., when Fail2ban was set to block an IP address on the first failed attempt and never unblock it. Moreover, DBB generated fewer false positives across all values of *maxretry* for all three sites. With increasing *maxretry*, the false positive rate of Fail2ban did progressively reduce to converge with that of DBB. However, the improvement came at the expense of reduced blocking effectiveness.

Overall, DBB's dictionary-based approach outperformed Fail2ban's rate-based approach. DBB can thwart nearly all incoming BFAs by blocking more aggressively without incurring many false positives because the approach is based on specific attacker behavior.

8.4 Number of Collectors

To examine the effect of the number of collectors on blocking performance, we computed dictionaries from C_n number of collectors $C_n \in \{[1,10] \cup \{20,30,40,50\}\}$ at each site in *Log2*. We then performed simulations using these reduced dictionaries as the *UBL*s of the hosts at the respective sites.

For each C_n, we ran the simulation nine times with randomly-selected collectors from all available nodes. Across these runs, the minimum and maximum blocked BFAs for *Site-A*, *Site-B*, *Site-C* were (98.2%, 99.5%), (97.7%, 99.5%) and (97.6%, 99.3%), respectively. All minima occurred at C_n = 1 and maxima at C_n = 50. DBB achieved most of its benefit with just one collector, with results within 1-2% of the performance when using *all* nodes as collectors. While there were slight increases in blocking with an increase in the number of collectors (full graphs are in Appendix C), the overall results show that DBB does not need a large number of collectors to achieve high performance. Over ten weeks, the *UBL* grew by only a few usernames per day, starting at ≈200 usernames per site and ending with 460 at the conclusion of the experiment.

Next, we checked whether *UBL*s created at one site are effective at other sites by testing all combinations in which each of the three sites was either a collector or deployment site. In all cases, DBB blocked at least 99.41% of the BFAs with at most 19 false positives (full details are provided in Appendix A). These results demonstrate that defending sites can implement DBB by simply using *UBL*s obtained from a trusted third party.

8.5 Deployment of DBB

We deployed DBB for three weeks on the *Emulab* [2] cluster of CloudLab (which consists of ≈400 nodes) with the three sites *Site-A*, *Site-B*, and *Site-C* as collectors. A single defending host removed locally valid usernames from the *UBL* and collected SSH logs from the other hosts in the cluster. Upon receiving a login attempt with any username in the local sanitized *UBL*, the defending host added the corresponding IP address to the blocklist. The blocklist was copied to the *Emulab*-wide firewall to block all further traffic from that IP address. For practicality, we updated the firewall blocklist on an hourly basis. The consequent gap between identifying attackers and subsequently blocking their access at the firewall can result in allowing attackers to keep trying SSH BFAs without restrictions for up to an hour in the worst case.

Before deploying DBB, we recorded the SSH traffic on *Emulab* for three weeks. Defense mechanisms already in place during this period consisted of two strategies: *lazy-fail2ban* and a firewall subscribing to a variety of public IP-address blocklists recommended by pfSense [8]. *lazy-fail2ban* adds an IP address to the blocklist at the firewall if the number of failed SSH login attempts from it crosses a threshold (i.e., *maxretry*=10). Unlike Fail2ban in standard configuration, *lazy-*

fail2ban operates without a finite *findtime*, instead using *findtime*=∞ and *bantime*=24 hours. When an IP address successfully logs in, the *maxretry* counter is reset to zero. *Emulab*'s firewall blocks *all* access from any IP address contained in the IP-address blocklists; this proactive measure is taken not only to counter SSH BFA but also to defend against attacks attempting to exploit other protocols and vulnerabilities.

Operational Effectiveness: It was possible to compute the precise percentage of BFAs blocked when assessing DBB performance in the simulation because we knew the total number of failed SSH attempts. However, blocking IP addresses in an operational system prevents subsequent failed attempts from that source, making it challenging to count the precise number of attack attempts that *would have been* logged without DBB. We therefore relied on the three-week record of failed attempts from *Emulab* to establish a baseline for estimating the proportion of attacks blocked. Importantly, these constraints mean that measurements of DBB performance blocking BFAs in operation provide lower bounds.

We found that DBB is significantly more effective than the existing blocking mechanism at *Emulab* during DBB deployment. DBB reduced failed SSH login attempts by 79.5%—from 80.6K to 16.5K per day—suggesting that it blocked four-fifths of the attacks not caught by the other defenses at *Emulab*. During the DBB deployment period, no legitimate user contacted the administrators about being blocked by DBB while two users contacted them because of blocking by *lazy-fail2ban*. One of the two blocked by *lazy-fail2ban* had mistakenly attempted to log in with a username resembling a university identification number, surpassing the *lazy-fail2ban* threshold and getting blocked. However, DBB did not block the user because the identification number used as the username was not in the *UBL*. The incident illustrates that the design of DBB has the advantage of minimizing the chances of blocking legitimate users because of inadvertent errors such as usernames with typos or those from other services.

8.6 Practical Considerations

DBB is a lightweight mechanism that involves negligible overhead for collectors or coordinators. The only requirement for DBB is that defending hosts have the ability to check username membership in a relatively small set. Moreover, the compact size of the *UBL* makes it well-suited for deployment on resource-constrained devices such as IoT devices. As a result, DBB can be easily deployed at larger scales by appropriately considering several practical factors.

Dictionaries Distribution: We propose hashing usernames before sharing with the coordinator. The approach permits easily testing membership in the *UBL* but avoids leaking valid usernames, thus making it difficult to use public *UBL*s to target high-value usernames. Moreover, hashed usernames prevent broader dissemination of newly discovered vulnerabilities pertaining to a username known only to a few attackers.

Dictionary Collection: Since dictionaries do not include IP addresses and locally valid usernames are removed before sharing, collectors can be set up anywhere on the Internet without raising privacy concerns. A collector need not reveal its own IP address and can be easily moved elsewhere to prevent adversaries from discovering it. However, DBB functions well even if the adversaries know the identities of some, or even *most*, collectors as long as there are a *few* collectors that are not known to the attackers. Since DBB does not place a high degree of trust in dictionary providers, attempting to deny service by inducing false inclusion of a username in a dictionary is ineffective. Therefore, a large network of volunteer collectors can operate with fairly light oversight.

Block Evasion: To evade DBB, an attacker must avoid high-value usernames in the public *UBL*, likely reducing the effectiveness of the attack. Attacking with a username not in the *UBL* requires that the username not be present in *any* dictionary. In other words, evading DBB requires avoiding *every* collector or avoiding using the same set of usernames from any source. The safest strategy for attempting to evade blocks is to mix high-value new usernames with unique "chaff" usernames in every attempt. Such an approach would slow the attacker down. Moreover, it would not help to have multiple attacking sources. In fact, it would require greater coordination or increased chaff to avoid reusing username sets.

9 Conclusion

While new cyberthreats emerge daily, attackers across the world continue to rely on simple, traditional approaches such as SSH BFAs. Yet, existing approaches face challenges in blocking such attacks with high accuracy without also blocking sizable numbers of legitimate users. The observed trends indicate an evolution of attacks on various accounts, software and devices, with attackers shifting from traditional *generic* attacks. Analyzing multi-year login attempt logs enables the effective identification of malicious activity by discerning differences in attacker and legitimate user behavior. We have shown that such insight can be applied for designing a lightweight blocking mechanism that can be deployed at scale with little overhead. Our approach outperforms the state-of-the-art in host-based SSH blocking, pointing the way to a new class of more effective defenses against BFAs.

Acknowledgements

We extend our sincere gratitude to the anonymous reviewers, our shepherd Bruce Davie, and the CloudLab administrators for their invaluable contributions and dedicated efforts in reviewing and supporting our research. Their insightful feedback have enriched the quality of our work. This material is based upon work supported by the National Science Foundation under Award Nos. 1743363, 1801446, and 2027208.

References

[1] Denyhosts. https://denyhosts.sourceforge.net/. Accessed 2024-03-03.

[2] Emulab. https://www.emulab.net/. Accessed 2024-03-03.

[3] Fail2ban. https://github.com/fail2ban/fail2ban. Accessed 2024-03-03.

[4] Hydra. https://www.kali.org/tools/hydra/. Accessed 2024-03-03.

[5] IPinfo. https://ipinfo.io/. Accessed 2024-03-03.

[6] Ncrack. https://nmap.org/ncrack/. Accessed 2024-03-03.

[7] The Netfilter.org "iptables" project. https://www.netfilter.org/projects/iptables/index.html.

[8] pfsense. https://www.pfsense.org/. Accessed 2024-03-03.

[9] SSH Guard. https://www.sshguard.net/. Accessed 2024-03-03.

[10] A. Abdou, D. Barrera, and P. C. van Oorschot. What lies beneath? Analyzing automated SSH bruteforce attacks. In F. Stajano, S. F. Mjølsnes, G. Jenkinson, and P. Thorsheim, editors, *Technology and Practice of Passwords*, pages 72–91, Cham, 2016. Springer International Publishing. DOI: 10.1007/978-3-319-29938_6.

[11] Akamai Security Research. Panchan's mining rig: New Golang peer-to-peer botnet says "Hi!". https://www.akamai.com/blog/security/new-p2p-botnet-panchan, Jun 2022. Accessed 2024-03-03.

[12] S. Alrwais, X. Liao, X. Mi, P. Wang, X. Wang, F. Qian, R. Beyah, and D. McCoy. Under the shadow of sunshine: Understanding and detecting bulletproof hosting on legitimate service provider networks. In *2017 IEEE Symposium on Security and Privacy*, IEEE S&P 2017, pages 805–823, 2017. DOI: 10.1109/SP.2017.32.

[13] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. Understanding the Mirai botnet. In *26th USENIX Security Symposium*, USENIX Security 17, pages 1093–1110, Vancouver, BC, Aug 2017. USENIX Association. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis.

[14] ARIN. American Registry for Internet Numbers. https://www.arin.net/. Accessed 2024-03-03.

[15] T. Barron and N. Nikiforakis. Picky attackers: Quantifying the role of system properties on intruder behavior. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC '17, pages 387–398, New York, NY, USA, 2017. Association for Computing Machinery. DOI: 10.1145/3134600.3134614.

[16] M. S. Bohuk, M. Islam, S. Ahmad, M. Swift, T. Ristenpart, and R. Chatterjee. Gossamer: Securely measuring password-based logins. In *31st USENIX Security Symposium*, USENIX Security 22, pages 1867–1884, Boston, MA, Aug. 2022. USENIX Association. https://www.usenix.org/conference/usenixsecurity22/presentation/sanusi-bohuk.

[17] P. M. Cao, Y. Wu, S. S. Banerjee, J. Azoff, A. Withers, Z. T. Kalbarczyk, and R. K. Iyer. CAUDIT: Continuous auditing of SSH servers to mitigate Brute-Force attacks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 667–682, Boston, MA, Feb. 2019. USENIX Association. https://www.usenix.org/conference/nsdi19/presentation/cao.

[18] Check Point Research. Stopping serial killer: Catching the next strike. https://research.checkpoint.com/2021/stopping-serial-killer-catching-the-next-strike/, Jan 2021. Accessed 2024-03-03.

[19] P. J. Criscuolo. Distributed denial of service: Trin00, tribe flood network, tribe flood network 2000, and stacheldraht. *Department of Energy Computer Incident Advisory Capability (CIAC)*, UCRLID-136939, Rev. 1, Feb 2000.

[20] Dahua Technology. Dahua Intelligent Solutions. https://www.dahuasecurity.com/. Accessed 2024-03-03.

[21] Dahua Technology. Security advisory — Buffer overflow vulnerability found in some Dahua IP camera devices. https://www.dahuasecurity.com/support/cybersecurity/details/617, 2019. Accessed 2022-03-01.

[22] Dahua Technology. Security advisory — Some products of Dahua have security risks. https://www.dahuasecurity.com/support/cybersecurity/details/637, 2019. Accessed 2022-03-01.

[23] Dahua Wiki. Dahua default usernames and passwords. https://dahuawiki.com/UsernameandPassword. Accessed 2022-03-01.

[24] F. Dang, Z. Li, Y. Liu, E. Zhai, Q. A. Chen, T. Xu, Y. Chen, and J. Yang. Understanding fileless attacks on Linux-based IoT devices with HoneyCloud. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, pages 482–493, New York, NY, USA, 2019. Association for Computing Machinery. DOI: 10.1145/3307334.3326083.

[25] Dr. Web. Linux.muldrop.14. https://vms.drweb.com/virus/?i=15391790, 2017. Accessed 2024-03-03.

[26] M. Dražar. Protocol-independent detection of dictionary attacks. In T. Bauschert, editor, *Advances in Communication Networking*, pages 304–309, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-40552-5_30.

[27] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, July 2019.

[28] K. Egevang and P. Francis. The IP Network Address Translator (NAT). *RFC 1631*, 1994. https://www.rfc-editor.org/rfc/rfc1631 Accessed 2024-03-03.

[29] D. Garn. Eight ways to protect SSH access on your system. https://www.redhat.com/sysadmin/eight-ways-secure-ssh, Oct 2020. Accessed 2024-03-03.

[30] V. Ghiette, H. Griffioen, and C. Doerr. Fingerprinting tooling used for SSH compromisation attempts. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses*, RAID 2019, pages 61–71, Chaoyang District, Beijing, Sept. 2019. USENIX Association. https://www.usenix.org/conference/raid2019/presentation/ghiette.

[31] J. Hancock, T. M. Khoshgoftaar, and J. L. Leevy. Detecting SSH and FTP brute force attacks in big data. In *20th IEEE International Conference on Machine Learning and Applications*, ICMLA 2021, pages 760–765, 2021. DOI: 10.1109/ICMLA52953.2021.00126.

[32] L. Hellemons, L. Hendriks, R. Hofstede, A. Sperotto, R. Sadre, and A. Pras. SSHCure: A flow-based SSH intrusion detection system. In R. Sadre, J. Novotný, P. Čeleda, M. Waldburger, and B. Stiller, editors, *Dependable Networks and Services*, pages 86–97, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-30633-4_11.

[33] R. Hofstede, L. Hendriks, A. Sperotto, and A. Pras. SSH compromise detection using NetFlow/IPFIX. *SIGCOMM Comput. Commun. Rev.*, 44(5):20–26, Oct 2014. DOI: 10.1145/2677046.2677050.

[34] R. Hofstede, M. Jonker, A. Sperotto, and A. Pras. Flow-based Web application brute-force attack and compromise detection. *Journal of Network and Systems Management*, 25(4):735–758, 2017. DOI: 10.1007/s10922-017-9421-4.

[35] R. Hofstede, A. Pras, A. Sperotto, and G. D. Rodosek. Flow-based compromise detection: Lessons learned. *IEEE Security & Privacy*, 16(1):82–89, 2018. DOI: 10.1109/MSP.2018.1331021.

[36] S. Hogg. IPv6 security vulnerability scanning. https://blogs.infoblox.com/ipv6-coe/ipv6-security-vulnerability-scanning/, Sep 2016. Accessed 2024-03-03.

[37] M. D. Hossain, H. Ochiai, F. Doudou, and Y. Kadobayashi. SSH and FTP brute-force attacks detection in computer networks: LSTM and machine learning approaches. In *5th International Conference on Computer and Communication Systems*, ICCCS 2020, pages 491–497, 2020. DOI: 10.1109/ICCCS49078.2020.9118459.

[38] K. Hynek, T. Beneš, T. Čejka, and H. Kubátová. Refined detection of SSH brute-force attackers using machine learning. In M. Hölbl, K. Rannenberg, and T. Welzer, editors, *ICT Systems Security and Privacy Protection*, pages 49–63, Cham, 2020. Springer International Publishing. DOI: 10.1007/978-3-030-58201-2_4.

[39] International Standards Organization. ISO 3166-1:2020: Codes for the representation of names of countries and their subdivisions — Part 1: Country codes. https://www.iso.org/iso-3166-country-codes.html, 2020.

[40] L. Izhikevich, M. Tran, M. Kallitsis, A. Fass, and Z. Durumeric. Cloud watching: Understanding attacks against cloud-hosted services. In *Proceedings of the 2023 ACM on Internet Measurement Conference*, IMC '23, pages 313–327, New York, NY, USA, 2023. Association for Computing Machinery. DOI: 10.1145/3618257.3624818.

[41] P. Jaccard. The distribution of the flora in the Alpine zone. *New Phytologist*, 11(2):37–50, 1912. DOI: 10.1111/j.1469-8137.1912.tb05611.x.

[42] M. Jonker, R. Hofstede, A. Sperotto, and A. Pras. Unveiling flat traffic on the Internet: An SSH attack case study. In *2015 IFIP/IEEE International Symposium on Integrated Network Management*, IM 2015, pages 270–278, 2015. DOI: 10.1109/INM.2015.7140301.

[43] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas. DDos in the IoT: Mirai and other botnets. *Computer*, 50(07):80–84, Jul 2017. DOI: 10.1109/MC.2017.201.

[44] J. Lane Thames, R. Abler, and D. Keeling. A distributed active response architecture for preventing SSH dictionary attacks. In *IEEE SoutheastCon 2008*, pages 84–89, 2008. DOI: 10.1109/SECON.2008.4494264.

[45] T.-H. Lee, L.-H. Chang, and C.-W. Syu. Deep learning enabled intrusion detection and prevention system over SDN networks. In *2020 IEEE International Conference on Communications Workshops*, ICC 2020 Workshops, pages 1–6, 2020. DOI: 10.1109/ICCWorkshops49005.2020.9145085.

[46] P. Muncaster. Massive Qbot botnet strikes 500,000 machines through WordPress. *Infosecurity Magazine*, Oct 2014. https://www.infosecurity-magazine.com/news/massive-qbot-strikes-500000-pcs/.

[47] C. Munteanu, S. J. Saidi, O. Gasser, G. Smaragdakis, and A. Feldmann. Fifteen months in the life of a honeyfarm. In *Proceedings of the 2023 ACM on Internet Measurement Conference*, IMC '23, pages 282–296, New York, NY, USA, 2023. Association for Computing Machinery. DOI: 10.1145/3618257.3624826.

[48] M. M. Najafabadi, T. M. Khoshgoftaar, C. Calvert, and C. Kemp. Detection of SSH brute force attacks using aggregated netflow data. In *IEEE 14th International Conference on Machine Learning and Applications*, ICMLA 2015, pages 283–288, 2015. DOI: 10.1109/ICMLA.2015.20.

[49] M. M. Najafabadi, T. M. Khoshgoftaar, C. Kemp, N. Seliya, and R. Zuech. Machine learning for detecting brute force attacks at the network level. In *2014 IEEE International Conference on Bioinformatics and Bioengineering*, pages 379–385, 2014. DOI: 10.1109/BIBE.2014.73.

[50] National Vulnerability Database. CVE-2014-8423 detail. Nov 2014. https://nvd.nist.gov/vuln/detail/CVE-2014-8423.

[51] National Vulnerability Database. CVE-2019-18670 detail. Dec 2019. https://nvd.nist.gov/vuln/detail/CVE-2019-18670.

[52] National Vulnerability Database. CVE-2019-9676 detail. June 2019. https://nvd.nist.gov/vuln/detail/CVE-2019-9676.

[53] National Vulnerability Database. CVE-2019-19494 detail. Jan 2020. https://nvd.nist.gov/vuln/detail/CVE-2019-19494.

[54] National Vulnerability Database. CVE-2020-8566 detail. Dec 2020. https://nvd.nist.gov/vuln/detail/CVE-2020-8566.

[55] National Vulnerability Database. CVE-2021-26067 detail. Jan 2021. https://nvd.nist.gov/vuln/detail/CVE-2021-26067.

[56] S. News. Poorly secured SSH servers targeted by Chalubo botnet. https://news.sophos.com/en-us/2018/10/24/poorly-secured-ssh-servers-targeted-by-chalubo-botnet/, Oct 2018. Accessed 2024-03-03.

[57] A. Noroozian, J. Koenders, E. van Veldhuizen, C. H. Ganan, S. Alrwais, D. McCoy, and M. van Eeten. Platforms in everything: Analyzing ground-truth data on the anatomy and economics of bullet-proof hosting. In *28th USENIX Security Symposium*, USENIX Security 19, pages 1341–1356, Santa Clara, CA, Aug. 2019. USENIX Association. https://www.usenix.org/conference/usenixsecurity19/presentation/noroozian.

[58] J. Owens and J. Matthews. A study of passwords and methods used in brute-force SSH attacks. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats*, LEET, 2008.

[59] S. Pastrana and G. Suarez-Tangil. A first look at the crypto-mining malware ecosystem: A decade of unrestricted wealth. In *Proceedings of the Internet Measurement Conference*, IMC '19, pages 73–86, New York, NY, USA, 2019. Association for Computing Machinery. DOI: 10.1145/3355369.3355576.

[60] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams. Security smells in Ansible and Chef scripts: A replication study. *ACM Trans. Softw. Eng. Methodol.*, 30(1), Jan 2021. DOI: 10.1145/3408897.

[61] Raspberry Pi Documentation. Configure a user manually. https://www.raspberrypi.com/documentation/computers/configuration.html#configuring-a-user. Accessed 2024-03-03.

[62] G. K. Sadasivam, C. Hota, and B. Anand. Classification of SSH attacks using machine learning algorithms. In *6th International Conference on IT Convergence and Security*, ICITCS 2016, pages 1–6, 2016. DOI: 10.1109/ICITCS.2016.7740316.

[63] S. Schechter, Y. Tian, and C. Herley. StopGuessing: Using guessed passwords to thwart online guessing. In *2019 IEEE European Symposium on Security and Privacy*, EuroS&P 2019, pages 576–589, 2019. DOI: 10.1109/EuroSP.2019.00048.

[64] Z. Shamsi, D. Zhang, D. Kyoung, and A. Liu. Measuring and clustering network attackers using medium-interaction honeypots. In *2022 IEEE European Symposium on Security and Privacy Workshops*, EuroS&PW 2022, pages 294–306, 2022. DOI: 10.1109/EuroSPW55150.2022.00036.

[65] A. Sharma, A. Z. Kalbarczyk, Sharma, R. Iyer, and J. Barlow. Analysis of credential stealing attacks in an open networked environment. In *2010 Fourth International Conference on Network and System Security*, pages 144–151, 2010. DOI: 10.1109/NSS.2010.56.

[66] B. Toulas. Unpatched Dahua cams vulnerable to unauthenticated remote access. https://www.bleepingcomputer.com/news/security/unpatched-dahua-cams-vulnerable-to-unauthenticated-remote-access/, Oct 2021. Accessed 2024-03-03.

[67] J. Vykopal, T. Plesnik, and P. Minarik. Network-based dictionary attack detection. In *2009 International Conference on Future Networks*, pages 23–27, 2009. DOI: 10.1109/ICFN.2009.36.

[68] Wikipedia. LG Uplus. https://en.wikipedia.org/wiki/LG_Uplus. Accessed 2024-03-03.

[69] Wikipedia. Lumen Technologies. https://en.wikipedia.org/wiki/Lumen_Technologies. Accessed 2024-03-03.

[70] Wikipedia. Selectel. https://en.wikipedia.org/wiki/Selectel. Accessed 2024-03-03.

[71] Y. Wu, P. M. Cao, A. Withers, Z. T. Kalbarczyk, and R. K. Iyer. Mining threat intelligence from billion-scale SSH brute-force attacks. In *Workshop on Decentralized IoT Systems and Security (DISS)*, 2020.

[72] T. Ylönen. SSH—Secure login connections over the Internet. In *Proceedings of the Sixth USENIX Security Symposium*, pages 37–42.

[73] W. Yurcik and C. Liu. A first step toward detecting ssh identity theft in hpc cluster environments: discriminating masqueraders based on command behavior. In *IEEE International Symposium on Cluster Computing and the Grid, 2005*, volume 1 of *CCGrid 2005*, pages 111–120, 2005. DOI: 10.1109/CCGRID.2005.1558542.

A Cross collector performance

Table A.1 shows the performance of DBB when the collector is not at the deployed site. We conducted simulations for all possible combinations of our sites.

Table A.1: Performance of DBB when collector and deployed site are different.

Deployed Site	Collector Site	% Attacks Blocked	False Positive
C	*A*	99.45	4
C	*B*	99.41	4
A	*C*	99.55	17
A	*B*	99.56	19
B	*C*	99.58	17
B	*A*	99.61	18

B Individual site performance in experiments S_1 and S_2

Figure B.1 presents the performance of DBB and F2B in experiments S_1 and S_2.

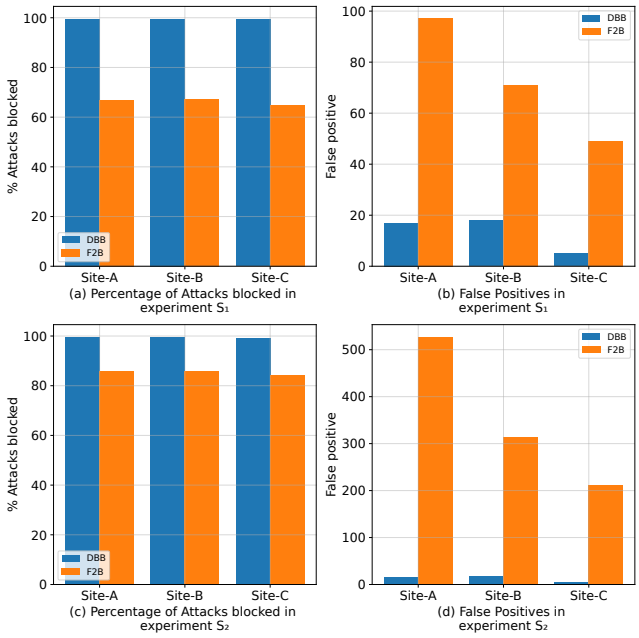


Figure B.1: DBB and F2B using settings S_1 and S_2.

C C_n : Number of Collectors

Figure C.1 depicts how the performance of DBB varies based on the number of collectors (C_n).

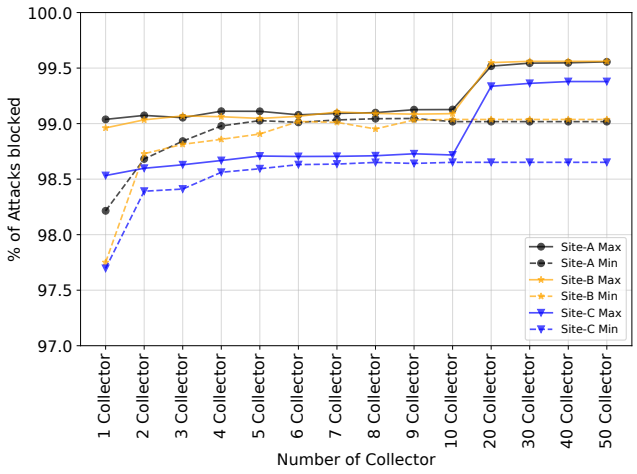


Figure C.1: Performance of DBB as the number of collectors increases.

D Top usernames in DG

Table D.1 shows the usernames present in at least 5% of the DGs.

Table D.1: Usernames in $\geq$ 5% of all DGs.

Username	Percentage of DGs	Username	Percentage of DGs
root	69.94	debian	8.65
admin	55.87	centos	8.65
user	36.07	demo	8.65
test	30.21	minecraft	8.36
support	26.83	zabbix	7.92
ubnt	24.93	odoo	7.92
oracle	23.02	server	7.92
ubuntu	22.87	ts3	7.92
postgres	20.38	apache	7.77
ftp	20.09	teamspeak	7.77
pi	19.21	dev	7.62
guest	18.48	vagrant	7.33
git	16.42	web	7.33
service	15.54	mother	7.33
usuario	14.96	test1	7.04
mysql	14.22	administrator	6.89
nagios	13.49	system	6.74
hadoop	12.76	weblogic	6.30
tomcat	11.88	steam	6.16
jenkins	11.29	svn	5.43
user1	10.12	ansible	5.28
www	9.82	test2	5.28
student	9.38	kafka	5.13
supervisor	9.24	alex	5.13
deploy	8.80	webmaster	5.13

E Username Classification

Table E.1 contains our manual classification of the 100 most-attempted usernames. (Usernames colliding with real CloudLab usernames have been omitted for privacy.)

Username	Percentage of Attempts	Cumulative Percentage	Category	Sub-category
root	47.81	47.81	admin	–
admin	3.97	51.78	admin	–
support	2.66	54.44	non-admin	role
ubnt	1.82	56.26	specific	network
user1	1.79	58.06	non-admin	role
default	1.78	59.83	non-admin	misc
MikroTik	1.72	61.56	specific	network
administrator	1.71	63.26	admin	–
admin1	1.70	64.97	admin	–
profile1	1.70	66.67	non-admin	role
demo	1.70	68.37	non-admin	role

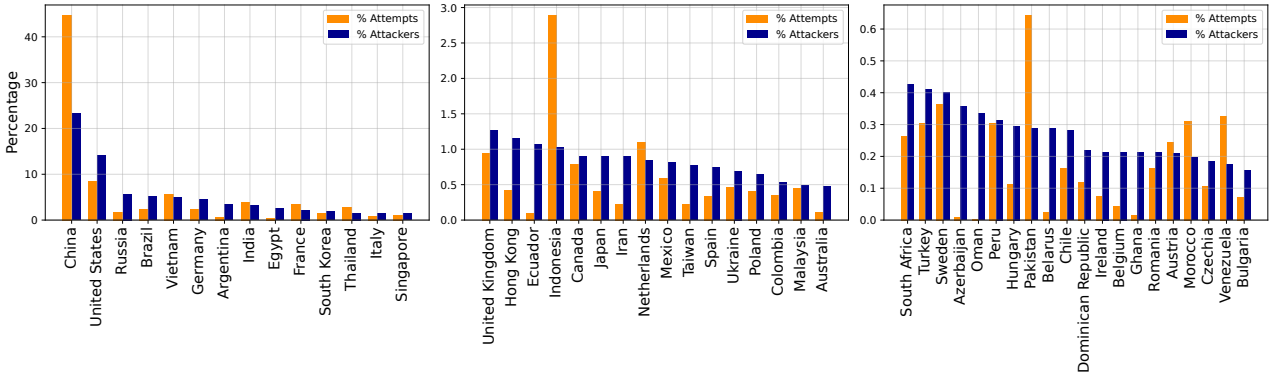
Username	Percentage of Attempts	Cumulative Percentage	Category	Sub-category
web	1.70	70.07	specific	service
tech	1.59	71.66	non-admin	role
telecomadmin	1.54	73.20	specific	network
oracle	0.78	73.98	specific	software
ubuntu	0.64	74.62	specific	distribution
ftp	0.58	75.20	specific	service
postgres	0.50	75.70	specific	software
pi	0.40	76.10	specific	distribution
git	0.35	76.45	specific	service
guest	0.31	76.75	non-admin	role
test1	0.29	77.05	non-admin	role
export	0.28	77.33	non-admin	role
usuario	0.28	77.61	non-admin	role
test2	0.26	77.86	non-admin	role
mysql	0.21	78.07	specific	software
hadoop	0.20	78.28	specific	software
deploy	0.18	78.45	non-admin	role
nagios	0.17	78.62	specific	software
jenkins	0.16	78.78	specific	software
dev	0.15	78.93	non-admin	role
www	0.14	79.07	specific	service
debian	0.13	79.20	specific	distribution
minecraft	0.12	79.32	specific	software
odoo	0.11	79.44	specific	software
ansible	0.11	79.55	specific	software
teamspeak	0.11	79.66	specific	software
student	0.11	79.76	non-admin	role
tomcat	0.10	79.87	specific	software
ts3	0.10	79.97	specific	software
server	0.10	80.07	non-admin	role
centos	0.09	80.16	specific	distribution
es	0.09	80.24	specific	software
zabbix	0.08	80.32	specific	software
weblogic	0.08	80.40	specific	software
steam	0.07	80.47	specific	software
vagrant	0.06	80.54	specific	software
elasticsearch	0.06	80.60	specific	software
elastic	0.06	80.66	specific	software
webadmin	0.06	80.72	specific	service
kafka	0.06	80.78	specific	software
ftpadmin	0.06	80.84	specific	service
webmaster	0.06	80.90	specific	service
vnc	0.06	80.96	specific	software
system	0.06	81.01	admin	–
contador	0.06	81.07	non-admin	role
ftptest	0.06	81.13	specific	service
service	0.06	81.18	non-admin	role
baikal	0.06	81.24	specific	software
ts	0.05	81.29	non-admin	misc
duni	0.05	81.34	non-admin	misc
temp	0.05	81.40	non-admin	misc
spark	0.05	81.45	specific	software

Username	Percentage of Attempts	Cumulative Percentage	Category	Sub-category
svn	0.05	81.50	specific	service
docker	0.05	81.55	specific	software
developer	0.05	81.60	non-admin	role
jira	0.05	81.66	specific	software
app	0.05	81.70	non-admin	misc
sinusbot	0.05	81.75	specific	software
apache	0.05	81.80	specific	software
sysadmin	0.05	81.84	admin	–
nexus	0.05	81.89	specific	software
uftp	0.04	81.93	specific	service
ec2-	0.04	81.98	non-admin	misc
bot	0.04	82.02	non-admin	misc
butter	0.04	82.06	specific	software
mcserver	0.04	82.10	specific	software
teamspeak3	0.04	82.14	specific	software
nginx	0.04	82.18	specific	software
csgo	0.04	82.22	specific	software
backup	0.04	82.25	non-admin	role
vbox	0.04	82.29	specific	software
csgoserver	0.04	82.33	specific	software
gpadmin	0.04	82.36	specific	software
info	0.03	82.40	non-admin	misc
hd	0.03	82.43	non-admin	misc
a	0.03	82.46	non-admin	misc
db	0.03	82.50	specific	service
teste	0.03	82.53	non-admin	misc
user2	0.03	82.56	non-admin	role
deployer	0.03	82.59	non-admin	role
daniel	0.03	82.63	non-admin	name
nvidia	0.03	82.66	specific	software
db2inst1	0.03	82.69	specific	software
ethos	0.03	82.72	specific	distribution
manager	0.03	82.75	non-admin	role
www-data	0.03	82.78	specific	service
wp	0.03	82.81	specific	software
redis	0.03	82.84	specific	software
testing	0.03	82.87	non-admin	role

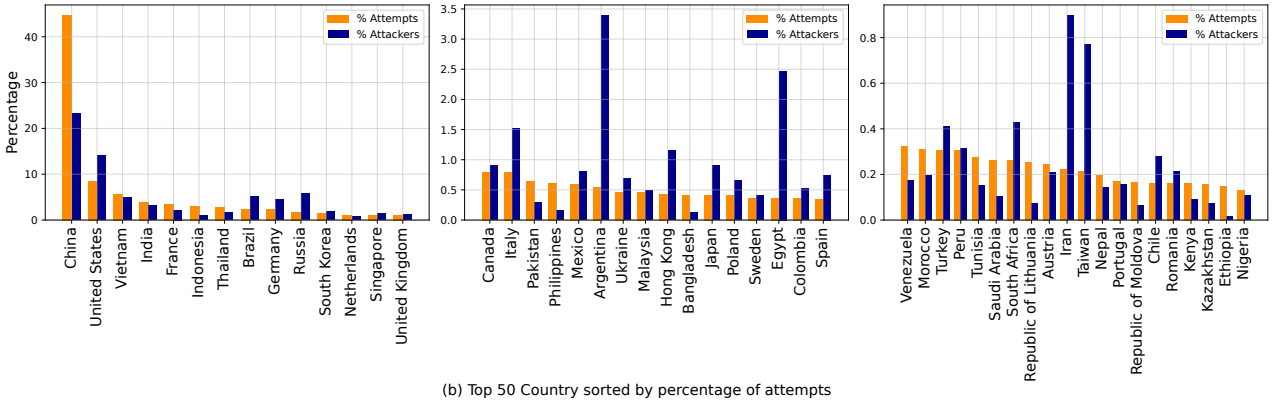
Table E.1: Username classification

F Top 50 Countries And Network Providers

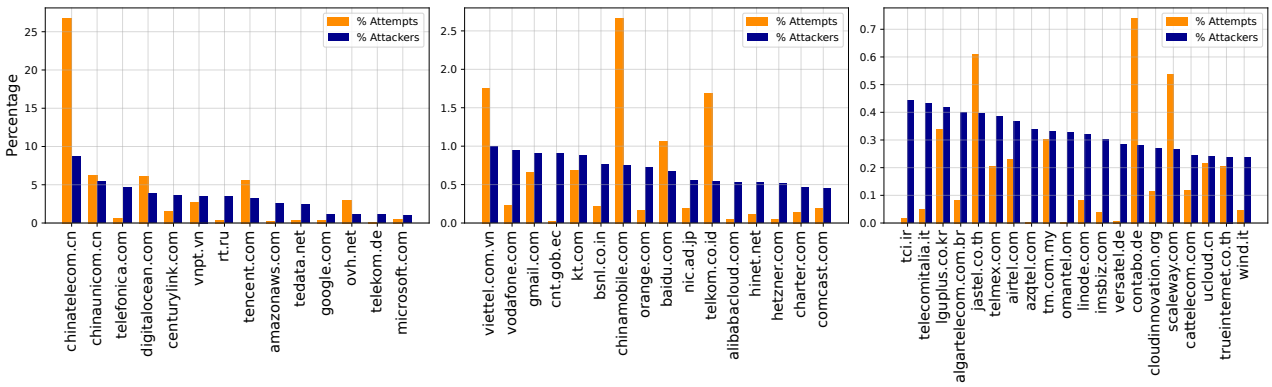
Figure F shows the top 50 countries and network providers based on the percentage of attackers and attempts.



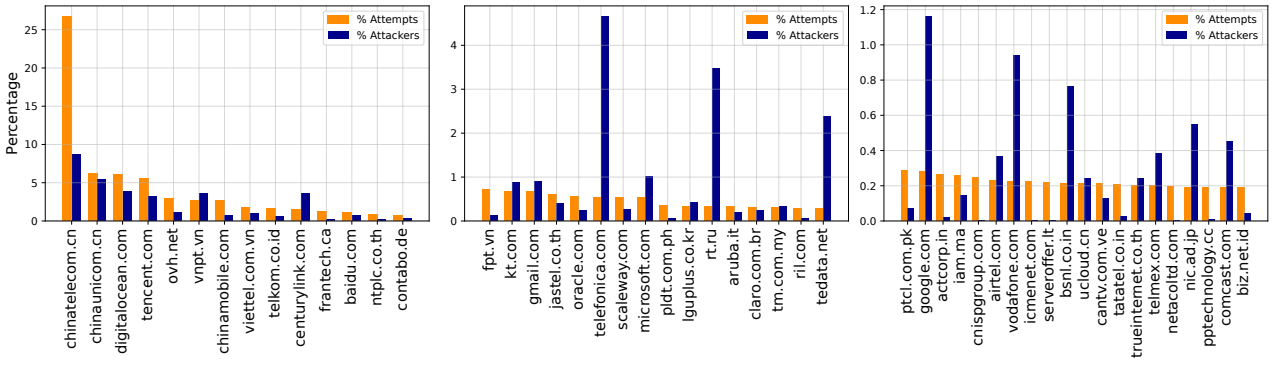
(a) Top 50 Country sorted by percentage of attackers



(b) Top 50 Country sorted by percentage of attempts



(c) Top 50 Network Providers sorted by percentage of attackers



(d) Top 50 Network Providers sorted by percentage of attempts

A System to Detect Forged-Origin BGP Hijacks

Thomas Holterbach[*], Thomas Alfroy[*], Amreesh Phokeer[†], Alberto Dainotti[‡], Cristel Pelsser[§]
[*]*University of Strasbourg,* [†]*Internet Society,* [‡]*Georgia Tech,* [§]*UCLouvain*

Abstract

Despite global efforts to secure Internet routing, attackers still successfully exploit the lack of strong BGP security mechanisms. This paper focuses on an attack vector that is frequently used: *Forged-origin hijacks*, a type of BGP hijack where the attacker manipulates the AS path to make it immune to RPKI-ROV filters and appear as legitimate routing updates from a BGP monitoring standpoint. Our contribution is DFOH, a system that *quickly* and *consistently* detects forged-origin hijacks in the *whole* Internet. Detecting forged-origin hijacks boils down to inferring whether the AS path in a BGP route is legitimate or has been manipulated. We demonstrate that current state-of-art approaches to detect BGP anomalies are insufficient to deal with forged-origin hijacks. We identify the key properties that make the inference of forged AS paths challenging, and design DFOH to be robust against real-world factors (e.g., data biases). Our inference pipeline includes two key ingredients: *(i)* a set of strategically selected features, and *(ii)* a training scheme adapted to topological biases. DFOH detects 90.9% of the forged-origin hijacks within only ≈5min. In addition, it only reports ≈17.5 suspicious cases every day for the whole Internet, a small number that allows operators to investigate the reported cases and take countermeasures.

1 Introduction

On 3 February 2022, the cryptocurrency platform KLAYswap was targeted by hackers who stole $1.9 million worth of digital assets [59]. More recently, on 17 August 2022, an attack to cBridge—a crypto-asset bridge—affected 32 victims, who lost $235,000 [4]. Both attacks were the result of a *forged-origin BGP hijack*, a type of routing hijack where the attackers announce forged AS paths towards a victim prefix by prepending the victim's origin AS number in order to make them appear legitimate. Clearly, BGP hijacking attacks are not a surprise anymore. They repeatedly make the headlines [1,2] and are known as attack vectors to steal cryptocurrency [8], obtain bogus certificates [15], or deanonymize Tor users [62]. The vulnerability they exploit is simply the result of BGP being designed without security in mind: An attacker can manipulate every attribute in a BGP message (including the AS path and its origin AS) and illegitimately announce a prefix owned by its victim so as to divert the traffic to its network.

Proactive solutions against BGP hijacks are being gradually deployed. However, forged-origin hijacks have been left uncovered by such solutions—despite these attacks being actively used in the wild. In fact, network operators attempt to proactively thwart BGP hijacks by configuring their routers to filter hijacked routes [46] using *(i)* RPKI-based Route Origin Validation (ROV) and *(ii)* data from Internet Routing Registries (IRR). Unfortunately, RPKI-ROV filters do not help to detect forged-origin hijacks, since the forged origin in the AS path is actually valid, while IRR-based filters are known to be inaccurate, incomplete [25], and far too often missing given the increasing number of observed BGP hijacks [7]. Today, network operators do not have many options left other than waiting for the deployment of new security extensions to BGP to consistently prevent forged-origin hijacks [44]. Such deployment—if it will happen at all—might take more than a decade, as in the case of RPKI-ROV [21].

In this paper, we present DFOH, the first *locally-deployable* system that *widely*, *quickly*, and *accurately* **D**etects **F**orged-**O**rigin **H**ijacks on the Internet. With a single deployment of DFOH on a commodity server, any attacker performing a forged-origin hijack is likely to be quickly detected, the hijack publicly reported, and the victim immediately notified. Being aware of the attack, the victim can apply countermeasures and the community can take actions to prevent similar attacks from happening again. Additionally, DFOH can detect past attacks, allowing the community to measure the frequency of such attacks or profile forged-origin hijackers.

DFOH is a passive system that processes the AS paths observed in publicly collected BGP routes to detect forged-origin hijacks. The problem of detecting forged-origin hijacks can be reduced to identifying whether a link between two ASes is real or fake. Unfortunately, there are multiple reasons why two ASes might connect, whereas there is no simple

property that indicates whether they are not supposed to. In fact, while *link prediction* in the AS graph might appear as a similar problem, approaches presented in literature do not translate well to revealing forged-origin hijacks [29, 68]. Alternatively, the only detection method specifically designed for forged-origin hijacks (ARTEMIS [56]) is solely capable of detecting hijacks targeting the network deploying it.

With DFOH, we address this challenge by first identifying and studying a rich set of domain-specific properties. Then, we derive classification features from them that—when carefully combined into an inference pipeline—enable us to effectively distinguish fake AS links from real ones. We observe that the key factors (e.g., data biases and discriminating power of classification features) that come into play when building the inference model, significantly vary depending on the characteristics of the two ASes whose hypothetical link is under examination. We take this observation into account when training DFOH to build a system that performs well in realistic settings. With DFOH, all attack scenarios (including adversarial inputs) are covered and legitimate cases are consistently not reported. DFOH is a solid line of defense against forged-origin hijacks: It detects 90.9% of them within ≈5min and only reports ≈17.5 suspicious cases every day, a small number compared to the size of the Internet.

Main contributions. Our main contributions are:

- An identification of the key factors to consider when designing a forged-origin hijack detection system that is effective in real-world scenarios (§4).
- The design of the DFOH inference pipeline, which quickly and accurately detects *any* forged-origin hijacks on the *whole* Internet (§5 and §6).
- An open-source implementation of DFOH, which we also run as a service (`https://dfoh.uclouvain.be/`).
- An evaluation of DFOH on synthetic and real data, which demonstrates that it is an effective line of defense against forged-origin hijacks and that its design is sound (§7).

2 Background

BGP Hijacks. As BGP was originally designed without security in mind, an AS can announce a prefix that it does not own with the result of diverting traffic towards itself, either accidentally or intentionally. This is referred to as a BGP hijack and results in some ASes adopting the false route when it is perceived to be better than the legitimate one. As a result, the traffic originated by many Internet users and destined to the hijacked prefix is sent to the hijacker AS instead of the AS owning the hijacked prefix. Consider the scenario in Fig. 1, which depicts an AS-level topology with nine ASes, one Content Delivery Network (CDN) and one Internet Exchange Point (IXP) and assume that they all configure routing policies following the Gao-Rexford model [28]. Assume that *AS6* hijacks `9.0.0.0/8` (without prepending the origin AS). Without defenses, the hijacked route is used by four ASes (*AS1*, *AS3*, *AS4* and *AS7*), which thus send traffic for that prefix to the hijacker AS. These hijacks are commonly called misorigin hijacks or "Type-0" hijacks following ARTEMIS's convention [56]. Observe that *AS6* could also hijack a more specific prefix, e.g., `9.0.100.0/24`, in which case *all* the ASes divert the traffic destined to `9.0.100.0/24` to the hijacker, since routes towards more specific prefixes are always preferred.

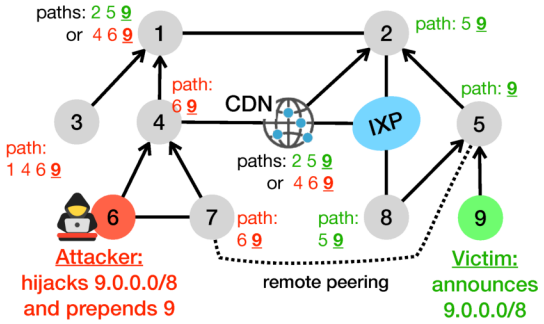


Figure 1: An example of an AS-level topology and the effect of a forged-origin hijack on the traffic destined to `9.0.0.0/8`. Nodes are ASes, straight arrows are customer to provider links and straight lines are peer to peer links.

Defenses against misorigin hijacks. Misorigin hijacks are often accidental and can be detected by looking at the origin of their AS path. Detecting them is effective today because there are many BGP vantage points from RIS [55] and RouteViews [51] that collect BGP routes received by many ASes and make them available to users. Sermpezis et al. show that the current public BGP monitoring infrastructure is able to observe all the hijacks with significant impact [56]. Thus, many hijack detection systems examine the collected BGP routes and raise an alarm when the origin AS is invalid [19, 35, 42, 56, 65]. Operators can also configure their routers to automatically deny routes with a wrong origin AS by using the Resource Public Key Infrastructure (RPKI) [43]. While RPKI-ROV is gradually being deployed [21], it only protects against misorigin hijacks and cannot help to detect AS-path manipulations.

3 Attack Model

Forged-origin hijack. Imagine that *AS6* in Fig. 1 is a clever attacker and knows that RPKI-ROV is increasingly adopted by operators [21]. After reading the state of the art about routing attacks in the literature [30], the attacker decides to prepend *AS9* in the AS path of the hijacked route (as shown in Fig. 1). Rapidly after applying this configuration, a significant part of the Internet traffic destined to `9.0.0.0/8` is deflected to

the attacker's AS. The attack is successful because the origin of the hijacked route is valid. Consequently, the RPKI-based filters miss the hijacked route, which also remains invisible from all the hijack detection systems that solely look at the origin AS. For instance, *AS4* in Fig. 1 receives a route for `9.0.0.0/8` with AS path *AS6-AS9* and accepts it given that its origin (*AS9*) is valid. Observe that *AS1* and the CDN receive the two routes for `9.0.0.0/8` with the same AS path length, in which case their local preferences determine whether they use the legitimate route or the hijacked one.

Definition: A forged-origin hijack is a BGP hijack where an attacker AS announces a route for an IP prefix that it is not authorized to originate and with an AS path that the attacker purposely manipulates so that the origin AS is valid.

The flip side of a forged-origin hijack is that it makes the AS path longer, which results in fewer ASes using the hijacked route compared to a misorigin (Type-0) hijack.

More specific prefixes are not better off. Despite RPKI, forged-origin hijacks can also succeed towards more specific prefixes. Operators oftentimes set a loose RPKI MaxLength [31] (sometimes confusingly [21]), allowing the attacker to launch successful forged-origin hijacks on more specific prefixes—a particularly harmful attack, since all traffic from *all* ASes is directed to the attacker [30, 32].

More advanced path manipulations are possible. An attacker can prepend more than one AS number. For instance, the attacker *AS6* in Fig. 1 could prepend *5 9*, in which case the origin is valid but the attacker AS appears in the third position from the origin. In this paper, we align with the state-of-art taxonomy [56] and define a Type-X hijack as follows:

Definition: A Type-X hijack is a forged-origin hijack where $X \geq 1$ indicates the position of the attacker's AS in the forged AS path, with the first AS (the origin) being at position zero.

In the (most common) cases where each AS number appears once in the AS path, X indicates the number of prepended ASes. Prepending *5 9* thus results in a Type-2 hijack. Intuitively, the higher is the number of prepended ASes, the lower is the impact of the attack (since the hijacked route reaches fewer ASes, as its AS path is longer).

4 The Case for DFOH

In this section, we highlight why DFOH is practically relevant (§4.1), and identify its key challenges and requirements (§4.2).

4.1 Lack of Defenses

DFOH is practically relevant because there is currently no effective mechanism to detect forged-origin hijacks.

Existing proactive defenses are not bulletproof. Because RPKI-ROV does not prevent forged-origin hijacks, the main proactive defense for network operators is to check whether the announcements of their customers are correct, i.e., each customer legitimately holds the AS numbers and IP address space they announce. This is achieved using route filters. In theory, these filters, recommended by MANRS [46], prevent an AS to propagate incorrect routing information when they are properly configured. Yet, in practice, they are often missing, inaccurate, or controllable by an attacker.

Missing filters: In April 2020, ROSTELECOM hijacked several prefixes, which impacted at least Amazon and Akamai. After investigation, it appears that at least two ASes, namely Rascom amd Cogent Co, did not configure proper filters that would have reduced the spread of the hijack [47]. Although an increasing—but still rather low—number of operators agree to follow routing security norms [26], we still observe many hijacks that widely propagate through the Internet, which indicates that these filters are often missing [48].

Inaccurate filters: Network operators often rely on peering information from the IRRs [39] to infer their customer cone, i.e., the set of ASes that can be reached only using customer links—a necessary information to build accurate filters. However, IRRs are known to contain inaccurate information [24, 25, 53], which inevitably results in inaccurate filters.

Controllable filters: Because the data from the IRRs is not verified [49], an attacker can intentionally pollute it to take control over which filters are added or removed in its provider's routers. More precisely, an attacker could pollute its `as-set` object, which is the IRR object that specifies a set of ASNs through which traffic can be routed, and is the object often used to automatically generate route filters [12]. This is supposedly what happened no later than in August 2022: An attacker (AS209243) added an Amazon's AS number into its `as-set` object before it started to announce an Amazon's prefix with an Amazon's AS number purposely prepended at the origin of the AS path [5].

Existing reactive defenses are narrowly focused. ARTEMIS is a system that detects forged-origin hijacks [56]. However, it is narrowly focused since it only detects hijacks that pertain to the AS deploying it. The key mechanism that ARTEMIS relies on to detect forged-origin hijacks consists in classifying new AS links as legitimate when observed in both directions, whereas it reports all the others as possible fake links. The problem is that ARTEMIS' approach cannot be directly extended to monitor *all prefixes*. Link bidirectionality can rule out a significant fraction of AS paths with new links when *they pertain to prefixes originated by the AS running ARTEMIS*. In fact, ARTEMIS likely observes both directions of the *neighboring* links as it combines BGP views from public collectors with local BGP views. However, when targeting attacks toward *any prefix* on the Internet, and using only public router collectors as vantage points, we find that bidirectionality is observable for only a small fraction of new links (see §6.2.4). Therefore, in the practical scenario that DFOH targets, solely relying on link bidirectionality results in a poor accuracy, as we show in §E.1.

4.2 Requirements

We now highlight four key challenges and requirements that DFOH must address to be effective and practical.

Requirement 1: A forged-origin hijack detection system must be fast and accept real-time and historical queries.

A fast detection (within a few minutes) enables mitigating the hijacks swiftly to limit their impact. Additionally, detecting past events is useful to e.g., lay hands on serial hijackers [64].

Requirement 2: A forged-origin hijack detection system must be accurate, both for pinpointing actual hijacks and avoiding triggering false alarms.

Accuracy matters because every detected hijack is likely to be manually investigated by the victim AS to confirm the attack and apply the appropriate mitigation scheme. A high number of false alarms would thus overwhelm network operators and limit the practicality of the system. Besides, the detection system should not miss actual hijacks.

Requirement 3: A forged-origin hijack detection system must be robust against missing, inaccurate and polluted data.

A forged-origin hijacks detection system must rely on a probabilistic inference as cryptographic-based AS paths verification is still missing today (BGPsec is not deployed at all [44]). The inferences are made using the data collected in various datasets including public peering information (PeeringDB and the IRR [39, 52]), which are known to often be inaccurate and incomplete [25, 53]. Worse, these datasets can be polluted by attackers, as they depend on voluntary and unverified contributions [49]. Consequently, care must be taken when designing the system to ensure robustness against any possible adversarial inputs or missing data.

Requirement 4: A forged-origin hijack detection system must be accurate in all attack and peering scenarios.

Internet routing involves various sort of players (e.g., ASes, CDNs, IXPs, as summarized in Fig. 1). Thus, there are many possible attack scenarios. Each induces a different fingerprint on the collected data and may require a different detection scheme. Typically, a densely-connected Tier1 AS that announces many prefixes (among which some on behalf of its customers) and hijacks a prefix owned by a stub AS might be harder to detect than a single-homed stub AS hijacking the prefix owned by another stub AS. Similarly, there are many legitimate peering scenarios (e.g., a remote peering session between two ASes located in different regions of the world) and none should exhibit a high level of false alarms.

5 Overview

We describe DFOH's workflow (§5.1) as well as the functions that it provides (§5.2) and how users can leverage them (§5.3).

5.1 Workflow

DFOH's workflow comprises the following three main components that are executed on a daily basis.

Zooming on the new AS link (§6.1). DFOH processes the publicly available BGP routes and pinpoints AS links that appear for the first time in their AS paths. DFOH zooms on those links because a forged-origin hijack is likely to trigger the appearance of a new AS link—typically the forged AS link that connects the attacker and the victim in case of a Type-1 hijack (we discuss the corner cases in §6.1). While a new AS link is an indicator to detect forged-origin hijacks, triggering an alert whenever a new link appears would result in a high number of inopportune alarms. In fact, legitimate factors such as new peering agreements or backup path activation can result in new AS links being visible too.

Unfortunately, we demonstrate in §7.4 that link prediction algorithms such as SEAL [68] do not translate well to discriminating the malicious new AS links from the real ones. These algorithms fail because they are generic whereas revealing malicious AS links is a problem with specific properties. Thus, DFOH includes the following components that aim to discriminate the fake new AS links from the legitimate ones.

Computing features (§6.2). DFOH uses a set of features that we carefully selected based on the key requirements identified in §4.2 and that are computed with security in mind: They remain correct even with adversarial input. We divide these features into the following four categories:

Topological features allow to quantify the change induced by a new link on the AS topology (following the reasoning in [63]) and pinpoint the suspicious ones (e.g., that do not follow the typical hierarchical structure of the AS topology).

Peering features are peering characteristics such as points of presence within a logical or geographical region that are computed on a per-AS basis. Intuitively, two ASes that exhibit similar peering characteristics have a higher chance to peer.

AS-Path-pattern features indicate whether the AS paths observed from vantage points and that include a new link are relevant based on the supposedly configured routing policies.

Bidirectionality features indicate whether an AS link is observed in both directions, which is a sign of legitimacy.

DFOH uses these features for the following two reasons. First, they enable to consistently (i.e., in all attack scenarios) discriminate whether a new AS link is a legitimate interconnection or caused by a forged-origin hijack (see §7.1). Second, to ensure that DFOH works even when some data is missing and some feature values cannot be precisely computed. We evaluate the relevance of the different categories of features in §7.4, and confirm that when one category is not relevant for a particular attack scenario or missing, the others compensate.

Inferring forged-origin hijacks (§6.3). Finally, DFOH builds an inference model that takes as input an AS link and its computed features and infers whether this AS link is the result of a

forged-origin hijack. DFOH builds the model using the typical training pipeline used for *link prediction* in a graph [45, 68]. A set of existing (legitimate cases) and nonexistent (malicious cases) links are sampled from the AS topology and used as ground truth to train the inference model. Unfortunately, naively using the training pipeline of existing *link prediction* frameworks falls short for detecting forged-origin hijacks in all attack scenarios (we confirm this in §7.4). The problem is that they sample the AS links without taking into consideration the biases observed in the AS topology caused by its hierarchical structure. In fact, they sample the links uniformly at random, which returns samples in which stub-to-stub links are overrepresented and links connecting the highly-connected ASes are underrepresented. This skewed effect is particularly critical when generating the negative sample, i.e., a set of nonexistent links labeled as malicious and used in the training pipeline. Following the recommendation in [66], DFOH builds a balanced negative sample, representative of all possible malicious AS links, to ensure that all the attack scenarios are covered.

5.2 Software functionalities

Ease of deployment and usage. DFOH is open source and can run on a commodity server or a VM. Once installed, DFOH first downloads the different datasets and saves the parsed data in a database, which we make publicly available at `https://dfoh.uclouvain.be/`. It then processes the data and builds inference models on a daily basis.

Real-time and historical detection. DFOH uses the precomputed inference models to detect past and real-time forged-origin hijacks. Upon detection of a new AS link, DFOH only needs to compute a few features before running the inference, which is swift because it relies on a simple model (a random forest). The most time-consuming operation is bootstrapping DFOH, because it needs to build the database and the inference models for many days (for historical detection).

Wide and public detection at no cost for users. DFOH detects and reports forged-origin hijacks for all possible attackers and victims on the Internet, and for any time period. Users can examine the list of suspicious cases and apply filters on them (e.g., to focus on one AS number) at no cost as we publicly disclose them at `https://dfoh.uclouvain.be/`. We describe a few interesting cases in §A.

5.3 Planned usages

Leveraging DFOH locally. Network operators can use DFOH to detect forged-origin hijacks targeting their prefixes. Our evaluation (§7.3) shows that in the median case, a given AS is only involved in zero or one suspicious case in a month, given network operators to ability to manually check each reported case and take the proper countermeasures if necessary. This usage is similar to how users use ARTEMIS [56]. However, unlike ARTEMIS which relies on a list of neighboring ASes provided by the user as well as feeds from local routers, DFOH does not require users to install any software and configure it.

Leveraging DFOH globally. DFOH enables a global detection of forged-origin hijacks, which is useful for the scientific and operational community. For instance, DFOH can help researchers to characterize forged-origin hijacks (e.g., their frequency, scope, or to profile serial hijackers [64]). Additionnally, DFOH helps to globally monitor Internet routing and is complementary to global BGP monitoring systems that detect misorigin (Type-0) hijacks (e.g., [35]), traffic delays, and disconnections (e.g., [37, 38]). Finally, we envision the output of DFOH to be used in the BGP decision process as an alternative to BGPSec and ASPA [9, 44], which may take years to be deployed. Network operators could deprioritize (or drop) suspicious routes over legitimate ones to prevent them from propagating to other networks.

6 Design

In this section, we motivate and explain the design of the DFOH's internal components and algorithms.

Terminology. Throughout this section, we consider the undirected graph $G_{d,k} = (V_{d,k}, E_{d,k})$ as the AS topology inferred at a given day d from the AS paths collected during the k days prior day d. $V_{d,k}$ is the set of ASes, and $E_{d,k} \subseteq V_{d,k} * V_{d,k}$ are the links between the ASes. We also consider that a new link (v_1, v_2) appears at day d, and is visible in a set A of AS paths.

Datasets. We collect BGP routes using BGPKIT [11] from 287 RIS [55] and RouteViews [51] vantage points that we carefully select using MVP [6]. We collect the AS paths that CAIDA uses to build its AS relationship dataset [3]. For now, we collect the data from one IRR (RADb) as it is the only one that makes available archives of its database. We collect daily snapshots of PeeringDB from the CAIDA website [17]. We explain how we clean and combine these datasets in §B.

6.1 Zoom on new AS links

We start by explaining how DFOH detects new AS links.

6.1.1 Under ideal conditions

Consider that an attacker controlling AS a launches a Type-1 hijack on a prefix owned by the victim AS v, with $a, v \in V_{d,k}$. Rapidly after, the BGP vantage points start to observe the hijacked routes and record their AS path. They likely observe different AS paths because they are scattered everywhere on the Internet. Yet, they all observe an AS path that ends with the attacker-to-victim link, here $(a-v)$, which is a new AS link, i.e., $(a-v) \notin E_{d,k}$. This is the case in Fig. 1, where the BGP

routes induced by the forged-origin hijack launched by *AS6* all have an AS path that ends with *AS6-AS9*. DFOH follows ARTEMIS' approach to detect new AS links: It considers the AS topology $G_{d,k}$, with $k = 300$, i.e., sufficiently high to avoid missing existing links, and classifies an observed link (x, y) as new if $(x, y) \notin E_{d,k}$. We detail how DFOH builds $G_{d,k}$ in §B.

6.1.2 In the real world

DFOH has to deal with real world factors, e.g., attackers could advertise carefully manipulated BGP updates to thwart DFOH. We now list the scenarios in which a forged-origin hijack does not create a new AS link, and show that either DFOH includes mechanisms to avoid them or that they occur only when the impact of the attack is greatly limited.

Scenario 1: The attacker hijacks the prefix owned by an AS with which it legitimately peers.

This attack scenario is akin to a route leak. For instance, an attacker announcing a route learned from one provider to another provider is defined as Type-1 route leak according to RFC 7908 [60]. Route leaks are outside the scope of DFOH because there already exists tools that aim to detect them in the wild [10,23,61]. Besides, thwarting DFOH by legitimately peering with its victim makes the attack harder to perform as it requires *(i)* additional resources for the attacker (e.g., being present in a peering facility where its victim is present too) and *(ii)* to convince the victim to peer with it (unless if the victim peers with the route server of an IXP [54]).

Scenario 2: The attacker pollutes DFOH's database by advertising legitimate routes but with fake AS paths.

Past AS path manipulations carried in legitimate routes could pollute the graph $G_{d,k}$ by adding fake AS links, preventing DFOH to classify them as new AS links any longer. DFOH thwarts this scenario by filtering out links that it inferred as fake from past inferences, i.e., these links are not in $G_{d,k}$. Observe that a link incorrectly inferred as supicious can be recurrently inferred as supicious over time, polluting DFOH's output. To prevent incorrect inferences from piling up over time, DFOH only considers past inferences for up to one month, after which fake links are added in $G_{d,k}$ and not considered as new link anymore. This one-month delay gives enough time for operators to examine the suspicious cases and protect their network against a potential future attack. Additionally, DFOH's website provides filters for users to omit these recurrent cases so as to prevent them from polluting the output.

Scenario 3: The attacker announces a fake path that comprises an existing path between the victim and the attacker.

The attacker could prepend a path (ideally the shortest one) from the victim AS to its own AS that exists in the AS topology inferred from the routes collected by the BGP vantage points. For instance, in Fig. 1, the attacker could prepend *AS7 AS5 AS9* to avoid triggering a new link. However, prepending more ASes increases the length of the AS path, inducing a trade-off between *visibility* and *impact* of the attack (which we highlight in §C) and compelling the attacker to significantly reduce the impact of its attack. In fact, Type-1 hijacks are *impactful* (31.3% of the ASes are polluted in the median case) but also *visible* by DFOH (100% induce a new link in the median case) whereas Type-2 are slightly less *visible* (98.8% induce a new link in the median case) but also less *impactful* (1.3% of the ASes are polluted in the median case).

Scenario 4: The attacker ensures that its fake announcements bypass the public BGP route collectors.

Previous works show that this is possible (unless DFOH relies on some private collectors) [50]. However, the attacker must prepend additional AS numbers in the AS paths. As shown in previous works [50,56] (and confirmed in §C), this significantly reduces the impact of the attack by diverting less traffic to the attacker. For instance, the impact of hijack types larger than Type-1 is very limited or negligible (e.g., less than 10% of the ASes see the hijacked route for Type-2 hijacks [56]).

6.2 Features computation

Upon detection of a new link or an explicit user query, DFOH computes feature values. DFOH uses as input the new link and a set of AS paths that include this new link. These AS paths are inferred from the public BGP routes or directly provided by the user. We now explain how DFOH computes the feature values for the four types of features.

6.2.1 Topological features

The topological features aim to quantify how the new link changes the AS topology [63]. Table 2 (Appendix) lists the topological features that DFOH uses to capture different dimensions of the topological changes. The topological features are either relative to a node (node-based) or a pair of nodes (pair-based). DFOH uses seven node-based features that we classify into three categories. The first one quantifies how central and connected a node is in the graph; the second quantifies how connected are the neighboring nodes; and the third quantifies the topological patterns (e.g., triangles) that include the node. We classify the four pair-based features into two categories. The first one measures how close are two nodes based on their neighboring nodes whereas the second measures how close they are using their shortest distance. We omit other topological features as they are either redundant with the selected ones or too slow to compute.

Computing the feature values. DFOH computes the difference induced by the new link on the feature scores. More formally, assume a set F_n of node-based features and a set F_p of pair-based features. The feature values computation

differs depending on the feature type. Note that for each type of feature, DFOH uses $k = 300$ to build the AS topology $G_{d,k}$.

Node-based features: Consider feature $f_i \in F_n$ and $f_i(x, G_{d,k})$ its score for node x on $G_{d,k}$, with i the feature index in Table 2. The feature value $v(f_i, d, v_1)$ is the difference induced by the new link (v_1, v_2) on the score of feature f_i for node v_1 on day d, and DFOH computes it using the following equation.

$$v(f_i, d, v_1) = f_i(v_1, G_{d,k}) - f_i(v_1, G'_{d,k})$$

$G'_{d,k} = (E'_{d,k}, V'_{d,k})$ is the graph $G_{d,k}$ that includes link (v_1, v_2), that is $E'_{d,k} = E_{d,k} \cup (v_1, v_2)$. DFOH computes the feature values for both nodes v_1 and v_2. Given that there are seven node-based features, the resulting 14-dimensional feature vector $T_{node_based}(d, v_1, v_2)$ is the following:

$$T_{node_based}(d, v_1, v_2) = [v(f_0, d, v_1), v(f_0, d, v_2), \ldots, v(f_6, d, v_1), v(f_6, d, v_2)]$$

Pair-based features: Consider feature $f_i \in F_p$ where $f_i(x, y, G_{d,k})$ is its score for the pair of nodes x, y, with i the feature index in Table 2. The feature value $v(f_i, d, v_1, v_2)$ is the difference induced by the new link (v_1, v_2) on the feature score f_i for the pair of node v_1, v_2 at day d, and DFOH computes it using the following equation.

$$v(f_i, d, v_1, v_2) = f_i(v_1, v_2, G_{d,k}) - f_i(v_1, v_2, G'_{d,k})$$

Given that there are four pair-based features, the resulting 4-dimensional feature vector $T_{pair_based}(d, v_1, v_2)$ is:

$$T_{pair_based}(d, v_1, v_2) = [v(f_7, d, v_1, v_2), \ldots, v(f_{10}, d, v_1, v_2)]$$

6.2.2 Peering features

The peering features evaluate the likelihood that two ASes peer based on peering information collected from PeeringDB [52] and BGPView [13]. DFOH considers the five peering information listed in Table 1. The first three features stem from the fact that two ASes registered in the same country, connected to the same IXP, or present in the same facility are more likely to peer. The last two features stem from the fact that ASes that are not present in the same facilities but that have point of presence that are geographically close (e.g., same city) are more likely to peer. Of course, these intuitions are not always true and an obvious counterexample is remote peering. Fortunately, the different categories of features compensate between each other so that DFOH remains accurate even when one is less relevant (see §7).

Dealing with adversarial inputs and polluted data. The peeringDB data is sometimes missing because participation is voluntary. Besides, the integrity of the data is unverified and an attacker could populate deceitful peering information. DFOH addresses those two problems with the following strategy. Instead of computing the feature scores for a hypothetical attacker *ASX*, it computes the scores for the neighboring ASes, for which *ASX* has no control over the peering information. In fact, an operator can only update the peering information relative to its own organization. Besides, as ASes often have several neighbors (the average node degree of the AS topology is 12 and the median is 2), focusing on the neighboring ASes helps find relevant peering information even if a few of them do not add peering information into peeringDB.

Index	Description
1	The countries where *ASX*'s neighbors are registered
2	The IXPs to which *ASX*'s neighbors are connected to
3	The facilities to which *ASX*'s neighbors are present
4	The cities of the facilities to which *ASX*'s neighbors are present
5	The countries of the facilities to which *ASX*'s neighbors are present

Table 1: List of peering features used by DFOH along with their description. We consider features computation for *ASX*.

Computing the feature values. Consider the vector $f_{v,i,d}$ that contains information about feature i for node v at day d. For each feature i, DFOH builds two vectors $f_{v_1,i,d}$ and $f_{v_2,i,d}$ based on peering information collected at day d. On September 19, 2022, for features 1 and 5, the vectors have 271 dimensions and each dimension corresponds to one of the 271 countries found in peeringDB. Similarly, for feature 2, the vectors have 944 (number of IXPs) dimensions, whereas for feature 3 they have 3558 (number of facilities) dimensions, and for feature 4 they have 1482 (number of cities) dimensions. The value of $f_{v,i,d}$ at index j is the number of v's neighbors that are in the country/IXP/facility/city that corresponds to index j.

DFOH then normalizes the two vectors $f_{v_1,i,d}$ and $f_{v_2,i,d}$ such that they become comparable even if v_1 and v_2 have a different number of neighbors (normalization operation ∇), and removes indexes for which the values in both vectors are zero (feature reduction operation $\ominus$). Finally, DFOH computes the feature value for the link (v_1, v_2) and a feature i by computing the cosine distance between the two vectors, which quantifies how similar the two vectors are (operation α). We use the cosine distance because we are interested in the direction of these vectors, not in their actual values, which depend on the number of neighbors, an irrelevant information for DFOH. For a given link (v_1, v_2) and a day d, DFOH computes the following 5-dimensional feature vector.

$$P(d, v_1, v_2) = [\alpha(\ominus(\nabla(f_{v_1,1,d}), \nabla(f_{v_2,1,d}))), \ldots, \alpha(\ominus(\nabla(f_{v_1,5,d}), \nabla(f_{v_2,5,d})))]$$

6.2.3 *AS-path-pattern* feature

DFOH uses the AS paths of the hypothetically observed hijacked routes to compute the *AS-path-pattern* features. More precisely, DFOH checks whether the sequence of AS degree

and customer cone size in an observed AS path $p \in A$ is relevant given the following two assumptions. First, as the AS topology exhibits a hierarchical pattern with Tier1 ASes at the top, we expect that ASes higher in the hierarchy exhibit a higher AS degree and customer cone size. Second, because the majority of the inter-domain routing policies follow the Gao-Rexford model [33], we expect the AS paths to have a valley-free pattern. When these two assumptions are valid, the sequence of AS degree and customer cone size in an AS path follows a strong and identifiable up-and-down pattern. For instance, between two stub ASes, we expect the AS degree and customer cone size to increase until the path reaches the Tier1 ASes, and then to decrease until it reaches the destination AS.

Inferring the suspicious AS paths. Unsurprisingly, these two assumptions do not always hold. For instance, a CDN may have a higher node degree than some of its providers. DFOH thus trains an inference model that computes the probability that a sequence of AS degree or customer cone size is legitimate or caused by a forged-origin hijack, based on historical (for the legitimate cases) and artificial (for the hijack cases) data used as ground truth. More precisely, we select a set of existing and nonexistent AS links. The existing links are selected randomly whereas the nonexistent links are selected following the sampling scheme described in §6.3, which ensures that the distribution of the nonexistent links follows the distribution of the existing links. Then, for each existing link, we randomly pick an AS path that includes this AS link and where one end of the link is at the origin. For the nonexistent links, we randomly define the attacker and the victim and pick an existing AS path for which the origin is the hijacker AS and add the victim as a new origin.

Computing the feature values. DFOH trains a random forest on sequences of AS degree and customer cone size inferred from the created AS paths. DFOH finds the best parameters of the random forest using a cross-validated grid search over a parameter grid. The degree of an AS is computed from the AS topology that DFOH builds on a daily basis, and the customer cone size is obtained from ASRank [18]. DFOH computes the following 3-dimensional feature vector.

$$J(d, v_1, v_2, p) = [\rho(d, v_1, v_2, p), \sigma(d, v_1, v_2, p), \tau(d, v_1, v_2, p)]$$

where ρ is the result of the fake AS path inference considering the AS degree for link (v_1, v_2), AS path $p \in A$ and day d. Similarly, σ is the result of the inference for the customer cone size, and τ is the result of the inference when AS degree and customer cone size are combined. We find that combining both AS degree and customer cone size improves the inference on some scenarios where, e.g., CDNs are involved. Note that each feature value is computed using its own independently-trained inference model that DFOH updates every day.

6.2.4 The *bidirectionality* feature

Identifying an AS link in both directions is a strong sign that it is legitimate [56]. However, checking for link bidirectionality is more challenging for DFOH compared to local detection systems such as ARTEMIS [56]. This is because the routes collected by the public BGP vantage points only allow to observe a small fraction of the new AS links as bidirectional. DFOH improves state-of-the-art techniques by combining the information from the public BGP data and the IRR to observe more AS links in both directions. We explain our methodology and demonstrate its *safety* against adversarial inputs and *benefits* in §D.2 due to space constraints. Computing the bidirectionality feature results in the following 1-dimensional feature vector.

$$B(d, v_1, v_2) = [bidir(d, v_1, v_2)]$$

Where $bidir(d, v_1, v_2) = 1$ if the link (v_1, v_2) is bidirectional at day d, else it is equal to zero.

6.3 Inference

We now explain how DFOH runs (§6.3.1) and trains (§6.3.2) its inference model using balanced samples (§6.3.3).

6.3.1 Detecting forged-origin hijacks

After computing the feature values for a new link (v_1, v_2) and the observed AS path $p \in A$ that includes the new link, DFOH concatenates the resulting feature vectors and obtains the following 27-dimensional feature vector.

$$F(d, v_1, v_2, p) = T_{node_based}(d, v_1, v_2) \oplus T_{pair_based}(d, v_1, v_2) \\ \oplus P(d, v_1, v_2) \oplus J(d, v_1, v_2, p) \oplus B(d, v_1, v_2)$$

Where $\oplus$ is the concatenation operation. DFOH uses this feature vector as input to its inference model, which is a supervised binary classifier. The classifier relies on a random forest as this algorithm returns a slightly better performance compared to others (e.g., neural networks, decision tree or SVM), is easier to understand, and is fast to train and query.

DFOH refines its inference using many vantage points. A new AS link is often visible from different BGP vantage points, and the AS paths that include this new link may be different. DFOH computes the *AS-path-pattern* features for all these AS paths, runs inferences for this new link and for every observed AS path using the computed *AS-path-pattern* features, and triggers an alarm if half or more of the inferences detect a forged-origin hijack. Observe that DFOH performs well even if only one AS path is used ($|A| = 1$), which is what we use to evaluate DFOH in §7.1.

6.3.2 Training the classifier

DFOH trains its classifier following a supervised training approach used in state-of-the-art link prediction frameworks [36,

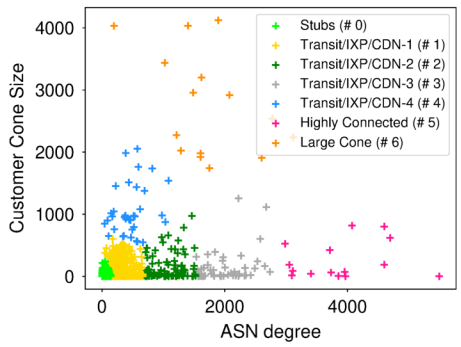


Figure 2: Computed clusters of ASes on April 30, 2022.

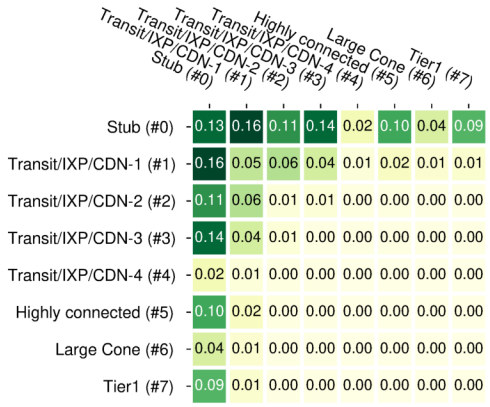


Figure 3: Link distribution within and between clusters. Each cell indicates the proportion (green means high proportion).

68]. From the graph $G_{d,k}$, DFOH samples a set of existing and nonexistent links, i.e., node pairs from the graph which have no link connecting them. DFOH samples 60 000 existing and nonexistent links that it uses to train the random forest. The AS paths presumably observed by the BGP vantage points and that include the new AS link (required to compute the *AS-path-pattern* feature) are generated using the same technique as in §6.2.3. For the training, DFOH estimates the best parameters of the random forest classifier using a cross-validated grid search over a parameter grid on 25% of the sampled links. Then, it trains the random forest classifier using the remaining 75% sampled links and with the optimal parameters. Note that DFOH uses different samples of links to train this classifier and the one used to generate the *AS-path-pattern* features (§6.2.3). DFOH trains and saves a random forest classifier on a daily basis so that it remains accurate over time.

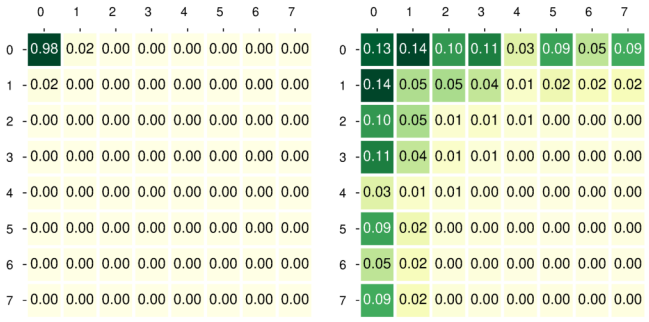
DFOH trains its classifier on representative samples. A key factor for a classifier to be accurate is the quality of the sample used for the training. A good sample should be a representative subset of the scenarios that the model to train will encounter in practice. There is no exception with link prediction frameworks, for which the quality of the samples of existing and nonexistent links plays a key role [67]. In the case of DFOH, a randomly generated sample of existing links turns to be a representative sample (e.g., stub-to-stub links are similarly represented in the initial and sampled set). However, randomly selecting a few nonexistent links returns a sample containing mostly stub-to-stub links and only a few links that involve e.g., transit networks. The sample is skewed because the AS topology is hierarchical, with many more lowly-connected ASes (e.g., stub ASes) than highly-connected ASes (e.g., Tier1 ASes). Thus, DFOH applies a balanced sampling scheme that aims to build a sample of nonexistent links that is representative of all the attacker scenarios.

6.3.3 Sampling all attack scenarios

An attack scenario is defined by the attacker and the victim AS and the link that appears between these two ASes. There exists many categories of ASes (e.g., stub or transit ASes) and within each category, the ASes exhibit different properties (e.g., high/low degree, or a different position in the topology). Sampling all types of attack scenarios is a challenge that DFOH addresses using the following two-steps approach.

Step #1: DFOH clusters ASes based on their degree and customer cone size. DFOH computes the degree of every AS based on the AS topology $G_{d,k}$ and obtains their customer cone size from ASRank [18]. It then uses K-means to cluster the ASes on a daily basis. We omit the Tier1 ASes listed in the CAIDA AS-level map [3] in the clustering but manually class them into their own dedicated "Tier1" cluster. We configure K-means to use the highest number of clusters as long as each cluster contains at least ten ASes. Fig. 2 shows the result of the clustering on April 30, 2022. There are seven clusters to which we assign different labels. For instance, we label the ASes with a low degree and customer cone size (the cluster in the bottom-left corner) as "Stub" ASes. ASes with a high degree but a customer cone size that is not as high are labeled as "Highly Connected" ASes and often correspond to CDNs. "Large Cone" ASes correspond to ASes at the top of the Internet hierarchy. Finally, the remaining ASes can be Transit, IXP, or CDNs and are divided into four clusters.

Step #2: DFOH samples links based on their distribution within and between clusters. Fig. 3 shows the link distribution, within and between clusters, observed in the AS topology. The number at the i-th line and j-th column indicates the proportion of links that connect an AS in cluster i with an AS in cluster j (the grid is symmetric). For instance, 10% of the links are connecting a "Stub" AS with a "Highly connected" AS. While a random sample of the *existing* links returns an inter-cluster link distribution similar to the one obtained when considering all the existing links (i.e., similar to the one in Fig. 3), a random sample of the *nonexistent* links returns a skewed inter-cluster link distribution. This skewed sampling is visible in Fig. 4, which shows the inter-cluster link distribution of the nonexistent links obtained depending on the sampling algorithm used. Most of the randomly sam-



(a) Random sample. (b) DFOH's sample.

Figure 4: Link distribution within and between clusters for the **nonexistent** links sampled using a random (4a) or DFOH's balanced (4b) sampling. The proportion of sampled links is written in each cell. Greener is a cell higher is the proportion.

pled nonexistent links connect two stub ASes because stubs are largely dominant in the AS topology (Fig. 4a). With this skewed sampling, DFOH (and also the state-of-the-art link prediction frameworks) detect forged-origin hijacks involving stub ASes, but performs poorly for other attack scenarios (see §7.4). Consequently, DFOH uses a sampling distribution for the *nonexistent* links that is proportional to power $3/4$ of the inter-cluster distribution of existing links, as suggested by Yang et al. [67]. The resulting sample includes nonexistent links for various inter-cluster scenarios (Fig. 4b)—enabling DFOH to work in all attack scenarios.

7 Evaluation

In this section, we show that DFOH is accurate (§7.1)—even on remote peering sessions (§7.2.1)—and practical (§7.3). We also demonstrate that the design choices of DFOH are relevant (§7.4). In the appendix, we show that DFOH is fast (§E).

Taxonomy. Throughout this section, we focus on the accuracy of DFOH in terms of *True Positive Rate* (TPR) and *False Positive Rate* (FPR). The TPR is the proportion of actual forged-origin hijacks (i.e., *Positive* cases) that are detected by DFOH, whereas the FPR is the proportion of legitimate (i.e., *Negative*) cases incorrectly inferred as malicious.

7.1 Accuracy

The challenge to evaluate the accuracy of DFOH is the lack of ground truth. In fact, while forged-origin hijacks occur in practice, they may not be publicly disclosed, as they negatively impact the victim's reputation. To address this challenge, we evaluate DFOH on both synthetic data and real cases.

Methodology. We measure the TPR and FPR of DFOH using synthetic data. As we have (synthetic) ground truth, we can measure the overall, and specific, performance of DFOH upon various attack scenarios. The synthetic data is built identically to the training data used in §6.3.2, i.e., it is constituted of a sample of (existing and nonexistent) links different from those in the training set. We also run DFOH on the two forged-origin hijacks reported in the news [5, 59] and which are interesting case studies as they involve different types of ASes.

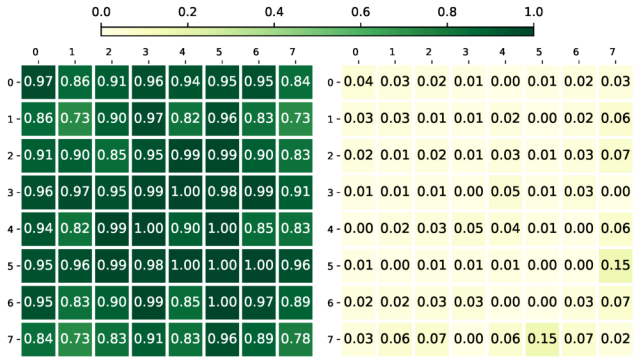
DFOH is accurate in any attack scenario. We measure the accuracy of DFOH on 9000 existing and nonexistent links selected using our balanced sampling (§6.3.3). We use DFOH's inference model computed on April 30, 2022 to align with Fig. 2. DFOH correctly detects 8181 synthetic forged-origin hijacks (TPR=0.909) and incorrectly inferred as forged-origin hijack 171 legitimate AS links (FPR=0.019). We further tested DFOH on various dates (including in 2023) and always found a comparable accuracy. Observe that only one AS path is considered for every inference with the synthetic data. In practice, this scenario happens when only one BGP vantage point sees the hijacked route. DFOH could refine its inference and possibly exhibit an higher TPR and lower FPR with more AS paths, i.e., if more vantage points see the attack (e.g., see §7.2.1).

We now investigate the accuracy of DFOH for specific classes of attack scenarios in Fig. 5. We consider the clusters of ASes as computed in §6.3.3 to classify the attack scenarios and measure the TPR and FPR within and between the clusters. Here, we sample 100 links for every attack scenario, unless fewer than 100 exist, in which case we take all of them (the minimum number of links for a scenario is 58). The number at the i-th line and j-th column indicates the TPR or FPR when the attacker is in cluster i and the victim in cluster j. DFOH is accurate in any attack scenario: The lowest TPR for an attack scenario is 0.73, and occurs when an AS in the "Tier1 (#7)" cluster hijacks another AS in the "Transit/IXP/CDN-1 (#1)" cluster, or when an AS in the "Transit/IXP/CDN-1 (#1)" hijacks an AS either in the same cluster or a "Tier1 (#7)" AS. The highest FPR is only 0.15 and occurs when a "Tier1 (#7)" AS hijacks an AS in the "Highly Connected (#5)" cluster.

DFOH detects actual forged-origin hijacks. DFOH detects[1] the Type-1 hijack launched by AS209243 on AS14618 (one of Amazon's ASes) [5]. Both the victim and the attacker ASes are classified as "Stub" ASes. We use the only available reported AS path: 34854 1299 209243 14618. The fact that DFOH detects this hijack despite the attacker manipulating its IRR data to presumably fake a peering link with AS14618 confirms its robustness against adversarial inputs.

DFOH also detects[2] the Type-1 hijack launched by AS6461 on AS9457 [59]. We used the four distinct AS paths reported in [59] as input. This use case highlights the ability of DFOH to detect forged-origin hijacks that involve central ASes (the attacker is a Tier1 AS and attacked an AS classified as "Stub"), which can be more challenging to detect compared to others.

[1] https://dfoh.uclouvain.be/cases/2022-08-17_14618_209243
[2] https://dfoh.uclouvain.be/cases/2022-02-03_6461_9457



(a) TPR. (b) FPR.

Figure 5: Performance of DFOH upon various attack scenarios. The y- and x-axis indicate the cluster in which the attacker and the victim are classified, respectively.

7.2 Robustness

We measure the accuracy of DFOH on cases that are intuitively harder to classify. We first focus on remote peering sessions (§7.2.1) and then on peering sessions through IXP (§7.2.2).

7.2.1 Remote peering sessions

New *remote* peering sessions are inherently harder to infer as legitimate for DFOH as they exhibit different characteristics than the more common local peering sessions. We now show that DFOH remains accurate in two remote peering scenarios: tunneled sessions and remote peering at IXPs.

BGP tunnels. We measure the accuracy of DFOH on tunneled BGP sessions using a four-steps approach: *(i)* We take four tunnel brokers (Securebit AG, Serveperso, iFog, August Internet); *(ii)* We collect the AS links that they have with their downstream ASes (based on the dataset from `bgpview.io`); *(iii)* We intersect these links with the new AS links observed in 2022; and *(iv)* we run DFOH on the resulting new AS link events (155 events)—each one thus likely corresponds to a new tunneled BGP session. DFOH only reports 12.9% of them as suspicious and all others as legitimate events (FPR=0.129).

We also tested the ability of DFOH to detect forged-origin hijacks launched by ASes connected to the Internet through BGP tunnels. We randomly took 100 victim ASes in every cluster computed by DFOH (see §6.3.2). For every victim, we randomly pick one AS that is a downstream of a tunnel broker and assume it is the attacker. We then take the AS paths observed from the BGP vantage points and for which the assumed attacker is at the origin, and artificially create a forged-origin hijack by adding the victim AS at the origin. DFOH detected 97.5% of these attacks (TPR=0.975).

Finally, we tested DFOH against forged-origin hijacks launched by BGP tunnel providers. We follow the same approach to synthetically generate attacks and found that DFOH detects 72% of them (TPR=0.72).

Remote peering at IXPs. The ground truth in [34] enabled us to identify 42 new remote peering sessions that appeared at several IXPs between January and March 2018. We ran DFOH on them and found that it only classifies eight as suspicious (FPR=0.19). We also measured the ability of DFOH to detect forged-origin hijacks launched by ASes that remotely peer at an IXP. We follow the same approach as with the ASes peering through BGP tunnels (see above) and found that DFOH finds 87.7% of the attacks (TPR=0.877).

7.2.2 Fake peering sessions at IXPs

Fake AS links between two ASes that have an IXP in common are challenging for DFOH as they appear legitimate from a topological and peering perspective.

Methodology. We measure the accuracy of DFOH on 1000 fake links connecting two ASes sharing an IXP but that do not peer through this IXP. We build a fake link using the following methodology. We select an IXP using a weighted random selection where the weight depends on the number of participants of the IXP (i.e., largest IXPs are prioritized). Then, we randomly select two of its participants that do not appear directly connected from a BGP standpoint and define one as the victim and the other as the attacker. Finally, we create a fake AS path similarly as in §7.1, i.e., we take an existing AS path where the attacker is at the origin and add the victim AS as the new origin.

DFOH detects 65.6% of these attacks. We find that it is the *AS-path-pattern* features that enable DFOH to detect most of the attacks. In fact, when omitting this feature category, the accuracy drops to 26.2% whereas when omitting another feature category, DFOH is still able to detect ≈65% of the attacks. This is an intuitive observation as fake AS paths resulting from these attacks are likely to violate the expected routing policies, which the *AS-path-pattern* features can detect. On the contrary, *topological* and *peeringDB* features tend to misclassify fake links at IXPs as two ASes sharing a common IXP could legitimately peer from a topological or peering perspective.

More AS paths improves the accuracy. In fact, when building ten AS paths (instead of one) for every case, the accuracy increases to 73%. This is a logical observation as more paths increase the chance to infer some as being fake.

7.3 Practical use

We now show that DFOH is usable in practice: It pinpoints suspicious events without reporting too many incorrect alarms.

DFOH reports $\approx 13.1\times$ ***fewer suspicious cases than a naive approach.*** We ran DFOH on 2022 and compare it to a naive approach that simply reports all the AS links that appeared during this time frame and have not been observed during the

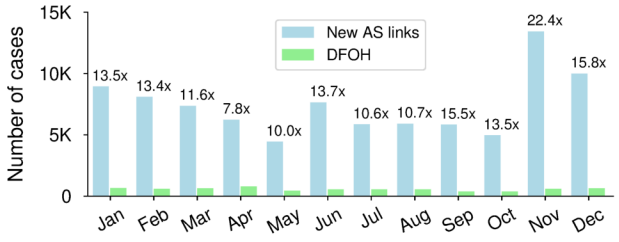


Figure 6: Number of new AS links and reported cases by DFOH for every month of 2022. We indicate the reduction factor at the top of the bars.

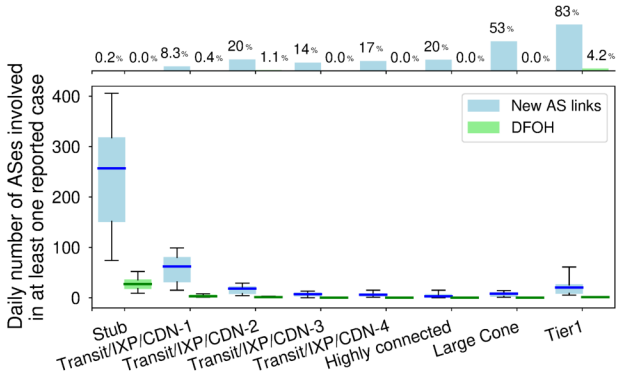


Figure 7: Number of ASes involved in at least one event for every AS category. We show at the top the proportion of ASes involved in at least one reported case every day (median).

last 300 days (to align with DFOH's design choices, see §6.1). As shown in Fig. 6, the number of suspicious cases reported by DFOH remains stable over time. In the median case (resp. 90th percentile), 180 (resp. 390) new links appeared *every day*, and DFOH only classifies 17.5 (resp. 31) of them as being suspicious. In 2022, the reduction factor (i.e., the ratio between the new AS links and the cases reported by DFOH) is 13.1 and ranges between 7.8 (April) and 22.4 (November).

DFOH is practical for each category of ASes. Fig. 7 shows the daily number of ASes involved in at least one event, as a function of their cluster (see §6.3.3). The line in a box depicts the median value; the whiskers show the 5 and the 95th percentile. Regardless of the category of ASes, DFOH always reduces by at least 8.4× the number of reported cases. For instance, in a single day, the naive approach reports 257 stub ASes involved in at least one new link, whereas DFOH only reports 27 stub ASes (median case). Besides, only a tiny fraction of the ASes (max. 4.2% for "Tier1" ASes) are involved in at least one reported case with DFOH every day. In July 2022, we count that the highest number of alarms that an operator can receive is 15, i.e., one every two days. The vast majority (99.8%) would receive zero or one alarm only during the month.

Unsurprisingly, the naive approach detects a higher number of cases involving stub ASes as they are overrepresented compared to others. Yet, when looking at the proportion of ASes involved in at least one reported case (top part), the naive approach seems to perform particularly poorly for the "Large Cone" and "Tier1" ASes as a significant portion of these ASes (53% and 83%, respectively) are involved, every day, in at least one reported case. On the contrary, DFOH exhibits high accuracy for every legitimate peering scenario.

DFOH pinpoints possible forged-origin hijacks. We manually inspected some of the detected suspicious cases and found cases that strongly seem malicious. In §A, we describe four of them and provide all the reported cases in 2022. However, it is impossible to precisely measure the FPR of DFOH when run on real BGP updates because of the lack of ground truth. What makes it even harder is that AS path manipulations are not limited to forged-origin hijacks. Among others, one can poison an AS path by prepending an AS number to prevent a BGP route to go through that AS or can remove AS numbers from an AS path to attract more traffic. DFOH may detect any sort of AS path manipulations that triggers the appearance of a new—but fake—AS link. Since these AS path manipulations often have malicious purposes, we argue that detecting them is useful for the hypothetical victim.

7.4 Relevance of the design choices

In this section, we show that our design choices are relevant.

Every feature category is useful but none is vital. We evaluate DFOH with all the four feature categories activated but one. The resulting TPR and FPR for a few different attack scenarios are reported in Table 3 and further discussed in §E.1. In short, each feature category helps improving the TPR or FPR by a significant factor in at least one attack scenario. We also observe that the accuracy of DFOH never appears particularly flawed when one feature category is missing. Thus, these feature categories compensate each other, allowing DFOH to remain accurate when there is missing or polluted data.

Training DFOH on a balanced sample is necessary. We trained DFOH on a randomly generated sample of existing and nonexistent links instead of using our balanced sampling scheme (r-DFOH in the following) and measured its performance on two synthetic datasets. The first dataset is built using a random sample of existing and nonexistent links, whereas the second is built using our balanced sampling presented in §6.3.3. Unsurprisingly, r-DFOH is accurate on the randomly generated dataset (the TPR is 97.3% and the FPR is 0.3%). However, it performs poorly when tested on the balanced dataset. In fact, the overall TPR is only 49.07%, and the FPR is 0.17%. Besides, r-DFOH is particularly inaccurate for the attack scenarios that are underrepresented in the random sample used for the training, e.g., between "Transit/IXP/CDN-1" ASes and "Tier1" ASes, the TPR is only 1.5%. When trained on a balanced sample, DFOH is accurate regardless of on which dataset it is tested (see §7.1), which demonstrates that

our balanced sampling scheme is a key ingredient to obtain high accuracy in every attack scenario.

DFOH outperforms state-of-the-art link prediction algorithms. We compared DFOH to SEAL, a state-of-the-art link prediction framework that could be an alternative to infer whether a new AS link is legitimate or fake [68]. We run SEAL on the AS topology inferred on August 1, 2022, and configure it to use 20 000 existing and nonexistent links for the training. We then evaluate it on 10 000 other existing and nonexistent links that are selected using our balanced sampling scheme. SEAL returns a TPR of 19.3%, and an FPR of 5.5%, a better accuracy than a random classifier that would return ≈50% of TPR *and* FPR, but a significantly lower accuracy than DFOH.

The accuracy of SEAL turns out to be very skewed (as we show in §E.2) because it uses a random sampling scheme for the training. In fact, the attack scenarios for which SEAL poorly performs are the ones involving the highly-connected ASes, i.e., the ones that are underrepresented in a random sample. We thus implemented *b*-SEAL, a modified version of SEAL that uses our balanced sampling scheme for the training instead. With a TPR of 80.6% and a FPR of 30.8%, *b*-SEAL is still significantly less accurate that DFOH. Yet, we find that its accuracy is more consistent across all attack scenarios. Thus, we conclude that *(i)* the balanced sampling is necessary for a consistent accuracy, regardless of the inference model used, and *(ii)* our selection of features based on domain-specific properties is relevant. We give more details in §E.2.

8 Related work

Misorigin (Type-0) hijacks detection. Prior works that analyze control-plane information to detect MOAS hijacks [19, 35, 42, 58, 65] can detect accidental hijacks but not the malicious ones induced by forged-origin hijacks. Prior works that detect hijacks from data-plane information [16, 70, 71] often can only be deployed per AS, precluding global analysis.

Forged-origin hijacks detection. ARTEMIS detects forged-origin hijacks involving the AS deploying the tool, but cannot be extended for global detection (see §4.1). Cho et al. introduce algorithms based on the AS hegemony [27] to classify reported hijacks as forged-origin hijacks [20]. The proposed global hegemony feature is similar in essence to our *AS-path-pattern* feature. Yet, without our key ingredients, this technique alone results in a low and skewed accuracy when used for *globally detecting* forged-origin hijacks (see §7.4). This is confirmed by the authors themselves, who acknowledge that their algorithm fails to classify hijacks that involve highly-connected ASes such as in the KlaySwap incident [59]. Kruegel et al. propose to detect anomalous BGP updates by combining geographical and topological information about the ASes in the path [40]. However, little is known about how this technique would work to detect forged-origin hijacks.

Link prediction applied to the AS topology. SEAL is a framework for link prediction in a graph but it does not apply well to detect fake AS links (see §7.4) [68]. Giakatos et al. compare link prediction algorithms based on graph-based prediction models on Internet routing data [29]. More precisely, they compute a set of features for every AS and feed them either into a GNN model or a graph embedding model such as bgp2vec [57]. The authors acknowledge that the AS topology and its hierarchical structure is challenging for a GNN or a graph embedding model, and their inference models do not translate well to detecting fake AS links. Finally, Shapira et al. proposed a deep-learning approach with a recurrent neural network based on node embedding computed using bgp2vec [57]. Yet, the performance of the proposed solution is evaluated on the small and biased dataset used in [20].

New protocols and architectures. BGPSec is an extension to BGP where routers cryptographically verify the validity of the AS path [44]. However, it is not deployed *at all*, as it requires expensive cryptographical operations in the routers. ASPA is a proposal to extend RPKI and use it for AS path validation but it is not extensively deployed [9]. Finally, new secure inter-domain protocols and architectures such as SCION [69] are challenging to widely deploy.

9 Conclusion

We present DFOH, the first system that consistently detects forged-origin hijacks on the Internet. DFOH only reports ≈17.5 cases every day, a small number that allows operators to manually investigate each case and take the proper countermeasures. We believe DFOH triggers interesting follow-up works, such as measuring the frequency of these events, profiling the forged-origin hijackers, or analyzing how often the data traffic is diverted to the supposed attacker.

10 Acknowledgments

We are grateful to the NSDI anonymous reviewers for their insightful comments. We are also grateful to our shepherd, Italo Cunha, for its detailed feedback which helped us to improve the quality of the paper. We thank the InetLab platform from the ICube laboratory for providing us a VM which we used to run DFOH.

Thomas Holterbach is partially funded by the Internet Society (through the MANRS Initiative) and by Région Grand Est. Thomas Alfroy is funded by ArtIC project "Artificial Intelligence for Care" (grant ANR-20-THIA-0006-01) and co funded by Région Grand Est, Inria Nancy - Grand Est, IHU of Strasbourg, University of Strasbourg and University of Haute-Alsace. This project has been made possible in part by a grant from the Cisco University Research Program Fund, an advised fund of Silicon Valley Foundation.

References

[1] Amazon once again lost control (for 3 hours) over the IP pool in a BGP Hijacking attack. https://research.securitum.com/amazon-once-again-lost-control-for-3-hours-over-the-ip-pool-in-a-bgp-hijacking-attack.

[2] BGP Hijack of Amazon DNS to Steal Crypto Currency. https://medium.com/oracledevs/bgp-hijack-of-amazon-dns-to-steal-crypto-currency-a90dd29cb3ab.

[3] CAIDA AS relationships. https://catalog.caida.org/dataset/as_relationships_serial_2. 2022-12-1.

[4] Celer Bridge incident analysis. https://www.coinbase.com/blog/celer-bridge-incident-analysis.

[5] Yet another BGP hijacking towards AS16509. https://mailman.nanog.org/pipermail/nanog/2022-August/220320.html.

[6] Thomas Alfroy, Thomas Holterbach, and Cristel Pelsser. MVP: Measuring internet routing from the most valuable points. In *IMC'22*.

[7] BGP, RPKI, and MANRS: 2020 in review. https://blog.apnic.net/2021/02/05/bgp-rpki-and-manrs-2020-in-review/.

[8] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking bitcoin: Routing attacks on cryptocurrencies. In *Security and Privacy*, 2017.

[9] Alexander Azimov, Eugene Bogomazov, Randy Bush, Keyur Patel, Job Snijders, and Kotikalapudi Sriram. BGP AS_PATH Verification Based on Autonomous System Provider Authorization (ASPA) Objects. Internet-draft, 2023.

[10] Marcelo Bagnulo, Alberto García-Martínez, Stefano Angieri, Andra Lutu, and Jinze Yang. Practicable route leak detection and protection with ASIRIA. *Computer Networks*, 2022.

[11] Fast, Extensible, On-premise Global BGP Monitoring. https://bgpkit.com/.

[12] bgpq4 - BGP filtering automation tool. https://github.com/bgp/bgpq4.

[13] BGPview. https://bgpview.io/".

[14] Bias in Internet Measurement Infrastructure. https://ripe84.ripe.net/archives/video/768/.

[15] Henry Birge-Lee, Yixin Sun, Anne Edmundson, Jennifer Rexford, and Prateek Mittal. Bamboozling certificate authorities with BGP. In *USENIX Sec*, 2018.

[16] Tobias Bühler, Alexandros Milolidakis, Romain Jacob, Marco Chiesa, Stefano Vissicchio, and Laurent Vanbever. Oscilloscope: Detecting BGP Hijacks in the Data Plane, 2023.

[17] Daily snapshots of historic PeeringDB data. https://publicdata.caida.org/datasets/peeringdb/.

[18] CAIDA AS Rank. http://as-rank.caida.org/.

[19] Massimo Candela. BGPAlerter, 2019.

[20] Shinyoung Cho, Romain Fontugne, Kenjiro Cho, Alberto Dainotti, and Phillipa Gill. BGP hijacking classification. In *TMA*, 2019.

[21] Taejoong Chung, Emile Aben, Tim Bruijnzeels, Balakrishnan Chandrasekaran, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, Roland van Rijswijk-Deij, John Rula, and Nick Sullivan. RPKI is Coming of Age: A Longitudinal Study of RPKI Deployment and Invalid Route Origins. In *IMC*, 2019.

[22] CIDR REPORT for 12 Oct 22. https://www.cidr-report.org/as2.0/".

[23] How we detect route leaks and our new Cloudflare Radar route leak service. https://blog.cloudflare.com/route-leak-detection-with-cloudflare-radar/.

[24] B Du, G Akiwate, C Testart, A Snoeren, k claffy, K Izhikevich, and S Rao. IRRegularities in the Internet Routing Registry. In *IMC'23*.

[25] Ben Du, Gautam Akiwate, Thomas Krenc, Cecilia Testart, Alexander Marder, Bradley Huffaker, Alex C. Snoeren, and KC Claffy. IRR Hygiene in the RPKI Era. In *PAM*, 2022.

[26] Ben Du, Cecilia Testart, Romain Fontugne, Gautam Akiwate, Alex C. Snoeren, and kc claffy. Mind Your MANRS: Measuring the MANRS Ecosystem. In *IMC*, 2022.

[27] Romain Fontugne, Anant Shah, and Emile Aben. "The (Thin) Bridges of AS Connectivity: Measuring Dependency Using AS Hegemony". In *PAM*, 2018.

[28] Lixin Gao and Jennifer Rexford. Stable Internet Routing without Global Coordination. ACM SIGMETRICS, 2000.

[29] Dimitrios Panteleimon Giakatos, Sofia Kostoglou, Pavlos Sermpezis, and Athena Vakali. Benchmarking Graph Neural Networks for Internet Routing Data, 2022.

[30] Yossi Gilad, Avichai Cohen, Amir Herzberg, Michael Schapira, and Haya Shulman. Are We There Yet? On RPKI's Deployment and Security. In *NDSS*, 2017.

[31] Yossi Gilad, Sharon Goldberg, Kotikalapudi Sriram, Job Snijders, and Ben Maddison. The Use of maxLength in the Resource Public Key Infrastructure (RPKI). RFC 9319, October 2022.

[32] Yossi Gilad, Omar Sagga, and Sharon Goldberg. MaxLength Considered Harmful to the RPKI. In *CoNEXT*, 2017.

[33] Phillipa Gill, Michael Schapira, and Sharon Goldberg. A Survey of Interdomain Routing Policies. *ACM SIGCOMM CCR*, 2014.

[34] Vasileios Giotsas, George Nomikos, Vasileios Kotronis, Pavlos Sermpezis, Petros Gigis, Lefteris Manassakis, Christoph Dietzel, Stavros Konstantaras, and Xenofontas Dimitropoulos. O Peer, Where Art Thou? Uncovering Remote Peering Interconnections at IXPs. *IEEE/ACM ToN*, 2021.

[35] GRIP. https://grip.inetintel.cc.gatech.edu/.

[36] Aditya Grover and Jure Leskovec. node2vec: Scalable Feature Learning for Networks, 2016.

[37] Internet Health Report. https://ihr.iijlab.net/ihr/en-us.

[38] IODA. https://ioda.inetintel.cc.gatech.edu/.

[39] Internet Routing Registry. https://www.irr.net/.

[40] Christopher Kruegel, Darren Mutz, William Robertson, and Fredrik Valeur. Topology-Based Detection of Anomalous BGP Messages. In *Recent Advances in Intrusion Detection*, 2003.

[41] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian. Delayed internet routing convergence. *ACM SIGCOMM CCR*, 2000.

[42] Mohit Lad, Daniel Massey, Dan Pei, Yiguo Wu, Beichuan Zhang, and Lixia Zhang. PHAS: A Prefix Hijack Alert System. In *USENIX Sec*, 2006.

[43] Matt Lepinski and Stephen Kent. An Infrastructure to Support Secure Internet Routing. RFC 6480, February 2012.

[44] Matt Lepinski and Kotikalapudi Sriram. BGPsec Protocol Specification. RFC 8205.

[45] David Liben-Nowell and Jon Kleinberg. The link prediction problem for social networks. In *CIKM*, 2003.

[46] Mutually Agreed Norms for Routing Security (MANRS). https://www.manrs.org/about/.

[47] Not just another BGP Hijack. https://www.manrs.org/2020/04/not-just-another-bgp-hijack/.

[48] MANRS blogpost: BGP security in 2021. https://www.manrs.org/2022/02/bgp-security-in-2021/.

[49] Do we still need the IRR? An analysis and comparison of IRR data across databases. https://ripe85.ripe.net/wp-content/uploads/presentations/71-10-RIPE85-IRRAnalysis.pdf.

[50] Alexandros Milolidakis, Tobias Bühler, Kunyu Wang, Marco Chiesa, Laurent Vanbever, and Stefano Vissicchio. On the Effectiveness of BGP Hijackers That Evade Public Route Collectors. *IEEE Access*, 2023.

[51] University of Oregon. Route Views Project, 2016. www.routeviews.org/.

[52] The Interconnection Database. https://www.peeringdb.com/.

[53] Richard Steenbergen. Examining the validity of IRR data. https://archive.nanog.org/meetings/nanog44/presentations/Tuesday/RAS_irrdata_N44.pdf.

[54] Philipp Richter, Georgios Smaragdakis, Anja Feldmann, Nikolaos Chatzis, Jan Boettger, and Walter Willinger. Peering at Peerings: On the Role of IXP Route Servers. In *IMC'14*.

[55] RIPE RIS Raw Data, 2016. https://www.ripe.net/data-tools/stats/ris/.

[56] Pavlos Sermpezis, Vasileios Kotronis, Petros Gigis, Xenofontas Dimitropoulos, Danilo Cicalese, Alistair King, and Alberto Dainotti. ARTEMIS: Neutralizing BGP Hijacking Within a Minute. *IEEE/ACM ToN*, 2018.

[57] Tal Shapira and Yuval Shavitt. BGP2Vec: Unveiling the Latent Characteristics of Autonomous Systems. *IEEE TNSM*, 2022.

[58] Xingang Shi, Yang Xiang, Zhiliang Wang, Xia Yin, and Jianping Wu. Detecting Prefix Hijackings in the Internet with Argus. In *IMC'22*, 2012.

[59] Aftab Siddiqui. KlaySwap – Another BGP Hijack Targeting Crypto Wallets. https://www.manrs.org/2022/02/klayswap-another-bgp-hijack-targeting-crypto-wallets/.

[60] Kotikalapudi Sriram, Doug Montgomery, Danny R. McPherson, Eric Osterweil, and Brian Dickson. Problem Definition and Classification of BGP Route Leaks. RFC 7908, June 2016.

[61] Shen Su, Beichuan Zhang, Lin Ye, Hongli Zhang, and Nathan Yee. Towards real-time route leak events detection. In *IEEE ICC*, 2015.

[62] Yixin Sun, Anne Edmundson, Laurent Vanbever, Oscar Li, Jennifer Rexford, Mung Chiang, and Prateek Mittal. RAPTOR: Routing Attacks on Privacy in Tor. In *USENIX Sec*, 2015.

[63] Mattia Tantardini, Francesca Ieva, Lucia Tajoli, and Carlo Piccardi. Comparing methods for comparing networks. *Scientific Reports*, 2019.

[64] Cecilia Testart, Philipp Richter, Alistair King, Alberto Dainotti, and David Clark. Profiling BGP Serial Hijackers: Capturing Persistent Misbehavior in the Global Routing Table. In *IMC*, 2019.

[65] He Yan, Ricardo Oliveira, Kevin Burnett, Dave Matthews, Lixia Zhang, and Dan Massey. BGPmon: A Real-Time, Scalable, Extensible Monitoring System. In *Cybersecurity Applications Technology Conference for Homeland Security*, 2009.

[66] Zhen Yang, Ming Ding, Chang Zhou, Hongxia Yang, Jingren Zhou, and Jie Tang. Understanding Negative Sampling in Graph Representation Learning, 2020.

[67] Muhan Zhang and Yixin Chen. Weisfeiler-Lehman Neural Machine for Link Prediction. In *KDD '17*, 2017.

[68] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. In *Advances in Neural Information Processing Systems*, 2018.

[69] Xin Zhang, Hsu-Chun Hsiao, Geoffrey Hasker, Haowen Chan, Adrian Perrig, and David G. Andersen. SCION: Scalability, Control, and Isolation on Next-Generation Networks. In *S&P*, 2011.

[70] Zheng Zhang, Ying Zhang, Y. Charlie Hu, Z. Morley Mao, and Randy Bush. ISPY: Detecting IP Prefix Hijacking on My Own. *IEEE/ACM Trans. Netw.*, 2010.

[71] Changxi Zheng, Lusheng Ji, Dan Pei, Jia Wang, and Paul Francis. A Light-Weight Distributed Scheme for Detecting IP Prefix Hijacks in Real-Time. In *ACM SIGCOMM*, 2007.

Appendix

A A sample of the most suspicious cases

We now describe three suspicious cases that DFOH detected in 2022. These cases illustrate that DFOH is exploitable, and that reporting the suspicious cases and notifying the operators is beneficial. Note that we did not manually inspect all the reported cases in 2022. Thus, even more suspicious cases might exist.

January 1, 2022:[3] AS267548, a small Peruvian AS, appears between Sprint (AS1239), a Tier1 AS, and AS199524, a large content provider. However, AS267548 is not supposed to

[3] https://dfoh.uclouvain.be/cases/2022-01-01_1239_267548

provide transit between these two ASes. This is suspicious as AS267548 is not connected to any IXP and not present in any facility according to PeeringDB. Besides, it is only connected to Brazilian ASes according to bgp.tools.

January 30, 2022:[4] This case involves AS138263, a small India AS, which according to an observed AS path provided transit between AS32934 (Meta) and AS1828 (a large global software-defined network) for Meta's prefix `31.13.79.0/24`. This case is suspicious for two reasons. First, the hypotetical victim (Meta) does not share any IXP with the hypothetical attacker (AS32934). Second, its unlikely that a small AS provides transit for two large ASes such as Meta and AS1828.

April 1, 2022:[5] New BGP routes for prefix `14.0.48.0/24`, owned by AS54994, have been observed from ten BGP vantage points. In their AS path, we observe that the (valid) origin AS54994, a CDN based in the US, appears to peer with AS132116, a rather small Indian ISP that only peers with Indian networks. Even though the two ASes are both connected to DE-CIX in Mumbai, these routes are a solid sign of either AS-path manipulation or route leak.

B Datasets

In this section, we give more details on which datasets DFOH downloads, cleans, and combines to build the main data structures that it uses for its inferences. We make publicly available the parsed data that DFOH uses as input for its inferences at the following URL: `https://dfoh.uclouvain.be/`.

Collecting and cleaning up the public BGP data. We use 287 BGP vantage points from RIS [55] and RouteViews [51] to collect BGP updates. These vantage points are selected using MVP [6] to maximize the utility of the collected routes, which is useful as our Ubuntu 20.04 server with 16 CPUs and 64GB of RAM does not have enough resources to collect BGP routes from all the $\approx$ 2500 vantage points in a timely manner. Observe that collecting BGP routes from all the vantage points would improve the accuracy of DFOH. We also use the AS paths that CAIDA uses to update its AS relationships dataset every month and makes publicly available [3]. DFOH applies the following actions to clean up the BGP routes before using them for inferences. First, it removes the redundant and identical AS numbers that appear consecutively due to AS path prepending. Second, it removes private AS numbers. Finally, it removes the AS numbers owned by an IXP. We obtain (and update on a daily basis) the list of IXPs from PeeringDB.

Building the AS topology. DFOH builds the AS topology (the undirected graph $G_{d,k}$ presented in §6) by combining three datasets. First, it builds the graph from AS links observed in the AS paths of all the BGP updates collected from the 287 vantage points during the ten months ($k = 300$ as in ARTEMIS [56]) preceding day k. DFOH takes the data from previous days to include the past and transient links in the AS topology, which prevents incorrectly classifying them as new AS link. Then, it also takes the AS paths observed in the RIB of every one of these 287 vantage points on the first day of the month at which the graph is built. Finally, it also takes the AS paths that CAIDA uses to update its AS relationships dataset [3]. For instance, the AS topology $G_{d,k}$ with $d =$April 15, 2023, and $k = 30$ (instead of 300 to simplify) is the combination of the AS paths observed between March 15, 2023 and April 14, 2023 in the BGP updates, the AS paths observed in the RIBs on April 1, 2023, and the AS paths used by CAIDA to build their AS relationship dataset on April 1, 2023.

Finding new AS links. To find the new AS links for the day d, DFOH collects the BGP updates from the 287 BGP vantage points observed at day d, parses the AS paths, and checks whether every AS link in the AS paths is in the AS topology $G_{d,k}$. AS links not in $G_{d,k}$ are classified as new.

[4] https://dfoh.uclouvain.be/cases/2022-01-30_32934_138263

[5] https://dfoh.uclouvain.be/cases/2022-02-24_54994_132116

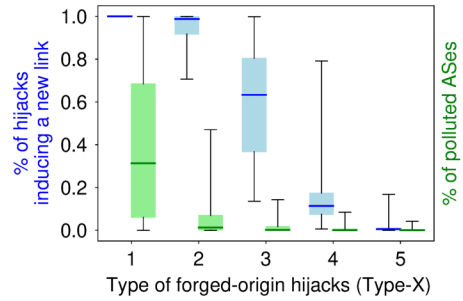


Figure 8: There is a tradeoff between hijack *visibility* (blue boxplots) and *impact* (green boxplots). Most of the impactful forged-origin hijacks are visible to DFOH. A polluted AS is an AS that prefers the hijacked route over the legitimate one.

C Tradeoff between visibility and impact

Aside from prepending the victim AS number in the AS path of the hijacked route, it is hard for an attacker to further manipulate the AS path to thwart DFOH (e.g., scenarios 3 and 4 in §6.1.2). In fact, prepending more ASes increases the length of the AS path, inducing a trade-off between *visibility* and *impact* of the attack, which we highlight in Fig. 8. This figure shows, for the different Type-X hijacks with X ranging from 1 to 5, the proportion of forged-origin hijacks that induce a new AS link (*visibility*) and the number of polluted ASes, i.e., that use the hijacked route (*impact*). Observe that computing the *visibility* of a Type-X hijack boils down to computing the proportion of ASes that are reachable from the attacker's AS in X hops or less, whereas computing the *impact* boils down to measuring for how many ASes the path to the attacker AS is shorter than the path to the victim AS. We compute *visibility* and *impact* for 5000 attacker ASes randomly selected, and for each, we take a random sample of 2000 victims. The results

Type	Categorie	Name	Index	Description
Node-based	**Centrality Metrics**	Degree centrality	0	Fraction of nodes connected to v
		Closeness centrality	1	Average length of the shortest path between v and all other nodes
		Harmonic centrality	2	Sum of the reciprocal of the shortest path distances from all nodes to v
	Neighborhood Richness	Average neighbor degree	3	Average degree of all the neighbors of v
		Eccentricity	4	Max distance from v to all other nodes
	Topological Pattern	Number of Triangles	5	Number of triangles that include v
		Clustering	6	Fraction of possible triangles including v that exist
Pair-based	**Closeness Metrics**	Jaccard	7	Similarity between the neighbors of v_1 and v_2
		Adamic Adar	8	Closeness of v_1 and v_2 based on their shared neighbors
		Preferential attachment	9	Likelihood of v_1 and v_2 to be connected based on their degree
	Distance	Shortest Path	10	Length of the shortest path between v_1 and v_2

Table 2: List of topological features used by DFOH along with their description. In the description, we consider for the node-based features a node v in the AS topology whereas we consider two nodes v_1 and v_2 for the pair-based features.

are aggregated in the box plots. Clearly, there is no sweet spot where an attack has high *impact* and low *visibility*. For Type-1 and 2, *impact* is high but *visibility* is high too, and vice versa for Type-3, 4 and 5. An attacker launching a forged-origin hijack thus often cannot prevent the AS path of the hijacked route to include a new AS link—giving DFOH the ability to detect it.

D Features computation (extension)

D.1 Topological features

Table 2 describes the ten topological features that DFOH uses in its inference model. We explain how DFOH computes the feature values in §6.2.

D.2 Bidirectionality feature

Observing an AS link in each direction is a strong sign that it is legitimate. In fact, consider the forged-origin-hijacked route with the AS path $x_1,\ldots,x_n,v_1,v_2$ where v_1 is the attacker and v_2 the forged origin. v_1 can only forge the upstream part of the AS path (i.e., the part on the right side of v_1), and has no control over the downstream part. Note that v_1 could prepend v_2,v_1 on a route to another prefix that it owns to fake a bidirectional link. However, the AS path would contain a loop and would be either denied by BGP routers or easily detectable by BGP monitoring systems. A challenge when assessing the bidirectionality of the AS links is that the AS topology derived from the AS paths in the BGP routes is incomplete (e.g., backup links can be missing). Thus, only a small fraction of the links (≈25 000 , i.e., ≈4.8% of the visible links) are visible in both directions.

Using the IRR data to supplement the BGP routes. DFOH parses the IRR data to infer more peering links that are not visible from the collected BGP routes. More precisely, DFOH parses the `aut-num` objects of every AS in the routing registries. For now, DFOH only uses RADb as it is the only one that makes available archive of its database. However, we envision to use all the registries listed in [39] for real-time detection. An `aut-num` object related to *ASX* may include (partial) information about the export and import policies of *ASX*. These policies generally indicate the AS number or an `as-set` objects to/from which *ASX* is exporting/importing routes. In the case of an `as-set` object, DFOH recursively parses the object (an `as-set` object can include other `as-set` objects) until it finds all the ASes in this `as-set`. With the IRR data, DFOH infers peering information that when combined with BGP data, allows identifying ≈10 000 more bidirectional links compared to with BGP data only.

The bidirectionality feature is beneficial. Even after parsing the IRR data, the number of bidirectional links remains small compared to the total number of AS links. Yet, they are worth the effort because they help DFOH to correctly classify new links as legitimate in some particular peering scenarios. In fact, as the location of BGP collectors is typically skewed with many of them located in the core of the Internet [14], many bidirectional links pertain to highly-connected ASes. We observe the same effect on the bidirectional links inferred from the IRR data, as network operators of the highly-connected ASes tend to populate their IRR data more frequently than others. The bidirectional feature thus improves the accuracy of DFOH upon peering scenarios that involve the highly-connected ASes—scenarios that are hard to accurately classify with the other features (see §E.1).

The bidirectionality feature is safe. The IRR data has two drawbacks: It can be *inaccurate* and it is *unverified*. This is not an issue for DFOH for two reasons. First, the number of possible attacker and victim pairs is $C^2_{74000} \approx 2.74$ billions (74 000 is the number of ASes [22]) whereas the number of inconsistencies in the IRR is by far lower [25]. Consequently, the bidirectional links incorrectly inferred because of these in-

	w/o AS-Path-based		w/o Bidirectionality		w/o Topological		w/o PeeringDB		All Features	
	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR
Stubs - Stubs	85.0 %	4.2 %	97.0 %	4.2 %	95.0 %	4.2 %	87.0 %	2.8 %	**97.0 %**	**4.2 %**
Stubs - Tier1	74.0 %	3.1 %	82.0 %	4.1 %	83.0 %	3.1 %	76.0 %	6.2 %	**83.0 %**	**3.1 %**
Transit/IXP/CDN-2 - Transit/IXP/CDN-3	82.0 %	1.0 %	95.0 %	1.0 %	94.0 %	1.0 %	92.0 %	2.0 %	**95.0 %**	**1.0 %**
Transit/IXP/CDN-3 - Highly Connected	96.2 %	1.0 %	98.1 %	1.0 %	98.1 %	2.0 %	98.1 %	1.0 %	**98.1 %**	**1.0 %**
Transit/IXP/CDN-4 - Tier1	58.0 %	15.6 %	83.0 %	6.2 %	83.0 %	12.5 %	82.0 %	12.5 %	**84.0 %**	**6.2 %**
Large Customer Cone - Tier1	35.0 %	13.3 %	89.0 %	13.3 %	88.0 %	6.7 %	94.0 %	13.3 %	**89.0 %**	**6.7 %**
All types of Links	74.0 %	2.1 %	90.9 %	2.0 %	90.6 %	2.3 %	86.2 %	2.9 %	**90.9 %**	**1.9 %**

Table 3: Accuracy of DFOH for a few selected attack scenarios when all the features but one are used in the inference pipeline.

consistencies only have very little impact on the performance of DFOH. Second, as an attacker can only change the IRR data for its own organization, she can only fake one direction of an AS link (attacker $\rightarrow$ victim)—the same direction as when prepending the victim's AS number in a BGP announcement.

Computing the feature values. Upon a historical request for a day d, DFOH infers link bidirectionality from the AS topology computed on day d (i.e., $G_{d,k}$) combined with the BGP and IRR data collected during the days *following* d (up to 30 days). Considering the following days allows DFOH to find more bidirectional links that only appear e.g., once BGP converges. Upon a real-time query on day d, DFOH considers the graph $G_{d,k}$, and the IRR data collected on day d. Observe that for the bidirectionality feature, $G_{d,k}$ is a *directed* graph. Computing the bidirectionality feature results in the following 1-dimensional feature vector.

$$B(d, v_1, v_2) = [bidir(d, v_1, v_2)]$$

Where $bidir(d, v_1, v_2) = 1$ if the link (v_1, v_2) is bidirectional at day d, else it is equal to zero.

E Detection speed

DFOH automatically downloads all the data and trains the inference model on a daily basis. Upon launching an inference, DFOH uses the inference model trained the day before. Thus, the detection speed depends on *(i)* the time to compute the feature values for the link and AS paths given as input, and *(ii)* the time to run the inference. The inference is fast (<1s) because it relies on a simple random forest. However, the time to compute the feature values depends on whether DFOH is used for real-time detection or to detect past forged-origin hijacks. We now differentiate the two cases.

DFOH detects past forged-origin hijacks in a few seconds. We measured the time needed by DFOH to compute the feature values for all the 18 000 synthetic cases used to evaluate the overall performance of DFOH in 7.1. We use an Ubuntu 20.04 LTS version server with 16 cores and 64 GB of memory. DFOH needs 7510 seconds to compute the feature values (i.e., $\leq 1s$ for a single case). The topological features are the most time consuming to compute (7155 seconds). Observe that the *AS-path-pattern* features are fast to compute, thus DFOH remains fast even when it must compute these features for many different AS paths.

DFOH detects new forged-origin hijacks in a few minutes. The only difference when running DFOH in real time pertains to the bidirectionality feature. In fact, a new peering interconnection may be visible in both directions from the BGP routes only when BGP has converged. As the BGP convergence typically takes a few minutes [41], DFOH waits a few minutes (five, identically to ARTEMIS [56]) to let BGP converge before computing the bidirectionality feature.

E.1 Discriminate power of classification features

In this section, we examine the discriminatory power of the classification features used in DFOH. We show that our selection is sound: Every feature is useful in at least one attack scenario. Besides, none of the features alone is able to detect forged-origin hijack consistently and for all attack scenarios.

Every feature category is useful. Table 3 shows the performance of DFOH when one feature category is deactivated

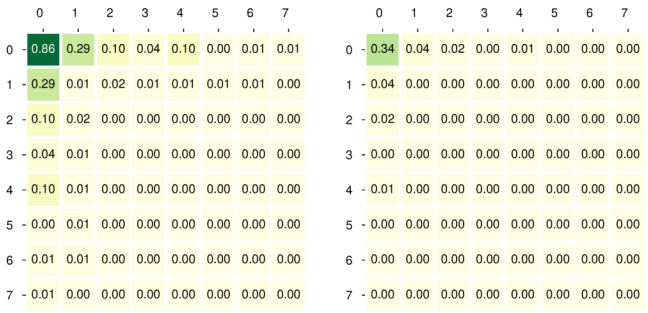
	AS-Path-based		Bidirectionality		Topological		PeeringDB		All Features	
	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR
Stubs - Stubs	83.0 %	2.8 %	0.0 %	0.0 %	79.0 %	5.6 %	84.0 %	4.2 %	**97.0 %**	**4.2 %**
Stubs - Tier1	70.0 %	10.3 %	0.0 %	0.0 %	46.0 %	9.3 %	65.0 %	1.0 %	**83.0 %**	**3.1 %**
Transit/IXP/CDN-2 - Transit/IXP/CDN-3	88.0 %	2.0 %	0.0 %	0.0 %	56.0 %	0.0 %	61.0 %	0.0 %	**95.0 %**	**1.0 %**
Transit/IXP/CDN-3 - Highly Connected	98.1 %	1.0 %	0.0 %	0.0 %	94.2 %	1.0 %	88.5 %	0.0 %	**98.1 %**	**1.0 %**
Transit/IXP/CDN-4 - Tier1	65.0 %	15.6 %	0.0 %	0.0 %	63.0 %	31.2 %	37.0 %	15.6 %	**84.0 %**	**6.2 %**
Large Customer Cone - Tier1	81.0 %	13.3 %	0.0 %	0.0 %	52.0 %	20.0 %	18.0 %	0.0 %	**89.0 %**	**6.7 %**
All types of Links	79.7 %	2.8 %	0.0 %	0.0 %	56.0 %	3.8 %	65.4 %	2.4 %	**90.9 %**	**1.9 %**

Table 4: Accuracy of DFOH for a few selected attack scenarios when only one feature is used in the inference pipeline.

and for a few attack scenarios. We can see that every feature matters: Removing one feature category from the inference pipeline significantly reduces the accuracy of DFOH for at least one attack scenario. The *AS-Path-based* features appear to be the most useful: They are useful in every scenario. The *peeringDB* features also significantly improve the accuracy in many scenarios. For instance, in the "Stubs - Tier1" scenario, they divide the FPR by two. The *topological* features reduces the FPR in the scenarios "Transit/IXP/CDN-3 - Highly connected" and "Transit/IXP/CDN-4 - Tier1" by a factor of two. Finally, the bidirectionality feature is mainly useful to reduce the FPR upon scenarios that involve highly-connected ASes. For instance, it reduces the FPR from 13.3% to 6.7% in the "Large Customer Cone - Tier1" scenario.

DFOH remains accurate when one feature category is missing. Unsurprisingly, removing one feature category from the inference pipeline reduces the accuracy. Yet, the accuracy of DFOH never appears particularly flawed when one feature category is missing. In fact, the minimum FPR when one feature is missing across all the possible scenarios is 15.8%, and the TPR is only below 50% for four out of the $36 * 4 = 144$ scenarios (36 is the number of attack scenarios and 4 is the number of feature categories). Consequently, these feature categories compensate each other, allowing DFOH to remain accurate when there is missing or polluted data.

None of the features alone is enough. Table 4 shows that none of the feature is able to accurately detect forged-origin hijacks when used individually. The *AS-path-pattern* feature returns the best accuracy when used individually, yet performs rather poorly in some cases. For instance, using this feature only returns 15.6% of FPR in the scenario that involves a



(a) TPR. (b) FPR.

Figure 9: Performance of SEAL on every attack scenario when trained on a randomly generated sample of links.

"Transit/IXP/CDN-4" and a "Tier1" AS. Observe that the *bidirectionality* feature always returns 0% of TPR and FPR. In fact, when used individually in the DFOH inference pipeline, the resulting classifier simply classifies all the links under examination as legitimate. This is the consequence of *(i)* the bidirectionality feature being helpful to assess the legitimacy of a link, but not its maliciousness, and *(ii)* the tiny fraction of AS links observed in both directions (only 5-10% of all the links are bidirectional, see §6.2.4).

E.2 Performance of SEAL for revealing forged-origin hijacks

In this section, we provide more details about the performance of SEAL [68] to detect fake AS links. Fig. 9 shows the performance (in terms of TPR and FPR) of SEAL for every attack

(a) TPR. (b) FPR.

Figure 10: Performance of *b*-SEAL on every attack scenario when trained on a sample of links generated with the balanced sampling scheme used by DFOH.

scenario when trained on a *randomly* generated set of existent and nonexistent links (the default implementation of SEAL). The number at the i-th line and j-th column indicates the TPR or FPR for the cases that involve an AS in cluster i, and an AS in cluster j. The grids are symmetric, and the greener a cell higher the rate. Clearly, the accuracy of SEAL is skewed: It is only capable of detecting fake links that involve two "Stub" ASes (the TPR is 0.86 for this attack scenario). For the majority of the other attack scenarios, SEAL classifies the links as legitimate, resulting in a low TPR and FPR.

Fig. 10 shows the performance of *b*-SEAL, a modified version of SEAL where it is trained on a set of existent and nonexistent links generated using our *balanced* sampling scheme (see §6.3.3). The performance of *b*-SEAL is more balanced: The minimum TPR is 0.55, which illustrate that our balanced sampling is useful. Yet, the FPR is also high for many of the attack scenarios, which illustrates that *b*-SEAL does not translate well to revealing fake AS links.

NetVigil: Robust and Low-Cost Anomaly Detection for East-West Data Center Security

Kevin Hsieh[*1] Mike Wong[*2,1] Santiago Segarra[1,3] Sathiya Kumaran Mani[1]
Trevor Eberl[1] Anatoliy Panasyuk[1] Ravi Netravali[2] Ranveer Chandra[1] Srikanth Kandula[1]
[1]*Microsoft* [2]*Princeton University* [3]*Rice University*

Abstract– The growing number of breaches in data centers underscores an urgent need for more effective security. Traditional perimeter defense measures and static zero-trust approaches are unable to address the unique challenges that arise from the scale, complexity, and evolving nature of today's data center networks. To tackle these issues, we introduce NetVigil, a robust and cost-efficient anomaly detection system specifically designed for east-west traffic within data center networks. NetVigil adeptly extracts security-focused, graph-based features from network flow logs and employs domain-specific graph neural networks (GNNs) and contrastive learning techniques to strengthen its resilience against normal traffic variations and adversarial evasion strategies. Our evaluation, over various attack scenarios and traces from real-world production clusters, shows that NetVigil delivers significant improvements in accuracy, cost, and detection latency compared to state-of-the-art anomaly detection systems, providing a practical, supplementary security mechanism to protect the east-west traffic within data center networks.

1 Introduction

The modern era of digitalization has brought about unprecedented growth in data center networks, ushering in a period where the need for robust security measures is more critical than ever before. Recent high-profile breaches, such as the Equifax breach [1] and the SolarWinds attack [3, 82] have exposed vulnerabilities in data center network security and demonstrate catastrophic consequences that can arise from a lack of adequate protection. As cyber threats continue to evolve and grow in sophistication, there is an urgent need to develop innovative solutions to safeguard sensitive information and ensure the integrity of data center networks.

One such area that warrants particular attention is the security of east-west traffic within data centers. Traditional security measures have focused primarily on securing north-south traffic at the network perimeter, which has left internal systems susceptible to lateral movements and persistent threats from compromised nodes, SSH keys, or other credentials [73, 75]. Zero-trust architecture [44, 66] aims to mitigate these risks by securing east-west communication and data flows. However, current zero-trust solutions, such as micro-segmentation [40, 59, 70, 77], rely only on static access control rules and are ill-equipped to detect dynamic and unusual behaviors, leaving networks exposed to potential attacks and lateral movement.

Addressing the critical gap in data center network security necessitates the development of an effective and always-on network anomaly detector specifically tailored for east-west traffic. Although existing network anomaly detection solutions have achieved significant progress in handling north-south traffic [32, 47], they struggle to overcome the unique challenges associated with east-west traffic. First, the majority of these solutions require capturing and analyzing network packets, leading to a *cost* that becomes prohibitively high when implemented across all nodes. For example, a recent high-throughput malicious traffic detection system [31] demands at least 17 cores and 10 GB of memory per node to secure a 10 Gb NIC, resulting in an annual cost of six million dollars for an application comprising 1,000 nodes. Second, existing solutions are known for generating *false alarms* [43, 57], an issue further amplified by the dynamic nature of east-west traffic. Even a marginal false alarm rate can significantly escalate operational overhead at scale, causing security teams to inadvertently neglect genuine threats. Third, a multitude of solutions depend on labeled malicious datasets or signatures for training their detectors [22, 46, 51, 53], a strategy that is not only impractical at scale but also renders systems susceptible to novel zero-day attacks [23, 55].

Objectives and Techniques. We introduce NetVigil, a novel anomaly detection system explicitly designed for securing east-west traffic in large-scale networks. In light of the challenges and limitations associated with existing solutions, NetVigil is designed to fulfill three primary objectives: (a) guaranteeing cost-effectiveness when monitoring numerous nodes, (b) precisely identifying anomalous behaviors while emphasizing the reduction of false alarms, and (c) demonstrating robustness to normal traffic changes without depending on prior knowledge of malicious attacks.

NetVigil achieves these objectives with three core ideas:

(1) *Deriving security-focused graph features from flow summaries.* To ensure cost-efficiency, NetVigil leverages low-cost flow summaries, available at both network level (e.g., VPC Flow Logs [21] and NSG Flow Logs [54]) and service level

* Equal contribution.

(e.g., Calico flow logs [6] and Cilium/Hubble [9]). These loggers offer substantial cost savings over packet traces by logging only aggregated statistics. NetVigil adeptly extracts security-oriented graph features from these summaries, effectively compensating for the absence of packet-level information. This approach enables scalable monitoring of large networks without sacrificing anomaly detection accuracy.

(2) *Leveraging graph neural networks (GNNs) and domain-specialized contrastive learning for context-aware robust anomaly detection.* NetVigil employs GNNs to model complex relationships between nodes in the network. The key insight is that different nodes within a network (e.g., microservices) carry diverse contextual information. By integrating contextual information from adjacent nodes, GNNs can detect anomalous behaviors that might be overlooked by traditional solutions focusing on individual flows [31, 56]. To further strengthen the model's resilience against normal traffic fluctuations, NetVigil adopts graph contrastive learning [85] with domain-specialized data augmentation. This approach guides the model toward capturing meaningful representations of standard traffic patterns, enabling it to distinguish between benign and malicious behaviors with greater precision.

(3) *Adapting to temporal dynamics via smoothing and continuous retraining.* NetVigil addresses the evolving nature of network traffic by integrating a temporal loss that encourages similarity between embeddings of temporally-adjacent graphs. Moreover, NetVigil continuously retrains its model using recent clean logs by excluding anomalous flows. This approach keeps the model updated with network behavior, maintaining high detection accuracy over time.

Implementation and Evaluation. We build NetVigil as an end-to-end streaming data pipeline, continuously analyzing network flow summaries and dynamically updating its model. To evaluate its effectiveness, we design a new east-west security benchmark, Yatesbury, using a microservice demo application [11], generating a diverse array of live traces and simulating evasive attack scenarios. Our extensive evaluation, including the benchmark and week-long to month-long traces from production clusters, demonstrates that NetVigil significantly outperforms existing malicious traffic detectors [31,56]. We achieve an average AUC (area under the ROC Curve) improvement of 0.22 (up to 0.62) and reduce operational costs by 2.7 – 16.7× for our 16-VM deployment. We release our benchmark Yatesbury [1] to enable researchers to explore novel attack scenarios in a cloud environment and contribute innovative solutions in this crucial domain.

Contributions. We make the following contributions:

- We introduce a novel network anomaly detection architecture designed to secure east-west traffic within data centers. This architecture utilizes low-cost network flow logs, security-oriented graph features, and graph neural networks (GNNs) to achieve cost-effectiveness and robustness.

[1] https://github.com/microsoft/Yatesbury

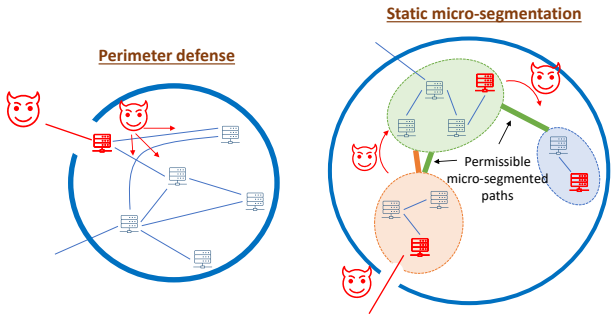


Figure 1: **Difference between perimeter defense and static micro-segmentation**

- We propose an innovative end-to-end training mechanism that combines graph representation learning, graph contrastive learning, and temporal smoothing. This approach enables the learning of east-west traffic dynamics for accurate anomaly detection.
- We build our solution alongside an east-west security benchmark tailored for cloud deployment, validating the performance of our proposed solution through various attack scenarios and long-term traces from two production clusters.

2 Background and Motivation

We briefly discuss key concepts and challenges in securing east-west traffic in data center networks. We focus on the importance of this task and the limitations of traditional security measures, and we provide an overview of network anomaly detection techniques specific to east-west traffic.

2.1 Securing East-West Traffic with Zero-Trust Solutions

Traditional security measures, such as firewalls [69] and intrusion detection systems (IDSes) [43], have primarily focused on securing north-south traffic (interactions between data center nodes and external systems) at the network perimeter, as illustrated in Figure 1. While these measures are essential, they often do not protect east-west traffic (communication between nodes within a data center). Traditional security measures tend to rely on static access control rules or attack signatures, which can be easily bypassed by attackers using compromised credentials or exploiting network vulnerabilities. Once inside the network, an attacker can perform lateral movement with relative ease. The limited visibility into the communication between nodes within the data center renders traditional security measures less effective in detecting and responding to emerging threats.

The zero-trust security model [44, 66] represents a paradigm shift in network security, aiming to address the limitations of traditional security measures and improve the protection of east-west traffic within data center networks.

IDS	Low compute overhead	Low network overhead	Easy management	Zero-day attack protection	Computation cost (CPU cores/100Gbps) or Licensing cost (US dollars)
Zeek (formerly Bro) [61]	✗	✗	✓	✗	400 [2]
Snort [65]	✗	✗	✗	✗	250 [88]
Pigasus [88]	✓	✗	✗	✗	5 + 1 FPGA [88]
Suricata [15]	✗	✗	✗	✗	53 + 1 SmartNIC [14]
Whisper [31]	✗	✓	✓	✓	170
VMware NSX [17]	✗	✓	✓	✗	$4,495/processor/yr [18]
Aviatrix DCF [5]	✗	✗	✓	✗ [4]	~$3,000/gateway/yr

Table 1: **Characteristics of popular IDSes. The characteristics of commercial options are obtained from their websites, which are subject to changes. The licensing costs associated with commercial solutions may not directly correspond to their computation costs.**

The main principles of zero-trust include: (a) least privilege access [68], where users and devices are granted the minimum access rights necessary to perform their tasks; (b) micro-segmentation [70, 77], which divides the network into smaller segments or zones to limit lateral movement (see Figure 1); and (c) continuous monitoring and validation of user and device behavior to ensure compliance with security policies.

Existing solutions [40, 59] primarily focus on micro-segmentation to limit the attack surface and the potential impact of a breach while preventing lateral movement of attackers. However, continuous monitoring and validation of user and device behavior remain a challenge due to the massive scale and dynamic nature of east-west traffic, requiring advanced techniques and tools to effectively detect and respond to anomalies in real time. Without this missing piece, an attack can still find a way to propagate from one node to another through permissible, micro-segmented paths, causing significant damage to critical infrastructure, data loss, or unauthorized access to sensitive information.

2.2 Challenges of Network Intrusion Detection Systems on East-West Traffic

A significant body of research [43, 48, 53] has been dedicated to the study of network intrusion detection systems (IDSes), which are tactically positioned at network choke points (e.g., routers or gateways) to safeguard network perimeters. Contemporary IDSes can be classified into signature-based and anomaly-based systems, each offering unique advantages. Signature-based systems efficiently detect known attack patterns using their distinct signatures, while anomaly-based systems focus on recognizing normal patterns to identify novel attacks. Although signature-based systems are effective for known attacks, the rise of zero-day attacks [23, 55] has highlighted the importance of anomaly-based systems. By employing deep learning techniques [32, 47], anomaly-based systems serve as a valuable complement to signature-based systems. Both systems have their merits, and a combination of the two can offer a more robust security solution. Table 1 summarizes key attributes of popular IDSes. Although these solutions work well for north-south traffic, their application to east-west traffic presents fundamental challenges.

Challenge 1: Excessive Compute Overhead in Network Packet Analysis. Past studies [16, 67] showed that only 17% of data center traffic was attributed to north-south traffic. The increasing adoption of micro-service architectures, cloud storage, and software-defined networks has since exacerbated the disparity between north-south and east-west traffic. Consequently, applying existing security solutions to east-west traffic would result in an unsustainable increase of already considerable costs. For example, Pigasus [88] employs Field-Programmable Gate Arrays (FPGAs) to substantially reduce operational expenses for rule-based IDSes. Nevertheless, it still requires five CPU cores and one FPGA to secure a 100 Gbps network. Commercial solutions like VMware NSX [17] tackle this problem by operating IDSes on every hypervisor. Although this method handles east-west traffic better, it results in substantial computational expenses for *all* data center nodes. The situation is no better for anomaly-based IDSes. Whisper [31], an efficient malicious traffic detection system, surpasses its predecessor Kitsune [56] by achieving a 100-fold increase in throughput. Despite this, securing a single 10 Gbps network necessitates 17 processing cores, potentially doubling or even tripling the application's operational costs.

The primary reason for the high overhead in existing solutions is their dependence on capturing and analyzing network packets, which offer rich and fine-grained information on network traffic. Nevertheless, as Table 1 shows, this approach is expensive and not scalable for large data center networks. A cost-effective alternative involves using network or service flow logs [6, 9, 21, 54], which utilize aggregation intervals to condense network packet telemetry into periodic 5-tuples and statistics. While service flow logs offer valuable application-level data, such as service names and request types, they often necessitate an agent in guest VMs or specialized environments like Kubernetes [24]. In this study, we focus on network flow logs due to their widespread availability in a cloud environment and consider integrating service flow logs in future work.

To evaluate this approach, we compare the effectiveness of two contemporary solutions, Whisper [31] and Kitsune [56], on network packet traces and network flow logs for similar set of attacks. As Figures 2 and 3 illustrate, applying existing solutions to network flow logs significantly compromises their accuracy due to the absence of packet-level information.

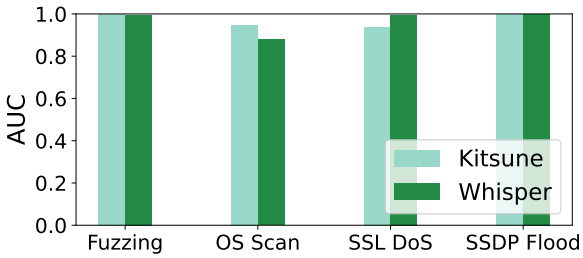


Figure 2: **AUC scores with packet traces**

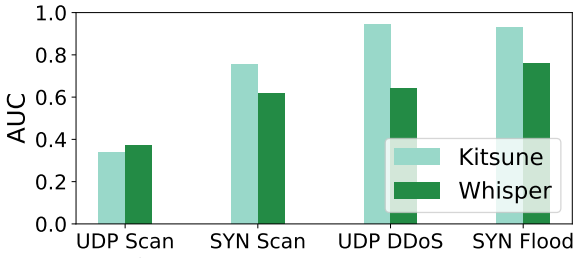


Figure 3: **AUC scores with flow traces**

Challenge 2: Complexity and Congestion in Networking. Numerous popular IDSes [15, 65, 88] are implemented by redirecting (or hair-pinning) traffic to IDS appliances. However, for east-west traffic, this method generates excessive network overheads, as traffic between VMs on the same machine or within the same rack must be rerouted to another cluster, potentially causing significant congestion in the network. Moreover, this approach adds complexity to network management, as hair-pinning must be executed for all communication paths within a data center, making routing substantially more complicated. Thus, commercial offerings [5] that adopt this approach primarily focus on egress traffic rather than encompassing all east-west traffic.

Challenge 3: Elevated False Alarms During Prolonged Deployment. Applying existing solutions to secure east-west traffic presents another challenge: an increased false alarm rate over extended deployment periods. As network traffic patterns evolve, security solutions must adapt to these changes to maintain high detection accuracy. However, many current network security systems struggle to keep up with the dynamic nature of network traffic, such as load variation and workload migration. This results in a high rate of false alarms.

We conduct an evaluation of Whisper [31] using two long-term traces from our first-party production clusters (see Section 6.5 for more detailed information). We find that there is a substantial increase in false alarms over time, while the continuous daily retraining of these solutions yields only marginal improvements (not shown in plot). These false alarms not only utilize significant resources for investigation but also undermine trust in the system, potentially resulting in the disregard of authentic threats.

Summary. The absence of a cost-efficient, robust and effective intrusion detection solution is a crucial impediment in securing east-west traffic. A comprehensive solution must: (a) ensure processing efficiency by eliminating dependence on all network packets, (b) avoid incurring network congestion or routing complexity, (c) proactively adapt to normal traffic fluctuations and (d) demonstrate high efficacy on previously unknown but malicious occurrences. These considerations are the foundation of our design requirements for NetVigil.

3 Overview of NetVigil

We present a novel anomaly-based intrusion detection system, NetVigil, that is explicitly designed to secure east-west data center networks with cost efficiency and robustness against normal traffic fluctuations. NetVigil achieves low operational costs by extracting security-oriented graph features from network flow logs, effectively eliminating the need for fine-grained yet costly network packet traces. The insight of NetVigil lies in the fact that network nodes within a data center typically provide specific functionalities (e.g., microservices, storage, databases, etc.), and by employing graph neural networks (GNNs), we can learn contextual information to enhance anomaly detection accuracy. Furthermore, our system incorporates graph contrastive learning and temporal smoothing techniques to achieve high detection accuracy while maintaining low false alarm rates. Figure 4 provides an overview of the NetVigil architecture.

Inference Time. During the inference phase (depicted on the left side of Figure 4), cloud resources such as virtual machines (VMs) and compute clusters continuously generate network flow logs [6, 9, 21, 54] at intervals ranging from tens of seconds to minutes. These data streams are processed by our *security graph feature extractor* ($\mathbf{I}_1$ in Figure 4, more details in Section 4.1), which groups network flow logs based on their IP addresses, extracts crucial features, and transforms the results into a featurized communication graph. In this graph, each node represents an IP address, and each edge summarizes all flows between respective IP pairs. NetVigil subsequently feeds this featurized communication graph into the continuously trained *GNN autoencoder* ($\mathbf{I}_2$ in Figure 4) to compute anomaly scores for each edge. Edges identified as potentially anomalous, along with the corresponding communication graphs and network flow logs, are then forwarded to the security team for further investigation.

Training Time. At each retraining interval, which typically spans hours or days based on network dynamics, NetVigil gathers clean communication graphs from the inference phase (i.e., excluding anomalous nodes and edges detected by the model) to retrain the GNN autoencoder (depicted on the right side of Figure 4). The *GNN encoder* ($\mathbf{T}_1$ in Figure 4) learns to compress the features of each edge and its incident nodes into an embedding space, allowing the *GNN decoder* ($\mathbf{T}_2$ in Figure 4) to reconstruct these features with minimal reconstruction loss ($\mathbf{L}_1$ in Figure 4). Section 4.2 provides more details.

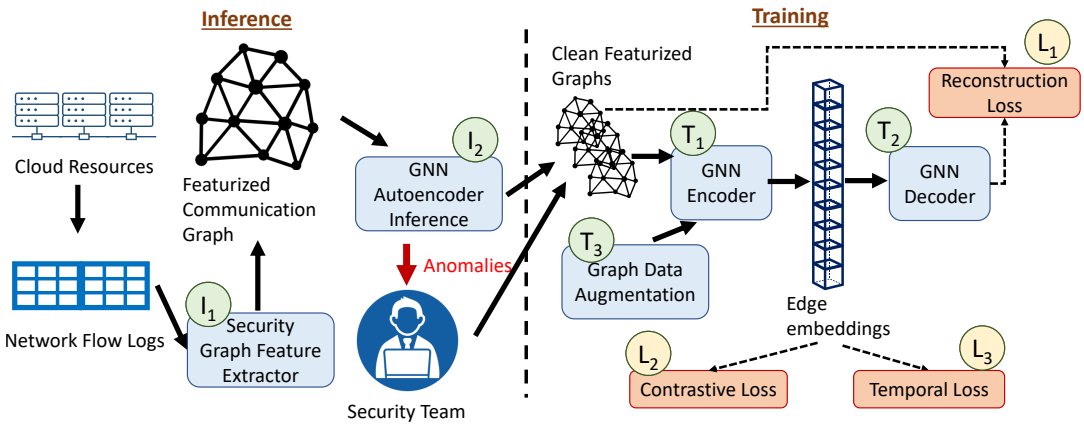


Figure 4: **Overview of NetVigil**

To enhance robustness, we design a *graph data augmentation* module ($\mathbf{T}_3$ in Figure 4) and train the system to encourage similarity between the embeddings of original and augmented graphs by minimizing the contrastive loss ($\mathbf{L}_2$, more details in Section 4.3). Furthermore, we introduce a temporal loss ($\mathbf{L}_3$, more details in Section 4.4) to promote embedding similarity between temporally-adjacent graphs. The entire training process is conducted end-to-end, resulting in an up-to-date GNN autoencoder for the subsequent inference phase.

Feature	Statistics
Number of transmitted packets Number of received packets Total received bytes Total transmitted bytes	min, max, mean, sum, std
Number of TCP flows Number of UDP flows Number of local unseen ports Number of global unseen ports Number of ports	count

Table 2: **Features obtained for each distinct IP pair**

4 Design Details of NetVigil

We provide a comprehensive discussion of the key modules within NetVigil and explain how these components collaborate to fulfill our design objectives. We discuss the security-oriented graph feature extractor (Section 4.1), graph representation learning (Section 4.2), domain-specialized graph contrastive learning (Section 4.3), and temporal smoothing techniques (Section 4.4) employed by NetVigil to ensure a cost-effective, accurate, and robust anomaly detection system for east-west data center networks.

4.1 Security Graph Feature Extractor

Our security graph feature extractor aims to gather security-oriented features from network flow logs, prioritizing cost efficiency while maintaining essential information for downstream anomaly detection. Within each aggregation interval (e.g., one minute), network flow logs typically contain the following information: (a) 5-tuple data, encompassing protocol, source and destination IP addresses, and source and destination ports, (b) the number of transmitted and received packets, and (c) the volume of transmitted and received bytes.

Our graph feature extractor operates at the IP address level instead of the network flow (IP and port) level for two primary reasons. First, operating at the network flow level leads to much larger graphs and increases the burden on both inference and training processes. For instance, we observe orders of magnitude increases in the number of nodes and edges in our production traces when constructing communication graphs at the network flow level (139k nodes and 115k edges) compared to the IP level (300 nodes and 10-20k edges). Furthermore, aggregating at the IP address level facilitates the identification of correlations between flows associated with the same IP address (e.g., a notable increase in the number of flows or usage of different ports). This, in turn, simplifies the detection of anomalous attacks, such as port scanning.

Our graph feature extractor offers additional operational cost reductions through a tunable *detection window* (e.g., two or three minutes). By generating a single communication graph per detection window, the feature extractor effectively balances detection latency with cost efficiency. Although a larger detection window might marginally impact detection accuracy for evasive attacks, this approach enables network operators to harmonize the requirement for prompt detection with limited resource constraints. Consequently, it becomes a suitable solution for large-scale network monitoring and security applications.

Table 2 summarizes the features we obtain for each distinct IP pair. We exclude ephemeral ports from the port-related features, as they do not provide learnable information. In addition to common features like the number of packets and bytes, we monitor critical security-oriented features such as the number of flows and unseen ports, which strongly indicate unusual occurrences. We maintain a record of globally observed ports

across all flows, as well as the locally observed ports for each IP pair within the training dataset. By maintaining port information as statistics, our feature extractor can function at the IP address level without sacrificing essential information. This approach allows us to capture crucial correlations among distinct flows associated with the same IP address while maintaining scalability and facilitating the detection of anomalous attacks involving multiple flows.

4.2 Graph Representation Learning

In order to discover relationships within the communication graph, NetVigil employs Graph Neural Networks (GNNs). These advanced machine learning models are specifically designed to analyze and extract patterns from complex graph-structured data. The key insight is that nodes within our communication graph correspond to various roles in an application and, as a result, each node and its neighbors exhibit a particular pattern over time. This contextual information enables NetVigil to identify anomalous behaviors that may seem normal if analyzed individually (e.g., C&C communication patterns or DNS amplification) without requiring the costly processing of granular packet-level information such as packet sizes, arrival times, and payloads. The ability to forego detailed packet-level data stems from the sufficiency of flow-level information for communication graph construction, capturing many key characteristics that distinguish a malicious flow from a normal one, including traffic volume, flow count, and interactions with various ports and IPs. Similar to prior work, E-GraphSage [51], we first aggregate edge features on each node as contextual information, and then concatenate the original edge features with this contextual information as the input to our edge encoder. Since E-GraphSage relies on supervised training, which is not practical at scale, we build our solution using an autoencoder by mapping each concatenated edge feature into a compressed embedding space.

Algorithm 1 presents the pseudocode for our GNN autoencoder. Lines 1–3 aggregate edge features using an AGG function, which can be mean, median, or element-wise pooling. Line 5 concatenates the aggregated contextual information with the original edge features, and Lines 6–8 encode the concatenated edge features into edge embeddings (Line 9). Lines 11-19 decode the embeddings back to the original edge features, and the reconstruction loss between e_{uv} and $\tilde{e}_{uv}$ corresponds to the $\mathbf{L}_1$ in Figure 4. Formally, for a (mini)batch $\mathcal{B}(\mathcal{G})$ of graphs, we have that

$$\mathbf{L}_1 = \frac{1}{\sum_{i|\mathcal{G}_i\in\mathcal{B}(\mathcal{G})}|\mathbb{E}_i|}\sum_{i|\mathcal{G}_i\in\mathcal{B}(\mathcal{G})}\sum_{(u,v)\in\mathbb{E}_i}\|e_{uv}-\tilde{e}_{uv}\|^2, \quad (1)$$

where $\mathbb{E}_i$ corresponds to the edge set of $\mathcal{G}_i$.

It is worth noting that although we employ simple graph convolutional networks (GCNs) [45] in Algorithm 1, the GNN architecture can be interchangeable, provided that the encoder takes into account both original edge features and aggregated neighboring features. While our experiments do not show a significant accuracy gain from using additional convolutional layers or alternate GNN architectures, some domain-specific GNN architectures may still perform better. We leave the exploration of domain-specific architectures for future work.

Algorithm 1 GNN Autoencoder for Edge Embedding

Input: Graph $\mathcal{G}(\mathbb{V},\mathbb{E})$
Input: Edge features $e_{uv}, \forall uv \in \mathbb{E}$
Input: Number of autoencoder layers L
Input: Encoder/decoder weights $W_E^l, W_D^l, \forall l \in 1,...,L$
1: **for** $v \in \mathbb{V}$ **do** ▷ Aggregate neighboring edge features
2: $h_v \leftarrow \text{AGG}(e_{uv}, \forall u \in N(v), (u,v) \in \mathbb{E})$
3: **end for**
4: **for** $(u,v) \in \mathbb{E}$ **do**
5: $h_{uv}^0 \leftarrow \text{CONCAT}(h_u, e_{uv})$
6: **for** $l \in 1,...,L$ **do** ▷ Edge Encoder
7: $h_{uv}^l \leftarrow \sigma(W_E^l \cdot h_{uv}^{l-1})$
8: **end for**
9: $z_{uv} = h_{uv}^L$ ▷ Edge embedding
10: **end for**
11: **for** $v \in \mathbb{V}$ **do** ▷ Broadcast edge embedding
12: $h_v \leftarrow \text{AGG}(z_{uv}, \forall u \in N(v), (u,v) \in \mathbb{E})$
13: **end for**
14: **for** $(u,v) \in \mathbb{E}$ **do**
15: $h_{e_{uv}}^0 \leftarrow \text{CONCAT}(h_u, e_{uv})$
16: **for** $l \in 1,...,L$ **do** ▷ Edge Decoder
17: $h_{uv}^l \leftarrow \sigma(W_D^l \cdot h_{uv}^{l-1})$
18: **end for**
19: $\tilde{e}_{uv} = h_{uv}^L$ ▷ Reconstructed edge features
20: **end for**

4.3 Domain-Specific Contrastive Learning

One of the primary challenges faced by existing network anomaly detectors is the generation of numerous false alarms for normal changes that were not encountered during the training process. A common approach to address this issue involves curating an extensive long-term dataset for training, with the expectation that all normal behaviors will be encompassed within this dataset. However, this approach is not scalable for east-west traffic, as they frequently experience normal changes, such as configuration updates, load variations, and node failures. To tackle this challenge, we employ graph contrastive learning [50], which augments the training data with general and domain-specific perturbations to enhance the model's generality. This approach allows the model to better accommodate and adapt to the dynamic nature of network traffic, thereby reducing the incidence of false alarms while accurately detecting genuine anomalies.

We find that most normal traffic fluctuations arise from (1) not all edges appearing consistently, (2) traffic volume varying over time, and (3) noisy behavior from non-application edges. Based on these observations, we employ the following data

augmentation strategies to enhance the model performance:

- Randomly removing edges and nodes: By presenting subgraphs to the model, the system gains the ability to more effectively analyze network communication patterns, attributable to the simplified structure and reduced noise.
- Adding noise to edge features: Edge features, such as the number of packets and the volume of transmitted/received bytes, are perturbed to test the model's robustness against variations in feature values.
- Removing non-application edges: It has been observed that application traffic within the network exhibits greater predictability compared to inter-service communications. Consequently, by removing nodes and edges unrelated to the application running within the network, the model can better learn and recognize application-level communication patterns, thereby enhancing its robustness and reducing the occurrence of false positives.

Formally, during training, a minibatch $\mathcal{B}(\mathcal{G})$ of graphs is randomly sampled. For every graph $\mathcal{G} \in \mathcal{B}(\mathcal{G})$, we generate two augmented versions $\mathcal{G}^{(1)}$ and $\mathcal{G}^{(2)}$ by randomly selecting two of the data augmentation strategies mentioned above. We denote the corresponding embeddings of an edge $uv \in \mathbb{E}$ by $z_{uv}^{(1)}$ and $z_{uv}^{(2)}$; see Line 9 in Algorithm 1. Recalling that the cosine similarity between two vectors x and y is given by $\cos(x,y) = x^\top y/(\|x\|\|y\|)$, we define the contrastive loss of a given edge uv as [85]

$$\ell_{uv} = -\log\left(\frac{\exp(2\cos(z_{uv}^{(1)}, z_{uv}^{(2)}))}{\sum_{u'v'}\exp(2\cos(z_{uv}^{(1)}, z_{u'v'}^{(2)}))}\right), \qquad (2)$$

where the negative edges $u'v'$ are randomly selected from augmented versions of other graphs in the minibatch. Notice that minimizing ℓ_{uv} promotes $z_{uv}^{(1)}$ and $z_{uv}^{(2)}$ to be similar, i.e., the embeddings corresponding to the same edge for two different augmented versions should be close to each other. Moreover, minimizing ℓ_{uv} also promotes $z_{uv}^{(1)}$ and $z_{u'v'}^{(2)}$ (the embeddings of different edges in augmented versions of different graphs) to be different from each other. In this way, we avoid the collapse of different embeddings into a common representation and encourage the full utilization of the embedding space. Our contrastive loss ($\mathbf{L}_2$ in Figure 4) is given by the average value of ℓ_{uv} over all edges in the minibatch

$$\mathbf{L}_2 = \frac{1}{\sum_{i|\mathcal{G}_i\in\mathcal{B}(\mathcal{G})}|\mathbb{E}_i|}\sum_{i|\mathcal{G}_i\in\mathcal{B}(\mathcal{G})}\sum_{uv\in\mathbb{E}_i}\ell_{uv}, \qquad (3)$$

where $\mathbb{E}_i$ corresponds to the edge set of $\mathcal{G}_i$. As defined, the loss $\mathbf{L}_2$ depends on the randomly selected augmentation strategies to compute the contrastive pairs of every graph. During training, we randomly choose new augmentation strategies for each minibatch, ensuring every gradient step is based on new contrastive pairs, thereby promoting generalization.

4.4 Temporal Smoothing and Continuous Retraining

Another crucial aspect that our model seeks to capture is temporal dynamics. Through our analysis of several traces from production clusters, we observe that network traffic within a short time window (e.g., minutes) tends to exhibit similarity, while patterns can undergo significant changes over longer periods (e.g., hours or days). This observation aligns with the understanding that major network traffic changes are typically driven by rare events (e.g., failures), periodicity (e.g., time of day), or application changes (e.g., code updates), which do not generally occur within short time frames.

We incorporate these temporal dynamics with a two-fold strategy. First, we define a temporal loss ($\mathbf{L}_3$ in Figure 4) during training to encourage embedding similarity between temporal adjacent graphs. For every pair of temporally adjacent graphs $\mathcal{G}^t$ and $\mathcal{G}^{t+1}$, we minimize the norm of the difference between consecutive embeddings of the same edge. More precisely, we have that

$$\mathbf{L}_3 = \frac{1}{\sum_t|\mathbb{E}_t\cap\mathbb{E}_{t+1}|}\sum_t\sum_{uv|uv\in\mathbb{E}_t\cap\mathbb{E}_{t+1}}\|z_{uv}^t - z_{uv}^{t+1}\|^2, \qquad (4)$$

where $\mathbb{E}_t$ denotes the edge set of $\mathcal{G}^t$. By minimizing $\mathbf{L}_3$, we promote the embeddings of the same edge in two consecutive time steps to be close to each other. Notice that, as a result of the dynamic nature of our graph, it might be that an edge uv that exists at time t is no longer present at time $t+1$. Hence, in (4), we account for this by only considering edges that belong to the intersection of two consecutive edge sets.

Second, we employ a periodic training procedure (e.g., hours or days) to update the model with the most recent traffic patterns. In each retraining window, we compile clean communication graphs (i.e., nodes and edges without any potential anomalies) from the inference phase (left side of Figure 4), as well as the false alarms cleared by the security team, to form the training set. Subsequently, we retrain the model by minimizing the composite loss, $\mathcal{L}$, which combines the reconstruction loss $\mathbf{L}_1$, contrastive loss $\mathbf{L}_2$, and temporal loss $\mathbf{L}_3$. The composite loss is defined as $\mathcal{L} = \mathbf{L}_1 + \alpha\mathbf{L}_2 + \beta\mathbf{L}_3$, where α and β are hyperparameters that trade off the relative importance of the different losses. This approach helps the model stay up-to-date and can effectively detect genuine anomalies and adapt to fluctuations in network traffic patterns.

5 Benchmarks and datasets

Datasets and network traces for intrusion and anomaly detection are available, but they primarily consist of packet traces. As stated in Section 2.2, conducting inference for each packet leads to considerable computational burdens, particularly in east-west traffic where there is a larger volume of network traffic. In light of this, we collect flow-level logs that summarize

network traffic. We deploy a 16-VM scale set on Microsoft Azure, activate Azure Network Watcher, and enable network security group (NSG) flow logging [54]. NSG logs record essential flow-level information such as 5-tuple, timestamp, number of bytes and packets transmitted, similar to other commercial flow logging offerings (e.g., [21]). For our 16-VM scale set, we deploy a web-based e-commerce app [11] that allows users to browse clothing items, add them to the cart, and complete purchases. Our application consists of 11 microservices for components such as ad generation, product catalog, payment, and product cart, as well as a load generator to send GET and POST requests, simulating user behavior like viewing items, setting currency, adding items to the cart, and submitting payment information.

On top of this setup, we develop and introduce a new benchmark, Yatesbury [1], designed to evaluate the performance and efficiency of anomaly detectors in east-west network traffic. We evaluate NetVigil with 13 distinct attacks that accurately represent a variety of malicious behaviors. Table 3 enumerates these attacks, each of which involves one or more compromised malicious nodes. Each trace lasts for 1-2 hours, and we label each trace for each 2-minute window and mark a connection as anomalous if any malicious traffic is sent or received between the two nodes. Malicious nodes communicate with each other and send malicious network traffic to the benign nodes. We utilize traditional network attacks that encompass various port scan methods, including traditional exhaustive port scanning, distributed scanning of multiple targets, stealth scanning using SYN packets to bypass firewalls, and accelerated scanning using UDP packets. Other traditional attacks we implement are DoS attacks, such as SYN flooding, which inundate victims with SYN requests, DDoS attacks involving multiple attackers, and DDoS attacks using UDP packets.

Furthermore, we incorporate attacks featuring more intricate communication patterns, which better represent malicious activity in east-west traffic. For these attacks, analyzing each flow in isolation (as done in traditional IDSes) is less effective, as it does not provide a holistic and comprehensive view of the network. We first employ Infection Monkey [8], an open-source breach and attack simulator for evaluating data center resiliency to perimeter breaches and internal host infection. It supports a wide range of different features such as port scanning, credential exploitation, and lateral movement to infect hosts. We also incorporate C&C communication patterns, which consist of a C&C server sending file updates, periodic heartbeat messages, and commands to control compromised hosts. These patterns are indicative of the communication observed in high-profile data breaches [1,3]. Further, we employ DNS amplification attacks, where multiple attackers send DNS requests to a DNS server and direct the responses to the victims. The DNS requests are crafted so that the responses are much larger in size to overwhelm the target machine.

6 Evaluation

6.1 Methodology

Benchmarks and datasets. We evaluate NetVigil on Yatesbury with our demo microservice application along with live production traces from our two first-party compute clusters.

Implementation. We implement NetVigil as an end-to-end data streaming pipeline using 1,400 lines of Python code. The inference pipeline extracts featurized communication graphs utilizing NetworkX [10] and pandas [12] libraries, while the training pipeline is built on PyTorch [13] and the Deep Graph Library (DGL) [7].

Baselines. We compare NetVigil against two state-of-the-art anomaly-based IDSes: Kitsune [56] and Whisper [31]. To evaluate Kitsune on our datasets, we modify its autoencoders to ingest flow-level features. Due to Kitsune's slow runtime, we implement several optimizations to reduce unnecessary computation that improves inference time by 5–10×; we refer to this optimized version of Kitsune as Kitsune+. Whisper's frequency domain analysis necessitates packet-level traces. Modifying it to utilize connection-level traces is challenging, as it performs frequency domain analysis on individual packets within each flow. Using only a single data point (aggregated flow-level statistics) would render it ineffective. To make Whisper compatible with flow logs, we utilize aggregated flow-level statistics to convert flow logs into packet traces. To understand the difference between flow and packet traces, we also carry out additional evaluations using packet traces for both Kitsune+ and Whisper.

Metrics. To evaluate NetVigil, we use the area under the ROC curve (AUC) as our primary metric, along with the true positive rate (TPR) and false positive rate (FPR). Importantly, AUC provides a measure of how well the detector can distinguish between the positive and negative classes, across all possible threshold settings. TPR and FPR are also crucial because the goal is to detect as many anomalies as possible while minimizing false alarms, which, as mentioned in Section 2, can significantly lower trust in an anomaly detector and is a fundamental challenge due to the dynamism of network traffic. To get these numbers, we select the threshold that maximizes $(TPR - FPR)$. Additionally, we compare latencies in running each anomaly detector. All latency experiments were run on a single 36-core, 72-hyperthread, 256-GB RAM machine (Intel(R) Xeon(R) Gold 5220).

6.2 Overall Results

We first compare the detection accuracy of NetVigil with our baselines, Kitsune+ and Whisper. For all attacks except one, NetVigil yields significantly higher performance over the baselines with AUC scores ranging from 0.6400 to 1.000, resulting in AUC improvements of up to 0.6591 over Kitsune+ and up

Attack	Description	# flows	Ratio malicious
Vertical Port Scan	Run an exhaustive scan of open ports	1429	0.0265
SYN Flood	DoS attack where connections are rapidly initialized but not completed	2817	0.0184
SYN Flood DDoS	DoS attack where connections are rapidly initialized but not completed (multiple attackers)	2437	0.0439
UDP DDoS	DoS attack with UDP packets (multiple attackers)	1473	0.0081
Distributed Stealth Port Scan	Run a targeted stealth scan of several key ports across many nodes with SYN packets	4069	0.0058
Distributed Port Scan	Run a targeted scan of several key ports across many nodes	4054	0.0051
Distributed UDP Port Scan	Run a targeted stealth scan of several key across many nodes with UDP packets	4319	0.0050
Infection Monkey 1	Scans key ports and launches network exploits	2768	0.0122
Infection Monkey 2	Scans key ports and launches network exploits (target limited number of hosts)	1490	0.0107
Infection Monkey 3	Scans key ports and launches network exploits (mount limited number of exploits)	4677	0.0027
C&C communication	Compromised nodes receive commands, heartbeats, and file updates from C&C server	2163	0.0254
DNS amplification	Attackers send DNS requests and direct responses to victim	4410	0.0825

Table 3: **Attack datasets**

	Kitsune+			Whisper			NetVigil		
	AUC	**TPR**	**FPR**	**AUC**	**TPR**	**FPR**	**AUC**	**TPR**	**FPR**
Moderate									
Vertical Port Scan	0.9300	0.8684	0.0057	0.9049	**0.9736**	0.1315	**0.9843**	0.9473	**0.0000**
SYN Flood	0.9322	0.8653	0.0014	0.7609	0.7307	0.1414	**1.0000**	**1.0000**	**0.0000**
Medium									
SYN Flood DDoS	0.9455	0.8971	0.0141	0.9148	0.9719	0.1283	**1.0000**	**1.0000**	**0.0000**
UDP DDoS	0.9455	0.8971	0.0141	0.6403	0.4166	0.1457	**0.9998**	**1.0000**	**0.0000**
Distributed Port Scan	0.4059	0.0952	0.0300	0.3961	0.0476	0.0329	**0.9968**	**0.9523**	**0.0000**
Distributed Stealth Port Scan	0.7542	0.6666	0.0758	0.6186	0.4166	0.1070	**0.9892**	**0.8333**	**0.0000**
Distributed UDP Port Scan	0.3367	0.0000	0.0281	0.3732	0.0000	0.4449	**0.9958**	**0.9545**	**0.0183**
Difficult									
Infection Monkey 1	0.5586	0.1176	**0.0003**	0.4395	0.0588	0.0190	**0.9997**	**1.0000**	0.0029
Infection Monkey 2	0.7497	0.5000	**0.0006**	0.4396	-	-	**0.9997**	**1.0000**	0.0033
Infection Monkey 3	0.5000	-	-	0.5000	-	-	**0.9998**	**1.0000**	**0.0006**
C&C communication	0.6347	0.4727	0.1480	0.5000	-	-	**0.9301**	**0.7636**	**0.0896**
DNS amplification	0.3962	0.0000	**0.0244**	0.8149	**0.8928**	0.2913	**0.8915**	0.3736	0.0692
SQL injection	**0.8531**	**1.0000**	**0.2237**	0.0900	-	-	0.6400	0.6428	0.2648
Unauthorized DB access	0.5976	**0.7500**	0.4720	0.7214	0.5833	0.0275	**0.8000**	0.7083	**0.1732**

Table 4: **Comparison of Kitsune+, Whisper, and NetVigil for various attacks.**

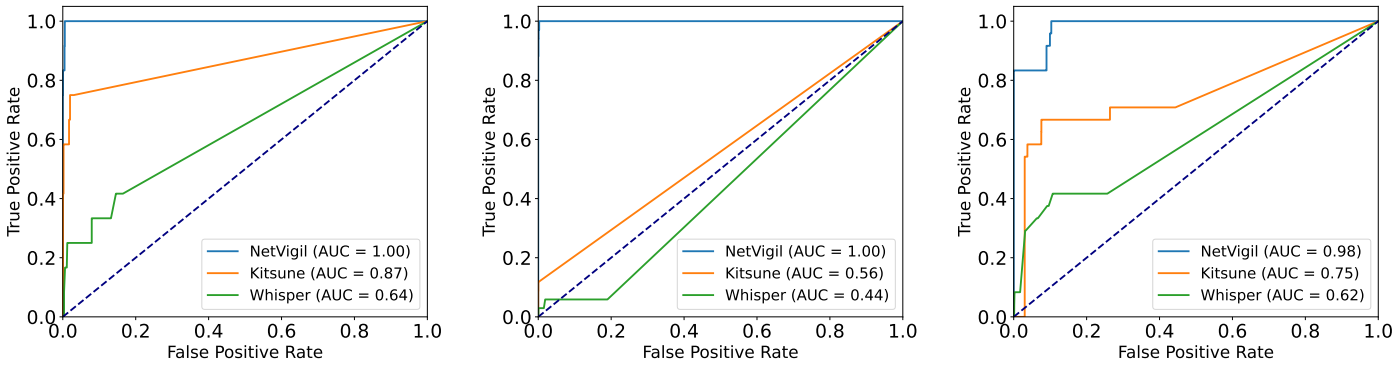
Attack	Kitsune+	Whisper	NetVigil
Vertical Port Scan	0.9817	0.9876	**0.9843**
UDP DDoS	0.9974	0.6414	**0.9998**
Dist. Stealth Port Scan	0.7267	0.6487	**0.9892**
Infection Monkey 1	0.8100	0.6188	**0.9997**
DNS Amplification	0.6759	0.8247	**0.8915**

Table 5: **AUC scores of Kitsune+ and Whisper using packet-level traces and NetVigil with flow-level logs for various attacks.**

to 0.6226 over Whisper. Table 4 presents the overall results for AUC, TPR, and FPR. Crucially, we observe that NetVigil outperforms the baselines because of two factors: (1) our novel security-centric feature extractor that effectively identifies lower-level malicious traffic characteristics in each connection that adversaries employ to fly under the radar, and (2) our use of graphs and a GNN architecture to obtain a holistic and comprehensive view of network behavioral patterns across many nodes.

Illustrating the efficacy of our feature selection approach, NetVigil exhibits strong performance in identifying DDoS attacks and vertical port scanning, achieving an AUC greater than 0.98 and an FPR approaching 0.0 for SYN Flood, SYN Flood DDoS, UDP DDoS, and Vertical Port Scanning. The extraction of packet- and connection-level statistics facilitates the detection of abnormal communications, such as substantial quantity of initiated connections in the SYN Flood scenario and the packet volume in other DDoS attacks.

Our GNN architecture excels in detecting reconnaissance patterns that span multiple nodes, an area where Kitsune+ and Whisper baselines demonstrate subpar performance. For Distributed Port Scan, Distributed Stealth Port Scan, and Distributed UDP Port Scan, Kitsune+ and Whisper yield AUC scores of 0.4059, 0.7542, 0.3367 and 0.3961, 0.6186, 0.3732,



(a) UDP DDoS (b) Infection Monkey 1 (c) Distributed Stealth Port Scan

Figure 5: **Area under the ROC curve for 3 sample attack traces.**

respectively. These scans involve an adversary probing a selected number of ports across various victim machines. The low traffic volume and variation from these network patterns enable them to evade detection. Similarly, our baselines exhibit inadequate performance in detecting infection monkey attacks due to similar reasons.

In contrast to these efforts, NetVigil uses its feature extractor in tandem with a graphical view of the network to successfully identify these scans and attacks. The features include previously unseen ports, a key characteristic in many scans and attacks. NetVigil also analyzes the number of ports as well as statistics on the number of bytes and packets that are sent/received. A large number of different ports with a comparatively small amount of traffic volume can be indicative of port scanning or of an adversary attempting multiple different exploits that target different ports/services. Furthermore, our GNN architecture detects higher-level behavioral patterns and relationships, rather than just relying on detecting each connection in isolation, as traditional host-based IDSes do. This is useful for detecting distributed port scanning and infection monkey attacks since, contrary to vertical port scanning and DoS attacks, each individual connection exhibits little abnormality in volume and variation, but each malicious actor makes connections to many different hosts, deviating from their typical communication patterns.

Although NetVigil performs well on detecting C&C communication and DNS amplification, it struggles to achieve the same performance as the other attacks. In addition to these scenarios encompassing behavioral patterns of many different nodes, each communication is more similar to traditional network traffic due to the file transfers and DNS queries, making them more difficult for our feature extractor to pick up.

Table 5 shows the results on several selected attacks in which we evaluated Kitsune+ and Whisper in their intended environment using packet-level traces. Overall, their AUC scores are significantly higher when using packet-level traces compared to flow-level traces, with Kitsune+ achieving near-perfect AUC scores for vertical port scanning and UDP DDoS. However, these approaches still fall short in matching NetVigil for other attacks due to the limitations of their extracted features and host-based detection models.

6.3 Efficiency Results

Figure 6 illustrate the wall clock and CPU times of NetVigil compared to Kitsune+ [56] and Whisper [31] for five different attacks. Each experiment involved feature extraction, training, and inference. NetVigil achieves key performance improvements through the following features:

1. Using flow-based features instead of packet-level data, reducing the amount of data significantly since only the aggregated statistics for each flow need to be processed.
2. Using a graph representation that aggregates features across multiple instances of the same connection.
3. An efficient GNN architecture with an autoencoder of two fully-connected layers.

Due to these components, the majority of time is spent during feature extraction. GNN inference takes only 2-3 seconds on average for a network trace with 16 VMs. Across 5 different attack traces, NetVigil achieves significantly lower execution times, yielding speedups of >= 37.59× and 2.87× – 7.38× over Kitsune+ and Whisper, respectively, for wall clock time, and speedups of >= 29.67× and 2.04× – 6.93× for CPU time. Further latency experiments where we varied the number of VMs and cores can be found in Appendix A.1.

6.4 System cost

When evaluating the system costs for Kitsune+ and Whisper, it is important to note that both of these tools necessitate packet traces, which can result in significant CPU and storage overheads to acquire. In addition, both baselines also incur considerable compute overheads during inference, requiring

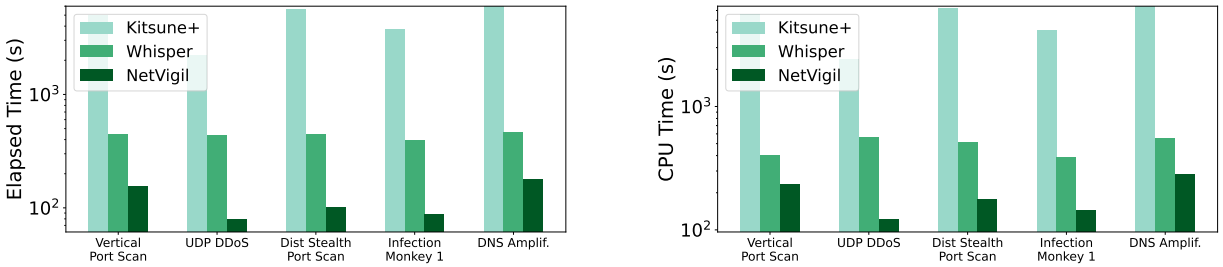


Figure 6: **Comparing detection latencies of Kitsune+, Whisper, and NetVigil in elapsed seconds and CPU time across several attacks. The y-axes use a logarithmic scale.**

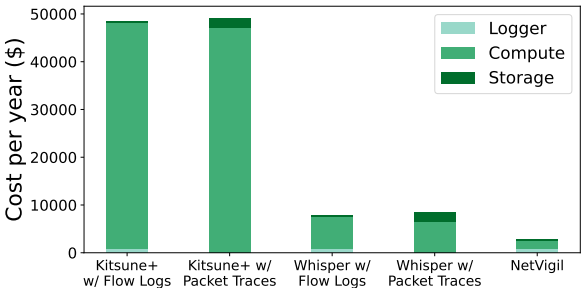


Figure 7: **System cost breakdown for Kitsune+, Whisper, and NetVigil for our 16-VM deployment**

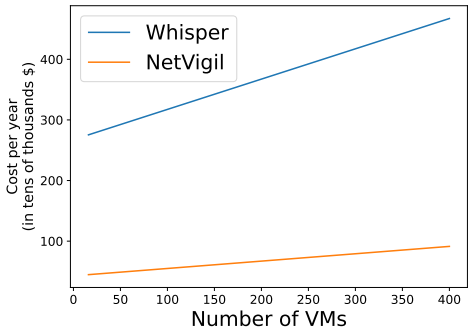


Figure 8: **Estimated monetary system cost for Whisper and NetVigil for production cluster of 400 VMs.**

an 8 vCPU VM for Whisper and a 56 vCPU VM for Kitsune+ to match the performance of NetVigil. Thus, we estimate, for our 16-VM deployment, a total system cost of $49159/year with packet traces and $48428/year with flow traces for Kitsune+ and a cost of $8602/year with packet traces and $7871 with flow traces for Whisper. To put these costs into perspective, we also analyze NetVigil. Because of the low cost of NetVigil, using a 2 vCPU VM is sufficient for performing all inference with flow logs, resulting in a total system cost of $2939/year. Figure 7 shows the system cost breakdown.

We perform a cost assessment of the system at larger scales by analyzing the trace data from our production cluster (see Section 6.5), which consists of 400 virtual machines (VMs). The outcomes are depicted in Figure 8. In comparison with Figure 7, it is clear that NetVigil exhibits superior cost-efficiency than Whisper under this scenario. This observation is primarily attributable to two factors. First, the network throughput in this setting exceeds that of our 16-VM deployment. As a result, Whisper's processing overhead, which is directly proportional to the number of packets, is substantially larger, while NetVigil's overhead remains independent of network throughput. Second, the production cluster primarily utilizes TCP connections, leading to a significantly reduced quantity of network flow records in contrast to the predominantly employed UDP connections in our 16-VM deployment.

6.5 Production Traces

We collect network flow records from two first-party compute clusters. The first cluster, `Service-Cluster`, contains approximately 400 VMs, and we gather traces for a week. The second cluster, `Compute-Cluster`, consists of around 200 VMs, and we acquire traces for two months. We confirm that no known attacks are present in these traces and use them to evaluate the false alarm rate of NetVigil.

We assess the number of false alarms without model retraining. For `Service-Cluster`, there are 4,356 false alarms on the last day of the week if the model isn't retrained, while model retraining reduces false alarms to 10. For `Compute-Cluster`, there are 1,231 false alarms without model retraining at the end of the week, and the number increases to 2,315 on the last day of the month. This cluster has less activity than `Service-Cluster`, explaining the lower dynamics. Model retraining reduces false alarms to fewer than 5 per day. The results from both product traces validate the importance of continuous retraining (Section 4.4).

Additionally, we inject attack traces into these production records to examine the performance of NetVigil. We incorporate Infection Monkey 1, 2, and 3, and replace the IP addresses and timestamps to blend the injected attack traces with normal ones. We observe that the detection accuracy of NetVigil remains consistent (similar to Table 4).

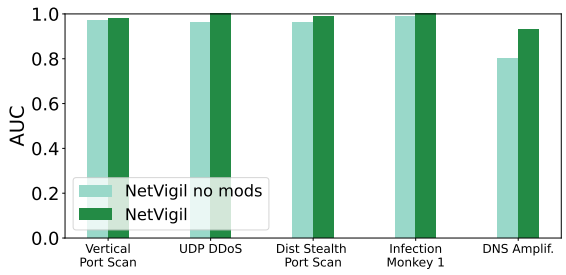


Figure 9: **AUC on several attack traces with and without temporal smoothing and data augmentation**

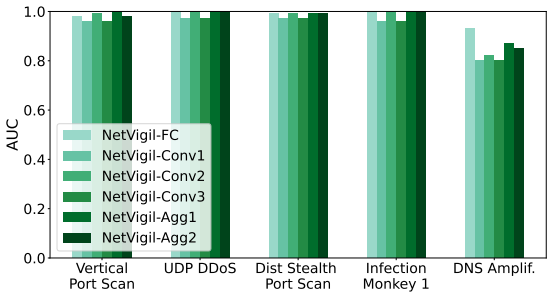


Figure 10: **AUC on several attack traces with different model modifications**

6.6 Ablation Study

We perform experiments to determine the importance of temporal smoothing and data augmentation. Figure 9 shows the AUC for NetVigil both with and without data augmentation and temporal smoothing. Using both techniques yields 1-2% AUC improvement for most baselines and 10% improvement for DNS amplification. Crucially, data augmentations add more heterogeneity to the dataset allowing it to become more robust to the dynamism in network traffic patterns. Temporal smoothing helps by ensuring that temporally similar graphs should be similar in structure and composition. Further, `NetVigil no mods` still performs highly, yielding over 0.95 AUC for all but 1 attack trace showing the efficacy of our approach on new network patterns even without any modifications.

We experiment further with different architectural modifications. Figure 10 shows the AUC results across 5 different attack traces with different model modifications. First we use a simple GNN architecture with fully-connected layers that operate on the graph's node and edge features (denoted by `NetVigil-FC`). We also used 3 different graph convolutional architectures (denoted by `NetVigil-Conv1`, `NetVigil-Conv2`, `NetVigil-Conv3`) that use graph attention layers [78], GraphSAGE layers [34], and EdgeConv layers [81]. We also try different aggregations for our message passing function using `min()` and `max()` instead of `mean()`, denoted by `NetVigil-Agg1` and `NetVigil-Agg2`, respectively. In summary, while `NetVigil-FC` performs the best overall due to its simpler architecture (and lower likelihood of overfitting) compared to the other architectures that use convolutional layers, all model variants perform similarly. We highlight that our approach is not tied to a particular model or architecture and that NetVigil is still able to reap significant performance gains on many different model variants.

7 Related Work

Related work on data center network security is discussed in Sections 1 and 2, and the advantages of NetVigil compared with the most relevant baselines are demonstrated and discussed in detail in Section 6. In this section, we focus on work related to GNNs, graph contrastive learning, and anomaly detection in graphs.

Graph Neural Networks. GNNs have gained significant attention in recent years as powerful tools for analyzing and modeling structured data represented as graphs. A considerable amount of research has been conducted in this field, and a variety of GNN architectures have been proposed. The most common ones include graph convolutional networks [45], graph attention networks [78], and GraphSAGE [34]. We refer the reader to relevant surveys [83, 87, 91] for further details on these and other related architectures. GNNs have achieved state-of-the-art performance in a series of problems in (wired and wireless) communication networks [28, 36, 41]. The NetVigil framework is agnostic to the specific GNN chosen for the encoder and the decoder. Thus, practitioners can seamlessly experiment with different architectures that might better accommodate their data.

Graph Contrastive Learning. Graph contrastive learning has emerged as a potential solution to several challenges faced by GNNs, such as heavy label reliance and weak robustness [50]. The core idea behind graph contrastive learning is to embed augmented versions of the same sample (node, edge, or graph) close to each other while trying to push away embeddings from different samples. Generating these augmented versions of a given sample can be challenging in a graph setting. Unlike images for which different augmented versions (contrastive pairs) can be generated by imposing different color filters or rotation operations, designing contrastive pairs can be challenging in graph settings. Some works use different parts of a graph to build these contrastive pairs [62, 71, 79] by comparing, e.g., nodes with subgraphs [38, 42]. Other works adopt graph data augmentations to generate contrastive pairs [35, 63, 72, 84, 86, 92]. Instead of relying on a generic set of augmentations, in NetVigil we leverage domain-specific knowledge to determine graph transformations that constitute valid contrastive pairs.

Anomaly Detection in Graphs. Anomaly detection is the data mining process that aims to identify patterns in data that do not conform to expected behavior [26]. If we focus on

the graph context, the objective is to find the graph objects (nodes, edges, or substructures) that are rare and that differ significantly from the majority of the reference objects in the given graph [20]. Graph anomaly detection has been applied to myriad settings, including telecom fraud [29], opinion spam [30], and malware detection [64]. Graphs bring specific challenges to the anomaly detection problem related to the inter-dependency of objects (a node being anomalous is not only a function of itself but also of its neighborhood) and the size of the search space (the search space of complex anomalies such as graph substructures is huge). Methods have been derived for the unsupervised [33] and (semi-)supervised [27] settings, for static [25] and dynamic [39] graphs, and for attributed [58] and plain (no attributes) [19] graphs. Notice that our problem falls in the most challenging category of unsupervised learning for dynamic and attributed graphs. Over the last five years, there has been an increasing interest in applying deep learning techniques to graph anomaly detection [52]. The idea is to depart from non-deep learning techniques with limited (linear) representation power [60] and use deep graph representation learning and GNNs to extract expressive representations such that graph anomalies and normal objects can be easily separated. Several methods have been developed for the simpler cases of static or plain graphs [37, 49, 80]. For the dynamic *and* attributed case, the existing techniques are limited [74, 89, 90]. Moreover, real-world networks (including our application) usually exhibit changes in *both* the network structure and node attributes. However, most existing works only consider changes in one of these aspects [74, 90]. To the best of our knowledge, we are the first to consider an autoencoder architecture enhanced by contrastive learning and temporal smoothing to tackle the challenging dynamic setting where both the network structure and attributes are changing.

8 Discussion

Privacy Consideration. Network flow logs may contain personally identifiable information (PII), such as user IP addresses, which are subject to data privacy compliance requirements [22, 76]. To address these privacy concerns, NetVigil employs two strategies. First, our model can be deployed using the Software as a Service (SaaS) model, where users continuously stream anonymized network flow logs to a server running our system. Anonymization can be achieved through encrypted IP addresses, as our model does not require plaintext IP addresses for anomaly detection, and users can interpret the encrypted results accordingly. Second, our model can be deployed within a user's cloud subscription as a standalone service, ensuring that all network flow logs remain entirely under the user's control. By implementing these strategies, we maintain a high level of privacy while still providing effective anomaly detection in network traffic patterns.

Initial Clean Training Set. NetVigil requires at least one clean dataset to train the initial model, with subsequent models obtained as discussed in Section 4.4. As with all anomaly-based intrusion detection systems, if a cloud deployment is already compromised from the outset, some anomalous behaviors might contaminate the model. Therefore, it is much safer to obtain the initial training set in a secure environment (e.g., a sandbox). This precautionary measure helps ensure that the model's foundation is built upon clean and reliable data, allowing it to effectively detect and adapt to genuine anomalies and fluctuations in network traffic patterns.

Applying to North-South Traffic. Although our primary focus in this study is on east-west traffic, the principles of NetVigil, such as employing GNNs for intrusion detection using flow-level logs, can potentially be extended to north-south traffic. This method could further decrease the significant computational costs of existing IDSes or create a comprehensive security solution for both east-west and north-south traffic simultaneously. North-south traffic typically exhibits increased node and contextual information variability, which may necessitate specialized learning techniques. Exploring this design would be an interesting future direction.

9 Conclusion

We present NetVigil, a novel network anomaly detection system specifically designed for securing east-west traffic in large-scale data center networks. Addressing the limitations of existing solutions, our approach focuses on three key objectives: (a) ensuring cost-effectiveness in monitoring numerous nodes, (b) accurately identifying anomalous behaviors while minimizing false alarms, and (c) exhibiting robustness against normal traffic fluctuations without reliance on prior knowledge of malicious attacks. By employing low-cost network flow logs, security-oriented graph features, graph neural networks, and a novel end-to-end training mechanism, our system achieves substantial improvements over existing malicious traffic detectors. We hope that the insights gained from our solution, along with the new east-west security benchmark, Yatesbury, will facilitate the validation of our proposed architecture and foster future research and innovation in this vital area of study.

Acknowledgments

We extend our gratitude to our shepherd, Bruce Davie, and the anonymous reviewers for providing invaluable and constructive feedback. Our thanks also go to our engineering and product collaborators, including Narayan Annamalai, Umair Aftab, Eliran Azulai, Wyman Chong, Jamie Lee, Kiran Muthabatulla, Mariana Alanis Tamez, and Roger Wong, for their contributions to production traces, data processing pipelines, and system requirements.

References

[1] 2017 Equifax data breach. https://en.wikipedia.org/wiki/2017_Equifax_data_breach, Retrieved on 2023-04.

[2] 2019 Zeek specs needed for 10gbps. http://mailman.icsi.berkeley.edu/pipermail/zeek/2019-September/014574.html, Retrieved on 2023-09.

[3] 2020 United States federal government data breach. https://en.wikipedia.org/wiki/2020_United_States_federal_government_data_breach, Retrieved on 2023-04.

[4] A Deeper Look at the Distributed Cloud Firewall: A Firewall for the Cloud Era. https://aviatrix.com/blog/a-deeper-look-at-the-distributed-cloud-firewall-a-firewall-for-the-cloud-era/, Retrieved on 2023-09.

[5] Aviatrix Distributed Cloud Firewall. https://aviatrix.com/distributed-cloud-firewall/, Retrieved on 2023-09.

[6] Calico flow logs. https://docs.tigera.io/calico-cloud/visibility/elastic/flow/, Retrieved on 2023-10.

[7] Deep Graph Library. https://www.dgl.ai/, Retrieved on 2022-11.

[8] Infection monkey. https://www.akamai.com/infectionmonkey, Retrieved on 2023-03.

[9] Introduction to Cilium & Hubble. https://docs.cilium.io/en/stable/overview/intro/, Retrieved on 2023-10.

[10] NetworkX: network analysis in python. https://networkx.org/, Retrieved on 2023-01.

[11] Online boutique. https://github.com/GoogleCloudPlatform/microservices-demo, Retrieved on 2022-07.

[12] pandas - python data analysis library. https://pandas.pydata.org/, Retrieved on 2023-01.

[13] PyTorch. https://pytorch.org/, Retrieved on 2022-11.

[14] Scaling Suricata performance to 100 Gbps with Napatech SmartNICs. https://www.napatech.com/support/resources/solution-descriptions/scaling-suricata-performance-to-100-gbps-with-napatech-smartnics/, Retrieved on 2023-09.

[15] Suricata. https://suricata.io/, Retrieved on 2023-09.

[16] Trends in data center security: Part 1 – traffic trends. https://blogs.cisco.com/security/trends-in-data-center-security-part-1-traffic-trends, Retrieved on 2023-09.

[17] VMware NSX. https://www.vmware.com/products/nsx.html, Retrieved on 2023-09.

[18] VMware NSX: The platform for network virtualization. https://www.virtualizationworks.com/NSX.asp, Retrieved on 2023-09.

[19] Leman Akoglu, Mary McGlohon, and Christos Faloutsos. Oddball: Spotting anomalies in weighted graphs. In *Advances in Knowledge Discovery and Data Mining: 14th Pacific-Asia Conference, PAKDD 2010, Hyderabad, India, June 21-24, 2010. Proceedings. Part II 14*, pages 410–421. Springer, 2010.

[20] Leman Akoglu, Hanghang Tong, and Danai Koutra. Graph based anomaly detection and description: a survey. *Data mining and knowledge discovery*, 29:626–688, 2015.

[21] Amazon. Logging IP traffic using VPC Flow Logs. https://docs.aws.amazon.com/vpc/latest/userguide/flow-logs.html, Retrieved on 2023-04.

[22] Behnaz Arzani, Selim Ciraci, Stefan Saroiu, Alec Wolman, Jack W Stokes, Geoff Outhred, and Lechao Diwu. PrivateEye: Scalable and privacy-preserving compromise detection in the cloud. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[23] Leyla Bilge and Tudor Dumitraş. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2012.

[24] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *Communications of the ACM*, 59(5), 2016.

[25] Deepayan Chakrabarti. Autopart: Parameter-free graph partitioning and outlier detection. In *Knowledge Discovery in Databases: PKDD 2004: 8th European Conference on Principles and Practice of Knowledge Discovery in Databases, Pisa, Italy, September 20-24, 2004. Proceedings 8*, pages 112–124. Springer, 2004.

[26] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3), jul 2009.

[27] Duen Horng Chau, Shashank Pandit, and Christos Faloutsos. Detecting fraudulent personalities in networks of online auctioneers. In *Knowledge Discovery*

in Databases: PKDD 2006: 10th European Conference on Principles and Practice of Knowledge Discovery in Databases Berlin, Germany, September 18-22, 2006 Proceedings 10, pages 103–114. Springer, 2006.

[28] Arindam Chowdhury, Gunjan Verma, Chirag Rao, Anantham Swami, and Santiago Segarra. Unfolding WMMSE using graph neural networks for efficient power allocation. *IEEE Transactions on Wireless Communications*, 20(9):6004–6017, 2021.

[29] Corinna Cortes, Daryl Pregibon, and Chris Volinsky. Communities of interest. In *Advances in Intelligent Data Analysis: 4th International Conference, IDA 2001 Cascais, Portugal, September 13–15, 2001 Proceedings 4*, pages 105–114. Springer, 2001.

[30] Hanbo Dai, Feida Zhu, Ee-Peng Lim, and HweeHwa Pang. Detecting anomalies in bipartite graphs with mutual dependency principles. In *2012 IEEE 12th International Conference on Data Mining*, pages 171–180, 2012.

[31] Chuanpu Fu, Qi Li, Meng Shen, and Ke Xu. Realtime robust malicious traffic detection via frequency domain analysis. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[32] Sunanda Gamage and Jagath Samarabandu. Deep learning methods in network intrusion detection: A survey and an objective comparison. *Journal of Network and Computer Applications*, 169, 2020.

[33] Jing Gao, Feng Liang, Wei Fan, Chi Wang, Yizhou Sun, and Jiawei Han. On community outliers and their efficient detection in information networks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 813–822, 2010.

[34] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[35] Kaveh Hassani and Amir Hosein Khasahmadi. Contrastive multi-view representation learning on graphs. In Hal Daumé III and Aarti Singh, editors, *37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 4116–4126. PMLR, 13–18 Jul 2020.

[36] Shiwen He, Shaowen Xiong, Yeyu Ou, Jian Zhang, Jiaheng Wang, Yongming Huang, and Yaoxue Zhang. An overview on the application of graph neural networks in wireless networks. *IEEE Open Journal of the Communications Society*, 2:2547–2565, 2021.

[37] Renjun Hu, Charu C. Aggarwal, Shuai Ma, and Jinpeng Huai. An embedding approach to anomaly detection. In *IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 385–396, 2016.

[38] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for pre-training graph neural networks. In *International Conference on Learning Representations*, 2020.

[39] Tsuyoshi Idé and Hisashi Kashima. Eigenspace-based anomaly detection in computer systems. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 440–449, 2004.

[40] Illumio. Zero trust: the security paradigm for the modern organization. https://www.illumio.com/solutions/zero-trust, Retrieved on 2023-04.

[41] Weiwei Jiang. Graph-based deep learning for communication networks: A survey. *Computer Communications*, 185:40–54, 2022.

[42] Yizhu Jiao, Yun Xiong, Jiawei Zhang, Yao Zhang, Tianqi Zhang, and Yangyong Zhu. Sub-graph contrast for scalable self-supervised graph representation learning. In *IEEE International Conference on Data Mining (ICDM)*, pages 222–231, 2020.

[43] Ansam Khraisat, Iqbal Gondal, Peter Vamplew, and Joarder Kamruzzaman. Survey of intrusion detection systems: techniques, datasets and challenges. *Cybersecurity*, 2(1), 2019.

[44] John Kindervag, Stephanie Balaouras, and Lindsey Coit. Build security into your network's DNA: The zero trust network architecture. *Forrester Research Inc*, 27, 2010.

[45] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

[46] Vinod Kumar and Om Prakash Sangwan. Signature based intrusion detection system using SNORT. *International Journal of Computer Applications & Information Technology*, 1(3), 2012.

[47] Donghwoon Kwon, Hyunjoo Kim, Jinoh Kim, Sang C Suh, Ikkyun Kim, and Kuinam J Kim. A survey of deep learning-based network anomaly detection. *Cluster Computing*, 22, 2019.

[48] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1), 2013.

[49] Ninghao Liu, Xiao Huang, and Xia Hu. Accelerated local anomaly detection via resolving attributed networks. In *26th International Joint Conference on Artificial Intelligence*, IJCAI'17, page 2337–2343. AAAI Press, 2017.

[50] Yixin Liu, Ming Jin, Shirui Pan, Chuan Zhou, Yu Zheng, Feng Xia, and Philip S. Yu. Graph self-supervised learning: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 35(6):5879–5900, 2023.

[51] Wai Weng Lo, Siamak Layeghy, Mohanad Sarhan, Marcus Gallagher, and Marius Portmann. E-graphsage: A graph neural network based intrusion detection system for IOT. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2022.

[52] Xiaoxiao Ma, Jia Wu, Shan Xue, Jian Yang, Chuan Zhou, Quan Z. Sheng, Hui Xiong, and Leman Akoglu. A comprehensive survey on graph anomaly detection with deep learning. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2021.

[53] Mohammad Masdari and Hemn Khezri. A survey and taxonomy of the fuzzy signature-based intrusion detection systems. *Applied Soft Computing*, 92, 2020.

[54] Microsoft. Flow logs for network security groups. https://learn.microsoft.com/en-us/azure/network-watcher/network-watcher-nsg-flow-logging-overview, Retrieved on 2023-04.

[55] Microsoft. Microsoft digital defense report 2022. https://www.microsoft.com/en-us/security/business/microsoft-digital-defense-report-2022, Retrieved on 2023-04.

[56] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[57] Chirag Modi, Dhiren Patel, Bhavesh Borisaniya, Hiren Patel, Avi Patel, and Muttukrishnan Rajarajan. A survey of intrusion detection techniques in cloud. *Journal of network and computer applications*, 36(1), 2013.

[58] Caleb C Noble and Diane J Cook. Graph-based anomaly detection. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 631–636, 2003.

[59] Palo Alto Networks. Prisma cloud: Cloud network security. https://www.paloaltonetworks.com/prisma/cloud/cloud-network-security, Retrieved on 2023-04.

[60] Guansong Pang, Chunhua Shen, Longbing Cao, and Anton Van Den Hengel. Deep learning for anomaly detection: A review. *ACM Comput. Surv.*, 54(2), mar 2021.

[61] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Comput. Netw.*, 31(23–24):2435–2463, dec 1999.

[62] Zhen Peng, Wenbing Huang, Minnan Luo, Qinghua Zheng, Yu Rong, Tingyang Xu, and Junzhou Huang. Graph representation learning via graphical mutual information maximization. In *Proceedings of The Web Conference 2020*, WWW '20, page 259–270. Association for Computing Machinery, 2020.

[63] Jiezhong Qiu, Qibin Chen, Yuxiao Dong, Jing Zhang, Hongxia Yang, Ming Ding, Kuansan Wang, and Jie Tang. GCC: Graph contrastive coding for graph neural network pre-training. In *26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 1150–1160. Association for Computing Machinery, 2020.

[64] Md Sazzadur Rahman, Ting-Kai Huang, Harsha V Madhyastha, and Michalis Faloutsos. Efficient and scalable socware detection in online social networks. In *USENIX Security Symposium*, pages 663–678, 2012.

[65] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, page 229–238, USA, 1999. USENIX Association.

[66] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. Zero trust architecture. Technical report, National Institute of Standards and Technology, 2020.

[67] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[68] Fred B Schneider. Least privilege and more computer security. *IEEE Security & Privacy*, 1(5), 2003.

[69] Rupam Kumar Sharma, Hemanta Kumar Kalita, and Biju Issac. Different firewall techniques: A survey. In *Fifth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, 2014.

[70] Nitin Singh Sikarwar and Dinesh Verma. Micro segmentation: today's success formulae. *International Journal of Operations Management and Services*, 2(1), 2012.

[71] Fan-Yun Sun, Jordan Hoffman, Vikas Verma, and Jian Tang. InfoGraph: Unsupervised and semi-supervised graph-level representation learning via mutual information maximization. In *International Conference on Learning Representations*, 2020.

[72] Susheel Suresh, Pan Li, Cong Hao, and Jennifer Neville. Adversarial graph augmentation to improve graph contrastive learning. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 15920–15933, 2021.

[73] Colin Tankard. Advanced persistent threats and how to monitor and deter them. *Network security*, 2011(8), 2011.

[74] Xian Teng, Yu-Ru Lin, and Xidao Wen. Anomaly detection in dynamic networks using multi-view time-series hypersphere learning. In *2017 ACM on Conference on Information and Knowledge Management*, pages 827–836, 2017.

[75] Zhihong Tian, Wei Shi, Yuhang Wang, Chunsheng Zhu, Xiaojiang Du, Shen Su, Yanbin Sun, and Nadra Guizani. Real-time lateral movement detection based on evidence reasoning network for edge computing environment. *IEEE Transactions on Industrial Informatics*, 15(7), 2019.

[76] European Union. General data protection regulation (GDPR). https://commission.europa.eu/law/law-topic/data-protection_en, Retrieved on 2023-04.

[77] Romans Vanickis, Paul Jacob, Sohelia Dehghanzadeh, and Brian Lee. Access control policy enforcement for zero-trust-networking. In *29th Irish Signals and Systems Conference (ISSC)*, 2018.

[78] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

[79] Petar Velickovic, William Fedus, William L. Hamilton, Pietro Liò, Yoshua Bengio, and R Devon Hjelm. Deep graph infomax. In *International Conference on Learning Representations*, 2019.

[80] Yanling Wang, Jing Zhang, Shasha Guo, Hongzhi Yin, Cuiping Li, and Hong Chen. Decoupling representation learning and classification for GNN-based anomaly detection. In *44th international ACM SIGIR conference on research and development in information retrieval*, pages 1239–1248, 2021.

[81] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, and Justin M. Solomon. Dynamic graph cnn for learning on point clouds. *ACM Trans. Graph.*, 38(5), oct 2019.

[82] Marcus Willett. Lessons of the SolarWinds hack. *Survival*, 63(2), 2021.

[83] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2021.

[84] Yuning You, Tianlong Chen, Yang Shen, and Zhangyang Wang. Graph contrastive learning automated. In Marina Meila and Tong Zhang, editors, *38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 12121–12132. PMLR, 18–24 Jul 2021.

[85] Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, and Yang Shen. Graph contrastive learning with augmentations. *Advances in neural information processing systems (NeurIPS)*, 33, 2020.

[86] Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, and Yang Shen. Graph contrastive learning with augmentations. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 5812–5823, 2020.

[87] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 34(1):249–270, 2022.

[88] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C Hoe, Vyas Sekar, and Justine Sherry. Achieving 100gbps intrusion prevention on a single server. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2020.

[89] Li Zheng, Zhenpeng Li, Jian Li, Zhao Li, and Jun Gao. Addgraph: Anomaly detection in dynamic graph using attention-based temporal gcn. In *28th International Joint Conference on Artificial Intelligence*, IJCAI'19, page 4419–4425, 2019.

[90] Panpan Zheng, Shuhan Yuan, Xintao Wu, Jun Li, and Aidong Lu. One-class adversarial nets for fraud detection. In *AAAI Conference on Artificial Intelligence*, volume 33, pages 1286–1293, 2019.

[91] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[92] Yanqiao Zhu, Yichen Xu, Feng Yu, Qiang Liu, Shu Wu, and Liang Wang. Graph contrastive learning with adaptive augmentation. In *Proceedings of the Web Conference*, WWW '21, page 2069–2080, 2021.

A Appendix

A.1 Additional efficiency results

We demonstrate the scalability of our approach to larger network log sizes. With a trace of 4 VMs, it takes 123 seconds for Whisper and 96 seconds for NetVigil. As the trace size increases, the execution time of Whisper also increases, resulting in 373 seconds to process a trace with 16 VMs. Meanwhile, the execution time of NetVigil only increases slightly to 140 seconds. Furthermore, as Whisper is allocated more cores, the CPU time increases from 508 seconds for 4 cores to 1520 seconds for 16 cores, while the wall clock time decreases marginally from 173 seconds with 4 cores to 164 seconds with 8 cores. In contrast, the runtime of NetVigil remains relatively stable when changing the number of allocated cores since it does not rely on parallelism for efficiency.

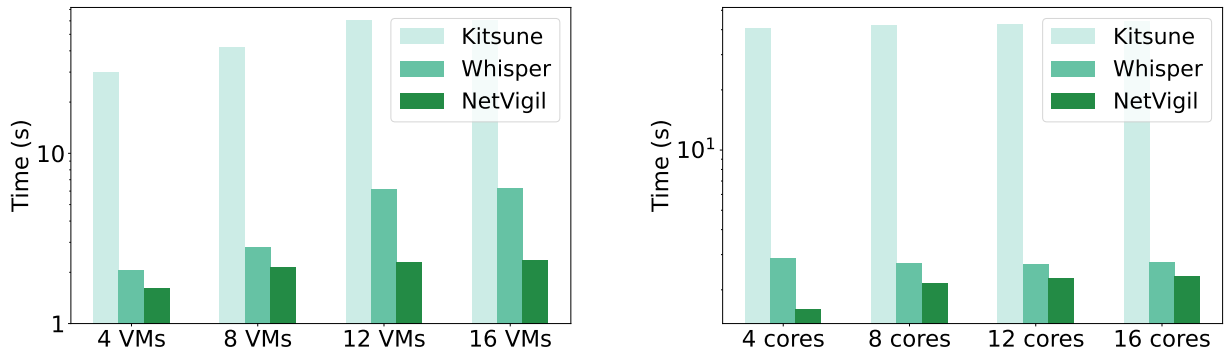


Figure 11: **Comparing detection latencies of Kitsune+, Whisper, and NetVigil in elapsed seconds while varying number of VMs in the network trace (left), number of cores used for processing each trace (middle), and attack traces (right).**

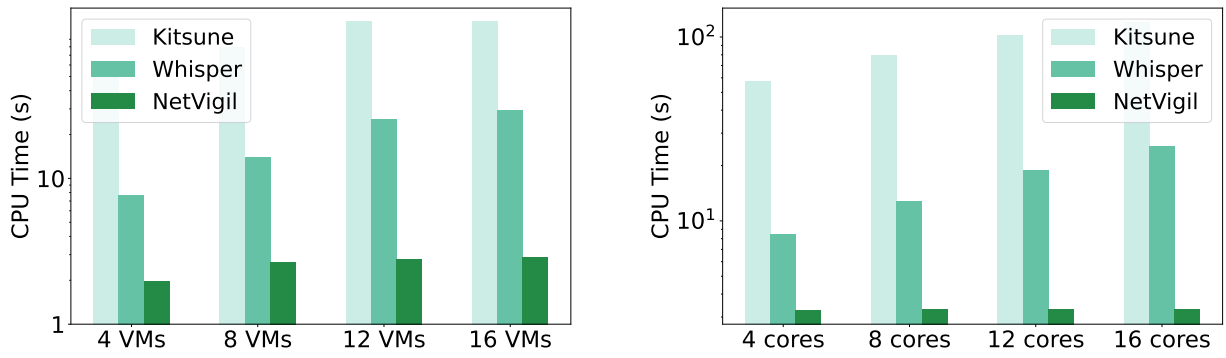


Figure 12: **Comparing detection latencies of Kitsune+, Whisper, and NetVigil in CPU seconds while varying number of VMs in the network trace (left), number of cores used for processing each trace (middle), and attack traces (right).**

TANGO: Secure Collaborative Route Control across the Public Internet

Henry Birge-Lee
Princeton University
birgelee@princeton.edu

Sophia Yoo
Princeton University
sophiayoo@princeton.edu

Benjamin Herber
Princeton University
bherber@alumni.princeton.edu

Jennifer Rexford
Princeton University
jrex@cs.princeton.edu

Maria Apostolaki
Princeton University
apostolaki@princeton.edu

Abstract

As the demands of modern latency-critical applications grow, major service providers are seeking to meet those demands by expanding their infrastructure to the edge and offering global connectivity through private WANs or Network-as-a-Service solutions. Unfortunately, these approaches are costly for smaller edge networks and lead to Internet consolidation. Worse, since the public Internet suffers from limited visibility and control over interdomain routing, smaller edges today are left with poor alternatives outside of joining the hypergiants. As a new alternative, we introduce TANGO, which enables smaller edges to expose paths and exert route control over the public Internet *without relying on third parties or cooperation from the Internet core*, to dynamically meet the performance needs of their customers. We show that, using *collaboration*, TANGO edges can jointly *(i)* expose more BGP-compliant wide-area paths via coordinated BGP advertisements; *(ii)* collect fine-grained, trustworthy telemetry using cryptographically-protected custom headers; and *(iii)* dynamically reroute traffic in the data plane. TANGO innovates in both the control and data planes, and runs on a programmable switch or in eBPF. Our Internet-scale experiments uncover rich path diversity, exposing paths that outperform the default BGP path 75-100% of the time for 20 edge pairs across multiple continents, while reducing latency by up to 39% compared to the default.

1 Introduction

Modern networked applications, from self-driving cars to online gaming and video conferencing, have strict requirements of high reliability and low latency [11, 34, 41, 54]. To satisfy these needs, hypergiants continually expand their private network infrastructure closer to the edge, effectively optimizing client experience. For instance, Google not only operates multiple Points of Presence (PoPs) globally, connecting data centers to the rest of the Internet via peering, but also partners with ISPs to deploy Google-supplied servers inside the ISP networks [37]. Similarly, Azure proposed deploying physical infrastructure inside enterprise premises to optimize ingress traffic [43]. Edge networks such as enterprises and small clouds, however, are unable to continuously expand their infrastructure and are forced to resort to alternatives such as Network-as-a-Service (NaaS) [7] from global cloud providers, outsourcing their traffic management while inheriting third-party practices and security/privacy policies. This private-WAN trend leads to increased industry consolidation, benefiting larger companies and well-served regions while leaving smaller edge networks with limited negotiation power, reduced growth opportunities, and increased vulnerability to outages [40, 75]. The natural question arises: *Is moving to private infrastructure the only way to meet growing application requirements, or can the public Internet rise to the challenge?*

To answer this, we must first understand the obstacles preventing an edge network from extracting more performance and route control in today's public Internet. Edge networks have very limited available path diversity. BGP (*i.e.,* the default Internet routing protocol) selects a single path per destination prefix based on crude (often performance-unaware) criteria [82]. While multi-homed ASes can optimize the first hop of their path [9, 10, 33], they are unable to tap into the Internet's full path diversity without collaboration from the Internet core. SD-WAN solutions [1, 2, 7] that combine multi-homing, overlay routing, and multicast techniques are still limited to BGP-default paths. Even assuming an edge network could forward traffic via adequately distinct paths, identifying performance opportunities requires accurate and trustworthy monitoring, which is impossible in practice for a single edge. Indeed, measurements at the hosts [53, 55] or even at the border of a stub network [15, 35] are affected by the performance of both the forward and reverse path and are inflated by the load on the receiver's access network, their hardware, and even the application itself, hiding performance differences between paths. Active probing might avoid some of this noise, but probes can be preferentially treated, hence unreliable [18]. Worse yet, some ASes might attempt to fool any measurement infrastructure to attract more traffic or hide their outages.

To overcome these obstacles, we present TANGO, an edge-to-edge route-control scheme that relies on *cooperation* between pairs of edge networks (*e.g.,* enterprises and data centers). We ob-

serve that collaboration between edge networks occurs naturally in today's Internet, either among sites of the same organization or between pairs of enterprises, creating many real-world opportunities for TANGO. TANGO exploits this collaboration to expose multiple routes per destination that are already installed in core routing tables (but not used by BGP) by advertising multiple prefixes for the same destination. While TANGO edges cannot explicitly change how on-path routers forward traffic, they can remotely guide the propagation of BGP advertisements for each prefix via surgical use of BGP communities and path poisoning, in an *automated manner and with no prior topology knowledge*.

TANGO also performs accurate and trustworthy wide-area monitoring between two edges, another building block towards reliable and real-time route control. Edges can operate TANGO nodes at their border gateways to piggyback telemetry information (metadata) on every packet at the sending edge [42]. This metadata is then stripped away at the receiving edge, before the packet is forwarded to its destination. In this manner, TANGO edges obtain accurate *wide-area*, *one-way* measurements unpolluted by reverse path metrics, noisy access networks, or application glitches. Further, relying on metadata makes TANGO *protocol-agnostic* (does not rely on TCP semantics) and scalable (does not keep per-flow or packet state [47, 79]). Most importantly, since TANGO operates over the untrusted public Internet, it provides *trustworthy* telemetry using shared book ciphers and secure OTP-protected route updates directly in the data plane.

TANGO has potential for many modern use cases. For example, a cloud provider without its own private WAN can run TANGO across the Internet between its data centers for dynamic route control. Emerging online gaming services can establish agreements with remote edge networks to optimize latency and jitter for their customers over the Internet, without investing in on-premise infrastructure or pairing with a cloud provider [80]. Edge networks leveraging federated learning to train models without sharing data with each other or with a cloud provider [19] can use TANGO to optimize communication between each participating network and the parameter server, which is often a bottleneck [46] or even security hazard [63]. Finally, datacenters running miners, validators, or decentralized exchanges can use TANGO to improve their pair latency and path diversity (hence their performance and security [13, 14, 29, 48, 60, 71]), without sacrificing their decentralized nature by moving to a single cloud.

As an end-to-end system, TANGO allows edges to optimize interdomain traffic according to their desired objectives, providing them with the means to *(i)* forward their traffic through paths they did not know existed; *(ii)* accurately measure relative loss and delay even in the presence of adversaries; and *(iii)* securely reroute traffic in real-time. This work does not seek to innovate on the *path-selection* algorithm, nor does it make claims on its stability and optimality. In fact, we find that TANGO can yield non-trivial benefits for TANGO edges in the wild even with a simple control loop, *e.g.,* selecting the path that maintains significantly lower latency for over 100ms. While, in theory, greedily optimizing routes based on local preferences might impact path conditions, we believe that in practice, independent edge pairs are less likely to affect other traffic by congesting links, especially compared to alternative large cloud systems with heavier traffic loads [43, 61, 76].

To reap all these benefits, two cooperative edges only need to *(i)* deploy lightweight TANGO logic at their border gateway, which controls routing between the cooperative edges and *(ii)* have access to a BGP speaker which can advertise a set of prefixes. While TANGO is highly deployable, since it can run on either a programmable switch or with eBPF, its modular design further eases adoption. In fact, each or a subset of the components can be independently deployed and directly benefit adopters. For example, TANGO's trustworthy telemetry scheme can be independently used to reliably measure loss and delay, verify service level agreements (SLAs), or detect violations of network neutrality. Similarly, clouds and distributed enterprises can use our TANGO's path-finding algorithm alone to expose paths they did not know existed.

In our ethically-conducted Internet-scale experiment between 23 pairs of TANGO nodes in globally-distributed Vultr data centers [5], TANGO's automated path discovery tool exposed 3-12 distinct paths beyond the BGP default. Interestingly, for 20 pairs, one or more TANGO-uncovered paths outperformed the default for 75-100% of the time, with some improving one-way latency by up to 39% (§6.1). Meanwhile for 6 pairs, an alternative path improved latency by at least 20% or more for over 10 hours on average. We also estimated with large-scale simulations across 999,000 randomly chosen pairs in the Internet topology that TANGO can expose at least two new paths for 98.6% of tested pairs, without collaboration from the Internet core (§6.2). Finally, we ran TANGO end-to-end between a switch and eBPF deployment on two continents, showing the practicality and performance of our real-time routing control in the wild (§7).

Contributions Beyond a Preliminary Workshop Paper: This work builds on our previous position paper that outlined the pressing need for edge networks to control their Internet routing and presented the high-level idea of cooperative routing [21]. This paper extends our earlier work by *(i)* developing an *automated* mechanism for identifying and exposing paths with no topology information; *(ii)* designing and implementing a data-plane mechanism for reliable monitoring that malicious on-path attackers cannot deceive; *(iii)* conducting an Internet-scale measurement study showcasing the benefits of real-time route control, in addition to a large-scale simulation showing the route diversity TANGO can expose.

2 TANGO Problem Setting

In this section, we explore important challenges of optimizing routing in today's Internet and highlight key requirements for a secure and practical interdomain route-control system.

2.1 Challenges of Today's Internet

Lack of Route Control: Despite the rich path diversity of the Internet (§6.2), the default interdomain routing choices of an edge network are limited to its direct neighbors. With standard BGP, each AS only exposes one path to each neighbor independently of performance, and so a single-homed network has no choice beyond the *single* BGP route its direct provider offers for each destination IP prefix. Meanwhile, a multi-homed network might only select among very few providers, which can have common bottlenecks, making such solutions limited [10, 15]. Any source routing protocol or multipath extension to BGP requires the participation of multiple ASes, making deployment difficult. Similarly, MPTCP only operates once paths are exposed, and it is protocol-specific.

Inaccurate Measurements: Collecting accurate performance measurements that are suitable for comparing wide-area paths is challenging. First, end-to-end measurements are often dominated by issues in the edge network (*e.g.,* wireless interference or local congestion) or on the end hosts themselves (*e.g.,* an overloaded machine) which, in essence, add noise to the wide-area path performance. Second, bidirectional metrics such as round-trip time (RTT)—whether collected by end-hosts or by network devices—are hard to decompose into separate metrics for the two one-way paths. Instead, separate measurements are required for path selection which is naturally one-way. Finally, measurement strategies often rely on protocol semantics (*e.g.,* TCP sequence and acknowledgment numbers), which do not generalize to all traffic, *e.g.,* QUIC [44], thus reducing the chances of reliable passive measurements or even ignoring the performance of certain protocols.

Untrusted Network: Route control over the public Internet (*i.e.,* an untrusted network) requires consideration of on-path adversaries. On-path adversaries may try to fool the monitoring infrastructure (or any data-driven system [49]) for monetary gains (*e.g.,* to attract more traffic to their path to generate higher revenue, perform traffic analysis, or hide poor performance to avoid SLA violations) [18]. Secure monitoring using cryptography at scale is very challenging in today's networking hardware.

Poor Deployability: The many proposals for optimizing Internet routing are notoriously hard to deploy in practice, creating a pressing need for low-cost and readily deployable solutions. Existing approaches often require core networks to run a new variant of BGP [26], deploy additional overlays [8], or run an entirely new routing protocol [56]. For instance, public overlay networks (*e.g.,* RON [12]) and future Internet architectures (*e.g.,* SCION [56]) require worldwide deployment, extra infrastructure (with associated costs), end-to-end coordination, and overheads for software processing on end-hosts.

2.2 TANGO Design Requirements

To overcome these challenges in the wide-area setting, we propose TANGO, a platform allowing edge networks (*e.g.,* small data centers and enterprises) to optimize interdomain routes. The following constraints drive TANGO's design.

Incentive Compatibility (§4, §5.1): TANGO should only expect cooperation from edge networks that actively benefit from routing optimizations (*i.e.,* source and destination networks of exchanged traffic). Thus, it should be transparent and should not rely on support from ISPs or intermediate ASes in the Internet core. In addition, the entry-level investment for individual networks to use TANGO should be minimal.

Plug-and-Play Control (§4): TANGO should enable edge networks to leverage Internet path diversity without requiring them to have multiple providers/peers, knowledge of the wide-area topology or expertise in advanced routing techniques. Observe that private-WAN approaches typically rely on highly connected PoPs and knowledge of the intermediate topology to control routing over available paths [43]. This is impractical for smaller edge networks connected over the Internet, where only incomplete topology approximations are available [6, 38]. Moreover, TANGO must *automatically* discover and expose paths *without* requiring manual input from the operator.

Accuracy & Timeliness (§5.1, §5.3): TANGO should allow participating networks to accurately measure paths and react in a timely manner to changes, dynamically choosing different routes based on collected performance measurements.

Trustworthiness (§5.2): TANGO should be robust against adversaries attempting to influence routing decisions by making a path appear more performant than it is. Concretely, we assume adversaries can intercept (and thus modify) packets on at most $n-1$ of n paths, and they can observe (eavesdrop) on any path. For more details on manipulation and replay attacks possible under our threat model, we refer readers to §A.

3 TANGO Overview

Using an intuitive example, we describe key insights and innovations TANGO employs to satisfy the above requirements.

Example: Consider an enterprise network ASX in Fig. 1 that wants to *temporarily* offload real-time computing of user information to a small cloud in ASY. This cloud is particularly reliable and meets ASX's computing needs, but does not operate an edge close to ASX. Despite the existence of alternative low-latency paths from ASX to ASY, the BGP default path via AS1-AS2-AS3 incurs prohibitively high tail latency for ASX's real-time needs. ASX could benefit from using the cloud in ASY, if only it could forward its traffic via one of the alternative paths (*e.g.,* via AS5). Unfortunately, under BGP, ASX cannot use an alternative path, and relying on an SD-WAN raises concerns with privacy and cost.

TANGO edges discover new paths and tunnel traffic over them: TANGO exposes path diversity by treating IP prefixes as *routes* (as opposed to distinct destinations), and using unique prefixes to reach the same destination via distinct paths. To

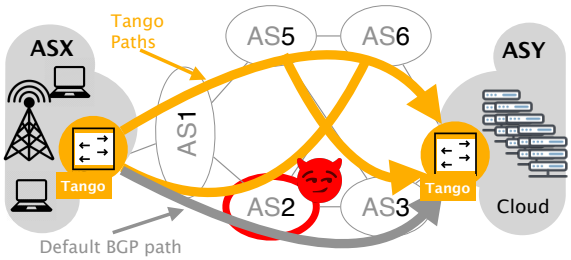


Figure 1: *ASX (enterprise) and ASY (cloud) communicate over the BGP default (gray) path via AS2-AS3.* TANGO *exploits collaboration between ASX and ASY to expose additional paths (orange).*

expose distinct paths, TANGO edges collaboratively use advanced BGP routing techniques available in today's networks (e.g., path poisoning and BGP communities). Note that TANGO *automatically* constructs BGP advertisements to discover paths making TANGO *plug-and-play* (§4). It also does not require the cooperation of any networks on the path other than the collaborative edges, and is thus *incentive compatible*.

As illustrated in Fig. 1, TANGO exposes three distinct paths from ASX to ASY by advertising three distinct prefixes from ASY, which are already installed in core BGP routing tables, but unused by BGP. The TANGO receiver (ASY) controls propagation of these advertisements through the Internet with BGP communities (§4). As a result, the TANGO sender (ASX), can control which path it uses to send traffic to ASY by tunneling application traffic over the preferred path (§5.1). ASY decapsulates tunneled packets and forwards them toward their final destination. Even though the TANGO sender can select a different path for each packet, the BGP announcements made by the TANGO receiver are **stable** and each statically represent a distinct path. Thus, there are no BGP updates when the TANGO sender chooses to change paths (preventing BGP route flapping). In practice, TANGO is symmetric, and both edges can optimize bidirectional traffic.

TANGO edges collaboratively measure delay and loss: Exposing path diversity is only the first step toward intelligent route control. ASX would need to monitor the performance of the four exposed paths to decide how to route traffic. TANGO provides highly accurate monitoring, as it operates at the edge of each network, avoiding access-network noise, *e.g.,* from wireless links. This gives TANGO an advantage over traditional end-host measurements that are notoriously inaccurate due to variable loss and delay within each network or probing techniques that can be deceived by ASes preferentially treating probes [18]. To passively measure delay and loss, TANGO adds the timestamp of when a packet left the sending edge network to every packet, along with a unique sequence number. Upon receipt, the receiving edge determines *(i) relative latency* between paths by calculating the difference between the time of packet receipt and the timestamp carried by the packet and comparing this with measurements from other paths; and *(ii)* loss by checking for missed segments (out-of-order TANGO sequence numbers).

TANGO offers trustworthy loss and delay measurements: A rational (or malicious) AS, say AS2 in Fig. 1, might try to fool TANGO into routing traffic through her infrastructure, not by improving the performance of her network but by compromising the monitoring or rerouting infrastructure. Since she cannot preferentially treat monitoring packets, as all packets are used for monitoring, she will try to fake lower delay by modifying the timestamps carried in the packets. Although adopting typical security primitives (*e.g.,* signatures, encryption) is challenging due to memory and computation constraints of modern high-speed hardware (*e.g.,* programmable switches), TANGO protects both timestamps and sequence numbers of each packet from tampering. To do so, TANGO leverage multiple insights. First, observe that timestamps and sequence numbers progress predictably. This allows TANGO to precompute and prepopulate signatures with more flexible, memory-rich software. Additionally, observe that adversaries want to make their path look superior and thus have nothing to gain from replaying old signatures (*e.g.,* from old timestamps). This enables TANGO to be resilient against replay attacks.

TANGO supports fast and secure route updates: While one-way measurements are collected at the TANGO receiver (ASY in Fig 1), the TANGO sender (ASX) decides which path packets will take *e.g.,* based on per-class performance objectives. Instead of sending raw or summarized measurements back to the TANGO sender (ASX), the receiver node ASY computes the best path for each traffic class according to ASX objectives and freshly collected measurements. If the newly computed best path is different from the current one, ASY issues a separate route update to the sender in the data plane, broadcasting the update packet over all paths to the sender, for increased update reliability. To prevent an on-path adversary from tampering with the reroute updates, we use one-time-pads directly in the dataplane.

We stress that the need for secure data-plane measurements is not particular to TANGO. In fact, many data-driven systems have been shown to be vulnerable to on-path adversaries [49]. Still, existing solutions focus solely on making monitoring scalable and accurate rather than secure [15, 35, 47, 79], effectively overlooking trustworthiness.

4 Unveiling Path Diversity with PATHFINDER

TANGO surpasses the limited path diversity offered with BGP by employing a novel recursive algorithm, PATHFINDER, which exposes BGP-compliant paths via *static* BGP announcements for different IP prefixes. Adhering to TANGO's design requirements (§2.2), PATHFINDER does not assume collaboration of the Internet core or knowledge of the wide-area topology. PATHFINDER runs on two TANGO edges with the minimum capability of announcing a single BGP prefix from one edge, and observing the AS path(s) for that prefix from the other edge. Increasing the test prefixes reduces the time it takes for PATHFINDER to expose all paths, but a typical duration using a single test prefix is $\approx$ 30 min. PATHFINDER will run at bootstrap of TANGO to expose paths and rerun on rare occasions *e.g.,* if the TANGO

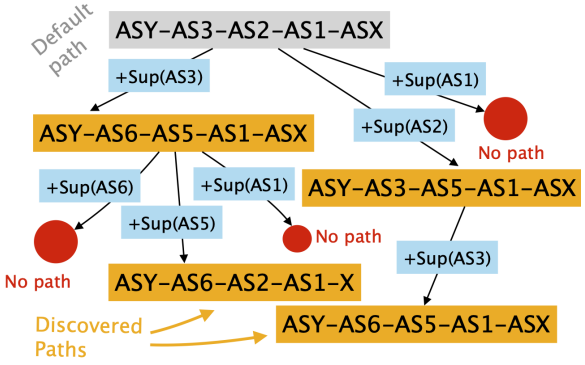


Figure 2: *Finding paths via iterative advertisement suppression.*

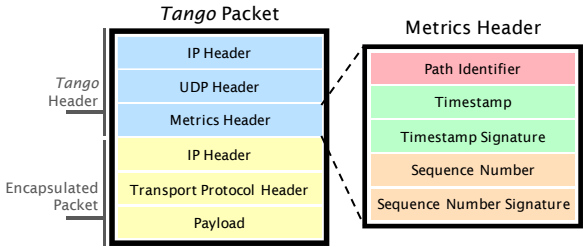


Figure 3: TANGO *tunnels application packets in an IP header specifying an interdomain path, a static UDP header to ensure consistent ECMP behavior, & a metrics header with signatures for integrity protection.*

sender receives a BGP update regarding a route exposed by PATHFINDER. Rerunning TANGO does not require TANGO to go offline as long there is at least one unused BGP prefix.

PATHFINDER exposes previously unknown paths through the Internet by advertising prefixes to the TANGO sender edge while *strategically blocking (suppressing) export of the BGP best-path*. PATHFINDER leverages two commonly supported route suppression methods: *(i)* BGP communities and *(ii)* BGP path poisoning. Community-based filtering involves attaching BGP communities supported by major transit providers (*e.g.,* those discussed in [24]) to suppress route exports (via no-export communities) or to lower route preferences so certain ASes do not use previously preferred routes. To support community-based filtering, PATHFINDER needs to be keyed with specific values of action communities supported by its upstreams and major transit providers. This can be obtained from publicly available routing guides. BGP path poisoning exploits BGP loop detection to prevent select ASes on the original path from importing the BGP announcements [22, 57]. While these techniques have different topology-dependent trade-offs[1], they are largely interchangeable from PATHFINDER's perspective, both accomplishing the algorithmic objective of suppressing a given BGP route.

PATHFINDER recursively updates the BGP advertisements announced by one edge (the destination) based on real-time feedback from the other edge (sender). Specifically, PATHFINDER finds unidirectional paths between two nodes: a traffic source and destination. PATHFINDER starts by making a "default" BGP announcement from theTANGO destination. This announcement reaches the TANGO source, which records the AS-path associated with this announcement. Note that without PATHFINDER, this would be the default and only BGP path available to the sender. Next, PATHFINDER *suppresses* the propagation of this route (using communities or path poisoning) to every AS on the recorded AS-path, effectively forcing the advertisement to find a different path to the source. For each path it finds, it recursively applies this algorithm.

[1]BGP communities can suppress individual links but are not honored over provider-customer or peer-peer links while AS-path-poisoning only suppresses at the AS granularity but affects the entire path.

As an illustration, to discover the paths in Fig. 1, ASX and ASY would need to jointly construct a graph like that in Fig. 2. In this graph, nodes contain BGP paths between ASX and ASY. The root node represents the default path, while leaves capture paths discovered by PATHFINDER by supressing ASes in the edges of the path to the node. Red dots represent advertisement attempts that did not result in a path (*i.e.,* the advertisement did not reach the sender)[2]. Recall from Fig. 1 that the default path from ASX to ASY is via ASX-AS1-AS2-AS3-ASY (advertisements are propagated via ASY-AS3-AS2-AS1-ASX). To discover additional paths, ASY will propagate multiple routes starting from one that suppresses the propagation to AS3. By poisoning AS3, the advertisement will follow the path via ASY-AS6-AS5-AS1-ASX, and thus this will be the route that ASX hears. Next, ASY suppresses AS6 in addition to AS3 which results in no available path to ASX (ASX will hear no route), as all paths traverse either AS3 or AS6. Having fully investigated routes that suppress AS3, PATHFINDER backtracks to advertising a new route suppressing AS2 and then AS1.

The algorithm described above finds paths between senders and receivers that are distinct at an AS level. Thus, PATHFINDER does not leverage path diversity *within* each AS, or the existence of multiple peering locations for an AS pair. These paths could be found by combining PATHFINDER with Paris Traceroute [16]) and could be used by TANGO by applying the source and destination port combos found by Paris Traceroute in the outer UDP header of the TANGO packets (the outer headers on TANGO packets are ignored by the receiving switch). However, the use and exploration of additional intra-domain paths is out of scope.

5 Secure, Metrics-Informed Dynamic Routing

With newly-exposed path diversity, TANGO can dynamically route traffic along paths best suited for given performance objectives, while being informed by fine-grained metrics. There are several challenges to accurate and trustworthy monitoring and rerouting. TANGO overcomes them with custom monitoring

[2]The graph is for illustrative purpose only and is constructed based on the information that PATHFINDER extracts (*i.e.,* is not part of its input).

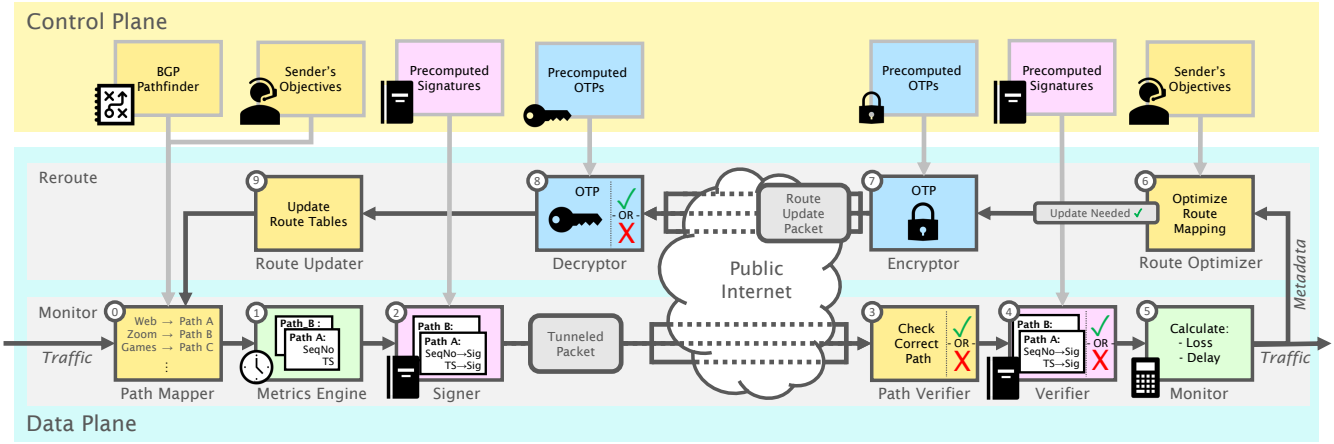


Figure 4: TANGO *is a hybrid control and data-plane approach to performant routing over the public Internet, informed by fine-grained, trustworthy telemetry.* TANGO *relies on cooperation: both edge networks share secret keys and per-class performance objectives.*

of one-way metrics (§5.1), trustworthy telemetry (§5.2), and real-time, tamper-proof route updates (§5.3). Fig. 4 illustrates the life cycle of a packet through each of these modules from a TANGO sender (left) to a TANGO receiver (right). We assume that TANGO routes traffic at the granularity of *traffic classes* to satisfy application-specific requirements.

5.1 Multi-Path Monitoring

Tunneling for Multi-Path Routing ⓪: TANGO tunnels packets through different physical paths by encapsulating them in a distinct per-path header (IPv4 or IPv6 headers) as shown in Fig. 3.[3] To route traffic according to its operator-designated traffic class objectives, TANGO maintains a mapping from class to path identifier (PID), which uniquely designates a path, specified by an IP header.

Custom Per-Path Monitoring ①⑤: In addition to the tunnel header, TANGO adds custom header fields to enable per-path performance monitoring [42]. Unlike prior works that rely on TCP semantics to measure performance [15, 28, 51, 62], TANGO adds a custom *Metrics Header* which contains a 3-bit PID, 12-bit timestamp, 32-bit PID and timestamp signature, 8-bit TANGO sequence number, and 1-bit sequence number signature. The *Metrics Header* is lightweight (7 bytes overall) and used to identify packet routes, calculate loss and delay, and facilitate trustworthy telemetry.[4] The timestamps and sequence numbers are defined as follows:

1. **Timestamps:** The TANGO sender tags a packet with its local time (t_1) in ms before tunneling it to its peer. Upon receipt, the peer node records its local time (t_2) and calculates the per-packet delay as $t_2 - t_1$. The timestamp carried by packets is at the ms granularity. Since Internet path latencies are on the order of tens of ms, more fine-grained measurements would not yield increased benefits. Also, since TANGO measures relative latency across paths, clock skew between nodes is irrelevant.
2. **Sequence Numbers:** The TANGO sender also tags packets with a monotonically increasing sequence number (*s_curr*) before tunneling. The receiver tracks the highest sequence number seen (*s_seen*), calculating loss as $s_curr - s_seen$. As TANGO sequences are not re-transmitted if dropped (unlike TCP sequence numbers), each dropped packet is only counted once. Out-of-order packets increase loss but should be rare among packets with a fixed header traveling edge-to-edge.

Aggregated Monitoring ⑤⑥: TANGO calculates per-path loss and delay metrics over an aggregation window of size *i*, by adding measurements to an aggregate sum. Upon the arrival of the *i*-th packet, the current sum will be persisted for later use in issuing reroutes, and the value will be reset. The operator may configure the window size, weighing tradeoffs in noise-resilience, network event response time, and security.

5.2 Secure Telemetry

Trustworthy Telemetry ②③④: To protect against tampering, the TANGO sender cryptographically signs all timestamps, PIDs, and sequence numbers, ensuring integrity and authenticity. Upon receipt, the TANGO receiver verifies each value to ensure they are untouched after transit over the public Internet. Signatures are path-specific, and hence secure against replay attacks: an on-path attacker cannot replace signatures with those they overhear from faster paths, as the signature is specific to the path (PID). Finally, any mapping (*i.e.,* pair of timestamps, sequence number, or PID to signature) that the adversary learns cannot be reused. The

[3] The TANGO header adds several bytes to each packet, which could potentially cause MTU issues. TANGO can resolve this in a similar way to other router-based encapsulation protocols by implementing Path MTU Discovery and responding with appropriate ICMP "fragmentation needed and DF set" packets, as specified in RFC 1191 [50].

[4] We developed these header sizes based on the parameters of the testbed we used. Even under more demanding production conditions, the anticipated header size will still be quite small compared to overall packet sizes.

sequence number changes per packet. The timestamp changes every ms and older timestamps are not useful as the attacker only tries to make her path appear faster.

Practical Challenges for Scalability: Cryptographically protecting metrics at packet line rate is nontrivial. Cryptographic primitives are resource-intensive to implement at scale. First, signing packets in the control plane requires laborious per-packet processing in software, while signing packets in the data plane would require multiple recirculations [27, 77, 78]—both too compute-intensive. Second, storing *precomputed* signatures directly on switch SRAM is too memory-intensive. Further, populating the data plane with precomputed signatures from the control plane creates read-write synchronization concerns between the data and control planes, as well as between TANGO sender and receiver.

The TANGO Signature Book ②④: To avoid online signature computation while still being memory-friendly, TANGO creates a *signature book* of precomputed signatures, and periodically populates smaller precomputed *signature blocks* from the signature book to the data plane. The signature book is used for two operations: signing (on the TANGO sender) and verifying (on the TANGO receiver). The following insights help us deal with the memory challenge. First, we observe that both sequence numbers and timestamps are predictable (monotonically increasing) over time, enabling efficient pre-computation and storage. Second, we observe that to reduce delay the adversary would only need to guess one timestamp per ms, while to hide a single packet drop she would need to correctly guess many consecutive signatures. Further, sequence numbers will eventually wrap around, thus making old mappings irrelevant to the adversary. Hence, TANGO uses 32-bit timestamp signatures and 1-bit sequence number signatures.

To deal with synchronization challenges, TANGO employs two strategies. First, it uses data-plane packets triggered by the control plane to quickly write signatures, instead of relying on control-plane write calls (*e.g.,* with gRPC). The control plane marshals precomputed signatures into large packets, which trigger block writes when processed in the data plane. Switch-specific constraints do not allow writing multiple indexes of the same register array, so signatures are sliced off packets in the data plane, which are recirculated until a null token is reached. This approach allows the control plane to update signatures faster than they are consumed by the data plane.

Second, TANGO splits its signature book into two blocks, where at any given time, one block is being written and one is being read, avoiding a race condition between reads and writes (see Fig. 5). This approach makes the sequence-to-signature mappings time-dependent and thus deterministic. Critically, however, sequence numbers are not reset for every new interval. While resetting would be more economical from the memory perspective, it would also allow the attacker to silently drop all packets at the end of each interval. Instead, TANGO stores the maximum signatures that can be consumed in each interval and ignores unused signatures. In other words, through every interval change, the sequence number continues to increase from its last value (Fig. 14 in §B).

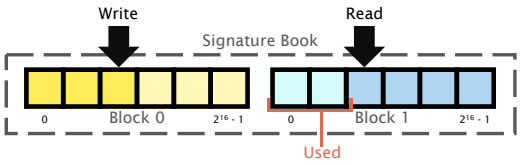


Figure 5: *To deal with synchronization issues between reads and writes,* TANGO *splits the signature book into two blocks, for reading and writing. The control plane writes fresh signatures to one block, while the data plane consumes the other for signing packets.*

5.3 On-Demand Reroutes in the Data Plane

Dynamic Route Updates ⑥⑨: The receiving TANGO node issues route updates to the sender based on aggregated path metrics and the sender's objectives for each traffic class. As reroutes should be infrequent, TANGO does not include these updates in the default TANGO header, rather opting for a custom packet containing the route update (*i.e.,* the new PID for the corresponding traffic class).

Trustworthy Route Updates ⑦⑧: Naturally, a route update could be the target of an on-path adversary that *(i)* tampers with route updates to direct traffic to desired paths; *(ii)* injects route updates to move traffic from a path; or even *(iii)* drops route updates to cause a denial-of-service to the reroute mechanism. Observe that the first two attacks are catastrophic for TANGO, since they have a direct effect on routing. Thus, to prevent tampering, the TANGO receiver (which issues the updates) encrypts route updates before transmission, while the original sender decrypts and verifies the update before applying. To also protect against dropping of route updates, TANGO broadcasts reroute packets over all available paths to ensure that at least one update will reach the sending node.[5]

OTP and Encoding Scheme: There are several challenges to encrypting updates: *(i)* limited compute and memory in the data-plane; and *(ii)* susceptibility of small ciphertexts to brute-force attacks. TANGO solves these problems by using precomputed OTPs with extension encoding scheme. This is a natural solution as route updates are relatively infrequent yet unpredictable. However, since the number of possible reroutes increases with both the number of paths and the number of traffic classes, storing route-update signatures could be memory inefficient. Most importantly, compromised security of route updates would be catastrophic even for a single packet, necessitating the perfect secrecy of OTP.

Concretely, a route update consists of an index that selects the OTP to XOR with the concatenated traffic class and PID (see Fig. 15 in §B). The control plane periodically updates OTPs, such that they are only used once. While OTP offers perfect secrecy, it is vulnerable to bit flips, meaning the attacker can change a few random bits in the traversed route update to change TANGO

[5] Threat model assumes at least one path with no on-path adversary (§2.2).

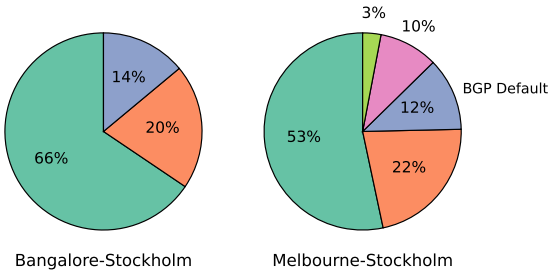


Figure 6: *Breakdown over 32 hours of how often a path had lowest relative latency. Here, the BGP default is beaten by alternative paths 100% of the time (Bangalore) and 88% of the time (Melbourne).*

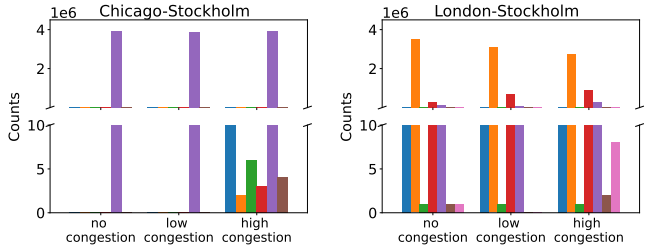


Figure 7: *Number of times different paths emerged as the best for a given congestion condition, per node pair. "no", "low", and "high" congestion conditions correspond to the lowest, middle, and upper tertiles, respectively, of one-way-delay experienced by the best path. While one path might dominantly be best under no/low congestion, during high congestion more paths emerged as the best path.*

behavior. To protect against this, TANGO independently encodes the traffic class and PID into 32-bit, sparse strings. The value to encode (*e.g.,* traffic class) selects a static, sparse-bit string and appends it to form a single, 32-bit string. The concatenation of the encoded bit strings are XOR'd with the OTP, and sent to the peer TANGO node alongside the update number. If an adversary tampers with the update number, traffic class bits, or the PID bits, the decrypted update or the encodings will be incorrect, and the update can be safely ignored. If the adversary were to now try and brute-force the update, they must now brute-force all encoding bits, which are tied to encoded values.

6 Internet-Scale Measurements

We showcase TANGO in the wild by deploying it with eBPF [17](§7) on 25 nodes in globally-distributed data centers of the cloud provider Vultr [5] (see Fig. 26 in §C). Through our Internet-scale measurement study (§6.1), we prove that TANGO uncovers *path diversity* even in a single-homed environment, where conventional BGP only offers a single path. We also show that TANGO uncovers *performance diversity* in these alternate paths, which often outperform the default BGP path by up to 100% of the time with up to 39% lower latency. To further validate and generalize these results, we also perform an Internet-scale simulation that confirms the rich path diversity available in the public Internet (§6.2). Together, these experiments showcase the benefit, practicality, and incremental deployability of TANGO.

Ethics Statement: We note that all testbed infrastructure in edge networks of our Internet-scale study are operated by the authors, and that traffic sent from TANGO nodes across the public Internet are transparent to intermediate ASes and can be processed as normal application traffic at reasonable sending rates (up to 1Mbps), raising no ethical issues.

6.1 Operational Deployment

Choice of Edge Network. We perform our measurement study with Vultr as a deployment convenience, however, many other types of edge networks can easily benefit from TANGO. Characteristics that would make an edge network particularly amenable to TANGO include having available prefix space (either IPv4 or IPv6) and either BGP communities *or* path poisoning capabilities. Note that the edge network does not require a provider that actively *engages* in BGP community announcements, as long as they are able to *transit* communities. This is often the default behavior in ISPs [68, 69].

Methodology: To measure available path diversity, we ran PATHFINDER on all bidirectional paths between 24 of the Vultr data centers (552 pairs in total)[6]. Vultr allows customers to make BGP announcements, supports several action communities (to control its own BGP behavior), and transits BGP communities (to potentially impact the behavior at remote ASes) [3, 4], but does not allow customers to do path-poisoning. Thus, we ran PATHFINDER using community-based suppression, inputting BGP suppression communities supported by Vultr as well as by several major transit providers (specifically ASes 3257, 6453, 4755, 3356, 1299, and 174).[7] While Vultr does export a full BGP route table to customers participating in its BGP services, it only provides a single next-hop at each datacenter: the Vultr upstream router. Thus, *by default, each source-destination pair would only have a single path, without* TANGO.

We also used TANGO to measure performance diversity between 23 Vultr pairs, with Vultr Stockholm fixed as the receiver. Specifically, we generated 1Mbps `iperf` UDP flows across 7 different TANGO-exposed interdomain paths for all 23 pairs, passively measuring latency and loss at 10 ms measurement intervals over a period of roughly 32 hours. We also repeated measurements with an additional 23 node pairs, using Vultr LA as the fixed receiver, and found similar results (§C). We present results for Stockholm measurements below.

6.1.1 Path Diversity over the Public Internet

How much path diversity can TANGO expose? Of the 552 node pairs explored, PATHFINDER exposed alternative paths for *503 pairs (91%), unveiling opportunities to benefit from multi-path routing*. The median number of paths available between two

[6] During PATHFINDER exploration, only 24 of 25 nodes were available.

[7] As there are no standard values for many BGP route suppression communities [24] these values had to be found by hand from routing guidelines.

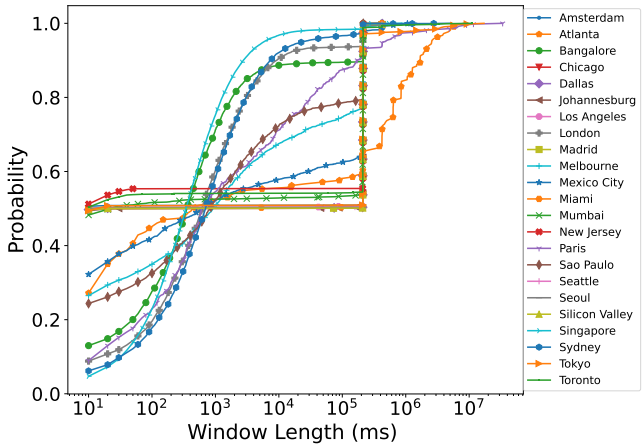


Figure 8: *On average across all 23 pairs, best paths lasted from 5-541s, with median durations of 10ms-105s.* TANGO *only chooses to perform dynamic rerouting for windows longer than 100ms.*

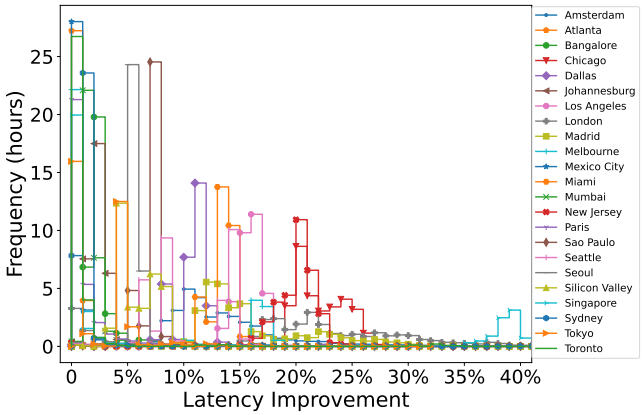


Figure 9: *Best paths that* TANGO *exposed outperform the BGP default by up to 22% on average, and up to 39% for some pairs.*

nodes was 3, and the average was 3.3. PATHFINDER also uncovered a sizeable long tail, as 84 different pairs had 6 or more paths and 3 pairs had 12 different paths between them (see Fig. 16 in §C). The average run time for PATHFINDER was quite low (<1h) as each pair only required 6.8 BGP announcements on average (with 5-min separations to account for propagation). While already encouraging, the number of paths can be significantly increased if Vultr were to allow BGP path-poisoning. Unlike community-based suppression which is only supported by some ASes and varies in implementation from AS to AS, AS-path poisoning is mandated by the BGP RFC [59], although some networks filter announcements with certain ASes poisoned [66].

6.1.2 Performance Diversity across Exposed Paths

How often are TANGO-discovered paths better than the default? We define the best path to be the one with lower relative one-way-delay compared to all other paths. Of the 23 pairs in our Vultr Stockholm measurements, *20 pairs had an alternative path that outperformed the default BGP path for a significant amount of time*: 100% of the time for 15 pairs, and 75-88% of the time for 5 pairs. This clearly shows the benefit of deploying TANGO. Of the remaining three pairs (Atlanta-Stockholm, Paris-Stockholm, Tokyo-Stockholm), the default path was dominantly best for only two pairs and was best only 55% of the time for the last pair. We also note that there were many pairs with *more than one* alternative path which outperformed the default, providing further path diversity and potential performance resilience: 7 pairs had two or more alternatives and 4 pairs had three or more alternatives. In Figs. 6 and 7, we visualize results for two node pairs, showing alternative paths outperforming the default. We include distribution charts for all 23 pairs in §C, Figs. 23 and 25.

How long is one path the best? We measured the window of time one path remained the best for each of the 23 Vultr pairs (Fig. 8). Path longevity varied across paths, likely due to performance variations from changing network conditions (*e.g.,* congestion). For 9 pairs, the median window duration was 350-925*ms* showing that dynamically updating the used path would be beneficial for forwarded traffic. For 11 pairs, the median duration was shorter at 10-30*ms*, most likely due to ECMP behavior between the nodes. The average duration for these 11 pairs ranged between 95-99*s* (from New Jersey, Mumbai), 121-221*s* (from Toronto, Tokyo), and 105-107*s* for the 7 other pairs. For the last 3 pairs, the median was much longer, lasting for 23-105*s*, while the average duration for each pair was 107s (from Johannesburg, Madrid, Chicago).

TANGO performs dynamic reroutes when a new best path emerges for longer than a given window threshold. The exact threshold can be configured per pair using such measurements. For instance, with this threshold, 10-30ms windows would not trigger a route update. Network operators can increase this threshold to provide more static long-term path optimizations, or further lower it for even faster dynamic updates.

By how much does the best path beat the BGP default? We measured the latency difference between the best and default paths, for each of the 23 Vultr pairs (Fig. 9). We found that for five pairs, the best path had at least 20% lower latency than the default for more than 6 hours within the 32-hour experiment. To better grasp the expected benefit of TANGO, consider that these nodes are in different sites of the same cloud. While as noted previously, there were two pairs for which the default path was best (Atlanta-Stockholm and Paris-Stockholm), for the remaining 21 pairs, *the non-default best path outperformed the default by an average of 1-22%* for durations of 0.9-24.54 *hours*. Further, the 75th percentile delay improvement was 24% for Chicago-Stockholm and 26% for London-Stockholm, lasting for 4.07*h* and 0.85*h*, respectively.

6.1.3 Event Analysis

While Fig. 9 demonstrates the benefit of running TANGO for the average-case, another major advantage of TANGO is to rapidly avoid problematic performance events by moving to other paths during disruptions. To quantify this benefit, we searched our measurements for significant performance degradation

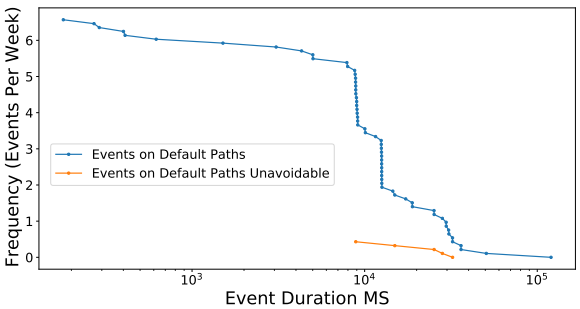


Figure 10: *Occurrence frequency vs. duration of high-loss performance degradation events on default BGP paths paths.* TANGO *could protect edges from four high-loss events on average per week.*

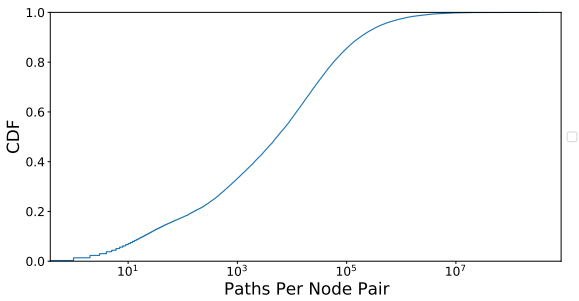


Figure 11: *The number of Gao-Rexford-compliant paths between 999k random topology node pairs.* PATHFINDER *exposed more than 2 unused BGP-compliant paths for 98.6% of pairs.*

events by filtering for 10 consecutive measurements (i.e., a 100ms window, with data points taken every 10ms apart) which were 20ms higher than the baseline one-way-delay on that path. We determined an event to be over once 10 consecutive measurements were below our event threshold. For each event, we computed the relative average one-way-delay and loss during the event. We also marked each event as avoidable if we saw there was another path between the same pair of nodes that was not experiencing a performance degradation event. We computed the frequency of events on paths by dividing the total number of events by the total duration of measurement collection.

From there, we further selected events with more significant performance degradation in loss or delay, potentially compromising real-time applications. We do expect that such events would cause customers to complain even if the average network performance is good. Concretely, we selected events with greater than 20% loss or 100% one-way-delay increase (latency events are discussed in §C.2). Fig. 10 presents a graph of event duration vs. frequency for loss events on default BGP paths (TANGO-discovered paths experienced similar events). In the absence of TANGO, events with 20% loss for longer than 8 seconds occurred over 5 times a week. Meanwhile, with TANGO-exposed paths and adaptive routing, such loss events can be reduced to *less than once a week on average*. It should also be noted that in our dataset, all unavoidable events occurred at a single sending node: Vultr Brazil (which may have seen a higher number of correlated events due to potentially less path diversity in developing regions like Brazil). *For all other nodes, every high-loss event encountered could have been avoided with* TANGO.

6.2 Internet-Wide Simulation

To understand potential path diversity across the Internet topology in an even more generalized setting, we counted Gao-Rexford-compliant [31] paths between randomly-chosen ASes in the March 2020 CAIDA Internet Topology Dataset [6]. To optimize the counting process, we separated a Gao-Rexford path into two sections: an initial part consisting of zero or more customer-provider links joined (optionally by a peer-peer link) to zero or more provider-customer links. With this in mind, for each BGP destination considered, we divided the topology into two sections: a "provider cone" that was reachable only traversing customer-provider links, and all other ASes outside the provider cone. We counted paths to all ASes in the provider cone via only customer-provider links by treating this as a Directed Acyclic Graph (DAG) based on the Gao-Rexford premise that there are no customer-provider loops [31]. We similarly counted paths from all other ASes to the source AS by treating the region outside the provider cone as a DAG but traversing provider-customer links (instead of customer-provider links). This gave us a path count to all ASes in the topology using either customer-provider or provider-customer links depending on whether the ASes were inside or outside the provider cone. Finally, we joined these two counts to form full paths by traversing each AS A in the provider cone and counting the potential paths that contained A in the last customer-provider link (the sum of the paths through ASes not in the provider cone of the peers and customers of A multiplied by the number of paths through the provider cone to reach A). This algorithm is scalabe and ensures counting distinctness, it is actually a lower bound on the number of paths in a given topology, as ASes in the provider cone can potentially be reached over peer-peer or provider-customer paths as well, which our counting methodology does not permit. Our counting is also limited by the accuracy of the CAIDA AS-Relationship dataset as is standard with many other Internet-scale simulation work [22–24, 70].

We ran our counting algorithm to count Gao-Rexford-compliant paths between 1000 randomly-chosen ASes producing 999,000 distinct source-destination pairs. We found today's Internet topology offers rich path diversity, as the median AS pair had 5,323 Gao-Rexford-compliant paths (Fig. 11). We largely attribute this richness to the high degree of certain vertices, including large IXPs, on the Internet graph; indeed, in a richly-connected topology paths grow exponentially. We, of course, do not advocate for TANGO to use more than a couple of them, yet this result shows high potential for multi-path routing across the wide area.

7 Internet-Scale Route Control

We showcase TANGO's real-time route control with an end-to-end experiment spanning two continents (North America and Europe), illustrating how TANGO leverages collected measurements to perform dynamic route control and avoid performance degradation

events (§7.1). We also present microbenchmarks that highlight the operational feasibility and efficiency of performing line-rate integrity protection with TANGO's switch prototype (§7.2).

Implementation: We implemented TANGO's data plane logic on modern switch hardware and with eBPF on standard Linux servers. Our switch prototype was implemented in 199 lines of Lucid code [67] and compiled to 1279 lines of P4 [25] targeted for an Intel Tofino programmable switch [36]. Our interchangeable eBPF version was written in 401 lines of code. Meanwhile, TANGO's control plane component and eBPF loading program was written in 524 lines. We have released TANGO's source code on GitHub[8].

Testbed: Our hardware testbed consisted of an Intel Tofino Wedge32X-BF programmable switch and several servers, each with a 20-core Intel Xeon Silver 4114 CPU and a Mellanox ConnectX-5 2×100Gbps NIC. Our sending edge network was based on Princeton University's campus in North America, with a server (running Ubuntu 20.04 and kernel version 5.4.0) sending main application traffic and generating background traffic flows, and the switch running TANGO data plane logic. Meanwhile, TANGO's receiving edge was deployed with eBPF on a standard Linux server at a Vultr data center in Stockholm, Sweden running Ubuntu 22.10 with kernel v5.19.0. The eBPF programs were built with Libbpf and Clang 15.0.6.

7.1 Dynamic Reroutes

To evaluate TANGO's end-to-end dynamic route control, we announced seven distinct IP prefixes from Vultr Stockholm via BGP, using community sets that we previously found were optimal for finding paths between Stockholm and other locations (§6.1). We did not have access to a BGP feed from our institution so we could not rerun our PATHFINDER algorithm specifically for this pair of nodes. We also noticed ECMP being used for outbound traffic from our institution. To benefit from this we explored the round-trip-time when sending traffic to different destination IPs within the same BGP prefix. We found two available ECMP paths for every announced BGP path. After enumerating these 14 paths, we observed some paths had seemingly-identical performance[9]. Ignoring redundant paths we had 12 distinct paths.

We generated keep-alive flows along all exposed paths while continually collecting per-path metrics. We also had one active flow, which we monitored for potential benefits from dynamic routing. We developed a simple control algorithm based on relative one-way-delay that would move the active flow to a different path if that path outperformed the current path for 10 consecutive measurements (i.e., 100ms for measurements taken every 10ms). We found this algorithm to be stable across the wide area, only rerouting during significant network degradation events. Once the control algorithm determined a route change was necessary, it produced a reroute data packet which was sent back to the TANGO switch node at our institution. This caused a register update on the switch, which rerouted the active flow to a different path.

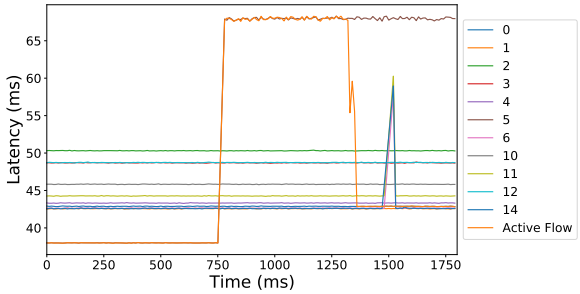


Figure 12: *One-way-delays of different paths and an active flow during our simulated performance degradation event*[10].

To observe TANGO's potential to dynamically avoid events like those discussed in § 6.1.3, we utilized another Intel Tofino switch running between our TANGO switch and the Internet to inject delay (with packet recirculations) on specific paths. To make our experiment more realistic, we replayed delay events that we had observed in the wild on paths that *(1)* had a similar one-way-delay as that of our institution to Stockholm and *(2)* had the largest number of disruptive events.

When we ran this experiment, our control algorithm started by moving the active flow from the BGP default path (path 0) to the optimally performing path (path 5). As the experiment continued, our delay mechanism began simulating a performance degradation event, causing path 5 to have its one-way-delay spike. *Within 650ms after the first sign of performance degradation, the observed active flow was moved back to path 0* (see Fig. 12). Even with a simple control loop, TANGO can reduce the duration of 90% of the high-loss events observed in §6.1.3, offering more than a 10-fold reduction in the median length (12s) of high-loss events. This response time could potentially be even faster if our prototype eBPF module generated reroute packets in the kernel as opposed to utilizing a separate user-space process (written in Python and taking approximately 400ms to initialize and execute), or if a more aggressive control algorithm was used.[11]

7.2 Data Plane Microbenchmarks

To demonstrate the operational efficiency of our integrity-protection scheme, we analyzed theoretical sizes for TANGO's signature book (§5.2), experimentally evaluating required control plane write speeds for book population. Our results prove that, for reasonable packet sizes, required block sizes refreshed every 5–15ms can comfortably be stored in the data plane and populated from the control plane (Fig. 13).

Book Block Hybrid Write Speed: We experimentally confirmed that the control plane can write an entire signature block across all

[8] https://github.com/PrincetonUniversity/tango-routing

[9] This is because we could not see the BGP path used by our institution so some BGP community combinations produced identical paths.

[10] It is likely a congestion-induced single-packet spike around 1500 ms.

[11] Our test flows ran UDP so reordering was not a prominent concern. If this rerouting algorithm is used on TCP traffic, flowlet boundaries can be taken into account to minimize overhead due to packet ordering changes.

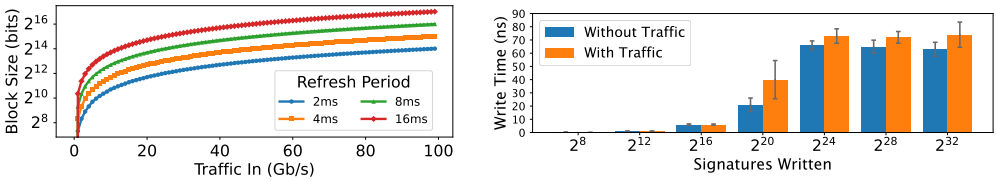


Figure 13: *Required signature block size* (left) *and write speed* (right) *reveal the practicality of a precomputed data plane signature scheme.*

books at speeds well below the refresh period, ensuring all signatures are refreshed before the data plane moves to the next block. Our first server issued control plane writes, the second generated background traffic at 100Gbps with DPDK [30], and the switch ran our slice-and-recirculate write behavior (§5.2) while tunneling out background traffic. We gathered write-time measurements by taking the time difference between the arrival of the first and last signature written to the block. As shown in Fig. 13, at a refresh period of 8ms at maximum port speed, the control plane can write 2^{16} signatures in less than 10ns over the data channel. Meanwhile, the required 2^{20} signatures for all blocks across all books can be written in approximately 20ns without background traffic and 40ns with it, while being well below the 8ms refresh limit.

8 Related Work

Performance-Driven Routing: Many prior works have explored performance routing; however, they suffer from poor deployability. Traffic engineering works such as TEXCP [39], EDGE FABRIC [61], and ESPRESSO [76] assume that the edge network is multi-homed. While EDGE FABRIC and ESPRESSO show the promise of SDN-based performance-driven routing, they can select between multiple already-existing BGP routes, thanks to global PoPs and peerings from Meta and Google. Such infrastructure is infeasible for smaller network operators and cloud providers. Other data-driven routing solutions such as BLINK [35] and SHORTCUT [64] allow fast failover in the data plane but do not improve performance. ROUTESCOUT [15] provides metric-driven dynamic routing in the data plane, but struggles to provide accurate metrics and also assumes multi-homed networks. AnyOpt [81] optimizes anycast catchment but does not advertise multiple prefixes for the same destination. PECAN [72] does advertise different routes to the same destination but only steers between them with DNS, preventing the fine-grained route control TANGO offers.

PAINTER [43] similarly uses multiple IP prefixes to advertise different routes for inbound traffic, but is designed for a cloud environment, which significantly changes the design space. First, as PAINTER runs in a highly-peered cloud, it can simply advertise distinct prefixes to different immediate neighbors, and does not attempt to find distinct paths after the initial hop (making it useless to single-homed networks). Second, PAINTER does not perform data-plane telemetry and leverages application-layer telemetry enabled by proxies running in edge networks. TANGO innovates on data-plane telemetry and does not require application proxies, while being protocol-agnostic and robust against malicious intermediate networks. Finally, TANGO is designed to be deployed in programmable data planes like P4 switches and smartNICs. PAINTER utilizes proxies running on traditional CPUs that do not have the scaling and cost benefits of data-plane hardware.

Secure Telemetry: Multiple solutions for real-time data-plane telemetry exist [45, 52, 58, 62], yet they are not designed with security in mind *i.e.,* their results could be compromised by an adversary. Moreover, they often require collaboration of all switches/routers in the path. For instance, INT (and later versions) [42, 65] collect fine-grained performance metrics at each hop, enabling informative network monitoring to operators, but require each switch to implement the protocol, which is not a reasonable assumption in the wide-area setting. A few secure telemetry solutions exist, such as Stealth Probing [18] and path-quality monitoring [32], but they do not provide the necessary fine-grained metrics for real-time, dynamic routing, and are not implementable in today's hardware.

Data-Plane Encyption: There are several general-purpose encryption schemes ported to the data plane; however, they are too resource-intensive. For example, the Advanced Encryption Standard *i.e.,* the de facto cipher for most Internet applications AES-TOFINO [27], utilizes the majority of Tofino memory resources and would need, optimally, 5 pipeline passes for each 16-byte block to encrypt. Even more lightweight cipher deployable on an ASIC such as SIMON AND SPECK [20], CHACHA [78], and HALFSIPHASH [77] are still too resource intensive. Beyond memory, they require several recirculations for every plaintext block to be encrypted. Other solutions, such as RAVEN [74] and PINOT [73], are application-specific and not easily extensible to other use-cases.

9 Conclusion

TANGO is the first route-control scheme to expose multiple wide-area paths without the cooperation of the Internet core, while offering accurate and trustworthy edge-to-edge measurements. Our Internet-wide experiments show there are significant benefits from optimizing routing on the public Internet, using TANGO-exposed paths not available with BGP. We show TANGO can run on a hardware switch or with eBPF, making it practical even for small networks.

Acknowledgments. We thank John Sonchack for his support with Lucid development. We are grateful to Ethan Katz-Bassett and Thomas Koch for their valuable feedback, and to our shepherd Italo Cunha and the anonymous reviewers for their insightful comments. This work was supported in part by Protocol Labs and by NSF GRFP Grant DGE-2039656.

References

[1] Managed sd-wan solutions for the cloud era. *https://www.aryaka.com/managed-wan-services/*.

[2] Versa networks. *https://versa-networks.com/sd-wan/cloud-wan/*.

[3] Announce your IP space with BGP and Vultr - Vultr.com. https://www.vultr.com/features/bgp/, 2022.

[4] AS20473 BGP customer guide. https://www.vultr.com/docs/as20473-bgp-customer-guide, 2022.

[5] SSD VPS servers, cloud servers and cloud hosting. https://www.vultr.com/, 2022.

[6] The CAIDA AS relationships dataset. https://www.caida.org/catalog/datasets/as-relationships/, 2023.

[7] HPE greenlake for aruba (NaaS). https://www.arubanetworks.com/solutions/naas/, 2023.

[8] Akamai. SureRoute. https://developer.akamai.com/article/sureroute, 2022.

[9] A. Akella, B. Maggs, S. Seshan, and A. Shaikh. On the performance benefits of multihoming route control. *IEEE/ACM Transactions on Networking (TON)*, 16(1):91–104, 2008.

[10] A. Akella, S. Seshan, and A. Shaikh. Multihoming Performance Benefits: An Experimental Evaluation of Practical Enterprise Strategies. In *USENIX Annual Technical Conference, General Track*, 2004.

[11] A. Alhilal, T. Braud, B. Han, and P. Hui. Nebula: Reliable Low-latency Video Transmission for Mobile Cloud Gaming. pages 3407–3417, 04 2022.

[12] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *ACM Symposium on Operating Systems Principles*, SOSP '01, page 131–145, 2001.

[13] M. Apostolaki, C. Maire, and L. Vanbever. Perimeter: A network-layer attack on the anonymity of cryptocurrencies. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part I 25*, pages 147–166. Springer, 2021.

[14] M. Apostolaki, G. Marti, J. Müller, and L. Vanbever. SABRE: Protecting Bitcoin against Routing Attacks. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[15] M. Apostolaki, A. Singla, and L. Vanbever. Performance-driven internet path selection. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, page 41–53, 2021.

[16] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding traceroute anomalies with paris traceroute. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, IMC '06, page 153–158, New York, NY, USA, 2006. Association for Computing Machinery.

[17] S. authors. Suricata - eBPF and XDP. https://suricata.readthedocs.io/en/latest/capture-hardware/ebpf-xdp.html, 2018.

[18] I. Avramopoulos and J. Rexford. Stealth probing: Efficient Data-Plane security for IP routing. In *USENIX Annual Technical Conference*, Boston, MA, May 2006. USENIX Association.

[19] G. Bao and P. Guo. Federated learning in cloud-edge collaborative architecture: key technologies, applications and challenges. *Journal of Cloud Computing*, 11, 12 2022.

[20] R. Beaulieu, S. Treatman-Clark, D. Shors, B. Weeks, J. Smith, and L. Wingers. The simon and speck lightweight block ciphers. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015.

[21] H. Birge-Lee, M. Apostolaki, and J. Rexford. It takes two to tango: cooperative edge-to-edge routing. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 174–180, 2022.

[22] H. Birge-Lee, Y. Sun, A. Edmundson, J. Rexford, and P. Mittal. Bamboozling Certificate Authorities with BGP. In *USENIX Security Symposium*, 2018.

[23] H. Birge-Lee, L. Wang, D. McCarney, R. Shoemaker, J. Rexford, and P. Mittal. Experiences deploying Multi-Vantage-Point domain validation at let's encrypt. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4311–4327. USENIX Association, Aug. 2021.

[24] H. Birge-Lee, L. Wang, J. Rexford, and P. Mittal. SICO: Surgical Interception Attacks by Manipulating BGP Communities. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[25] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. In *ACM SIGCOMM Computer Communication Review*, 2014.

[26] J. M. Camacho, A. García-Martínez, M. Bagnulo, and F. Valera. BGP-XM: BGP Extended Multipath for Transit Autonomous Systems. *Computer Networks*, 57(4):954–975, 2013.

[27] X. Chen. Implementing AES encryption on programmable switches via scrambled lookup tables. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, SPIN '20, page 8–14, New York, NY, USA, 2020. Association for Computing Machinery.

[28] X. Chen, H. Kim, J. M. Aman, W. Chang, M. Lee, and J. Rexford. Measuring tcp round-trip time in the data plane. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, SPIN '20, page 35–41, New York, NY, USA, 2020. Association for Computing Machinery.

[29] C. Decker and R. Wattenhofer. Information propagation in the bitcoin network. In *IEEE P2P 2013 Proceedings*, pages 1–10. IEEE, 2013.

[30] L. Foundation. Data plane development kit (DPDK), 2015.

[31] L. Gao and J. Rexford. Stable Internet Routing without Global Coordination. *IEEE/ACM Transactions on Networking*, 9(6):681–692, 2001.

[32] S. Goldberg, D. Xiao, E. Tromer, B. Barak, and J. Rexford. Path-quality monitoring in the presence of adversaries. In *ACM SIGMETRICS*, page 193–204, New York, NY, USA, 2008. Association for Computing Machinery.

[33] D. K. Goldenberg, L. Qiu, H. Xie, Y. R. Yang, and Y. Zhang. Optimizing cost and performance for multihoming. In *ACM SIGCOMM*, volume 34, pages 79–92. ACM, August/September 2004.

[34] F. Han, M. Wang, Y. Cui, Q. Li, R. Liang, Y. Liu, and Y. Jiang. Future Data Center Networking: From Low Latency to Deterministic Latency. *IEEE Network*, 36(1):52–58, 2022.

[35] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 161–176, Boston, MA, Feb. 2019. USENIX Association.

[36] Intel. Barefoot Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html.

[37] S. Jain. What's in a name? Understanding the Google Cloud network "edge". https://cloud.google.com/blog/products/networking/understanding-google-cloud-network-edge-points, 2021.

[38] J. Juen, A. Johnson, A. Das, N. Borisov, and M. Caesar. Defending Tor from Network Adversaries: A Case Study of Network Path Prediction. *Proceedings on Privacy Enhancing Technologies*, 2015(2):171–187, 2015.

[39] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *ACM SIGCOMM*, SIGCOMM '05, page 253–264, New York, NY, USA, 2005. Association for Computing Machinery.

[40] A. Kashaf, V. Sekar, and Y. Agarwal. Analyzing third party service dependencies in modern web services: Have we learned from the mirai-dyn incident? In *Proceedings of the ACM Internet Measurement Conference*, pages 634–647, 2020.

[41] M. A. Khan, E. Baccour, Z. Chkirbene, A. Erbad, R. Hamila, M. Hamdi, and M. Gabbouj. A Survey on Mobile Edge Computing for Video Streaming: Opportunities and Challenges. *IEEE Access*, 10:120514–120550, 2022.

[42] C. Kim, A. Sivaraman, N. P. Katta, A. Bas, A. Dixit, and L. J. Wobker. In-band network telemetry via programmable dataplanes. Industrial demo, ACM SIGCOMM '15, 2015.

[43] T. Koch, S. Yu, S. Agarwal, E. Katz-Bassett, and R. Beckett. Painter: Ingress traffic engineering and routing for enterprise cloud networks. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 360–377, New York, NY, USA, 2023. Association for Computing Machinery.

[44] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. B. Krasic, C. Shi, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. C. Dorfman, J. Roskind, J. Kulik, P. G. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, and W.-T. Chang. The quic transport protocol: Design and internet-scale deployment. 2017.

[45] M. Lee, S. Goldberg, R. R. Kompella, and G. Varghese. Finecomb: Measuring microscopic latency and loss in the presence of reordering. *IEEE/ACM Transactions on Networking*, 22(4):1136–1149, 2014.

[46] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith. Federated learning: Challenges, methods, and future directions. *IEEE signal processing magazine*, 37(3):50–60, 2020.

[47] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A better NetFlow for data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 311–324, Santa Clara, CA, Mar. 2016. USENIX Association.

[48] A. Maria, Z. Aviv, and V. Laurent. Hijacking Bitcoin: Routing Attacks on Cryptocurrencies. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[49] R. Meier, T. Holterbach, S. Keck, M. Stähli, V. Lenders, A. Singla, and L. Vanbever. (self) driving under the influence: Intoxicating adversarial network inputs. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 34–42, 2019.

[50] J. Mogul and S. Deering. Path mtu discovery. RFC 1191, RFC Editor, November 1990. http://www.rfc-editor.org/rfc/rfc1191.txt.

[51] G. C. M. Moura, J. Heidemann, W. Hardaker, P. Charnsethikul, J. Bulten, J. M. Ceron, and C. Hesselman. Old but gold: Prospecting TCP to engineer and live monitor DNS anycast. In *Proceedings of the Passive and Active Measurement Workshop*, page to appear, virtual, Mar. 2022. Springer.

[52] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *ACM SIGCOMM*, SIGCOMM '17, page 85–98, New York, NY, USA, 2017. Association for Computing Machinery.

[53] K. Nichols. pping (pollere passive ping), 2017.

[54] M. Osama, A. A. Ateya, S. Ahmed Elsaid, and A. Muthanna. Ultra-Reliable Low-Latency Communications: Unmanned Aerial Vehicles Assisted Systems. *Advances in Wireless Communications Systems, Information*, 13, 2022.

[55] S. Ostermann. tcptrace homepage. *http://www. tcptrace. org/*, 2007.

[56] A. Perrig, P. Szalachowski, R. M. Reischuk, and L. Chuat. *SCION: A Secure Internet Architecture*. Springer Verlag, 2017.

[57] A. Pilosov and T. Kapela. Stealing the Internet: An Internet-scale Man in the Middle Attack. *NANOG 44*, 2008.

[58] T. Qiu, J. Ni, H. Wang, N. Hua, Y. R. Yang, and J. J. Xu. Packet doppler: Network monitoring using packet shift detection. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, New York, NY, USA, 2008. Association for Computing Machinery.

[59] Y. Rekhter, T. Li, and S. Hares. A border gateway protocol 4 (bgp-4). RFC 4271, RFC Editor, January 2006. http://www.rfc-editor.org/rfc/rfc4271.txt.

[60] M. Saad, V. Cook, L. Nguyen, M. T. Thai, and A. Mohaisen. Partitioning attacks on bitcoin: Colliding space, time, and logic. In *2019 IEEE 39th international conference on distributed computing systems (ICDCS)*, pages 1175–1187. IEEE, 2019.

[61] B. Schlinker, H. Kim, T. Cui, E. Katz-Bassett, H. V. Madhyastha, I. Cunha, J. Quinn, S. Hasan, P. Lapukhov, and H. Zeng. Engineering Egress with Edge Fabric: Steering Oceans of Content to the World. In *ACM SIGCOMM*, 2017.

[62] S. Sengupta, H. Kim, and J. Rexford. Continuous in-network round-trip time monitoring. In *ACM SIGCOMM*, SIGCOMM '22, page 473–485, New York, NY, USA, 2022. Association for Computing Machinery.

[63] G. Severi, M. Jagielski, G. Yar, Y. Wang, A. Oprea, and C. Nita-Rotaru. Network-level adversaries in federated learning. In *2022 IEEE Conference on Communications and Network Security (CNS)*, pages 19–27. IEEE, 2022.

[64] A. Shukla and K.-T. Foerster. Shortcutting fast failover routes in the data plane. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, ANCS '21, page 15–22, New York, NY, USA, 2022. Association for Computing Machinery.

[65] G. Simsek, D. Ergenç, and E. Onur. Efficient network monitoring via in-band telemetry. In *2021 17th International Conference on the Design of Reliable Communication Networks (DRCN)*, pages 1–6, 2021.

[66] J. Snijders. Practical everyday BGP filtering with AS_PATH filters: Peer locking. *NANOG-67*, 2016.

[67] J. Sonchack, D. Loehr, J. Rexford, and D. Walker. Lucid: a Language for Control in the Data Plane. In *Proceedings of the 2021 ACM SIGCOMM Conference*. ACM, 2021.

[68] F. Streibelt, F. Lichtblau, R. Beverly, A. Feldmann, C. Pelsser, G. Smaragdakis, and R. Bush. Bgp communities: Even more worms in the routing can. In *Proceedings of the Internet Measurement Conference 2018*, IMC '18, page 279–292, New York, NY, USA, 2018. Association for Computing Machinery.

[69] F. Streibelt, F. Lichtblau, R. Beverly, A. Feldmann, C. Pelsser, G. Smaragdakis, and R. Bush. Bgp communities: Even more worms in the routing can. In *Proceedings of the Internet Measurement Conference 2018*, IMC '18, page 279–292, New York, NY, USA, 2018. Association for Computing Machinery.

[70] Y. Sun, A. Edmundson, L. Vanbever, O. Li, J. Rexford, M. Chiang, and P. Mittal. RAPTOR: Routing Attacks on Privacy in Tor. In *USENIX Security Symposium*, 2015.

[71] W. Tang, L. Kiffer, G. Fanti, and A. Juels. Strategic latency reduction in blockchain peer-to-peer networks. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(2):1–33, 2023.

[72] V. Valancius, B. Ravi, N. Feamster, and A. C. Snoeren. Quantifying the benefits of joint content and network routing. 41(1):243–254, jun 2013.

[73] L. Wang, H. Kim, P. Mittal, and J. Rexford. Programmable in-network obfuscation of traffic. *CoRR*, abs/2006.00097, 2020.

[74] L. Wang, H. Kim, P. Mittal, and J. Rexford. Raven: Stateless rapid ip address variation for enterprise networks. *Proceedings on Privacy Enhancing Technologies*, 2023, Jul 2023.

[75] Z. Yang, Z. Wu, M. Luo, W.-L. Chiang, R. Bhardwaj, W. Kwon, S. Zhuang, F. S. Luan, G. Mittal, S. Shenker, et al. {SkyPilot}: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 437–455, 2023.

[76] K.-K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, V. Lin, C. Rice, B. Rogan, A. Singh, B. Tanaka, M. Verma, P. Sood, M. Tariq, M. Tierney, D. Trumic, V. Valancius, C. Ying, M. Kallahalla, B. Koley, and A. Vahdat. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. SIGCOMM '17, page 432–445, New York, NY, USA, 2017. Association for Computing Machinery.

[77] S. Yoo and X. Chen. Secure keyed hashing on programmable switches. In *Proceedings of the ACM SIGCOMM 2021 Workshop on Secure Programmable Network INfrastructure*, SPIN '21, page 16–22, New York, NY, USA, 2021. Association for Computing Machinery.

[78] Y. Yoshinaka, J. Takemasa, Y. Koizumi, and T. Hasegawa. On implementing chacha on a programmable switch. In *Proceedings of the 5th International Workshop on P4 in Europe*, EuroP4 '22, page 15–18, New York, NY, USA, 2022. Association for Computing Machinery.

[79] L. Yuliang, M. Rui, K. Changhoon, and Y. Minlan. LossRadar: Fast Detection of Lost Packets in Data Center Networks. In *CoNEXT*, New York, NY, USA, December 2016. ACM.

[80] X. Zhang, H. Chen, Y. Zhao, Z. Ma, Y. Xu, H. Huang, H. Yin, and D. O. Wu. Improving Cloud Gaming Experience through Mobile Edge Computing. *IEEE Wireless Communications*, 26(4):178–183, 2019.

[81] X. Zhang, T. Sen, Z. Zhang, T. April, B. Chandrasekaran, D. Choffnes, B. M. Maggs, H. Shen, R. K. Sitaraman, and X. Yang. Anyopt: Predicting and optimizing ip anycast performance. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 447–462, New York, NY, USA, 2021. Association for Computing Machinery.

[82] Z. Zhang, M. Zhang, A. Greenberg, Y. C. Hu, R. Mahajan, and B. Christian. Optimizing cost and performance in online service provider networks. In *USENIX Networked Systems Design and Implementation*, 2010.

Appendix

A Adversarial Model

We describe in more detail a few concrete attacks that can be launched within the threat model outlined in §2.1.

1. **Manipulation Attacks**: An adversary on path P wants to make the one-way-delay of path P look smaller. The adversary launches the attack by modifying a timestamp t to timestamp $t+d$ which will make its one-way-delay appear to be reduced by d. Similarly, the adversary could modify the sequence numbers to hide loss and cover-up that some packets were dropped in her network.
2. **Replay Attacks**: An adversary is on paths P and Q. The latency from the sender to the adversary along P is shorter than the latency from the sender to the adversary along path Q. Latency from the adversary to the receiver is the same on path P and Q (P and Q may even share all hops after the adversary). At an instant t in time, the adversary sees packets with timestamp $t-p$ on path P and $t-q$ on path q. Since path P is faster, $t-p > t-q$. The adversary wants to improve the one-way-delay on path q and rewrites the timestamps for path Q to be $t-p$, reducing the delay on Q to the delay of P. Alternatively, the adversary could have a passive tap on path P.

B Additional TANGO Design Details

As shown in the pseudocode above (Alg. 1), TANGO senders perform the mapping from operator-provided objectives to routing traffic classes, add latency and loss metrics in custom headers, apply integrity-protecting signatures over the collected metrics, and tunnel the resulting packet over the public Internet to the destination edge.

Meanwhile, TANGO receivers decapsulate received metrics and verify signatures over the metrics, ensuring only tamper-free measurements are included in new optimization calculations (Alg. 2). After removing metadata, the receiver forwards the application traffic along to its final destination. It also issues route updates to the TANGO sender, if a better path has emerged and meets specified update thresholds (*e.g.,* the new path improves latency by at least 20% for more than 10 measurement cycles over 100 ms).

Algorithm 1 TANGO tunneling send behavior

```
1:  // Send an application packet to peer node
2:  function SEND(AppPacket)
3:      // Map packet to tunnel information
4:      TrafficClass ← GETCLASS(AppPacket)
5:      PathID ← GETPATH(TrafficClass)
6:
7:      // Collect metrics at time of processing packet
8:      Ts ← GETTIMENOWMS( )
9:      Seq ← GETANDINCREMENTSEQNUM( )
10:
11:     // Sign using books indexed by PathId and metrics
12:     TsSig ← TSBOOK[PathId][Ts]
13:     SeqSig ← SEQNUMBOOK[PathId][Seq]
14:
15:     // Form headers and send encapsulated packet
16:     { IpHdr, UdpHdr } ← GETTUNNELHEADERS(PathId)
17:     Metrics ← { PathId, Ts, TsSig, Seq, SeqSig }
18:     FORWARD(IpHdr, UdpHdr, Metrics, AppPacket)
19: end function
```

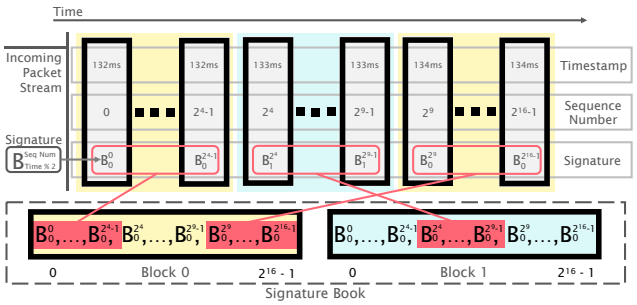


Figure 14: TANGO *ensures sequence numbers monotonically increase, even during block transitions, and only reset once all are expended. For example, assume block 0 is used on even timestamps and block 1 on odd. If, as the last sequence number of an even timestamp, the sequence number is* 2^4-1*, the next sequence number, now with an odd timestamp and therefore accessing block 1, is* 2^4*.*

C Supplementary Internet-Scale Results

C.1 Path Diversity: CDF

As described in §6.1.1, TANGO's PATHFINDER algorithm (detailed in §4) uncovered alternative paths between 503 globally-distributed Vultr pairs. The CDF of number of paths exposed per Vultr pair is included in Fig. 16. In some cases our BGP announcements appeared to be filtered as the source node used did not hear the BGP announcement made from the destination node. These

Algorithm 2 TANGO tunneling receive behavior

```
1:  // Verify tunneled packet and finish forwarding and reroute
2:  function RECEIVE(IpHdr, UdpHdr, Metrics, AppPacket)
3:      // Map packet to tunnel information
4:      TrafficClass ← GETCLASS(AppPacket)
5:      PathID ← Metrics.PathId
6:
7:      // Verify signatures
8:      ValidTsSig ← TSBOOK[PathId][Metrics.Ts]
9:      ValidSeqSig ← SEQNUMBOOK[PathId][Metrics.Seq]
10:     IsValidTs ← Metrics.TsSig == ValidTsSig
11:     IsValidSeq ← Metrics.SeqSig == ValidSeqSig
12:     if not IsValidTs or not IsValidSeq then
13:         return // Bail early if invalid
14:     end if
15:
16:     // Update metrics and finish forwarding packet
17:     { Delay, BestDelayId } ← UPDATEDELAY(Metrics.Ts)
18:     { Loss, BestLossId } ← UPDATELOSS(Metrics.SeqNum)
19:     FORWARD(AppPacket)
20:
21:     // Issue reroute request if path is not performant enough
22:     DoReroute ← CHECK(TrafficClass, Delay, Loss)
23:     if DoReroute then
24:         // Choose the best path for traffic class
25:         Id ← BEST(TrafficClass, BestDelayId, BestLossId)
26:         Update ← ENCRYPT(TrafficClass, BestPath)
27:
28:         // Request route update over all paths
29:         for { IpHdr, UdpHdr } in AllPaths do
30:             FORWARD(IpHdr, UdpHdr, Update)
31:         end for
32:     end if
33: end function
```

cases are recorded in the CDF as having a single available path because TANGO can still function by using the Vultr-provided IP address of instances to tunnel traffic. We also explored how many of these paths were found using only communities supported by our immediate transit provider Vultr and how many were found because of BGP community support at ASes further down the path (shown as separate traces in Fig. 16). If only Vultr-supported communities are used, the median pair of nodes only has two paths, but using transitive communities raises this by 50% to 3 paths. The maximum number of paths found between any two pairs of nodes increases from 6 to 12 showing the importance of community support at various nodes in the topology.

C.2 High-Latency Performance Degradation Events

In addition to looking for events with high loss we searched for events with high latency (i.e., a 100% latency increase over the baseline). We found these happened more frequently than high loss events. Fig. 17 shows these events. For high latency events, we found *all events were avoidable with* TANGO.

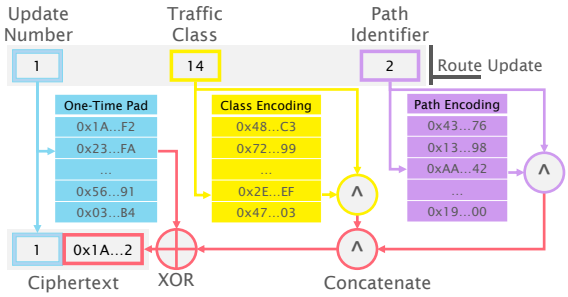


Figure 15: *Integrity-protected route updates. The encodings, if sufficiently sparse, prevent adversaries from brute-forcing the few bits encoding class/path, since overall encodings can easily be verified.*

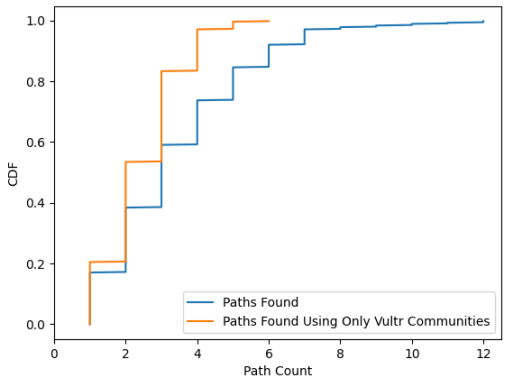


Figure 16: *Available paths identified with* PATHFINDER*, per Vultr pair using only communities supported by Vultr and communities supported by Vultr as well as ASes further down the path.*

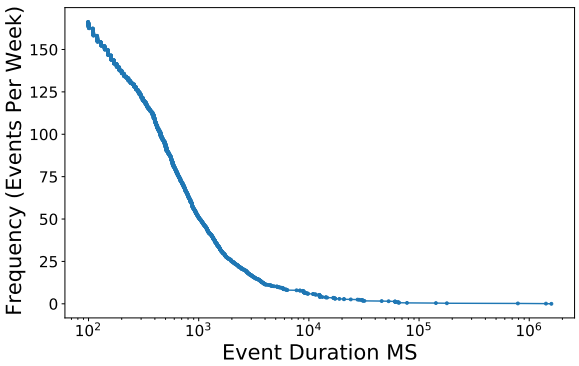


Figure 17: *Frequency vs duration of events with high latency.*

C.3 Performance Diversity: Vultr LA

We include results from our additional measurement study of 23 node pairs with Vultr LA as TANGO's fixed receiver.

How often are TANGO-discovered paths better than the de-

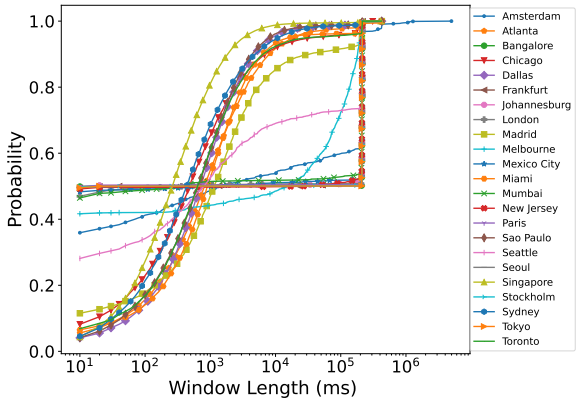


Figure 18: *Vultr LA Results: On average across all 23 pairs, best paths lasted from 1.8-107s, with median durations of 20ms-106s.* TANGO *only chooses to perform dynamic rerouting for best paths that last longer than 100ms.*

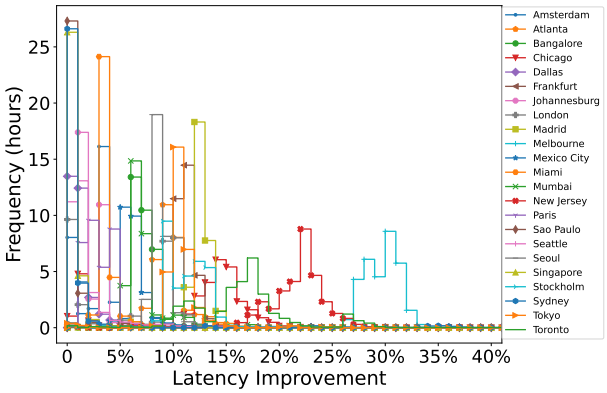


Figure 19: *Vultr LA Results:* TANGO*-exposed best paths outperform the BGP default by up to 29% on average, and up to 32% for some pairs.*

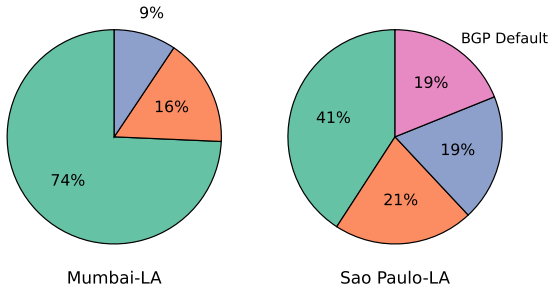


Figure 20: *Vultr LA Results: The BGP default path is often beaten by one or more alternative paths.*

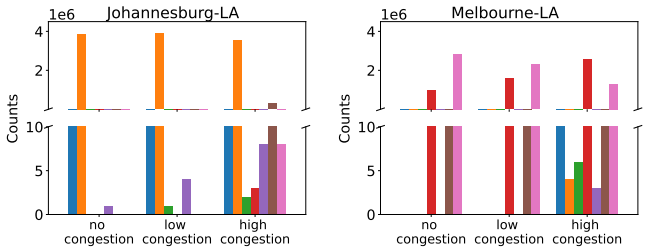


Figure 21: *Vultr LA Results: Number of times different paths contributed to being "best path" for a given congestion condition, per node pair. While one path might dominantly be best under no/low congestion, during high congestion more paths emerge as the best.*

fault? Of the 23 pairs in our Vultr Los Angeles (LA) measurements, *all 23 pairs had at least one alternative path that outperformed the default BGP path for a significant amount of time*: 100% of the time for 20 pairs, and 58-81% of the time for the remaining 3 pairs (Amsterdam-LA, Sao Paulo-LA, Singapore-LA). We also note that there were many pairs with *more than one* alternative path which outperformed the default, providing further path diversity and potential performance resilience: 18 pairs had two or more alternatives and 4 pairs had three or more alternatives. In Figs. 20 and 21, we visualize how often other paths emerged as best for a subset of pairs, chosen for their interesting breakdown. Distribution charts for all 23 pairs are included in Figs. 22 and 24.

How long is one path the best? We measured the window of time one path remained the best for each of the 23 Vultr pairs (Fig. 18). Many pairs showed path longevity, and for 17 pairs, the median window duration was 235*ms*-15*s*, with average durations of 1.8-10.8*s* (from Singapore, Sao Paulo, Dallas, Sydney, Atlanta, Miami, Chicago, Toronto), 20.1-59*s* (from Madrid, Seattle), and 100-107*s* (from Mumbai, Amsterdam, Mexico City, Frankfurt, London, Stockholm). For 3 pairs, the median duration was shorter at 20 *ms*, most likely due to ECMP behavior between the nodes. The average duration for these 3 pairs ranged between 105-106*s* (from Bangalore, Johannesburg, and Paris to LA). For the last 3 pairs out of all 23, the median and average duration were both much longer: 31*s* median and 104*s* average from New Jersey and 105*s* median with 106-107*s* average from both Seoul and Tokyo.

TANGO performs dynamic reroutes when a new path emerges as the best for longer than a given window threshold (e.g., 100ms), and it remains on the old best path for shorter windows. Thus, windows in the range of 20ms would not trigger a dynamic route update, while longer window lengths on the order of minutes would benefit from static path optimizations, instead of on-demand route updates. TANGO's on-demand reroutes would be most beneficial for the 17 pairs with median window durations between 235*ms*-15*s*.

By how much does the best path beat the BGP default? We measured the difference in latency between the best path and the default path, for each of the 23 Vultr pairs (Fig. 19). Of all 23 pairs, *the non-default best path outperformed the default by an average of 1-29% per pair*, for durations of 1.03-26.3 *hours*. The median improvement values also ranged from 0-22%. We saw even more opportunity for optimizing routing performance at the 95th percentile, where the TANGO-discovered best path *improved latency over the BGP default by 25-32%* for a duration of over half an hour (from Toronto) and between 1.3-1.6 hours (from New Jersey, Melbourne).

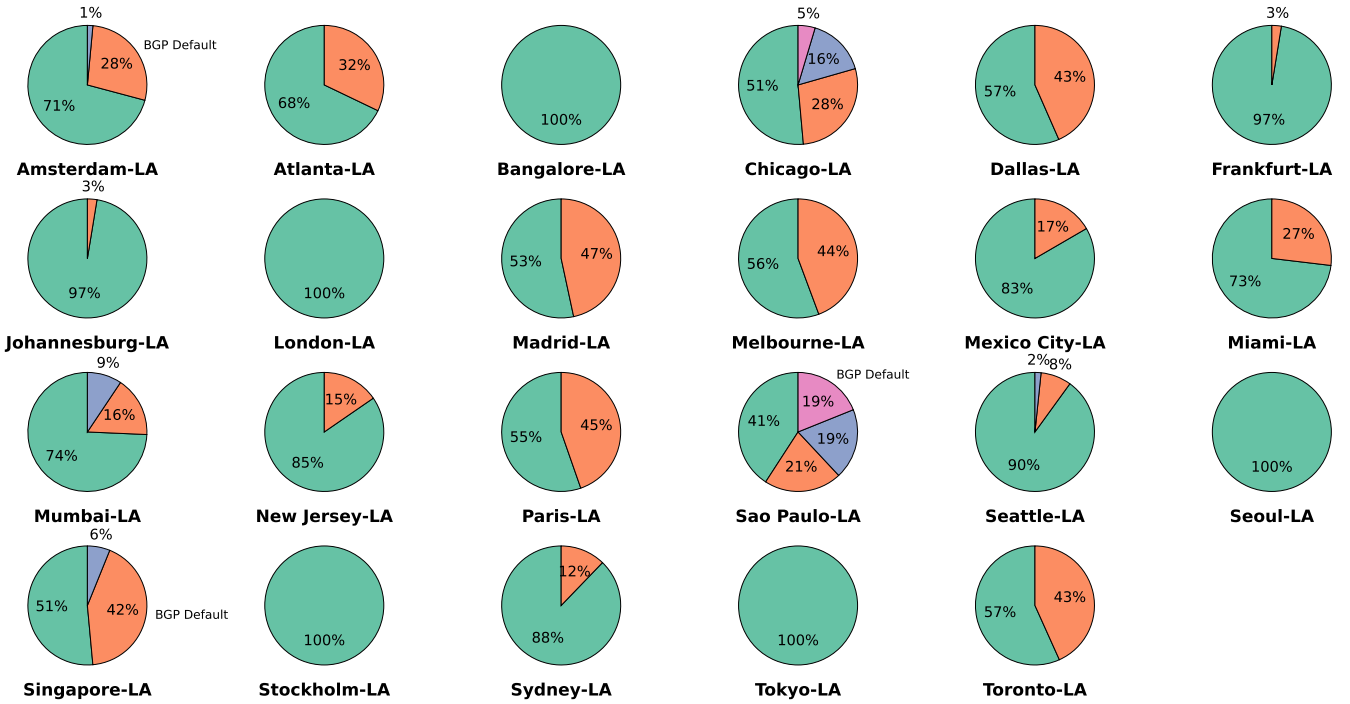


Figure 22: *Vultr LA Results: The BGP default path is beaten by one or more* TANGO*-exposed paths for all 23 node pairs, and for 4 out of 23 pairs it is beaten by 3 or more paths. On average, measurements from Vultr LA exposed richer path diversity across more nodes.*

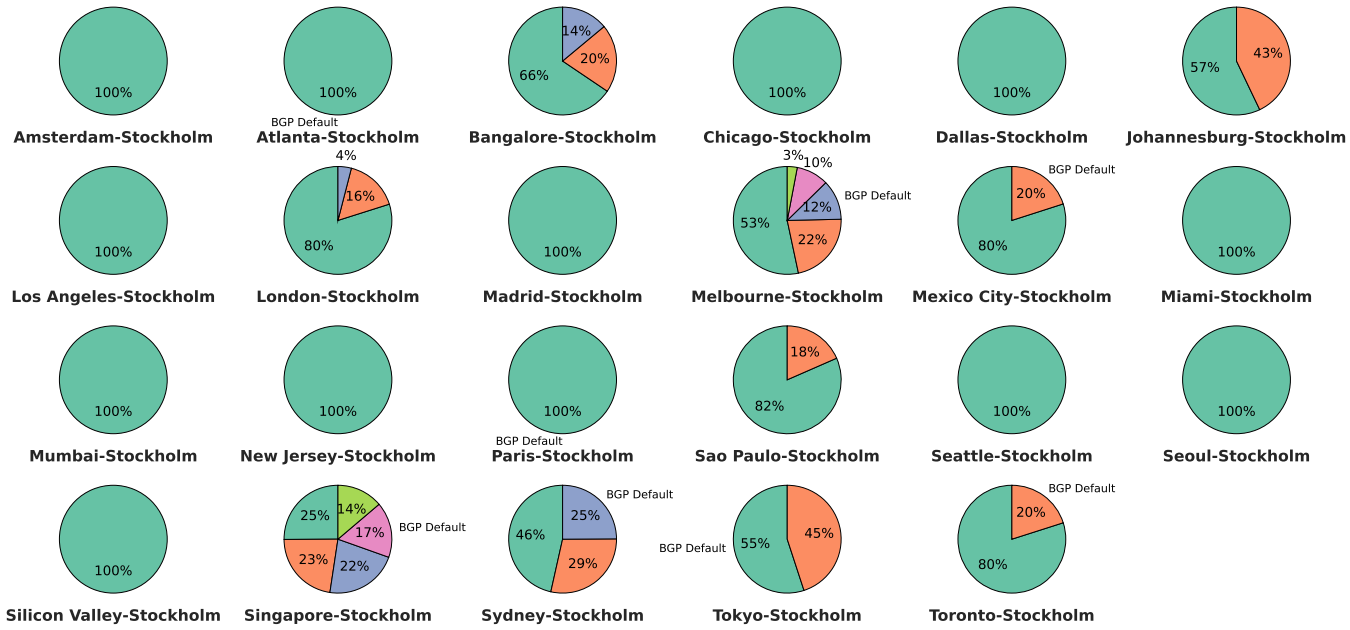


Figure 23: *Vultr Stockholm Results: The BGP default path is beaten by one or more* TANGO*-exposed paths for 20 out of 23 node pairs, for 4 out of 23 pairs it is beaten by 3 or more paths, while for 2 out of 23 pairs it is beaten by 4 alternative paths. On average, measurements from Vultr Stockholm exposed less path diversity across all nodes as compared to results from Vultr LA measurements.*

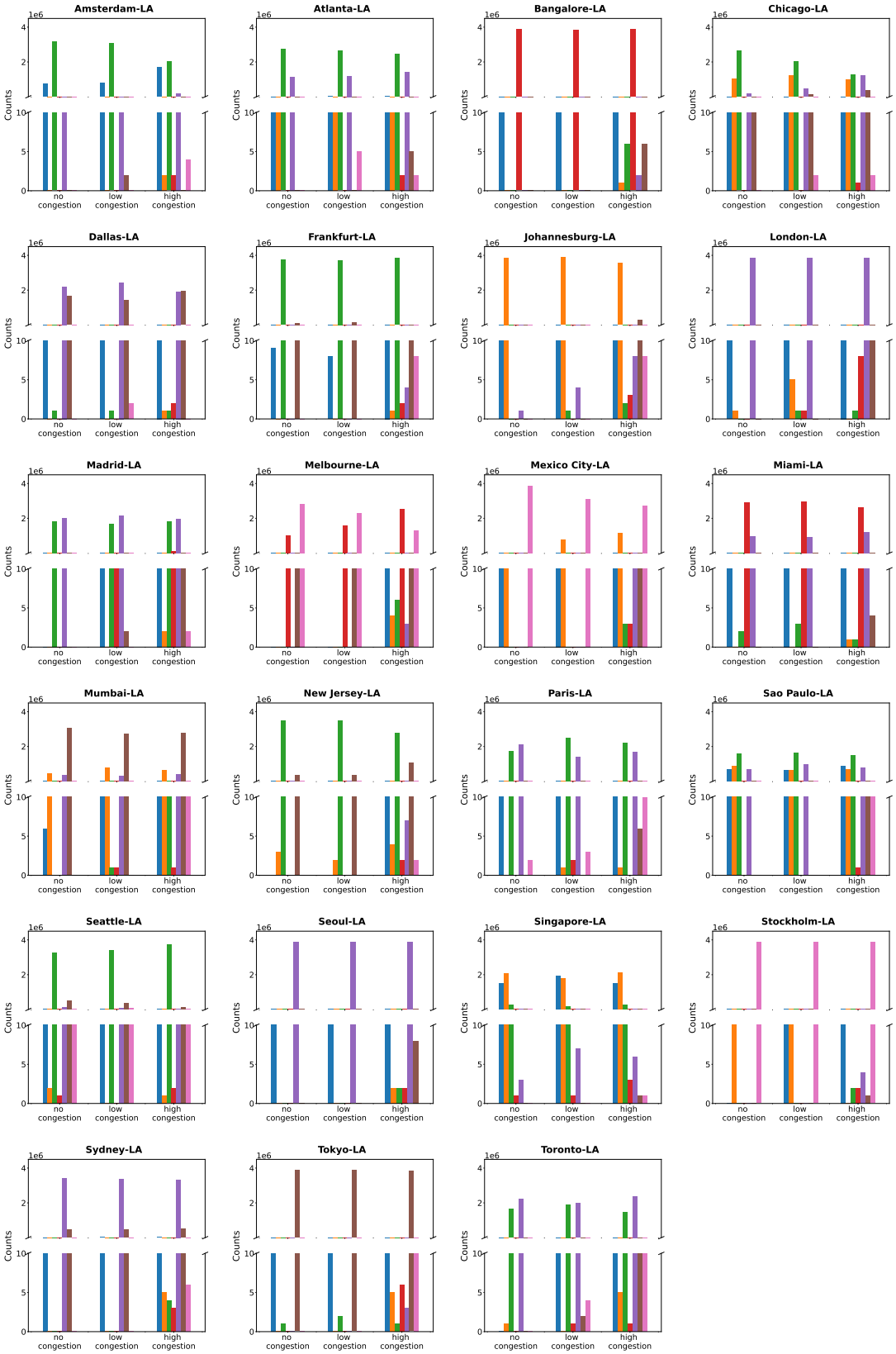


Figure 24: *Vultr LA Results: Number of times different paths contributed to being "best path" for a given congestion condition, per node pair. While one path might dominantly be best under no/low congestion, during high congestion more paths emerge as the best path.*

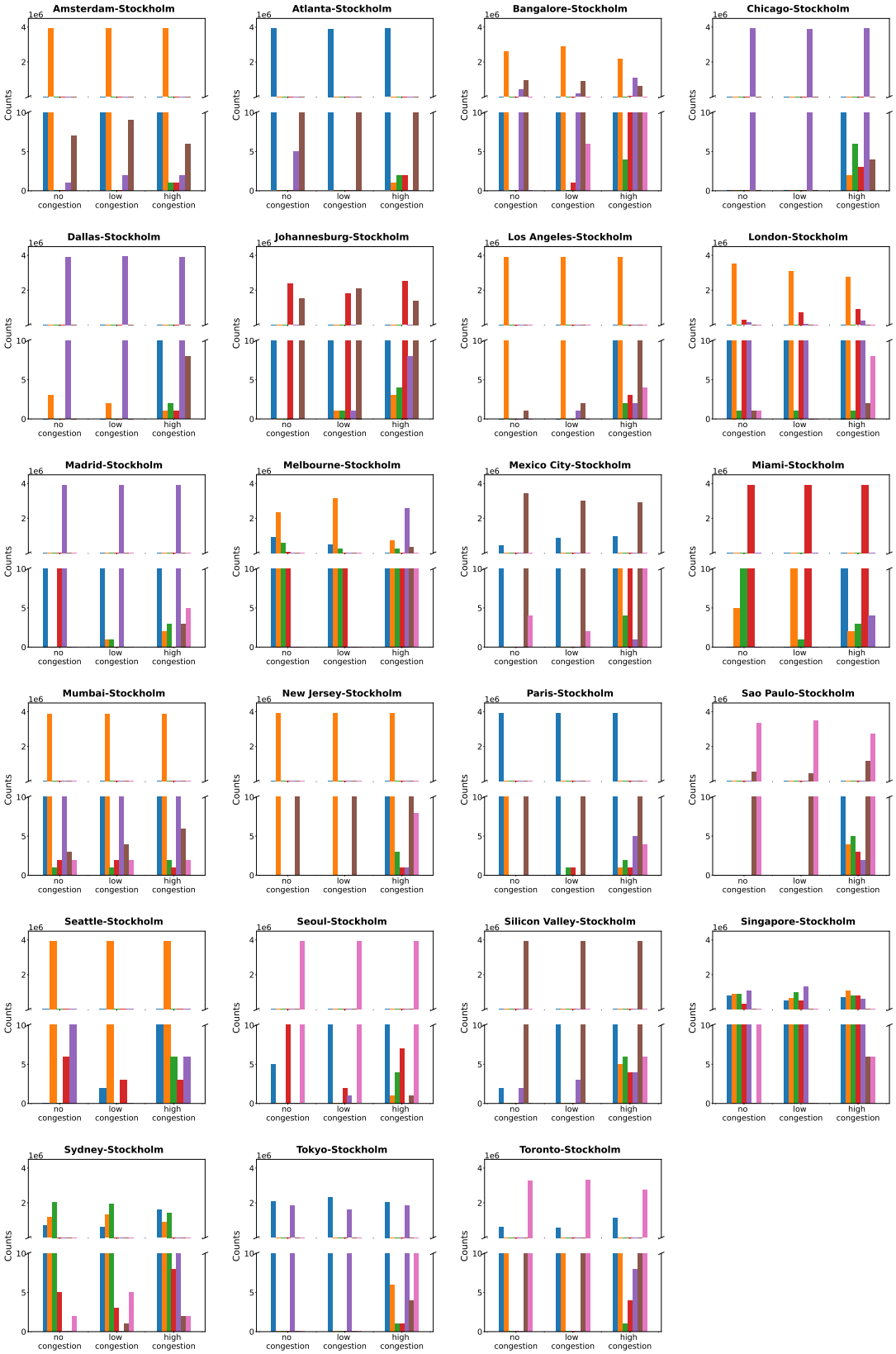


Figure 25: *Vultr Stockholm Results: Number of times different paths contributed to being "best path" for a given congestion condition, per node pair. While one path might dominantly be best under no/low congestion, during high congestion more paths emerge as the best path.*

Figure 26: *Geomap of 25 Vultr data centers running* TANGO *globally.*

Sidekick: In-Network Assistance for Secure End-to-End Transport Protocols

Gina Yuan
Stanford University

Matthew Sotoudeh
Stanford University

David K. Zhang
Stanford University

Michael Welzl
University of Oslo

David Mazières
Stanford University

Keith Winstein
Stanford University

Abstract

In response to concerns about protocol ossification and privacy, post-TCP transport protocols such as QUIC and WebRTC include end-to-end encryption and authentication at the transport layer. This makes their packets opaque to middleboxes, freeing the transport protocol to evolve but preventing some in-network innovations and performance improvements. This paper describes *sidekick protocols*: an approach to in-network assistance for opaque transport protocols where in-network intermediaries help endpoints by sending information adjacent to the underlying connection, which remains opaque and unmodified on the wire.

A key technical challenge is how the sidekick connection can efficiently refer to ranges of packets of the underlying connection without the ability to observe cleartext sequence numbers. We present a mathematical tool called a *quACK* that concisely represents a selective acknowledgment of opaque packets, without access to cleartext sequence numbers.

In real-world and emulation-based evaluations, the sidekick improved performance in several scenarios: early retransmission over lossy Wi-Fi paths, proxy acknowledgments to save energy, and a path-aware congestion-control mechanism we call *PACUBIC* that emulates a "split" connection.

1 Introduction

In the Internet's canonical model, transport is end-to-end and implemented only in hosts. Traditionally, routers and other network components forwarded IP datagrams without regard to their payloads or flow membership [12, 58]; only hosts thought about connections, reliable delivery, or flow-by-flow congestion control.

In practice, however, the best behavior for a transport protocol depends on the particulars of the network path. An appropriate retransmission or congestion-control scheme for a heavily-multiplexed wired network wouldn't be ideal for paths that include a high-delay satellite link, Wi-Fi with bulk ACKs and frequent reordering, or a cellular WWAN [25, 42].

By the 1990s, many networks had broken from the canonical model by deploying in-network TCP accelerators, also known as "performance-enhancing proxies" (PEPs) [26]. TCP PEPs can split an end-to-end connection into multiple concatenated connections [10, 17, 23, 28, 34], buffer and retransmit packets over a lossy link [2, 55], virtualize congestion control [14, 29, 49], resegment the byte stream, and enable forward error correction, explicit congestion notification, or other segment-specific enhancements. Because TCP isn't encrypted or authenticated, PEPs can achieve this transparently, without the knowledge or cooperation of end hosts. Roughly 20–40% of Internet paths cross at least one TCP PEP [21, 30].

While many flows benefit from PEPs, their use carries a cost: protocol ossification [21, 53]. When a middlebox inserts itself in a connection and enforces its preconceptions about the transport protocol, it can thwart the protocol's evolution, dropping traffic that uses an upgraded version or new options. TCP PEPs have hindered or complicated the deployment of many TCP improvements, such as ECN++, tcpcrypt, TCP extended options, and multipath TCP [30, 46, 56].

In response to this ossification, and to an increased emphasis on privacy and security, post-TCP transport protocols have been designed to be impervious to meddling middleboxes, by encrypting and authenticating the transport header. We call these newer transport protocols "opaque." The most prevalent is QUIC [32], found in billions of installed Web browsers and millions of servers [68]; other opaque transport protocols are used in WebRTC/SRTP [54], Zoom [69], BitTorrent [4], and Mosh/SSP [63].

This opacity means that middleboxes can't interpose themselves on a connection or understand the sequence numbers of packets in transit. This prevents PEPs from providing assistance, reducing—in some situations—the performance of opaque transport protocols [6, 7, 38, 42, 47]. It's possible to co-design protocols and PEPs to preserve security and privacy while permitting assistance from credentialed middleboxes [19, 24, 33, 59], but challenging to do so without tightly coupling these components, risking ossification and fragility.

In this paper, we propose a method for in-network assistance of opaque transport protocols that tries to resolve this tension. Our approach leaves the transport protocol unchanged on the wire: a secure end-to-end connection between hosts, opaque to middleboxes and free to evolve. No PEPs are credentialed to decrypt the transport protocol's headers.

Instead, we propose a second protocol to be spoken on an adjacent connection between an end host and a PEP. We call this the **sidekick protocol**, and its contents are *about* the packets of the underlying, or "base," connection. Sidekick PEPs assist end hosts by reporting what they've observed about the

packets of the opaque base connection, without coupling their assistance to the details of the base protocol. End hosts use this information to influence decisions about how and when to send or resend packets on the base connection, approximating some of the performance benefits of traditional PEPs. A similar functional separation was first proposed by [67], but this paper presents the first concrete realization of the idea and its nuanced interactions with real transport protocols.

One key technical challenge with this approach is how the sidekick can efficiently refer to ranges of packets in an opaque base connection. These packets appear random to the middlebox, and referring to a range of, e.g., 100 opaque packets in the presence of loss and reordering is not as simple as saying "up to 100" when there are cleartext sequence numbers. In Section 3, we present and evaluate a mathematical tool called a **quACK** that concisely represents a selective acknowledgment of opaque, randomly identified packets. The quACK is based on the insight that we can model the problem as a system of power sum polynomial equations if there is a practical bound on the maximum number of "holes" among the packets being ACKed. We created an optimized implementation [65], building on related theoretical work [22, 35, 50].

A second challenge is how the end host should use information from a sidekick connection to obtain a performance benefit for its base connection. Since the performance benefit comes from changing behavior at the end host rather than the middlebox, transport protocols need to incorporate this information into their existing algorithms for, e.g., loss detection and retransmission, which have gotten increasingly complex over time. To explore this, we designed a sidekick protocol we call Robin, and implemented it in three scenarios:

- A low-latency audio stream over an Internet path that includes a Wi-Fi path segment (low latency with loss), followed by a WAN path segment (higher latency with low loss). Can the sidekick PEP reduce the de-jitter buffer delay by triggering earlier retransmissions on loss?
- An upload over the same path. Can an opaque transport protocol like QUIC, aided by a sidekick PEP at the point between these two path segments, match the throughput of TCP over a connection-splitting PEP?
- A battery-powered receiver, downloading data from the Internet over Wi-Fi. If the Wi-Fi access point sends sidekick quACKs on behalf of the receiver, can it reduce the number of times the receiver's radio needs to wake up to send an end-to-end ACK?

A third technical challenge is how knowledge about *where* loss occurs along a path should influence a congestion-control scheme. The challenge in any such scheme is how to maximize the congestion window while sharing the network fairly with competing flows. We present a path-aware modification to the CUBIC congestion-control algorithm [27], which we call **PACUBIC**, that approximates the congestion-control behavior of a PEP-assisted split TCP CUBIC connection while making its decisions entirely on the host.

Summary of results. Concretely realized, the quACK expresses the equivalent of TCP's cumulative + selective ACK over opaque (randomly identified) packets in 48 bytes, tolerating up to 10 missing packets before the last "selective ACK." On a recent x86-64 CPU, it takes 33 ns/packet for a sidekick PEP to encode a quACK, and 3 μs for an end host to decode it. These overheads compared well with several alternatives (Section 3.5).

We implemented Robin in a low-latency media client based on the WebRTC standard, and an HTTP/3 client using the Cloudflare implementation of QUIC [13] and the `libcurl` [45] implementation of HTTP/3. We evaluated the three scenarios in real-world and emulation experiments. In real-world experiments using an unmodified local Wi-Fi network to access our nearest AWS datacenter, the sidekick was able to trigger early retransmissions to fill in gaps in the audio of a latency-sensitive audio stream, reducing the receiver's de-jitter delay from 2.3 seconds to 204 ms—about a 91% reduction (Figure 8). The sidekick was also able to improve the speed of an HTTP/3 (QUIC) upload by about 50%.

In emulation experiments of the "battery-powered receiver" scenario, the sidekick PEP was able to reduce the need for the receiver to send ACKs by sending proxy acknowledgments on its behalf—ACKs the sender used to advance its flow-control and congestion-control windows. The receiver only needed to wake up its radio to send occasional end-to-end ACKs, which the sender used to discard data from its buffer (Figure 4c).

Also in an emulation experiment, we confirmed that PACUBIC's performance approximates a split CUBIC connection (two TCP CUBIC connections separated by a PEP), responding to loss events on the different path segments similarly to how the individual CUBIC flows would (Figure 6). The results indicate that the sidekick protocol's gains do not come at the expense of congestion-control fairness relative to a split CUBIC connection.

The rest of this paper describes the sidekick's motivating scenarios (Section 2), explores the quACK's design and implementation (Section 3), discusses the concrete sidekick protocol we built around quACKs (Section 4) and its implementation in two base protocols (Section 5), and then evaluates the protocol in real-world and emulation experiments (Section 6).

2 Motivating Scenarios

We focus on three scenarios where end hosts benefit from in-network assistance. In each one, a proxy server provides feedback, called a quACK, to an end host: the data sender (Figure 1). Recall that a quACK is a "cumulative ACK + selective ACK" over encrypted sequence numbers. The data sender uses this feedback to influence its behavior on the base connection, without altering the wire format.

To be clear: the sidekick protocol is not tied to a specific base protocol nor to how the end hosts use the quACK infor-

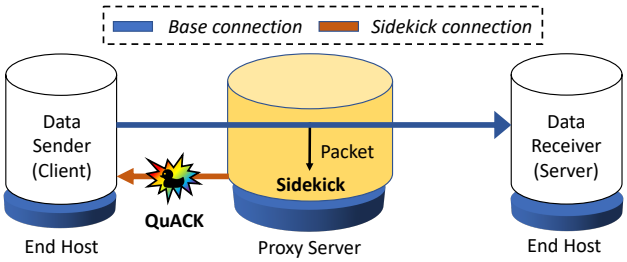


Figure 1: The proxy generates quACKs, in-network acknowledgments, based on the opaque packets it observes in the base protocol. It quACKs to an end host, the data sender, which sends or resends packets on the base protocol as a result. Although we only show one side of the connection, the sidekick could assist either end host of a bidirectional flow.

mation. The base protocol does not need to be reliable, nor to have unique datagrams—we implemented and evaluated the same sidekick protocol and the same middlebox behavior across the different scenarios in this paper.

2.1 Low-Latency Media

Consider a train passenger using on-board Wi-Fi to have a low-latency audio conversation, using WebRTC/SRTP [54], with a friend. The end-to-end network path contains a low-latency, high-loss "near" path segment (the Wi-Fi hop) followed by a high-latency, low-loss "far" path segment (the cellular and wired path over the Internet). The friend probably suffers from poor connection quality, experiencing drops in the audio stream or high de-jitter buffer delays from waiting for retransmitted packets to be played in order (Figure 4a in emulation, Figure 8a in real world).

In the sidekick approach, a sidekick on the Wi-Fi access point sends quACKs to the audio application on the user's laptop, assisting the base connection's data sender. The sender uses quACKs to retransmit packets sooner than they would have using negative acknowledgments (NACKs) from the receiver. The end result is similar to the effect of prior PEPs, such as Snoop [2] and Milliproxy [55], that leverage TCP's cleartext sequence numbers to trigger early retransmission on lossy wireless paths.

2.2 Connection-Splitting PEP Emulation

Consider the same train passenger as before but uploading a large file over the Internet with a reliable transport protocol. If the protocol were TCP, the train could deploy a split TCP PEP at the access point. The split connection allows quick detection and retransmission of dropped packets on the lossy Wi-Fi segment, while opening up the congestion window on the high-latency cellular segment.

However, opaque transport protocols like QUIC can't benefit from (nor be harmed by) connection-splitting PEPs. Without a PEP, QUIC relies on end-to-end mechanisms over the entire path to detect losses, recover from them, and adjust the congestion-control behavior. This leads to reduced upload speeds (Figure 4b in emulation, Figure 8b in real world).

With help from the same sidekick PEP, the QUIC sender combines information from quACKs and end-to-end ACKs to emulate the congestion-control behavior of a split TCP connection (Section 4.3.2). The application considers whether packets are lost on the near or far path segments, and adjusts the congestion window accordingly while respecting the opacity of the end-to-end base connection. The application also retransmits the packet as soon as the loss has been detected.

The only guarantee the proxy makes to the sender via the quACK is that it has received some packets. To respect the end-to-end reliability contract with the receiver, the sender does not delete packets that may need to be transmitted until it receives an ACK, even if the packet has been quACKed.

2.3 ACK Reduction

Now consider a battery-powered device downloading a large file from the Internet. To reduce how often the receiver's radio needs to wake up, saving energy, the base connection can reduce the frequency of end-to-end ACKs the device sends. ACK reduction has also been shown to improve performance by reducing collisions and contention over half-duplex links [16, 43]. The ACK frequency can be configured with a TCP kernel setting or proposed QUIC extension [31].

However, ACK reduction can also degrade throughput [15, 16] (Figure 4c in emulation). The sender receives more delayed feedback about loss, and has to carefully pace packets to avoid bursts in the large delay between ACKs. One proposal has the PEP acknowledge packets on behalf of the receiver [36], leveraging cleartext TCP sequence numbers, but it does not apply to opaque transport protocols.

In this case, a sidekick at the Wi-Fi access point (or a cellular base station) quACKs to the sender on behalf of the receiver. The receiver still occasionally wakes up its radio to send ACKs, but the sender uses the more frequent quACKs to advance its flow-control and congestion-control windows.

The sender respects the end-to-end reliability contract by only deleting packets in response to ACKs, but disregards the receiver's flow control by using quACKs to advance the flow-control window. If the sender only used ACKs to advance the window, it would waste time waiting between ACKs to send packets with too small a window, and need to pace sent packets on receiving a large ACK with too large a window.

3 QuACK

As previously illustrated, a sidekick needs to be able to refer to and efficiently acknowledge a set of opaque packets seen by a network intermediary. But this problem is technically challenging for middleboxes without access to cleartext sequence numbers or the ossification of other fields.

We start by mathematically defining the quACK problem. We discuss how to select an *identifier* to refer to a packet, and analyze strawman solutions to the quACK problem that

use too much space or computation. Finally, we present an efficient construction of a quACK based on the insight that we can model the problem as a system of power sum polynomial equations when we have a bound on the maximum number of missing elements, a threshold t. This solution is most similar to the deterministic solution to the *straggler identification* problem [22], and also builds on related theoretical work in set reconciliation [50], and coding theory and graph theory [35].

3.1 The QuACK Problem

We first describe the quACK problem. A data sender transmits a multiset[1] of elements S (these correspond to packets). At any given time, a receiver (such as a proxy server) has received a subset $R \subseteq S$ of the sent elements. We would like the receiver to communicate a small amount of information to the sender, who then efficiently decodes the missing elements—the set difference $S \setminus R$—knowing S. We call this small amount of information the "quACK", and the problem is: **what is in a quACK and how do we decode it?**

3.2 Packet Identifiers

In a networking context, how exactly do we refer to the elements in the quACK problem that have been sent or received? Traditional TCP middleboxes have been able to interpose their own concise, cumulative acknowledgments using cleartext sequence numbers, but this is not possible with modern, secure transport protocols. Even if a connection did expose an unencrypted numerical field, we would not want to refer to that field at risk of ossifying that protocol.

Instead, we need a function that deterministically maps a packet to a random b-byte *identifier*. The most trivial solution that applies to all base protocols is to hash the entire payload. Another option if the payload is already pseudorandom (e.g., QUIC) is to take the first b bytes from a fixed offset of that payload. Although the latter option would rely on those bytes to remain pseudorandom, it is computationally more efficient because it does not require reading the entire payload.

Collisions. The main considerations when selecting the number of bytes, b, in an identifier is the tolerance for collisions compared with the extra data needed to refer to these packets on the link. The larger b is, the lower the collision probability but the greater the link overhead.

Define the collision probability to be the probability that a randomly-chosen b-byte identifier in a list of n packets maps to more than one packet in that list. If we assume that identifiers are uniformly distributed, this probability is equal to $1-(1-1/256^b)^{n-1}$. When $n=25$, using 4 bytes results in an almost negligible chance of collision while using 2 bytes results in a 0.04% chance (Table 1).

When handling collisions, a sender who is decoding a quACK has a list of n packets it is trying to classify as received or missing (Section 3.5). Note that collisions are also known to the sender beforehand. If there is a collision between a packet that is received and a packet that is missing, the fate of that identifier is considered indeterminate. In our scenarios (Section 2), either the protocol can still function with approximate statistics (e.g., congestion control) or it can fall back to an end-to-end mechanism (e.g., retransmission).

[1] A "multiset" means the same element can be transmitted more than once.

Identifier Bytes	1	2	4	8
Collision Prob.	0.090	0.0004	5.6e-09	≈0

Table 1: Collision probabilities for $n = 25$.

3.3 Strawman Solutions

A problem that is simple with cumulative and selective acknowledgments of plaintext sequence numbers is deceivingly challenging for pseudorandom packet identifiers. Consider the following strawman solutions to the quACK problem:

Strawman 1: Echo every identifier. Strawman 1a, similar to [41, 44], echoes the identifier of every received packet in a new UDP packet to the data sender. Decoding is trivial given the identifiers are unmodified. This strawman adds significant link overhead in terms of additional packets. Additionally, since the strawman is not cumulative, losing a quACK means the end host could falsely consider a packet to be lost, creating a congestion event or spurious retransmission.

Strawman 1b echoes a sliding window of identifiers over UDP such that there is overlap in the identifiers referred to by consecutive quACKs. This solution is slightly more resilient to loss, but uses more bytes and is still not guaranteed to be reliable. Another variant batches identifiers to reduce the number of packets, but this solution is even less resilient to loss.

We also consider a Strawman 1c that echoes every identifier over TCP with `TCP_NODELAY` to send every identifier in its own packet. This ensures there are no false positives when detecting lost packets, but adds even more link overhead in terms of TCP headers and additional ACKs from the data sender (every other packet by default in the Linux kernel).

Strawman 2: Cumulative hash of every identifier. Strawman 2 sends a SHA-256 hash of a sorted concatenation of all the received packets in a UDP packet, and the sender hashes every subset of the same size of sent packets until it finds the subset with the same hash (assuming collision resistance). The strawman includes a count of the packets received to determine the size of the subset to hash. As the number of missing packets exceeds even a moderate amount, the number of subsets to calculate explodes, making the strawman impractical to decode.

One might also suggest the receiver send negative acknowledgments of the packets it has not received. However, unlike sequence numbers where one can determine a gap in received packets, there is no way to tell with random identifiers what packet is missing or should be expected next.

	Per-Packet Encode Time	Decode Time	Num Additional Proxy Packets	Packet Payload Size (bytes)	Cumulative?
Strawman 1a	Parse identifier	N/A	n	b	No
Strawman 1b	Parse identifier, move sliding window	N/A	n	$b \cdot window$	No
Strawman 1c	Parse identifier	N/A	n (TCP headers)	b	No
Strawman 2	Parse identifier, concatenate and hash	Concatenate and hash $\binom{n}{m}$ subsets	1	$32+4$ (hash and count)	Yes
Power Sums	Parse identifier, t modular multiplications and additions	Plug n candidate roots into a degree-m polynomial OR solve system of m polynomial equations	1	$4+b+b \cdot t$ (count, last value, t power sums)	Yes

Table 2: Strawmen compared to the power sum quACK representing n packets sent by the data sender, m missing packets, and b-byte identifiers. The power sum quACK uses the threshold t. The total data overhead of each quACK must consider the packet payload size along with transport headers. We evaluate the overheads in practice in Section 6.4.

3.4 The Power Sum Solution

Now we describe a solution to the quACK problem based on the insight that we can model the problem as a system of power sum polynomial equations when we have a bound on the maximum number of missing elements, a threshold t. Unlike the previous strawmen, this construction is efficient to decode, and its size is proportional only to t.

Consider the simplest case, when the receiver is only missing a single element. The receiver maps packet identifiers to a finite field, i.e. modulo the largest prime that fits in b bytes, and communicates the sum $\sum_{x \in R} x$ of the received elements to the sender. The sender computes the sum $\sum_{x \in S} x$ of the sent elements and subtracts the sum from the receiver, calculating:

$$\sum_{x \in S} x - \sum_{x \in R} x = \sum_{x \in S \setminus R} x,$$

which is the sum of elements in the set difference. In this case, the sum is exactly the value of the missing element.

In fact, we can generalize this scheme to any number of missing elements m. Instead of transmitting only a single sum, the receiver communicates the first m *power sums* to the sender, where the i-th power sum of a multiset R is defined as $\sum_{x \in R} x^i$. The sender then computes the first m power sums of S and calculates the respective differences d_i for $i \in [1, m]$, producing the following system of m equations:

$$\left\{ \sum_{x \in S \setminus R} x^i = d_i \mid i \in [1, m] \right\}.$$

Instead of transmitting an unbounded number of power sums, the receiver only maintains and sends the first t power sums. Efficiently solving these t power sum polynomial equations in t variables in a finite field is a well-understood algebra problem [22]. The solutions are exactly $x \in S \setminus R$.

Efficiency. The power sum quACK is efficient to decode, adds reasonable link overhead, and is a cumulative representation of the packets seen by the receiver (Table 2). Compared to Strawman 2, the power sum quACK can be decoded with simple algebraic techniques. Its link overhead is proportional only to the number of missing packets between consecutive quACKs, up to a configurable threshold. In comparison, the link overhead of Strawman 1 is necessarily proportional to the number of received packets. The power sum quACK is also resilient to mis-identifying a received packet as dropped, in the case a quACK is lost in transmission.

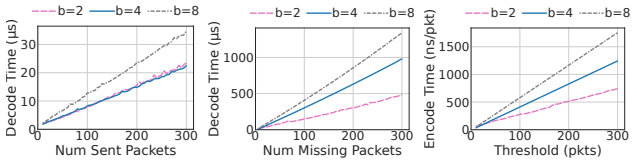
Interface. The actual format of the power sum quACK includes three fields: (i) t b-byte power sums, (ii) a 4-byte count of received elements, and (iii) the b-byte identifier of the last element received. We assume power sum quACKs to be sent over UDP, though the actual mechanism is not tied to the design. Since the decoder does not know m ahead of time, the decoder takes the difference between the number of packets it has sent and the count in the quACK to calculate m. Sending the last element received is an optimization that allows m to represent just the "holes" among the packets being selectively ACKed, excluding the possibly many consecutive elements that are in-flight (Section 4.3.1).

3.5 Microbenchmarks

We benchmark our optimized implementation of the power sum quACK [65] to demonstrate its practicality for in-line packet processing. Our microbenchmarks used an m4.xlarge AWS instance with a 4-CPU Intel Xeon E5 processor @ 2.30 GHz and 16 GB memory.

	Encode Time	Decode Time
Strawman 1a/1c	1 ns/pkt	0
Strawman 1b	51 ns/pkt	0
Strawman 2	27 ns/pkt	830 ms
Power Sum	**33 ns/pkt**	**2.82 μs**

Table 3: The CPU overheads of power sums are comparable to those of the strawmen, while being more efficient in space and computation. The encode time includes constructing and serializing the quACK(s), given n identifiers. The decode time includes finding the identifiers of either R or $S \setminus R$, given the quACK(s) and S. Parameters: $n = 25$, $t = 10$, $b = 4$.



(a) Evaluate a degree-$m = t = 10$ polynomial at n candidate roots.

(b) Plug $n = 300$ candidate roots into a degree-m polynomial.

(c) Update t power sum equations. Average of 1000 packets.

Figure 2: How power sum quACK performance depends on various parameters: bit width, threshold, number of sent and missing packets. Average of 100 trials.

A power sum quACK that represents $n = 25$ outstanding packets (packets in consideration on that path segment not yet known to be received or lost) and up to $t = 10$ missing packets with $b = 4$-byte identifiers adds 33 ns of encoding time per packet and takes 2.82 μs to decode (Table 3).

Decoding. The decode time must be comparable to the time it takes to process a typical ACK and modify the logic in the transport protocol. Decoding typically occurs on end hosts, compared to encoding which occurs in the middle of the path.

Finding the solution to the system of power sum polynomial equations boils down to applying Newton's identities (a linear algorithm) and finding the roots of a polynomial equation in a modular field [22]. Factoring a polynomial is asymptotically fast in theory, but the implementation is branch-heavy and complicated [3]. We found that plugging in and evaluating which of n candidate roots evaluated to zero was faster in practice for $n < 40{,}000$ roots. This is the method we use to decode the power sum quACK.

The decode time of this method is directly proportional to n (Figure 2a) and the number of missing packets m (Figure 2b). Decoding takes $2820/10/25 \approx 11$ ns/candidate/missing. Both n and m are typically a few hundred at most.

Encoding. The encode time per-packet is directly proportional to the threshold number of missing packets t (Figure 2c) at $33/10 \approx 3$ ns/power sum. Each power sum can be updated in a constant number of operations based on the previous power sum, so encoding an identifier requires t modular additions and multiplications for the t power sums.

Bit widths. Different bit widths have different implications for which instructions the CPU can use. Modular operations are efficient for 16- and 32-bit integers, fitting within the 64-bit word size (the number of bits that can be processed in one instruction) of most modern CPUs. For example, to multiply two 32-bit integers, we cast them to 64-bit integers, multiply, then take the modulus.

Figure 2 shows the best performance we achieved at different bit widths. For 16-bit identifiers only, we precomputed power tables that fit in the L3 cache. For 64-bit identifiers, we implemented Montgomery modular multiplication [51] to avoid an expensive hardware division for 128-bit integers. In the remainder of the paper, we use $b = 4$ as the preferred tradeoff between space and collision probability.

4 Sidekick Protocol

This section describes Robin, our design for a sidekick protocol built around quACKs. This includes the setup and configuration of a Robin sidekick connection, how a sender detects loss from a quACK, and a path-aware modification to CUBIC called PACUBIC, for congestion-controlled base protocols.

4.1 PEP Discovery Mechanism

Sidekick connections can be configured explicitly or implicitly. In systems that explicitly configure proxies, such as Apple's iCloud Private Relay [1] based on MASQUE [39, 40], proxies can simply negotiate sending quACKs during session establishment. In most other settings, such as 4G/5G cellular networks, PEPs have traditionally been deployed as transparent proxies, silently interposing on end-to-end connections. Senders therefore need a way to detect transparent sidekick proxies and inform them of where to send quACKs. Because of network address translation, all communication to the proxy must be initiated by the sender or use the same IP addresses and port numbers of the base connection.

Our current design has senders signal quACK support by sending a distinguished packet containing a 128-byte *sidekick-request* marker. Such inline signaling could confuse receivers, but sidekicks target protocols such as QUIC that discard cryptographically unauthenticated data anyway. It would be cleaner to signal support through out-of-band UDP options [61], which we hope to do once they are standardized.

The proxy replies to a sidekick-request packet by sending a special packet from the receiver's IP address and port number back to the sender. This packet contains a *sidekick-reply* marker, an opaque session ID, and an IP address and port number for communicating with the proxy. Upon receiving the sidekick-reply packet, the sender begins communicating directly with the proxy from a different UDP port. It initially sends back the session ID and configuration parameters to start receiving quACKs.

Security. A malicious third-party could execute a reflection amplification attack that generates a large amount of traffic while hiding its source. This is possible because the sender requests quACKs to a different port and (for some carrier-grade NATs) IP address from the underlying session. To mitigate this, each quACK contains a quota, initially 1, of remaining quACKs the proxy will send as well as an updated session ID. The quota and session ID ensure only the sender can increase the quota or otherwise reconfigure the session.

An adversarial PEP could send misleading information to the sender. Note that only on-path PEPs can send credible information, since they refer to unique packet identifiers. To mitigate this, the sender can consider PEP feedback along with end-to-end metrics to determine whether to keep using the PEP. The sender can always opt out of the PEP, and the

PEP cannot actively manipulate traffic any more than outside a sidekick setting.

4.2 Configuration Messages

The data sender can send various other messages to the proxy to configure the connection or reset bad state.

Protocol parameters. The sender configures (i) the quACK interval of the PEP and (ii) the threshold number of missing packets t, or otherwise selects sidekick-specific settings such as how an identifier is computed.

The quACK interval is expressed in terms of time or number of packets, e.g., every N milliseconds or every N packets, as in a TCP delayed ACK. The sender determines the desired interval based on its estimated RTT of the base connection and its application objectives, e.g., more frequently for latency-sensitive applications or lower-RTT paths.

The threshold represents the bound on the number of missing packets between quACKs, in practice the number of "holes" among the packets that are selectively ACKed. The threshold depends on the quACK interval, and should be set based on how precise loss detection needs to be and other qualities of the link. For example, the threshold is larger to detect congestive loss in the queue of a bottleneck link, or smaller to still detect transmission error on a lossy link.

Resets. Robin allows the sender to tell the PEP to reinitialize the quACK. This is helpful if the quACK becomes invalid, e.g., if m exceeds the threshold t. It is always safe to reset the quACK, or even to ignore the sidekick entirely and fall back to the base protocol's end-to-end mechanisms.

4.3 Sender Behavior

In this section, we discuss two particular sender-side behaviors that are enabled by the sidekick protocol and which are helpful across several scenarios: detecting packet loss from a decoded quACK and congestion control.

4.3.1 Detecting Loss

The sender knows definitively which packets have been received by the proxy from a decoded quACK. Next, it must determine from the remaining packets which ones have been dropped and which are still in-flight, including if there has been a reordering of packets. In-flight packets are later classified as received or dropped based on future quACKs.

When there is no reordering, the packets that are dropped are just the "holes" among the packets that are selectively ACKed by the quACK. In particular, these are the holes when considering sent packets in the order they were sent up to the last element received, which represents the last selective ACK. To identify these dropped packets, the sender encodes t cumulative power sums of its sent packets up to the last element received. The difference between these power sums and the power sums in the quACK represents the dropped packets. The sender "removes" the identifiers of dropped packets from its cumulative power sums, ensuring that the only packets that contribute to the threshold limit are those that went missing since decoding the last quACK.

To account for reordering in loss detection, Robin implements an algorithm similar to the 3-duplicate ACK rule in TCP [5, 60]. In TCP, if three or more duplicate ACKs are received in a row, it is a strong indication that a segment has been lost. Robin considers a packet lost only if three or more packets sent after the missing packet have been received. Other mechanisms could involve timeouts for individual packets similar to the RACK-TLP loss detection algorithm for TCP [11].

4.3.2 Path-Aware CUBIC Congestion Control

Congestion-controlled base protocols must have a congestion response to lost packets that they retransmit due to quACKs, similar to if the loss were discovered by the end-to-end ACK. This ensures friendliness with end-to-end congestion control algorithms that do consider the loss, such as CUBIC [27] in the presence of a connection-splitting TCP PEP. Here, we propose PACUBIC, an algorithm that emulates this "split CUBIC" behavior. PACUBIC uses knowledge of where loss occurs to improve connection throughput compared to end-to-end CUBIC, while remaining fair to competing flows.

Recall that CUBIC [27] reduces its congestion window by a multiplicative decrease factor, $\beta = \beta^* = 0.7$, when observing loss (a congestion event), and otherwise increases its window based on a real-time dependent cubic function with scaling factor $C = C^* = 0.4$:

$$cwnd = C(T-K)^3 + w_{max} \text{ where } K = \sqrt[3]{\frac{w_{max}(1-\beta)}{C}}.$$

Here, $cwnd$ is the current congestion window, w_{max} is the window size just before the last reduction, and T is the time elapsed since the last window reduction.

While a split CUBIC connection has *two* congestion windows, end-to-end PACUBIC only has *one* window representing the in-flight bytes of the end-to-end connection. Conceptually, we want an algorithm that enables PACUBIC's single congestion window to match the sum of the split connection's two congestion windows.

PACUBIC effectively makes it so that we reduce and grow $cwnd$ proportionally to the number of in-flight bytes on the path segment of where the last congestion event occurred. Let r be the estimated ratio of the RTT of the near path segment (between the data sender and the proxy) to the RTT of the entire connection (between end hosts). We use r as a proxy for the ratio of the number of in-flight bytes. If the last congestion event came from a quACK, we use the same real-time dependent cubic function but with the following constants[2]

$$\beta = 1 - r(1-\beta^*) \text{ and } C = \frac{C^*}{r^3}.$$

[2] See Appendix A for more intuition behind β' and C'.

If the last congestion event came from an end-to-end ACK, then we use the original β and C as above.

While this algorithm resembles the congestion behavior of split CUBIC, it is simply an approximation. PACUBIC does not know the exact number of bytes in-flight on each path segment, and the sum of the two congestion windows is simply a heuristic for an inherently different split connection. The main takeaway is that knowing where loss occurs can inform congestion control. We generally hope that quACKs can lead to the development of smarter, path-aware algorithms.

5 Implementation

Module	Language	LOC
QuACK library (Section 3.5)	Rust	1772
Media server/client + integration	Rust	478
`quiche` client integration	Rust	1821
`libcurl` client integration	C	1459
Proxy sidekick binary	Rust	833

Table 4: Lines of code.

We now describe our implementation of Robin [66] for several applications. We integrated sidekick functionality with a simple media client for low-latency streaming and an HTTP/3 (QUIC) client. The total implementation of the quACK library, and proxy and client integrations used 6363 LOC (Table 4).

5.1 Baselines and Applications

The baselines we evaluated against were the performance of two opaque transport protocols without proxy assistance, and the fairness of a split CUBIC connection.

Low-latency media application. We implemented a simple server and client in Rust for streaming low-latency media. The client sends a numbered packet containing 240 bytes of data every 20 milliseconds, representing an audio stream at 96 kbit/s. The sequence number is encrypted on the wire.

The server receives packets. If it receives a nonconsecutive sequence number, it sends a NACK back to the client that contains the sequence number of each missing packet. The client's behavior on NACK is to retransmit the packet. The server retransmits NACKs, up to one per RTT, until it has received the packet.

The server's application behavior is to store incoming packets in a buffer and play them as soon as the next packet in the sequence is available. The de-jitter buffer delay is the length of time between when the packet is stored to when it can be played in-order. Some packets can be played immediately.

HTTP/3 file upload application. We used the popular `libcurl` [45] file transfer library as the basis for our HTTP client, and an `nginx` webserver. The client makes an HTTP POST request to the server. Both are patched with `quiche` [13], a production implementation of the QUIC protocol from Cloudflare, to provide support for HTTP/3.

For our TCP baselines, we used the same file upload application with the default HTTP/1.1 server and client. We used a split-connection TCP PEP [10] that intercepts the TCP SYN packet in the three-way handshake, pretends to be the other side of that connection, and initiates a new connection to the real endpoint. Both clients use CUBIC congestion control.

5.2 Client Integration

In each application, we modified only the *client* to speak Robin and respond to in-network feedback. The server remained unchanged. The modifications were in two parts: following the discovery mechanism to establish bi-directional communication with the proxy, and using the information in the quACK to modify transport layer behavior.

Low-latency media client. The media client has two open UDP sockets: one for the base connection and one for the sidekick connection. When it receives a quACK, it detects lost packets without reordering and immediately retransmits them. The protocol does not have a congestion window nor a flow-control window. The client also sends reset and configuration messages over the sidekick connection.

HTTP/3 file upload client. The HTTP/3 client similarly has an adjacent UDP socket for the sidekick connection on which it receives quACKs and sends reset and configuration messages. The client passes the quACK to our modified `quiche` library, which interprets the quACK and makes transport layer decisions. From the client's perspective, `quiche` tells `libcurl` exactly what bytes to send over the wire.

Our modified `quiche` library uses the quACK to inform the retransmission behavior, congestion window, and flow-control window. The library immediately retransmits lost *frames* in a newly-numbered packet, as opposed to the lost *packet*, similar to QUIC's original retransmission mechanism. We implement PACUBIC, described in Section 4.3.2. We also move the flow-control window (without forgetting packets in the retransmission buffer), but only in the ACK reduction scenario, when the congestion window is nearly representative of that of the sidekick connection's path segment.

5.3 Proxy Integration

Our proxy sniffs incoming packets of a network interface using the `recvfrom` system call on a raw socket. It stores a hash table using Rust's standard library `HashMap` that maps socket pairs to their respective quACKs, and incrementally updates the quACKs for flows that have requested sidekick assistance. It also sends quACKs at their configured frequencies and listens for configuration messages.

6 Evaluation

We evaluated Robin to answer the following questions:

1. Can sidekicks improve the performance of opaque transport protocols in a variety of scenarios while preserving the opaque behavior of the base protocols?

Scenario	Link 1	Link 2	QuACK Interval	Threshold	Success Metric	Emulated?	Real-World?
#1 Low-latency media	1 ms delay, 3.6% loss, 100 Mbit/s	25 ms delay, 0% loss, 10 Mbit/s	2 pkts	8	Reduce tail latency of how long packets are queued in the data receiver's de-jitter buffer.	Yes	Yes
#2 Connection- splitting PEP emulation	1 ms delay, 1.0% loss, 100 Mbit/s	25 ms delay, 0% loss, 10 Mbit/s	30 ms	10	Achieve high throughput; match the performance, congestion control behavior, and fairness of connection-splitting TCP PEPs.	Yes	Yes
#3 ACK reduction	25 ms delay, 0% loss, 10 Mbit/s	1 ms delay, 0% loss, 100 Mbit/s	15 ms	50	Reduce ACK frequency of data receiver; achieve high throughput.	Yes	No

Table 5: Experimental scenarios. Link 1 connects the data sender (client) to the proxy, while Link 2 connects the proxy to the data receiver (server). The quACK interval and threshold represent our sidekick configuration.

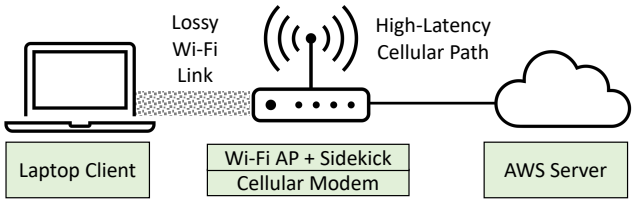


Figure 3: Real-world experimental setup.

2. Can a path-aware congestion control algorithm match the fairness of split TCP PEPs using CUBIC?
3. How do the CPU overheads of encoding quACKs impact the maximum capacity of a proxy with a sidekick?
4. What link overheads does the power sum quACK add and how does it compare to the strawmen?
5. Is Robin robust in a real-world environment?

6.1 Experimental Setup

We modeled the scenarios from Section 2 in both emulated and real-world environments. We answer questions 1-4 in emulation and question 5 in the real world. We use the same m4.xlarge AWS instance as before for the emulated experiments, and as the server in the real-world experiments.

Emulated environment. We emulated a two-hop network topology (Figure 1) in mininet, configuring the link properties using `tc`. In emulation, we represented each link by a constant delay (with variability induced by the queue), a random loss percentage, and a maximum bandwidth. Table 5 describes the parameters for each link to model—e.g., lossy Wi-Fi or a high-latency cellular path—as well as the metrics for success in that scenario. Link 1 connects the data sender (client) to the proxy, while Link 2 connects the proxy to the data receiver (server). On the proxy, we either run a sidekick, a connection-splitting TCP PEP [10], or nothing at all.

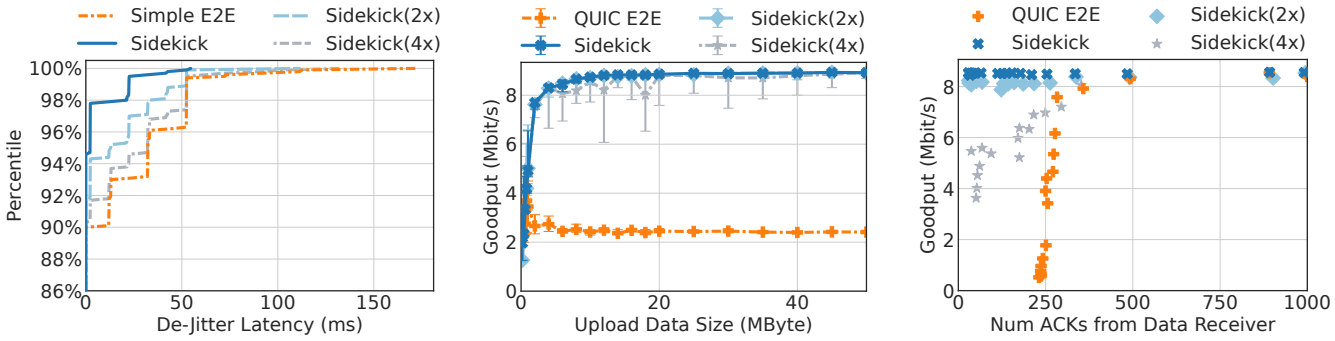
Real-world environment. To test its robustness, we also evaluated Robin over a real-world environment that resembled the scenario on the train (Figure 3). In this setup, a Lenovo ThinkPad laptop, running Ubuntu 22.04.3 with a 4-Core Intel i7 CPU @ 2.60 GHz and 16 GB memory, acted as a client to an AWS instance in the nearest geographical region. The ThinkPad used as an access point (AP) a Lenovo Yoga laptop, running Ubuntu 20.04.6 with a 4-Core Intel i5 CPU @ 1.60 GHz and 4 GB memory, with a 2.4 GHz Wi-Fi hotspot. The AP was connected to the Internet via a JEXtream cellular modem with a 5G data plan. The AP ran sidekick software.

We measured the link properties of each path segment to compare to our emulation parameters. We measured delay and loss using 1000 `pings` over a 100 second period, and bandwidth using an `iperf3` test. On the near segment between the ThinkPad client and the AP, the min/avg/max/stdev RTT was 1.249/37.194/272.168/54.660 ms at 49.8 Mbit/s bandwidth. We observed that loss increased the further away the AP. In our experiments, the client was located roughly 200 feet away in a different room, with 3.6% loss. The far segment between the AP and the AWS server was 48.546/64.381/92.374/6.806 ms with 0.0% loss at 30.9 Mbit/s. In both environments, the cellular link was the bottleneck link in terms of bandwidth, and the corresponding path segments in emulation had similar minimum RTTs and average loss percentages.

6.2 Performance Comparison to Baseline

We first evaluate Robin's main performance goal: In each of the motivating scenarios, we show that Robin can improve performance compared to the base protocol alone, which would not be able to benefit from existing PEPs. Each scenario has a different metric for success—tail latency, throughput, or number of packets sent by the data receiver (corresponding to energy usage or chance of Wi-Fi collisions)—demonstrating the versatility of the sidekick protocol.

Low-Latency Media. The sidekick can reduce tail latencies in a low-latency media stream, representing fewer drops and better quality of experience. The early retransmissions induced by the sidekick reduced the 99th percentile latency of the de-jitter buffer delay from 48.6 ms to 2.2 ms—a 95% reduction (Figure 4a). As long as the quACK interval is less than the end-to-end RTT, the connection benefits from the sidekick.



(a) Scenario #1: Low-latency media. Reduced tail latency of de-jitter delay with earlier retransmission. 5 minute trials.

(b) Scenario #2: Connection-splitting PEP emulation. Improved goodput. 20 trials median. Error bars are 1st and 3rd quartiles.

(c) Scenario #3: ACK reduction. High goodput independent of end-to-end ACK frequency. 10 MB upload.

Figure 4: Comparing the end-to-end baseline protocol to the same protocol with a sidekick connection, using the success metrics for the three scenarios described in Table 5. The Sidekick (Nx) data points show the performance at Nx the quACK interval (sent less frequently) and threshold of the default configurations specified in Table 5.

The sidekick is beneficial in this scenario because it enables the client to sooner detect and retransmit lost packets, and the server to sooner play packets from its de-jitter buffer. The end-to-end mechanism takes one additional received packet to notify of the loss and one end-to-end RTT to retransmit and play the packet (20+52=72ms), resulting in three delayed packets (the three "steps" in Figure 4a) in most cases. The sidekick takes up to two additional packets and one near path segment RTT (20+2=22ms or 20×2+2=42ms), delaying either one or two packets in comparison. Dropped ACKs and quACKs account for the $< 2\%$ of packets with even greater de-jitter latencies.

Connection-Splitting PEP Emulation. The sidekick improves upload speeds when there is a lossy, low-latency link by using quACKs to inform the sender's congestion control. In a scenario with 1% random loss on the link between the proxy and the data sender, the HTTP/3 (QUIC) client achieves 3.6× the goodput for a 10 MB upload with a sidekick compared to end-to-end QUIC (Figure 4b).

When there is no random loss, the sidekick does not impact the performance of QUIC. There are no logical changes to the base protocol in this case because all loss is on the bottleneck link on the far path segment, and the CPU overheads of processing quACKs are negligible.

Knowing *where* congestion occurs is an opportunity for creating smarter congestion control. In PACUBIC, identifying where the loss occured let the data sender reduce the congestion window proportionally to how many packets were in-flight on each path segment. In Section 6.3, we will show that our path-aware congestion control algorithm still matches the fairness of connection-splitting TCP PEPs.

ACK Reduction. Using quACKs in lieu of end-to-end ACKs allows the data receiver to significantly reduce its ACK frequency while maintaining high goodput. In our experiment, QUIC with a sidekick sent 96% fewer packets (mainly ACKs) than end-to-end QUIC before the goodput dropped below 8.5 Mbit/s (Figure 4c). The quACK enables the data sender to promptly move the flow-control window forward, as long as the last hop is reliable.

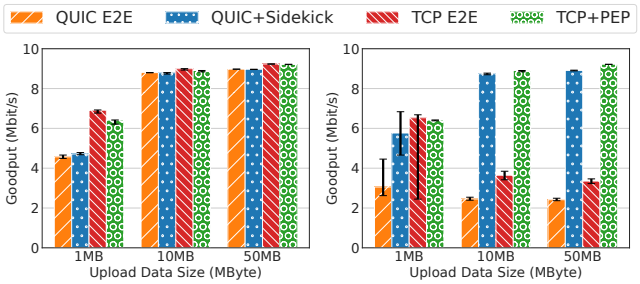
The goodput significantly degrades when reducing the end-to-end ACK frequency without a sidekick. When end-to-end QUIC reduces the ACK frequency to every 80 ms, the data receiver sends $247/138 = 1.8\times$ the packets at $4.5/8.4 = 0.5\times$ the goodput, worse than QUIC with the sidekick in both dimensions (Figure 4c). With a sidekick, the data sender also does not need to change packet pacing to avoid bursts in response to infrequent ACKs, which is why end-to-end QUIC cannot send fewer than ≈ 240 packets.

6.2.1 Configuring the Sidekick Connection

Table 5 shows the quACK interval and threshold we elected for each scenario based on the considerations in Section 4.2. In each experiment in Figure 4, we also show how with less frequent quACKs (2× and 4× the interval) and proportionally-adjusted thresholds, the protocol performs worse, or more variably. Less frequent quACKs means the client reacts later to feedback about the near path segment, and more often has to rely on the end-to-end mechanism. The performance particularly degrades when the quACK interval exceeds the end-to-end RTT. However, even in this case, the base protocol with any sidekick at all performs better than the base protocol alone.

6.3 Fairness Evaluation

It is easy to improve performance without regard to competing flows; however, we demonstrate that PACUBIC can match the fairness of split CUBIC in a TCP PEP connection. We evaluate fairness using Scenario #2 with varying amounts of loss on the near path segment.



(a) 0% loss. (b) 1% loss.

Figure 5: Median goodput for three upload data sizes with 0% and 1% loss on Link 1. 20 trials. Error bars are 1st and 3rd quartiles. With proxy assistance at 1% loss, both QUIC and TCP match the performance of when there is no loss at all.

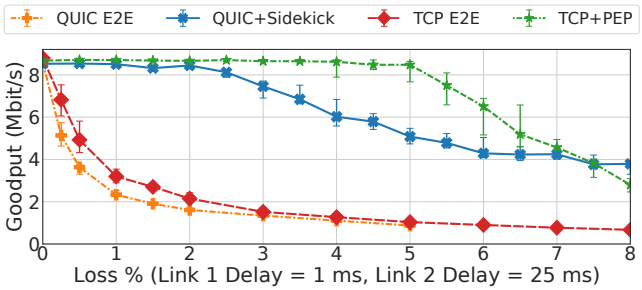


Figure 6: Connection-splitting PEP emulation as a function of near-segment loss rate. In this emulation experiment, QUIC+Sidekick (running PACUBIC) performs similarly to TCP+PEP (each connection running CUBIC) and improves goodput compared with end-to-end protocols. The graph shows median goodput of a 10 MByte upload. QuACK interval is 30 ms, threshold is 10. Error bars show IQR of 10 trials.

QUIC vs. TCP. We first compare QUIC to TCP without either PEP. As both connections use CUBIC, they exhibit similar congestion control behavior and achieve nearly maximum throughput in the emulated network with no random loss (Figure 5a). We attribute the differences to the slightly different retranmission and loss recovery behaviors of QUIC and TCP. The PEPs do not affect the performance.

With even a little loss on the near path segment, both QUIC and TCP dramatically worsen, respectively achieving 28% and 42% of the goodput at 0% loss, for a 10 MB upload (Figure 5b). In both protocols, CUBIC treats every transmission error as a congestion event, even though no amount of reducing the congestion window affects the error rate. QUIC and TCP perform similarly to each other with proxy assistance and 1% loss on the near path segment.

Sidekick vs. TCP PEP. Figure 6 shows that QUIC with a sidekick roughly matches—as intended—the behavior of TCP with a PEP-assisted split connection. At higher loss rates, the near path segment becomes the bottleneck link even with earlier feedback about loss, causing the performance of TCP with proxy assistance to drop. QUIC with a sidekick follows a similar pattern because of its path-aware congestion-control scheme (Section 4.3.2). The results indicate that the sidekick protocol's gains do not come at the expense of congestion-control fairness relative to the split TCP connection.

6.4 Proxy CPU Overheads

	25-Byte Payload		1468-Byte Payload	
	Cycles	%	Cycles	%
Sniff Packet	22417	97.6	22408	97.5
Table Lookup	247	1.1	251	1.1
Parse ID	23	0.1	22	0.1
Encode ID	74	0.3	69	0.3
Other	213	0.9	225	1.0
Total	*22974*	*100.0*	*22975*	*100.0*

Table 6: Breakdown of the CPU cycles spent processing each packet at the proxy. Most cycles are spent on general per-packet overheads as opposed to quACK-specific processing.

The main bottleneck of Robin on a proxy is the CPU. Table 6 shows a breakdown of the number of CPU cycles in each step. The largest overhead was reading the packet contents from the network interface (97.5% of the CPU cycles).

Encoding an identifier in a power sum quACK with $t = 10$ used 74 CPU cycles (0.9%). As a calculation of the theoretical maximum on a 2.30 GHz CPU, the proxy would be able to process 31 million packets/second on a single core. The hash table lookup used 251 cycles and parsing the pseudorandom payload as an identifier used 22 cycles.

In practice, we measured the maximum throughput of Robin to be 464k packets/s with 25-byte payloads and 5.5 Gbit/s (458k packets/s) with 1468-byte packet payloads on a single core (assuming 1500-byte MTUs). This experiment used multiple `iperf3` clients to simulate high load until Robin was unable to keep up with the load on a single core. The packet payload size did not seem to affect results.

We find these achieved throughputs acceptable for edge routers such as Wi-Fi APs and base stations. To deploy Robin on core routers, we would need to reduce the overhead of reading packets from the NIC, such as by bypassing the kernel/user-space protection boundary[3] [20, 48, 62] or using native hardware [8]. We could also scale on multiple cores using symmetric RSS hashing [64].

6.5 Link Overheads

The other cost in terms of using sidekick protocols is the additional data sent by the proxy to the data sender. Too many

[3] A kernel-bypass system like Retina [62] can achieve 25 Gbps on 2 cores while processing raw packets with a 1000-cycle callback (Figure 5(a) in [62]). The Sidekick equivalent would be a 500-cycle callback, and assuming all traffic has requested sidekick help. Throughput scales almost linearly with the number of cores using symmetric RSS hashing. Thus we don't expect proxy overheads to be an issue with modern 100 Gbps network speeds and an optimized implementation even on commodity hardware.

	Data Sender→		←Proxy		←Data Receiver		
	Pkts	**Bytes**	**Pkts**	**Bytes**	**Pkts**	**Bytes**	**Goodput**
QUIC E2E	1.00×	1.00×	1.00×	1.00×	1.00×	1.00×	1.00×
Strawman 1a	0.96×	1.01×	2.02×	1.56×	1.01×	1.03×	3.33×
Strawman 1b	0.94×	1.00×	2.00×	1.78×	1.00×	1.03×	3.53×
Strawman 1c	1.83×	1.06×	2.01×	1.83×	1.00×	1.03×	3.46×
Power Sum	**0.94×**	**1.00×**	**1.03×**	**1.07×**	**1.00×**	**1.03×**	**3.55×**

(a) Scenario #2: Connection-splitting PEP emulation.

	Data Sender→		←Proxy		←Data Receiver		
	Pkts	**Bytes**	**Pkts**	**Bytes**	**Pkts**	**Bytes**	**Goodput**
QUIC E2E	1.00×	1.00×	1.00×	1.00×	1.00×	1.00×	1.00×
Strawman 1a	0.96×	1.00×	9.94×	4.99×	0.04×	0.08×	1.02×
Strawman 1b	0.96×	1.00×	9.95×	7.13×	0.04×	0.08×	1.02×
Strawman 1c	1.91×	1.05×	9.73×	7.41×	0.04×	0.08×	0.97×
Power Sum	**0.96×**	**1.00×**	**1.09×**	**2.56×**	**0.04×**	**0.08×**	**0.98×**

(b) Scenario #3: ACK reduction.

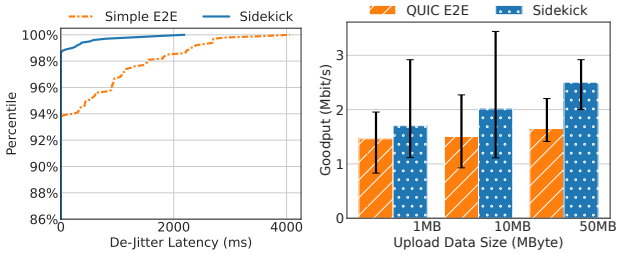
Figure 7: Link overheads for a 10 MB upload. The cells represent the multiplier relative to the end-to-end QUIC baseline for each type of quACK. Lower is better for number of packets and bytes sent on a link. Higher goodput is better. Robin's power sum quACK achieves the success metric for each scenario without incurring the link overheads of the strawmen. We did not evaluate the contrived protocol in Scenario #1.

additional bytes use up bandwidth, and additional packets use up CPU. Figure 7 shows the number of packets and bytes sent at each node comparing the strawmen and power sum quACK to no sidekick connection at all.

Using power sum quACKs increases the packets sent from the proxy to the data sender by 3-9%. These packets either consist mostly of end-to-end ACKs which are sent every packet in `quiche`, or end-to-end ACKs that have been replaced by quACKs in the ACK reduction scenario. We did not evaluate Scenario #1 because it is based on a contrived protocol that lacks many of these features, and the link overheads would not really make sense.

This overhead is representative of the CPU overhead at the client, since quACKs and ACKs take a similar number of cycles to process. In an experiment with Scenario #2 during a period of $\approx$ 90k incoming packets, ACKs took on average 26065 cycles to process while the quACKs took 26369 cycles, 1% more. These cycles come from, i.e., the complex recovery and loss detection algorithms implemented at the end host.

The strawmen have significantly higher link overheads compared to the power sum quACK. The proxy sends up to 10× more packets using Strawman 1a, and also slightly harms the goodput in the congestion control scenario. The reduced goodput is due to the sender mis-identifying received packets as dropped due to dropped quACKs. The proxy achieves higher goodput with Strawman 1b but sends more bytes. Strawman 1c increases the link overheads at both the proxy and the data sender due to larger TCP headers and TCP ACKs. We did not evaluate Strawman 2 due to its impractical decode time.



(a) Low-latency media. CDF of per-packet de-jitter latencies over 10 one-minute trials per protocol.

(b) Path-aware congestion control. Median of 20 trials. Error bars are 1st and 3rd quartiles.

Figure 8: Real-world results. Experiments were run in a moderately well-attended office environment over a Friday afternoon. Trials alternate between the baseline and the sidekick to account for variability in time of day.

6.6 End-to-End Real World Experiments

We discuss the results of our experiments replicating two of our scenarios in the real world, using as context these main differences between emulation and the real-world:

- The RTT is more variable as it depends on interactions in the wireless medium and the shared cellular path.
- Wireless loss can be more variable as nearby 2.4 GHz devices and physical barriers may interfere with the link. Wireless loss also tends to be more clustered in practice.
- The available bandwidth on the shared cellular path is more variable, and depends on the time of day.

Figure 8 shows the results of running the low-latency media and connection-splitting PEP emulation experiments in the real-world. The baseline protocol with a sidekick is able to reduce the 99th percentile de-jitter latency of an audio stream from 2.3 seconds to 204 ms—about a 91% reduction—and improve the goodput of a 50 MB HTTP/3 upload by about 50%. Although the improvements are more conservative compared to emulation in Figure 4a and Figure 4b, each case still benefits the base protocol under all circumstances, compared to end-to-end mechanisms alone.

Part of the difference can be attributed to the network setting. When there is no loss on the near path segment, as can occasionally happen in a real Wi-Fi link, we do not expect to see a difference with a sidekick. When there is more loss on the far path segment, which is variable and depends on the time, we expect the benefit of the sidekick to be less since this equally affects the performance of the base protocol.

The other part of the difference could be made up by future work that better adapts a sidekick connection to real-world variability: The client could improve path segment RTT estimation based on when the proxy receives packets, and use this dynamic estimate in the calculation of r used in β and C. The client could also use this estimate to dynamically adjust the quACK interval. Finally, we could analyze theoretically how PACUBIC responds to traffic patterns in the real world.

7 Limitations

The sidekick approach, and our experiments, are subject to some limitations, which we describe briefly here.

Multipath scenarios. We have only considered sidekick proxies along a single path, and not thought extensively about how quACKs would interact with protocols such as TCPLS [57] that use multiple paths or streams, or even multipath QUIC [18]. To begin thinking about this question, we would have a more complex model of the network: multiple PEPs along a single path, multiple paths each with varying numbers of PEPs, and so on. The proxy can include additional information in the sidekick-reply packet to indicate which path the PEP assistance is on, and the sender can infer from the RTT how far along a path each PEP is relative to others. New sidekick algorithms that come from this model could diagnose troublesome paths, or better allocate network traffic in a multipath connection. Existing algorithms could be applied to individual paths as if they were single-path connections.

Even more diverse network scenarios. The three scenarios we explored all consisted of a lossy Wi-Fi link and a high-latency WAN link. Not all scenarios will be favorable to the sidekick protocol we designed. If the "lossy" section of a network path were on the far path segment from the sender, the sender would not have any more information about the problematic link. To accomodate scenarios like this, sidekick protocols will need more features. For example, the *proxy* would need some way to receive quACKs from the data *receiver*, as well as a mechanism to buffer and retransmit packets [2, 10].

There are likely other scenarios that could benefit from sidekick protocols as described, but we did not evaluate them. For example, if we replaced the lossy Wi-Fi link with a modern wireless link that has a fluctuating physical capacity [9,37,52], the sender may be able to more quickly adapt and make data available for transmission whenever capacity intermittently becomes available.

Practical deployment. The implementation of Robin exists as a research system that has been evaluated in emulation and a limited set of real-world scenarios. Since sidekick protocols require the cooperation of middleboxes and client applications, more work will be needed to standardize the discovery protocol and wire format of sidekick messages described in Section 4, ideally with interest from the IETF. The standards will need to establish several design choices such as how identifiers are computed, how quACKs are transmitted, and the exact mechanisms for security and backwards compatibility. We may also want to standardize sender behavior for specific base protocols, though this could be opaque except to the sender.

The deployment of sidekick protocols can be gradual and backwards-compatible with parties that are either unaware of or do not want to participate in sidekick protocols. To migrate existing client applications, one needs to modify the code to discover a PEP and use information in a quACK to inform the base protocol. To migrate middleboxes, they would need to be modified to listen for sidekick-request markers, then accumulate and send quACKs for participating connections.

Deeper analysis of path-aware congestion control. The correspondence between endpoint-driven PACUBIC and "split CUBIC" is good, and both are better than end-to-end CUBIC in Figure 6), but not exact. The appropriateness of the PACUBIC heuristic, and in general the idea of path-aware congestion control, needs to be further explored. We discuss this more in Appendix A.

8 Conclusion

We presented *sidekick protocols*: an alternate approach to PEPs that leaves the underlying protocol opaque and unmodified on the wire. We described a mathematical technique called a *quACK* that enables middleboxes to refer to packets of the underlying connection without the ability to observe cleartext sequence numbers. We augmented a streaming protocol and a production QUIC implementation (Cloudflare quiche) to make use of information arriving from a proxy on a sidekick connection, including a path-aware congestion-control mechanism called *PACUBIC*. In emulation and a real-world evaluation, the sidekick protocol was able to improve the performance—tail latency, throughput, or energy usage—of these end-to-end base protocols without modifying the wire format or security properties.

Quacknowledgments

We thank Mary Wootters for identifying the relevance of error-correcting codes to the challenging quACK problem, Gerry Wan for his insights in high-speed packet processing that helped improve the proxy performance, and Deepti Raghavan, Matei Zaharia, our shepherd Akshay Narayan, and the anonymous reviewers for other invaluable feedback and discussion. This work was supported in part by NSF grants 2045714, 2039070, 2028733, 1931750, 1918056, 1763256, and DGE-1656518, DARPA contract HR001120C0107, a Stanford Graduate Fellowship, a Sloan Research Fellowship, and by Google, VMware, Dropbox, Amazon, and Meta Platforms.

References

[1] Apple Inc. iCloud Private Relay Overview. https://www.apple.com/icloud/docs/iCloud_Private_Relay_Overview_Dec2021.pdf, Dec. 2021.

[2] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz. Improving TCP/IP performance over wireless networks. In *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking*, MobiCom '95, page 2–11, New York, NY, USA, 1995. Association for Computing Machinery.

[3] C. Batut, K. Belabas, D. Bernardi, H. Cohen, and M. Olivier. *User's Guide to PARI-GP*. Université de Bordeaux I, 2000.

[4] BitTorrent Foundation. BitTorrent (BTT) White Paper v0.8.7. https://www.bittorrent.com/btt/btt-docs/BitTorrent_(BTT)_White_Paper_v0.8.7_Feb_2019.pdf, Feb. 2019.

[5] E. Blanton, D. V. Paxson, and M. Allman. TCP Congestion Control. RFC 5681, Sept. 2009.

[6] J. Border. Google QUIC over satellite links. Presentation, IETF PANRG interim, June 2020.

[7] J. Border, B. Shah, C.-J. Su, and R. Torres. Evaluating QUIC's performance against performance enhancing proxy over satellite link. In *2020 IFIP Networking Conference (Networking)*, pages 755–760, 2020.

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[9] H. Burchardt, N. Serafimovski, D. Tsonev, S. Videv, and H. Haas. VLC: Beyond point-to-point communication. *IEEE Communications Magazine*, 52(7):98–105, 2014.

[10] C. Caini, R. Firrincieli, and D. Lacamera. PEPsal: a performance enhancing proxy designed for TCP satellite connections. In *2006 IEEE 63rd Vehicular Technology Conference*, volume 6, pages 2607–2611, 2006.

[11] Y. Cheng, N. Cardwell, N. Dukkipati, and P. Jha. The RACK-TLP Loss Detection Algorithm for TCP. RFC 8985, Feb. 2021.

[12] D. Clark. The design philosophy of the DARPA internet protocols. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, page 106–114, New York, NY, USA, 1988. Association for Computing Machinery.

[13] Cloudflare, Inc. Quiche. https://github.com/cloudflare/quiche, Feb. 2024.

[14] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy. Virtualized congestion control. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 230–243, New York, NY, USA, 2016. Association for Computing Machinery.

[15] A. Custura, T. Jones, and G. Fairhurst. Impact of acknowledgements using IETF QUIC on satellite performance. In *2020 10th Advanced Satellite Multimedia Systems Conference and the 16th Signal Processing for Space Communications Workshop (ASMS/SPSC)*, pages 1–8, 2020.

[16] A. Custura, T. Jones, R. Secchi, and G. Fairhurst. Reducing the acknowledgement frequency in IETF QUIC. *International Journal of Satellite Communications and Networking*, 41(4):315–330, 2023.

[17] P. Davern, N. Nashid, C. J. Sreenan, and A. H. Zahran. HTTPEP: a HTTP performance enhancing proxy for satellite systems. *Int. J. Next Gener. Comput.*, 2(3), 2011.

[18] Q. De Coninck and O. Bonaventure. Multipath QUIC: Design and evaluation. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, page 160–166, New York, NY, USA, 2017. Association for Computing Machinery.

[19] F. R. Dogar and P. Steenkiste. Architecting for edge diversity: supporting rich services over an unbundled transport. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, page 13–24, New York, NY, USA, 2012. Association for Computing Machinery.

[20] DPDK: Data Plane Development Kit. https://www.dpdk.org/, Sept. 2023.

[21] K. Edeline and B. Donnet. A bottom-up investigation of the transport-layer ossification. In *2019 Network Traffic Measurement and Analysis Conference (TMA)*, pages 169–176, 2019.

[22] D. Eppstein and M. T. Goodrich. Straggler identification in round-trip data streams via Newton's identities and invertible Bloom filters. *IEEE Trans. on Knowl. and Data Eng.*, 23(2):297–306, Feb. 2011.

[23] V. Farkas, B. Héder, and S. Nováczki. A split connection TCP proxy in LTE networks. In R. Szabó and A. Vidács, editors, *18th European Conference on Information and Communications Technologies (EUNICE)*, volume LNCS-7479, Budapest, Hungary, Aug. 2012. Springer.

[24] B. Ford and J. R. Iyengar. Breaking up the transport logjam. In *Proceedings of the 7th ACM Workshop on Hot Topics in Networks*, HotNets '08, pages 85–90, New York, NY, USA, 2008. Association for Computing Machinery.

[25] P. Goyal, M. Alizadeh, and H. Balakrishnan. Rethinking congestion control for cellular networks. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets '17, page 29–35, New York, NY, USA, 2017. Association for Computing Machinery.

[26] J. Griner, J. Border, M. Kojo, Z. D. Shelby, and G. Montenegro. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135, June 2001.

[27] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[28] D. A. Hayes, D. Ros, and O. Alay. On the importance of TCP splitting proxies for future 5G mmWave communications. In *2019 IEEE 44th LCN Symposium on Emerging Topics in Networking (LCN Symposium)*, pages 108–116, 2019.

[29] K. He, E. Rozner, K. Agarwal, Y. J. Gu, W. Felter, J. Carter, and A. Akella. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 244–257, New York, NY, USA, 2016. Association for Computing Machinery.

[30] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, page 181–194, New York, NY, USA, 2011. Association for Computing Machinery.

[31] J. Iyengar, I. Swett, and M. Kühlewind. QUIC Acknowledgement Frequency. Internet-Draft draft-ietf-quic-ack-frequency-07, Internet Engineering Task Force, Oct. 2023. Work in Progress.

[32] J. Iyengar and M. Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021.

[33] J. R. Iyengar and B. Ford. Flow splitting with fate sharing in a next generation transport services architecture. *CoRR*, abs/0912.0921, 2009.

[34] A. Kapoor, A. Falk, T. Faber, and Y. Pryadkin. Achieving faster access to satellite link bandwidth. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–6, 2006.

[35] M. G. Karpovsky, L. B. Levitin, and A. Trachtenberg. Data verification and reconciliation with generalized error-control codes. *IEEE Transactions on Information Theory*, 49(7):1788–1793, 2003.

[36] D. Kliazovich, S. Redana, and F. Granelli. Cross-layer error recovery in wireless access networks: The ARQ proxy approach. *Int. J. Commun. Syst.*, 25(4):461–477, Apr. 2012.

[37] S. Koenig, D. Lopez-Diaz, J. Antes, F. Boes, R. Henneberger, A. Leuther, A. Tessmann, R. Schmogrow, D. Hillerkuss, R. Palmer, T. Zwick, C. Koos, W. Freude, O. Ambacher, J. Leuthold, and I. Kallfass. Wireless sub-THz communication system with high data rate. *Nature Photonics*, 7:977–981, Oct. 2013.

[38] M. Kosek, H. Cech, V. Bajpai, and J. Ott. Exploring proxying QUIC and HTTP/3 for satellite communication. In *2022 IFIP Networking Conference (IFIP Networking)*, pages 1–9, 2022.

[39] M. Kosek, T. Shreedhar, and V. Bajpai. Beyond QUIC v1: A first look at recent transport layer IETF standardization efforts. *IEEE Communications Magazine*, 59(4):24–29, 2021.

[40] Z. Krämer, M. Kühlewind, M. Ihlar, and A. Mihály. Cooperative performance enhancement using QUIC tunneling in 5G cellular networks. In *Proceedings of the Applied Networking Research Workshop*, ANRW '21, page 49–51, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Z. Krämer, S. Molnár, M. Pieskä, and A. Mihály. A lightweight performance enhancing proxy for evolved protocols and networks. In *2020 IEEE 25th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pages 1–6, 2020.

[42] N. Kuhn, F. Michel, L. Thomas, E. Dubois, and E. Lochin. QUIC: Opportunities and threats in SATCOM. In *2020 10th Advanced Satellite Multimedia Systems Conference and the 16th Signal Processing for Space Communications Workshop (ASMS/SPSC)*, pages 1–7, 2020.

[43] T. Li, K. Zheng, K. Xu, R. A. Jadhav, T. Xiong, K. Winstein, and K. Tan. Tack: Improving wireless transport performance by taming acknowledgments. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 15–30, New York, NY, USA, 2020. Association for Computing Machinery.

[44] Y. Li, X. Zhou, M. Boucadair, J. Wang, and F. Qin. LOOPS (Localized Optimizations on Path Segments) Problem Statement and Opportunities for Network-Assisted Performance Enhancement. Internet-Draft draft-li-tsvwg-loops-problem-opportunities-06, Internet Engineering Task Force, July 2020. Work in Progress.

[45] libcurl - the multiprotocol file transfer library. https://curl.se/libcurl/, Sept. 2023.

[46] A. M. Mandalari, A. Lutu, B. Briscoe, M. Bagnulo, and O. Alay. Measuring ECN++: Good news for ++, bad news for ECN over mobile. *IEEE Communications Magazine*, 56(3):180–186, 2018.

[47] A. Martin and N. Khademi. On the suitability of BBR congestion control for QUIC over GEO SATCOM networks. In *Proceedings of the Workshop on Applied Networking Research*, ANRW '22, New York, NY, USA, 2022. Association for Computing Machinery.

[48] S. McCanne and V. Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Technical Conference*, USENIX '93, page 2, USA, 1993. USENIX Association.

[49] A. Mihály, S. Nádas, S. Molnár, Z. Krämer, R. Skog, and M. Ihlar. Supporting multi-domain congestion control by a lightweight PEP. In *2017 International Conference on Internet of Things, Embedded Systems and Communications (IINTEC)*, pages 105–110, 2017.

[50] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*, 49(9):2213–2218, 2003.

[51] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[52] Y. Niu, Y. Li, D. Jin, L. Su, and A. V. Vasilakos. A survey of millimeter wave communications (mmWave) for 5G: opportunities and challenges. *Wireless Networks*, 21:2657–2676, 2015.

[53] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K.-J. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic, and S. Mangiante. Deossifying the internet transport layer: A survey and future perspectives. *IEEE Communications Surveys & Tutorials*, 19(1):619–639, 2017.

[54] C. Perkins, M. Westerlund, and J. Ott. Media Transport and Use of RTP in WebRTC. RFC 8834, Jan. 2021.

[55] M. Polese, M. Mezzavilla, M. Zhang, J. Zhu, S. Rangan, S. Panwar, and M. Zorzi. milliProxy: A TCP proxy architecture for 5G mmWave cellular systems. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*, pages 951–957, 2017.

[56] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? Designing and implementing a deployable multipath TCP. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, page 29, USA, 2012. USENIX Association.

[57] F. Rochet, E. Assogba, and O. Bonaventure. TCPLS: Closely integrating TCP and TLS. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 45–52, New York, NY, USA, 2020. Association for Computing Machinery.

[58] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, Nov. 1984.

[59] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 213–226, New York, NY, USA, 2015. Association for Computing Machinery.

[60] W. R. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC 2001, Jan. 1997.

[61] D. J. D. Touch. Transport Options for UDP. Internet-Draft draft-ietf-tsvwg-udp-options-28, Internet Engineering Task Force, Nov. 2023. Work in Progress.

[62] G. Wan, F. Gong, T. Barbette, and Z. Durumeric. Retina: analyzing 100GbE traffic on commodity hardware. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 530–544, New York, NY, USA, 2022. Association for Computing Machinery.

[63] K. Winstein and H. Balakrishnan. Mosh: An interactive remote shell for mobile clients. In *2012 USENIX Annual Technical Conference*, USENIX ATC '12, pages 177–182. USENIX Association, June 2012.

[64] S. Woo and K. Park. Scalable TCP session monitoring with symmetric receive-side scaling. *KAIST, Daejeon, Korea, Tech. Rep*, 144, 2012.

[65] G. Yuan. Quack. https://github.com/ygina/quack, Feb. 2024.

[66] G. Yuan. Sidekick. https://github.com/ygina/sidekick, Feb. 2024.

[67] G. Yuan, D. K. Zhang, M. Sotoudeh, M. Welzl, and K. Winstein. Sidecar: in-network performance enhancements in the age of paranoid transport protocols. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, HotNets '22, page 221–227, New York, NY, USA, 2022. Association for Computing Machinery.

[68] J. Zirngibl, P. Buschmann, P. Sattler, B. Jaeger, J. Aulbach, and G. Carle. It's over 9000: analyzing early QUIC deployments with the standardization on the horizon. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, page 261–275, New York, NY, USA, 2021. Association for Computing Machinery.

[69] Zoom Video Communications, Inc. Zoom Encryption White Paper. https://explore.zoom.us/docs/doc/Zoom%20Encryption%20Whitepaper.pdf, Aug. 2021.

A Intuitive Analysis of PACUBIC

Here, we dive deeper into the intuition behind the PACUBIC constants (Section 4.3.2), including how they were derived and why the PACUBIC algorithm achieves similar congestion behavior to the CUBIC algorithm in a split connection—we call this behavior "split CUBIC".

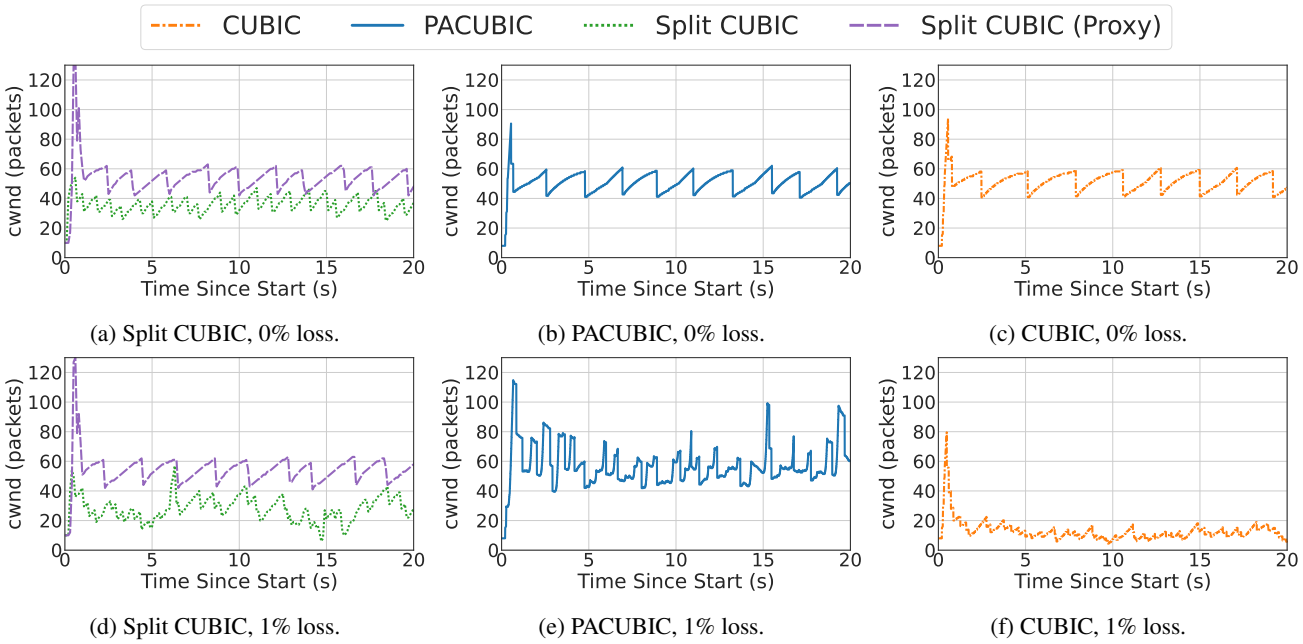
Consider the same network topology as Figure 1 in which a data sender uploads a large file to a data receiver, with help from a sidekick proxy in the middle of the connection. The near path segment connects the sender to the proxy, and the far path segment connects the proxy to the receiver. The near segment is low-delay with varying random loss, and the far segment is high-delay with no random loss. The far segment is the bottleneck link in terms of bandwidth. The actual link parameters are the same as in Scenario #2 of Table 5.

We first discuss how split CUBIC would behave in this setting to conceptually motivate PACUBIC. Consider the congestion windows of each half of the split connection, one taken at the data sender and one at the proxy (Figures 9a and 9d). The far path segment experiences only congestive loss, leading the window at the proxy to fluctuate around the segment's BDP regardless of the loss on the near path segment. The window at the data sender independently determines whether the packets that reach the proxy will be able to fully utilize the window set at the far path segment. The data sender is able to achieve this at low random loss rates, but becomes the bottleneck as loss rates increase (Figure 6).

While split CUBIC has two windows, PACUBIC only has one window representing the in-flight bytes of the end-to-end connection. PACUBIC considers loss detected from both quACKs and end-to-end ACKs. Conceptually, we want an algorithm that would enable PACUBIC's single congestion window to match the sum of CUBIC's two congestion windows, or the total number of in-flight bytes.

With no random loss on the near path segment, PACUBIC (Figure 9b) behaves the same as normal CUBIC (Figure 9c). The congestion window is entirely governed by end-to-end ACKs since the far path segment is the bottleneck link. Note that while the sender may be able to deduce that a loss occurred on the far path segment by combining info from the quACK with the end-to-end ACK, PACUBIC conservatively treats the loss as occurring anywhere on the path.

With some random loss on the near path segment, PACUBIC grows and reduces cwnd based on where the last congestion event occurs (Figure 9e). Note that if the congestion window *cwnd* represents the bytes in-flight in the end-to-end connection, then $r \cdot cwnd$ represents the proportion of bytes in-flight on the near path segment. At a high level, if the data sender discovers loss on the near path segment via the quACK, it holds the $(1-r) \cdot cwnd$ portion of the "far window" constant while applying the CUBIC algorithm to the remaining $r \cdot w_{max}$ of the "near window," representing the bottleneck link.



(a) Split CUBIC, 0% loss. (b) PACUBIC, 0% loss. (c) CUBIC, 0% loss.

(d) Split CUBIC, 1% loss. (e) PACUBIC, 1% loss. (f) CUBIC, 1% loss.

Figure 9: Congestion window of a long-running upload in Scenario #2 (Table 5) with 0% and 1% loss on the near path segmenet. The cwnd is measured at the data sender, except for split CUBIC whose split connection also has a cwnd at the proxy. PACUBIC reacts to every congestion event while keeping the cwnd high. CUBIC performs poorly when there is loss on the near path segment. CUBIC and PACUBIC are implemented in QUIC, while split CUBIC is implemented in TCP using a PEP.

Mathematically, instead of reducing w_{max}, the window size just before the last reduction, by $(1-\beta^*)\cdot w_{max}$, PACUBIC reduces it by only $[1-(1-r(1-\beta^*))]\cdot w_{max} = r(1-\beta^*)\cdot w_{max}$. That is r times the original reduction, a *smaller* amount. We use the RTT ratio r (near path segment to end-to-end) as a proxy for the ratio of the number of in-flight bytes.

Similarly, instead of using a cubic growth function with scaling factor C^* and inflection point $K = K^* = \sqrt[3]{w_{max}(1-\beta^*)/C^*}$, we use a larger scaling factor $C = C^*/r^3$ and thus a shorter inflection point

$$K = \sqrt[3]{\frac{w_{max}(1-\beta)}{C}} = \sqrt[3]{\frac{r\cdot w_{max}(1-\beta^*)}{C^*/r^3}} = r^{4/3}\cdot K^*.$$

The shorter inflection point leads the congestion window to *grow more quickly* since the sender also reacts to feedback about loss more quickly over the low-delay link.

At times, there can be loss detected both in quACKs and in end-to-end ACKs. The end-to-end ACKs have a greater effect since they reduce the congestion window by a larger proportion, until the remaining path segment with loss is the bottleneck link. In this scenario with loss, the bottleneck link at equilibrium is the near path segment. At this point, the quACK primarily determines the congestion window updates. If the far path segment were to become the bottleneck again, the data sender would detect a congestion event via the end-to-end ACK.

PACUBIC has several limitations. Although it beats end-to-end CUBIC, it still performs worse than split CUBIC, especially at high loss rates (Figure 6). Also, it doesn't consider loss on the far path segment any differently than original CUBIC, unlike split CUBIC which treats the two split connections independently. PACUBIC emulates the congestion control behavior and fairness of split CUBIC fairly well as a heuristic, but would benefit from an analysis in a wider variety of network scenarios. It would also benefit from a side-by-side fairness comparison against other congestion control algorithms that perform well in the same scenarios. We'd like the primary takeaway of PACUBIC to be that knowing where loss occurs can cleverly inform congestion control.

VILAM: Infrastructure-assisted 3D Visual Localization and Mapping for Autonomous Driving

Jiahe Cui[123], Shuyao Shi[2], Yuze He[2], Jianwei Niu*[1], Guoliang Xing[2] and Zhenchao Ouyang[34]
[1]*Beihang University* [2]*The Chinese University of Hong Kong*
[3]*Tianmushan Laboratory* [4]*International Innovation Institute of Beihang University*

Abstract

Visual Simultaneous Localization and Mapping (SLAM) presents a promising avenue for fulfilling the essential perception and localization tasks in autonomous driving systems using cost-effective visual sensors. Nevertheless, existing visual SLAM frameworks often suffer from substantial cumulative errors and performance degradation in complicated driving scenarios. In this paper, we propose VILAM, a novel framework that leverages intelligent roadside infrastructures to realize high-precision and globally consistent localization and mapping on autonomous vehicles. The key idea of VILAM is to utilize the precise scene measurement from the infrastructure as global references to correct errors in the local map constructed by the vehicle. To overcome the unique deformation in the 3D local map to align it with the infrastructure measurement, VILAM proposes a novel elastic point cloud registration method that enables independent optimization of different parts of the local map. Moreover, VILAM adopts a lightweight factor graph construction and optimization to first correct the vehicle trajectory, and thus reconstruct the consistent global map efficiently. We implement the VILAM end-to-end on a real-world smart lamppost testbed in multiple road scenarios. Extensive experiment results show that VILAM can achieve decimeter-level localization and mapping accuracy with consumer-level onboard cameras and is robust under diverse road scenarios. A video demo of VILAM on our real-world testbed is available at https://youtu.be/lTlqDNipDVE.

1 Introduction

Visual SLAM utilizes video streams from cameras as input, reconstructs the 3D map of the unknown environment, and simultaneously determines the position and orientation of cameras with respect to their surroundings [5, 10, 13, 38]. It holds the potential to enable the critical perception and localization tasks required in autonomous driving systems [6, 44], especially in challenging environments characterized by the absence of traffic semantics, the lack of high-precision localization and prior driving maps, or where the road surroundings undergoes frequent structural changes. However, as an online localization and mapping paradigm, visual SLAM systems are vulnerable to cumulative drift caused by inherent sensor noises of commodity cameras as well as errors from the feature point extraction and matching algorithms [23]. A recent study [45] shows that current visual SLAM systems can yield up to $75\,m$ of cumulative drift after mapping $2.6\,km$ on real roads. A similar result is also shown in our real-world case study where a state-of-the-art visual SLAM algorithm [4] exhibits a drift of over $10\,m$ in map construction after a $400\,m$ drive on a campus road. Such drift not only leads to significant deviations in vehicle localization but also causes inconsistencies in the constructed 3D map, posing a substantial challenge to the reliability of autonomous driving systems.

To address these challenges, existing studies propose to employ loop-closures [4, 46] or integrate high-precision GNSS locations [5, 7] as global references for error correction. However, loop-closure methods require the presence of looped paths in vehicle trajectories, which is uncommon in autonomous driving situations. GNSS-based methods rely on centimeter-level accuracy GNSS-RTK equipment that can cost up to $4,000 per suite [1] and the availability of GPS signals. Such limitations present significant barriers for adoption in a variety of application scenarios, such as self-parking in underground garages or autonomous driving in urban canyons.

In this work, we exploit intelligent roadside infrastructure as a promising solution for enhancing vehicular SLAM. Intelligent roadside infrastructures, equipped with LiDARs and compute units, are increasingly available not only on public roads [8] but also in places like campuses [20] and parking lots [11, 35]. In particular, LiDARs are being progressively deployed on roadside infrastructures due to their decreasing prices in recent years [2]. Thanks to their stationary nature, infrastructures can obtain accurate localization and measurements of the environmental structure, which can serve as reliable references to correct the cumulative drift in vehic-

*Corresponding author.

ular SLAM. By opportunistically exploiting the references from infrastructure nodes along the road, vehicles can achieve high-performance visual SLAM over long-distance driving without looped paths or GNSS devices.

This paper proposes VILAM, the first infrastructure-assisted vehicular SLAM system that harnesses LiDAR measurements from distributed roadside infrastructure in real time. In designing VILAM, we address several key challenges. First, due to the cumulative drift in visual SLAM, the local map constructed by the vehicle suffers irregular deformation, making its alignment with global references challenging. We address this challenge by devising a novel elastic alignment approach, which optimizes each part of the local map independently to maximum overlap with global references. Second, roadside infrastructures may be installed sporadically on roadways. Therefore, merely aligning the local map with the global references is insufficient for obtaining a consistent global map. VILAM adopts a novel factor graph optimization method to infer the global map efficiently based on the local map and the alignment results. Third, the vast volume of raw LiDAR point clouds makes it challenging to share with passing vehicles. Meanwhile, dynamic objects such as vehicles and pedestrians can significantly deteriorate the scene measurements on the infrastructure. VILAM proposes a lightweight solution, which removes the low-occupancy segments of the accumulated point cloud based on the mobility of objects and reduces the redundancy and amount of 3D points to be shared while ensuring the accuracy of the static scene measurement. Lastly, VILAM does not require high-end sensors or localization devices on vehicles. This facilitates a new mapping paradigm for autonomous driving, especially beneficial in environments lacking pre-loaded HD maps.

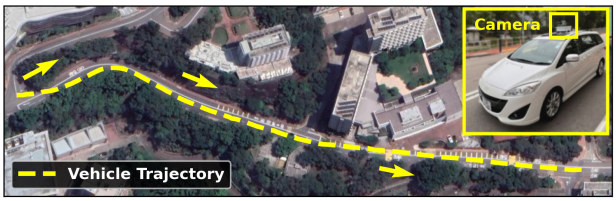
We have implemented VILAM end-to-end on a smart lamppost testbed, spanning multiple real-world road scenarios. We collect a new dataset that covers a total of 17.6 km of driving trajectories with 44 infrastructure nodes in five typical road scenarios. Our extensive evaluation shows that VILAM exhibits a localization error within 0.5 m and a mapping error within 0.7 m, which are less than 15% and 40% of the errors of state-of-the-art baselines, respectively, even when the coverage of roadside infrastructure is as low as 20%. Moreover, VILAM only transmits compact scene measurements at around 236 KB per frame, which reduces the data volume by about 90× and 17× compared with raw and down-sampled point clouds, respectively. Lastly, VILAM achieves an end-to-end system latency within 350 ms for global map correction while updating the local map and vehicle localization at 15 Hz. A demo video of VILAM on our real-world testbed is available at https://youtu.be/lTlqDNipDVE.

2 Related Work

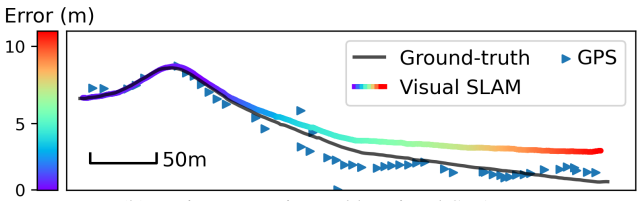
Visual SLAM. Visual SLAM is a specialized branch of SLAM that utilizes visual sensor data, typically from one or multiple cameras, to perform simultaneous mapping and localization. Existing visual SLAM methods [3, 7, 27, 49] build visual maps of 3D distinctive points by aggregating the extracted features from the input images. However, due to the noisy sensor measurements and the errors of feature matching, such methods suffer from cumulative errors over a period of exploration. Some works [4, 28, 38] utilize loop closure detection to eliminate such errors, which is not reliable in autonomous driving scenarios as vehicles rarely revisit the exact same locations in a short time. Some studies [17, 29, 48] construct globally consistent maps by merging the SLAM maps from multiple agents. However, they require the agents to frequently communicate with others and create overlapped maps actively. This requirement complicates the deployment of such methods in autonomous driving, where latency and computational efficiency are paramount. Other works [7, 25, 37] eliminate the cumulative errors by incorporating global constraints, such as global positioning from GNSS sensors. However, high-precision RTK-GNSS units are highly expensive, while the consumer-grade GNSS measurement is insufficient to effectively correct cumulative errors in SLAM. Furthermore, these methods tend to fail in environments where GNSS signals are not available, such as underground parking lots and urban canyons.

Camera-LiDAR Fusion. Previous research [32, 34, 42, 52] explore the fusion of camera images and LiDAR point clouds, with a focus on leveraging the strengths of these two sensors to enhance the perception abilities of autonomous vehicles across diverse scenarios. However, these fusion approaches assume that the camera and LiDAR are mounted together on the vehicle and are precisely calibrated. Therefore, they cannot cope with the unique fusion problem in infrastructure vehicle cooperative systems, where the LiDAR and cameras are typically separated, and the relative positions constantly change as the vehicle moves. Some studies [30, 39] can resolve the relative offset between the camera and LiDAR by registering camera images to LiDAR point clouds, thereby accomplishing sensor data fusion. However, the applicability of these methods is often challenged as their performance relies on the number of feature correspondences between the image and point cloud pair. Due to the significant difference in sensor perspectives between vehicles and infrastructure, there are generally fewer corresponding features.

Infrastructure-assisted Localization and Mapping. The utilization of smart roadside infrastructure to assist autonomous vehicles is an emerging paradigm [33, 47]. Previous works have utilized wireless communication devices like WiFi [19] and UWB [15] on the infrastructures to achieve vehicle localization. However, roadside devices can only provide vehicles with approximate single-point location information, making it challenging to aid vehicles in map construction tasks. Some studies [22, 31] utilize pre-installed visual indicators, such as QR code markers, as references for vehicle localization. However, their effectiveness diminishes as the distance between



(a) The real-world test scene and the test vehicle.



(b) Trajectory estimated by visual SLAM.

Figure 1: A motivational case study. (a) An example scenario where the test vehicle drives through a long road. (b) The errors of the vehicle's trajectory estimated by visual SLAM without correction and the GPS localization samples.

the vehicle and the marker increases, making them unreliable for practical driving scenarios. Some studies [16, 18, 43] achieve vehicle localization by aligning the semantic correspondences observed collectively by both infrastructure and vehicles, thereby accomplishing perception fusion. These studies predominantly focus on localization within the perception range of a single infrastructure and the construction of local maps. Moreover, to ensure alignment accuracy, these studies usually require the equipping of high-precision LiDAR on both vehicle and infrastructure sides.

3 Background and Motivation

3.1 Preliminaries of SLAM Systems

SLAM technology constructs a 3D feature point map of an unknown environment while simultaneously determining the vehicle's localization within the map using onboard visual-inertial sensors, such as cameras and IMUs. Practical SLAM systems exhibit considerable diversity in their implementations, yet most systems typically consist of the following three modules:

Tracking. The tracking module goes through several key steps. First, it detects 2D feature points within the current image frame. Each point corresponds to a 3D map point $\mathbf{m}$ in the environment and has a feature descriptor $\mathbf{f}$. Then, these feature points are matched with those in a previous keyframe [50] according to the similarity of $\mathbf{f}$, establishing associations between the current frame and the keyframes. By combining information on the feature associations and IMU measurements, this tracking module computes the relative pose between the current frame and the keyframe. This process constantly updates the vehicle's localization w.r.t the starting position.

Local Mapping. This module continuously collects tracking results from each individual image frame within a local sliding window. It then leverages Bundle Adjustment [4] to construct a 3D feature point map and estimate the vehicle's trajectory within that window. The feature point map is represented as a set of 3D map points and their feature descriptors: $\mathcal{M} = \{(\mathbf{m}_j \in \mathbb{R}^3, \mathbf{f}_j) | j < N\}$, where $\mathbf{m}_j$ is the 3D coordinates of the jth map point, and $\mathbf{f}_j$ is the corresponding feature descriptor. N denotes the total number of landmarks in the map. The estimated vehicle trajectory is a sequence of transformation matrices $\mathcal{T} = \{\mathbf{T}_0, \mathbf{T}_1, \cdots, \mathbf{T}_i, \cdots\}$. $\mathbf{T}_i = (\mathbf{R}_i, \mathbf{t}_i) \in \mathbb{SE}3$ is the estimation of the vehicle's pose at time i, where $\mathbf{R}_i$ and $\mathbf{t}_i$ denote the rotation and translation components of the pose, respectively. Note that $\mathcal{M}$ and $\mathcal{T}$ are generated in the local coordinate frame, whose origin is the starting point of the SLAM. In practical driving scenarios, $\mathcal{M}$ and $\mathcal{T}$ are typically approximated to the global coordinate frame using the vehicle's initial localization as the starting point.

Error Minimization. This module rectifies the accumulated errors in the estimated map and constructs a consistent global map. These errors mainly stem from factors like the sensor noises as well as feature mismatching during tracking and are integrated into the map frame-by-frame, eventually leading to an offset in the map. Most methods employ Loop Correction [23] to minimize such errors. When vehicles revisit a previously traversed area, the module identifies key features in the current frame, matches them with those stored in the map from previous visits, and then detects if a loop path exists. It then performs adjustments on the map points and the estimated trajectory to ensure consistency on the loop path, thereby mitigating accumulated errors. Some methods also utilize periodic global positional information, such as GNSS locations [5, 25, 37], to limit the growth of accumulated errors.

3.2 Limitations of SLAM

SLAM technologies can encounter unique challenges in autonomous driving scenarios. To illustrate these challenges, we present a motivating case study highlighting the limitations of SLAM in the context of autonomous driving.

We conducted a test drive using a vehicle along a campus road of around 450 m, as depicted in Fig. 1(a). The test vehicle is equipped with a RealSense camera, a consumer-level GPS receiver, and a high-precision RTK unit. We employ ORB-SLAM3 [4], a state-of-the-art visual SLAM algorithm widely adopted in both research and industry, to process vehicle camera images. Fig. 1(b) presents a comparison between the vehicle trajectory estimated by ORB-SLAM3 and the ground truth trajectory recorded by the high-precision RTK unit. Initially (e.g., within the first 100 m), the estimated trajectory closely aligns with the ground truth. However, as the vehicle continues to drive, a noticeable offset occurs. As discussed in Sec. 3.1, this offset is attributed to accumulated errors from the tracking module. In driving scenarios, due to the pres-

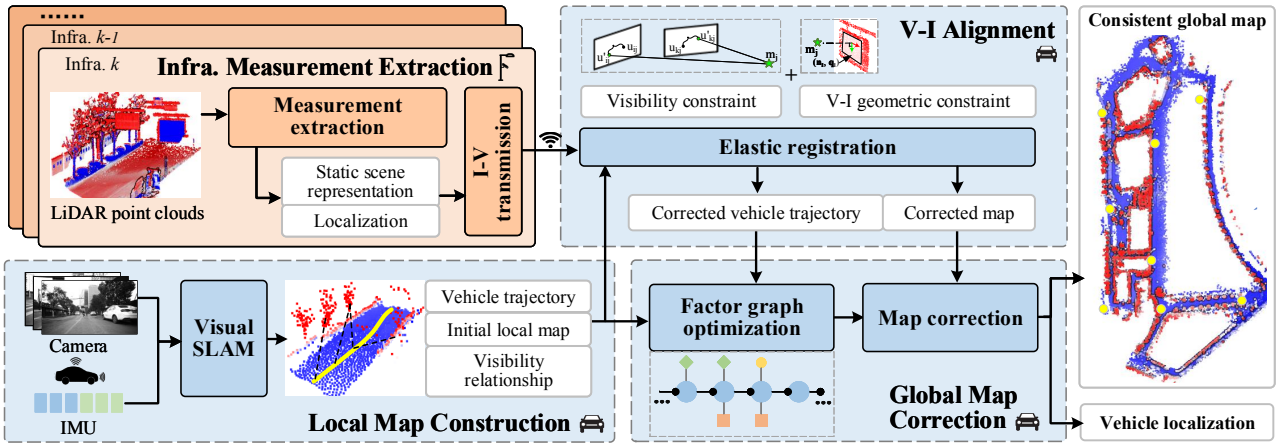


Figure 2: The architecture of VILAM. The orange and blue boxes denote infrastructure and vehicle operations, respectively.

ence of dynamic objects and the rapid perspective changes caused by high driving speed, incorrect feature associations between consecutive frames can lead to more severe pose estimation errors than in robotic applications. Lacking external supervision, these errors accumulate over time, leading to increasingly significant deviations in the vehicle's trajectory. For example, as shown in Fig. 1(b), when the trajectory length exceeds 400 m, the offset can exceed 10 m.

While Loop Correction can minimize the cumulative errors, it requires the vehicle to revisit previously traversed locations frequently to establish loop constraints. Such an approach might be practical in small-scale indoor localization scenarios. However, autonomous vehicles often travel long distances, and their trajectory planners aim to minimize redundant routes. Consequently, the commonly seen trajectory of the vehicles is as shown in Fig. 1(a). This characteristic reduces the frequency of loop closure opportunities compared to other SLAM applications like indoor robotics. Additionally, road and traffic conditions can change rapidly in autonomous driving scenarios. Conventional loop closure methods may struggle to handle dynamic objects and changing environments. For those methods that utilize global positioning to mitigate accumulated errors, Fig. 1(b) shows that consumer-level GPS cannot consistently provide accurate localization (with an average error exceeding 5 m). In certain scenarios (such as underground parking garages), vehicles can even fail to obtain GNSS signals for extended periods. Therefore, a significant gap remains between existing visual SLAM technologies and the vision of applying SLAM in autonomous driving.

4 System Design

4.1 System Overview

As discussed in Sec. 3.2, visual SLAM technology faces substantial challenges in driving scenarios due to significant mapping errors in complex traffic environments and the unreliability of error correction methods. This work addresses these challenges by utilizing intelligent roadside infrastructure to enhance vehicle SLAM. The stationary nature of the roadside infrastructure enables it to obtain accurate measurements of the environmental structure and global localization, which serve as reliable references to correct the accumulative errors in visual SLAM. By receiving and exploiting such references when continuously passing infrastructure nodes, vehicles can achieve high-performance SLAM over extensive distances in the absence of looped paths or precise global localization.

We propose VILAM, the first infrastructure-assisted vehicle SLAM system that harnesses LiDAR measurements from distributed roadside infrastructures as global references to enhance visual SLAM on the vehicle in real time. Fig. 2 shows the overview of VILAM. Specifically, VILAM consists of one module on infrastructure and three modules on the vehicle.

On the infrastructure side, the *infrastructure measurement extraction* module (Sec. 4.3) periodically extracts a lightweight scene representation from the accumulated infrastructure LiDAR point clouds. By filtering out dynamic objects and merging redundant points, it obtains a high-quality yet compact measurement of the environmental structure. This refined measurement, along with the infrastructure's location, are utilized as *global references* and broadcast to all nearby vehicles.

On the vehicle side, the *local map construction* (Sec. 4.2) module employs existing visual SLAM modules based on the onboard sensor to keep constructing an initial 3D local map. To be compatible with diverse visual SLAM frameworks, it extracts three types of data that are common to most of the existing approaches: the feature point map, the historical vehicle localization (i.e., the vehicle trajectory), and the visibility relationship between them, for the following processing with the infrastructure measurement. Once the vehicle receives the global reference from the infrastructure, the *vehicle-infrastructure alignment* (Sec. 4.4) module aligns the local feature point map with the infrastructure measurement to correct the overlapped part of the local map and the latest

vehicle trajectory. In particular, we devise a novel elastic alignment approach to address the deformation in local 3D maps that renders conventional rigid-oriented point cloud registration methods ineffective. Finally, the *global map correction* (Sec. 4.5) module leverages the corrected part of the vehicle trajectory to amend the historical vehicle trajectory and the entire feature point map. This module adopts a novel factor graph-based representation to encode only the lightweight vehicle trajectory and the alignment results. By optimizing the factor graph to iteratively correct the historical trajectory, it can infer a consistent global map efficiently based on the visibility relationship between the vehicle trajectory and the initial feature point map. To minimize compute overhead, the *local map construction* module runs continuously to update the local map and vehicle trajectory. Concurrently, the *vehicle-infrastructure alignment* and *global map correction* modules, triggered by global references from the infrastructure, operate in parallel. This parallel processing strategy guarantees efficient computation and timely updates for the map and localization.

4.2 Local Map Construction

We employ the tracking and the local mapping modules from visual SLAM frameworks (see Sec. 3.1) on the vehicle to continuously estimate the vehicle localization and construct an initial local map based on the camera image sequences. However, existing visual SLAM frameworks exhibit diverse designs due to varying sensor configurations and feature extraction methods [4, 38, 53], resulting in a multitude of map representations and data formats. To ensure robust compatibility across these visual SLAM frameworks, VILAM carefully extracts three data types: (i) The initial 3D feature point map $\mathcal{M}$. (ii) The estimated vehicle trajectory $\mathcal{T}$. (iii) The visibility relationship $\mathcal{C}$ between $\mathcal{M}$ and $\mathcal{T}$. This visibility relationship indicates that a feature point in $\mathcal{M}$ can be observed by the vehicle at different positions along $\mathcal{T}$, denoted by a set of tuples as

$$\mathcal{C} = \left\{(i,j,u,v) \middle| \mathbf{T}_i \in \mathcal{T}, \mathbf{m}_j \in \mathcal{M}, u,v \in \mathbb{Z}\right\} \quad (1)$$

where the tuple (i,j,u,v) indicates that the map point $\mathbf{m}_j$ can be observed by vehicle at pose $\mathbf{T}_i$, and this observation corresponds to a 2D feature point (u,v) on the camera image captured at $\mathbf{T}_i$. These three types of data are common to most existing visual SLAM pipelines, depicting all the map point information in the initial local map and the spatial-temporal relationships between them.

4.3 Infrastructure Measurement Extraction

This module runs on each roadside infrastructure node to generate an accurate 3D measurement of the surrounding environment with precise localization, utilizing the data from the LiDAR on the infrastructure. This measurement is a set of points processed from LiDAR point clouds over a period of time. Thanks to the precise nature of LiDAR and the immovable location of the infrastructure, these points have global coordinates with errors within millimeters. Therefore, measurements from infrastructure distributed along the vehicle trajectory can serve as ideal global references to assist visual SLAM.

Measurement Extraction. LiDAR point clouds can be highly dynamic in traffic scenarios due to moving objects such as vehicles and pedestrians. Points belonging to moving objects may not have correspondences in the 3D local map constructed by the vehicle and thus cannot be used as global references for visual SLAM. Existing methods [14] for filtering moving objects, which typically use deep neural networks, can cause substantial computation overhead. Inspired by [24], we design a lightweight approach to eliminate the dynamic points, utilizing the accumulated point cloud over a period of time. Moreover, such accumulated point clouds have higher resolution to achieve a more accurate measurement of the environment structure. Specifically, we accumulate the LiDAR frames over a time period (e.g., 5 seconds) as a dense point cloud. We then voxelize this point cloud into voxel grids and calculate the occupancy of each grid (i.e., the number of points located in that grid). Static objects are present at consistent positions across all the LiDAR frames so the corresponding voxels have higher occupancy, and vice versa. Therefore, we filter out the points in voxels with low occupancy and merge the rest of the points to the center of each voxel, thus obtaining an accurate but extremely compact measurement of the static scene. Finally, we estimate the planar feature (i.e., the parameter of the tangent plane) for each point in the remaining point cloud and combine it as the feature of this point. Note that planar features not only are presented in the structures like roads and walls, but also can be extracted by differentiation from curved surfaces such as tree trunks. We denote the final point set from the kth infrastructure node as $\mathcal{P}_k$.

I-V Transmission. Given that $\mathcal{P}_k$ comprises only information regarding static scenes around the infrastructure, it can maintain consistency over a period of time. This allows for low-frequency measurement extraction on the infrastructure and a one-shot operation to transmit the measurement from the infrastructure to each vehicle. This approach substantially diminishes the computational and communication overhead on the infrastructure. Moreover, the measurement of an infrastructure node can be broadcast to all passing vehicles, enhancing the scalability of this infrastructure-assisted SLAM framework.

4.4 Vehicle-Infrastructure Alignment

When the vehicle passes an infrastructure node, there is usually a significant overlap between the field of views of the sensors on the vehicle and the infrastructure, which means that the LiDAR measurement from the infrastructure (i.e., $\mathcal{P}_k$)

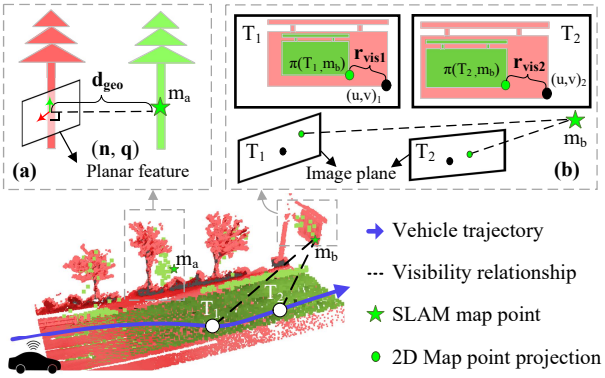


Figure 3: Illustration of (a) *geometry constraint* and (b) *visibility constraint* in the vehicle-infrastructure alignment. The red point cloud is the global reference from the infrastructure, and the green one is the constructed local map.

and the corresponding part of the vehicle's local map (i.e., $\mathcal{M}$) depict the same scene. Since $\mathcal{P}_k$ and $\mathcal{M}$ are both represented by 3D point sets with real-world coordinates, by aligning $\mathcal{M}$ to the global reference $\mathcal{P}_k$, the vehicle can correct the errors in the overlapped part of $\mathcal{M}$ and the corresponding part of vehicle trajectory $\mathcal{T}$. We denoted these local map and vehicle trajectory segments as $\tilde{\mathcal{M}}$ and $\tilde{\mathcal{T}}$, respectively.

Conventional point cloud alignment algorithms, also known as point cloud registration, are predicated on the rigid-body assumption of point clouds, making them ill-suited for this alignment task. These algorithms assume a rigid-body transformation relationship between $\tilde{\mathcal{M}}$ and $\mathcal{P}_k$, suggesting that the misalignment for each map point $\mathbf{m}_j$ in $\tilde{\mathcal{M}}$ is uniform. However, as discussed in Sec. 3.1, $\mathcal{M}$ is accumulated frame-by-frame during the visual SLAM process on the vehicle, with errors from both feature point extraction and vehicle pose estimation being integrated and compounded in $\mathcal{M}$. As a result, the misalignment distribution between each segment of $\tilde{\mathcal{M}}$ and the corresponding segment in $\mathcal{P}_k$ is not uniform.

To address this issue, we design an elastic point cloud registration approach. Our key idea is to optimize each part of $\tilde{\mathcal{M}}$ independently to achieve maximum overlap with $\mathcal{P}_k$ after alignment. This optimization is conducted based on two types of constraint: *geometry constraint* and *visibility constraint*. As illustrated in Fig. 3, the *geometric constraint* restricts the points in $\tilde{\mathcal{M}}$ to their closest planes in $\mathcal{P}_k$, while the *visibility constraint* further optimizes the position of each point in the plane using its coordinates from different viewpoints on the trajectory.

Geometry Constraint. We first associate each map point $\mathbf{m}_j$ with a plane in $\mathcal{P}_k$ by searching its nearest plane feature $(\mathbf{n}_j, \mathbf{q}_j)$ estimated during infrastructure measurement extraction (see Sec. 4.3), where $\mathbf{n}_j$ represents the plane's normal vector and $\mathbf{q}_j$ denotes the plane's center. The distance from $\mathbf{m}_j$ to the plane can be calculated by:

$$\mathbf{d_{geo}}(\mathbf{m}_j) = (\mathbf{m}_j - \mathbf{q}_j)^T \cdot \mathbf{n}_j \tag{2}$$

which denotes the misalignment between $\mathbf{m}_j$ and its corresponding object in $\mathcal{P}_k$. Therefore, we aim to minimize $\mathbf{d_{geo}}(\mathbf{m}_j)$ across the map segment $\tilde{\mathcal{M}}$ to constrain all the map points closer to $\mathcal{P}_k$.

Visibility Constraint. The geometry constraint can locate $\mathbf{m}_j$ to a plane but cannot determine its position within the plane. We propose to utilize the vehicle's visibility of $\mathbf{m}_j$ at different positions on $\tilde{\mathcal{T}}$ to further constrain its coordinate. As discussed in Sec. 3.1, a map point $\mathbf{m}_j$ can be observed by the vehicle from different locations on the trajectory, which is represented by the visibility relationship $\mathcal{C}$. We project $\mathbf{m}_j$ back to the images captured at these locations. If $\mathbf{m}_j$ is accurate, its projection point should be consistent with the corresponding 2D feature point. We calculate the distance between them by:

$$\mathbf{r_{vis}}(\mathbf{m}_j, \tilde{\mathcal{T}}) = \sum_i \left\| \pi\left((\mathbf{T}_i)^{-1}\mathbf{m}_j\right) - (u,v)^T \right\| \tag{3}$$

where $\pi(\cdot)$ denotes the projection function and $(i, j, u, v) \in \mathcal{C}$. Therefore, minimizing $\mathbf{r_{vis}}(\mathbf{m}_j, \tilde{\mathcal{T}})$ provides another constraint on the map points to ensure their consistency from different viewpoints on the vehicle trajectory.

By combining the two types of constraints across all map points, we formulate the point cloud alignment of $\tilde{\mathcal{M}}$ and $\tilde{\mathcal{T}}$ as an optimization problem:

$$\tilde{\mathcal{T}}^*, \tilde{\mathcal{M}}^* = \arg\min_{\tilde{\mathcal{T}}, \tilde{\mathcal{M}}} \sum_j \left\| \begin{bmatrix} \mathbf{d_{geo}} \\ \mathbf{r_{vis}} \end{bmatrix} \right\|_2^2 \tag{4}$$

where we leverage 2-norm to combine $\mathbf{d_{geo}}$ and $\mathbf{r_{vis}}$ as they can be approximated as the normal and radial distances between $\mathbf{m}_j$ and the groundtruth, since $\mathbf{d_{geo}}$ are usually parallel to the direction of the vehicle due to the depth estimation error, while the $\mathbf{r_{vis}}$ are within the image plane, which is usually perpendicular to the vehicle direction. Given the high dimension of the optimization variables (there can be thousands of map points in $\tilde{\mathcal{M}}$), we employ the sparse Levenberg-Marquardt algorithm [40] to solve this optimization problem.

4.5 Global Map Correction

After the alignment, we have the latest part of the initial local map and the vehicle trajectory corrected (i.e., $\tilde{\mathcal{M}}^*$ and $\tilde{\mathcal{T}}^*$) based on the global references from the infrastructure. Ideally, if the roadside infrastructure nodes are densely distributed with their field of view covering the entire road, an accurate global map can be simply obtained by stitching all the corrected map segments. However, given the complexity of real-world road environments and the varied density of roadside infrastructure distribution, it is challenging for roadside sensors to achieve comprehensive coverage of roadways. Therefore, we need to utilize $\tilde{\mathcal{M}}^*$ and $\tilde{\mathcal{T}}^*$ to correct the historical part of $\mathcal{M}$ and $\mathcal{T}$ and obtain a consistent global map. However, directly conducting the correction on $\mathcal{M}$ may incur

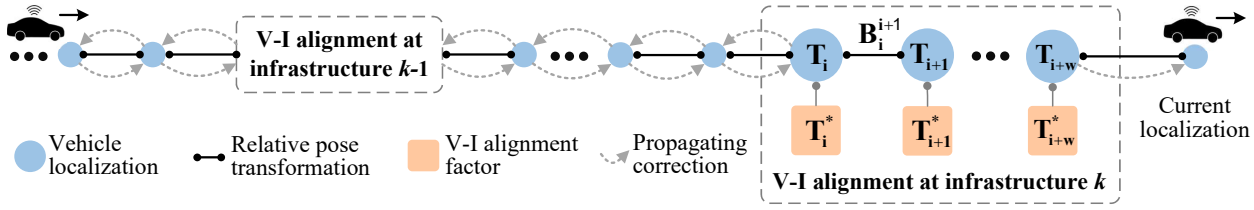


Figure 4: Illustration of the factor graph construction and optimization. The nodes and factors denote the poses along the original vehicle trajectory and the trajectory corrected with global references, respectively. By optimizing this factor graph, VILAM can efficiently correct historical and upcoming vehicle poses.

significant compute overhead since numerous map points in $\mathcal{M}$ have to be adjusted one by one based on $\tilde{\mathcal{M}}^*$. To address this challenge, we adopt a factor graphed-based representation to encode only the original vehicle trajectory $\mathcal{T}$ and its corrected part $\tilde{\mathcal{T}}^*$. By optimizing this lightweight factor graph, we can obtain the corrected historical trajectory and then infer the entire feature point map efficiently based on it.

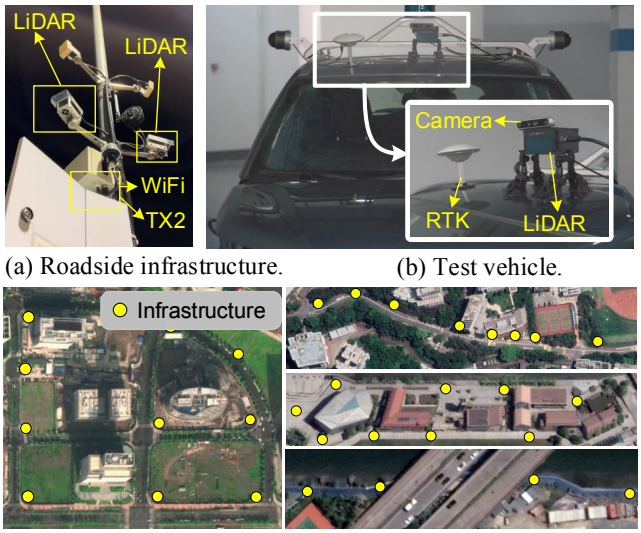
Specifically, Fig. 4 shows the construction and optimization of the factor graph. The nodes in the graph denote the 3D poses along the original vehicle trajectory $\mathcal{T}$, and the binary edge $\mathbf{B}_i^{i+1}$ between two neighbor nodes represents the relative pose transformation between them. For the trajectory segment $\tilde{\mathcal{T}}$ corrected by the global references from the infrastructure, the corrected coordinate $\mathbf{T}_i^*$ acts as the factor of the node $\mathbf{T}_i$. We aim to optimize the value of nodes without factors based on all the binary edges and the factors. It can be described by the following optimization problem:

$$\mathcal{T}^* = \underset{\mathcal{T}}{\operatorname{argmin}} \sum_{\mathbf{T}_i \in \mathcal{T}} \left\| E\left(\mathbf{T}_{i+1}, \mathbf{T}_i \mathbf{B}_i^{i+1}\right) \right\|^2 + \sum_{\mathbf{T}_i^* \in \tilde{\mathcal{T}}^*} \left\| E\left(\mathbf{T}_i, \mathbf{T}_i^*\right) \right\|^2, \tag{5}$$

where $E(\cdot,\cdot)$ calculates the errors between two poses. This problem can be efficiently solved by the incremental factor graph optimization method [21], which iteratively corrects each node value under the constraints of its connected edges in a propagation manner. After such optimization processing over the entire graph, we obtain the entire corrected trajectory $\mathcal{T}^*$. Utilizing the visibility relationship $\mathcal{C}$ between the trajectory and the feature points in $\mathcal{M}$, we can infer the map points based on $\mathcal{T}^*$ and finally reconstruct a consistent global map.

5 Testbed and Dataset

Testbed. Fig. 5 (a) and (b) show the setups of the roadside infrastructure and the test vehicle in the real-world. Each roadside infrastructure is equipped with two Livox Horizon LiDARs installed at a height of 3.5m, covering both sides of the facility in a rear-facing configuration. It should be viable to apply VILAM's idea to 3D cameras on the infrastructure, and we have left this to future work. Additionally, it has an Nvidia Jetson TX2 computing module with Wi-Fi, capable of simple local information processing and communicating



(a) Roadside infrastructure. (b) Test vehicle.

(c) Test scenarios.

Figure 5: A real-world smart infrastructure testbed deployed in diverse scenarios for data collection and system evaluation.

with the vehicle. The vehicle is equipped with a Realsense D455 camera (with built-in IMU). The camera is only used to collect raw images and IMU measurements, without depth information. The on-vehicle computing/communication unit is a laptop with an AMD Ryzen 2.90 GHz CPU and an NVIDIA RTX 2060 GPU. Additionally, the test vehicle is equipped with a Livox HAP LiDAR and an Asense ins570d RTK-GNSS receiver for ground truth collection. Specifically, we repeatedly scan the roads with LiDAR and merge the point clouds offline using an existing mapping system [41] to create the ground-truth map. The vehicle's ground-truth trajectory is obtained by aligning the point cloud frames captured by the LiDAR with the ground-truth map.

Real-world Dataset. We collect an extensive real-world dataset across five typical scenarios using our infrastructure and vehicle testbeds, as shown in Fig. 5(c). Table 1 details the data collected in each scenario. Each of these scenarios poses challenges for existing visual SLAM systems: (i) In the open roads and campus scenarios, there are few closed loops in the vehicle trajectories. (ii) The industrial park and underpasses have limited GNSS signal coverage ($\sim$ 60%), which is completely absent in the underground parking facil-

Table 1: Summary of the test scenarios, where "Traj." represents the vehicle trajectories, "GPS" represents the coverage of GNSS signals in the scenario, "Infra." represents infrastructure.

Scenario	Traj. length	Images	GPS	Infra. nodes
Open Road	6.6 km	45.1 k	~90%	13
Campus	1.3 km	15.4 k	~80%	8
Industrial park	5.9 km	40.2 k	~60%	9
Underpasses	0.4 km	3.6 k	~60%	4
Underground parking	3.4 km	33.0 k	0%	10

ity. The average infrastructure coverage across all scenarios is around 60%. We quantify the infrastructure coverage by calculating the proportion of the vehicle's ground-truth trajectory positions that fall within the infrastructure's point cloud. The inter-infrastructure node distance lies between $20\,m$ and $170\,m$. The vehicle's average speed is $5\,m/s$ (up to $10\,m/s$) due to the speed limits in most test scenarios. In summary, our dataset covers a total of $17.6\,km$ in driving trajectories, including $137.3\,k$ image frames captured by vehicles and $55.2\,k$ point cloud frames from 44 infrastructure nodes. This research has been granted IRB approval.

6 Evaluation

6.1 Evaluation Setup and Metrics

6.1.1 Experiment Setup

On the infrastructure side, the data rate of the LiDAR point cloud is 10 fps. We utilize a 5-second point cloud sequence for measurement extraction and set the voxel grid size to $0.5\,m$. On the vehicle side, for the *local map construction* task, we configure the vehicle camera to capture images at a data rate of 15 fps and IMU measurements at 200 fps. We set the image resolution to 768x480. For the *V-I alignment* task, we set the length of the local map segment to $30\,m$.

6.1.2 Evaluation Metrics

APE, ARE and RTE. We employ Average Positioning Error (APE) and Average Rotation Error (ARE) [36] to evaluate the absolute accuracy of the trajectory estimated by the SLAM algorithm. Specifically, APE quantifies the translational discrepancies between the estimated trajectory and the ground truth trajectory at each frame: $APE = \frac{1}{n}\sum_i^n \left\| E_t\left(trans(\mathbf{T_i^{est}}), trans(\mathbf{T_i^{gt}})\right)\right\|$, where n denotes the frame number. Similarly, ARE measures the rotational discrepancies between the estimated and the ground truth trajectory: $ARE = \frac{1}{n}\sum_i^n \left\| E_r\left(rot(\mathbf{T_i^{est}}), rot(\mathbf{T_i^{gt}})\right)\right\|$. Furthermore, we leverage the Relative Trajectory Error (RTE) from the KITTI Benchmark [12] to compute the degree of trajectory drift over time.

Chamfer Distance. We adopt the Chamfer Distance (CD) to assess the difference between the point cloud map X constructed by the SLAM algorithm and the ground truth map Y. CD is computed by summing the squared distances between nearest neighbor correspondences of two point clouds: $CD(X,Y) = \sum_{x\in X} \min_{y\in Y} ||x-y||_2^2 + \sum_{y\in Y} \min_{x\in X} ||x-y||_2^2$.

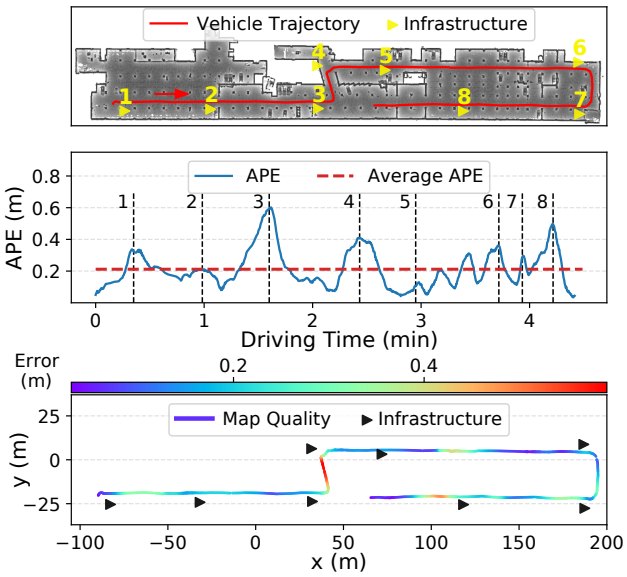


Figure 6: An end-to-end evaluation in the underground parking scenario. Above: the vehicle trajectory and infrastructure locations in the test scene. Middle: the error of the estimated vehicle localization during the driving trace. Bottom: the final map quality along the vehicle trajectory.

6.1.3 Baselines.

We implement three state-of-the-art visual SLAM algorithms as baselines. OpenVINS [13] is a conventional local mapping framework without error minimization. ORB-SLAM3 [4] and GVINS [5] employ loop-closures and GNSS locations, respectively, as global constraints to correct the cumulative drift. VILAM can incorporate all these algorithms within the *local map construction* module. However, due to the diverse GNSS coverage across different test scenarios, we adopt ORB-SLAM3 in VILAM's implementation for evaluation unless otherwise noted.

6.2 End-to-end System Evaluation

We evaluate the end-to-end system performance of VILAM in real-world scenarios described in Table 1. Fig. 6 illustrates the performance of VILAM in the Underground Parking scenario. A video clip of the complete localization and mapping result of the test vehicle is available at https://youtu.be/lTlqDNipDVE. In the upper subfigure, the grey background represents the floorplan of the parking lot extracted from the ground truth map, the red line depicts the driving trajectory, and the yellow dots represent the locations of the infrastructure nodes. The node is numbered in the

Table 2: Trajectory accuracy comparison of VILAM with existing SLAM methods across various scenarios. The unit of the APE metric is meters.

Method	Open roads		Campus		Industrial park		Underpasses		UDG parking		**All Scenarios**	
	APE	RTE	APE	RTE	APE	RTE	APE	RTE	APE	RTE	APE	RTE
OpenVINS [13]	8.170	3.54%	10.406	9.92%	4.772	1.48%	11.30	6.71%	6.144	3.37%	8.158	5.01%
OpenVINS + VILAM	**0.355**	**1.46%**	**0.493**	**3.93%**	**0.380**	**0.60%**	**0.347**	**3.04%**	**0.294**	**0.71%**	**0.373**	**1.94%**
ORB-SLAM3 [4]	5.745	2.91%	2.646	2.45%	2.556	1.60%	3.568	3.85%	3.278	2.06%	3.558	2.58%
ORB-SLAM3 + VILAM	**0.351**	**1.87%**	**0.427**	**1.75%**	**0.296**	**1.16%**	**0.359**	**2.63%**	**0.219**	**0.76%**	**0.330**	**1.63%**
GVINS [5]	2.456	3.37%	1.710	3.52%	3.505	2.41%	4.219	3.69%	4.059	2.78%	3.190	3.15%
GVINS + VILAM	**0.403**	**2.30%**	**0.354**	**2.38%**	**0.325**	**0.65%**	**0.340**	**3.13%**	**0.249**	**0.42%**	**0.335**	**1.77%**

order the vehicle passes by. The middle subfigure shows the real-time localization accuracy (APE) of VILAM. The numbered vertical lines represent the moment the vehicle encounters an infrastructure node. We observe that, in the areas between infrastructures, there is a slight upward trend in the localization error. The error is corrected once the vehicle passes an infrastructure and optimizes its local map through *V-I alignment*. The lower subfigure shows the trajectory estimated by VILAM and the accuracy of the constructed map. We color-code the vehicle trajectory using the map-construction error (Chamfer Distance) of the corresponding region. It can be seen that, with *global map correction*, the error distribution of the map maintains consistent uniformity in both infrastructure-covered and uncovered regions. We further evaluate the impact of the coverage of infrastructure deployment on VILAM's performance in Sec. 6.6.

6.3 Performance of VILAM

6.3.1 Trajectory Estimation

As mentioned in Sec. 4, VILAM tracks the vehicle by associating image frames to derive the vehicle trajectory. Therefore, the accuracy of the estimated trajectory indicates VILAM's continuous real-time localization performance. Table 2 presents a comparison of trajectory accuracy between the baselines and VILAM implementations based on them across five testing scenarios. The results show that all VILAM implementations outperform the corresponding baseline algorithms. Specifically, OpenVINS exhibits substantial errors (i.e., over 8.1 m on average) in all scenarios. ORB-SLAM3 and GVINS enhance accuracy by introducing global constraints but still have significant errors, especially in scenarios that lack loop paths (e.g., open roads and underpasses) and GNSS coverage (e.g., underground parking), respectively. In contrast, VILAM achieves an APE within 0.4 m and an RTE within 2% on average across all scenarios while demonstrating robustness to the algorithms used in the *local map construction* module.

Fig. 7 shows a further evaluation of the benefits of the VILAM framework exploiting global references from roadside infrastructure. "Baseline" and "VILAM " refer to the original baseline algorithms and the VILAM implementations using these algorithms in the *local map construction* module, respectively. "w/ Landmark" and "w/ Infra." indicates the modified baseline algorithms. "w/ Landmark" correct SLAM by utilizing the detected infrastructure positions from vehicle-side images as reference landmarks. "w/ Infra." correct the SLAM results by directly integrating the point clouds from the infrastructure through a conventional point cloud registration method [26]. We can find that simply utilizing information from the infrastructure can improve the performance of the baseline algorithms. However, such improvements are minor for ORB-SLAM3 and GVINS, as they utilize loop detection and GNSS to correct some errors, respectively. In contrast, VILAM significantly outperforms "w/ Infra." and "w/ Landmark" across all three basic algorithm implementations, achieving over an 80% reduction in APE. This is because VILAM's elastic alignment approach overcomes the deformation in the local map, thereby significantly improving alignment accuracy. Moreover, the *global map correction* module of VILAM utilizes the alignment results to optimize both the historical map and the real-time localization results. We delve further into the performance of these two modules in Section 6.5.

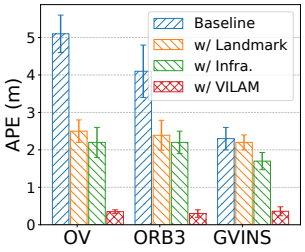


Figure 7: Performance improvement of VILAM using different SLAM methods for *local map construction*.

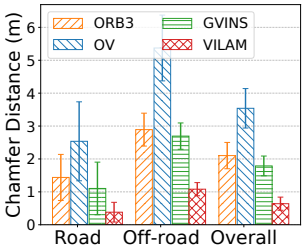


Figure 8: The Chamfer Distance between the constructed map and the ground truth map.

6.3.2 Map Quality Evaluation

In this section, we evaluate the quality of the map constructed by VILAM. Like mainstream SLAM methods, VILAM builds a 3D feature point map, primarily used for vehicle relocalization by reusing the map. Therefore, we mainly focus on the accuracy of the reconstructed map points' coordinates and the overall consistency of the map.

Fig. 8 compares the CD between the estimated map of

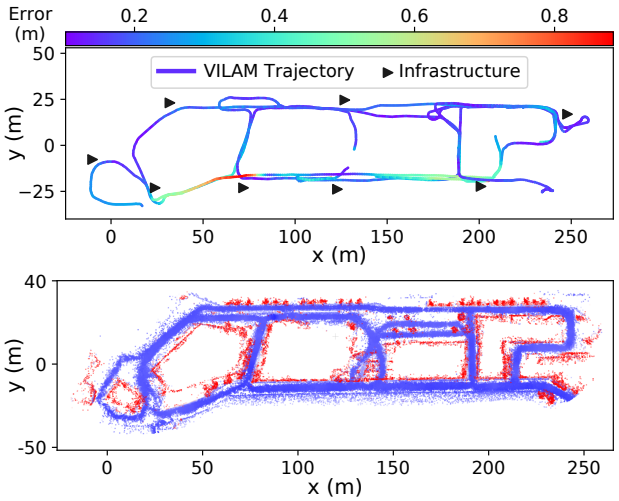


Figure 9: Visualization of the trajectory and the 3D feature point map estimated by VILAM. Above: the infrastructure locations and the estimated vehicle trajectory by VILAM. Bellow: the bird-eye view of the 3D map built by VILAM, where colors(blue to red) indicate different point heights.

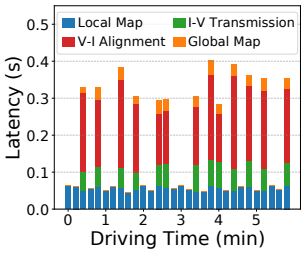


Figure 10: End-to-end latency of VLIAM on the vehicle side.

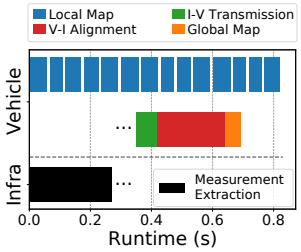


Figure 11: The detailed runtime of each module.

Table 3: The average size and transmission time of the shared data measured on an 802.11ac network. PC denotes "Point Cloud", and SSR denotes "Static Scene Representation."

Data type	Trans mode	Sync	Size	Time
Raw PC	Continuous	Required	21.1 MB	3.93s
Downsampled PC	Continuous	Required	4.1 MB	0.78s
VILAM SSR	**Single-shot**	–	**236.3 KB**	**0.04s**

VILAM / baseline SLAM and the ground-truth map. The higher the CD, the greater the error of the estimated map. As autonomous driving perception tasks primarily focus on road information and relocalization tasks mainly concern off-road three-dimensional structure and texture features, we divide the map points into two main categories: road and Off-road structure. Consistent with the trajectory evaluation results, baseline methods incorporating global constraints such as loop-closing or GNSS (ORB-SLAM and GVINS) exhibit an improvement over approaches relying solely on Local Mapping. However, the overall map accuracy of these methods still falls short when compared to the ground truth. VILAM addresses this limitation through its *global map correction* module. By effectively integrating corrections from multiple infrastructure-based VI-Alignment with the factor graph, VILAM enhances global consistency and improves map accuracy. We visualize the point cloud map constructed by VILAM in Fig. 9. Due to the high accuracy and consistency of the map, the structures of buildings around the road are clearly discernible.

6.4 System Overhead

System Latency. In Fig. 10, we present the end-to-end latency distribution of VILAM throughout a driving trace. Here, "Local Map" represents *local map construction*, and "Global Map" represents *global map correction* . The infrastructure-related tasks, namely *I-V Transmission*, *V-I alignment*, and *global map correction*, are only triggered when the vehicle encounters an infrastructure. VILAM operates with minimal computational overhead during intervals without infrastructure coverage, as only the *local map construction* task remains active. A detailed timeline as the vehicle passes through one of the infrastructures is illustrated in Fig. 11. Thanks to VILAM's concurrent design, *local map construction* is not blacked by the infrastructure-related tasks. This ensures it outputs continuous vehicle localization based on previously corrected maps, maintaining real-time system performance. It is critical for downstream tasks that require real-time localization, such as trajectory planning. The average latency for infrastructure-related tasks is about 0.35 s, ensuring a near real-time correction of both the map and localization after the vehicle passes through an infrastructure.

I-V Transmission. We further evaluated the data transmission overhead associated with the vehicle's acquisition of point clouds from infrastructure. The results are presented in Table. 3. When directly streaming a raw point cloud sequence (about 100 frames) to the vehicle, the data transfer volume exceeds 20 MB, and it is also burdened by a considerable transmission latency. After performing voxel downsampling on each point cloud frame, the number of points is reduced, yet the final data volume still surpasses 4 MB. This can be attributed to the fact that points in each frame contain redundant measurements of static structures within the scene. As discussed in Sec. 4.3, VILAM addresses this by merging the points of static structures across multiple frames, effectively eliminating redundancy. Consequently, the final transmitted Static Scene Representation requires a mere 236 KB of data.

Influence of Infrastructure Measurement Precision. In Sec. 4.3, the voxel grid size influences the size and precision of the extracted representation, which in turn affects the latency of I-V transmission and the subsequent performance of the *V-I alignment*. As illustrated in Fig. 12, when the voxel size is increased, more redundant points occupying the same spatial location are merged, leading to a consistent reduc-

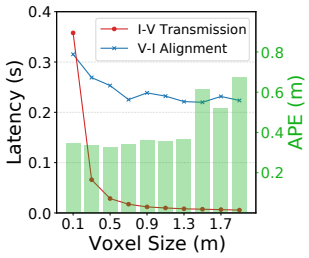


Figure 12: The impact of voxel size on the system overhead and the performance of VILAM.

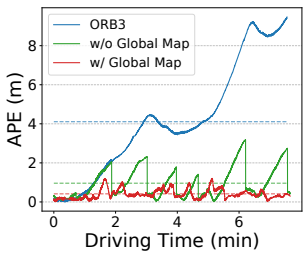


Figure 13: The influence of the *global map correction* module on the performance of VILAM.

tion in the size of the representation and thereby significantly decreasing the transmission latency. Meanwhile, the latency of *V-I alignment* only reduces at first. This reduction is due to the decreased point number after the voxel grid partition, resulting in faster neighboring searches during alignment. However, when the voxel size exceeds about $0.7\,m$, this latency reduction plateaus because the reduced precision of the representation at larger voxel sizes means that the joint optimization process requires more time to converge. The green bars in Fig. 12 depict the influence of voxel size on the accuracy of the *V-I alignment*. There is a slight degradation in the accuracy with the voxel size increases. In our practical implementation, we set the voxel size to $0.5\,m$, which strikes a balance by ensuring lower latency without compromising accuracy.

6.5 Micro Benchmarks

Performance of V-I Alignment. The result of the *V-I alignment* directly influences the accuracy of the global constraints that VILAM derives from the infrastructure. In this section, we provide a detailed evaluation of the performance of *V-I alignment*. We compare the following two registration approaches with the proposed *V-I alignment* method. 1) image-to-point (I2P): The continuous frame images acquired by the vehicle near an infrastructure are sequentially registered with the infrastructure point cloud. 2) Point-to-point (P2P): Align the vehicle's local map with the infrastructure point cloud using 3D point cloud registration algorithms.

As shown in Table. 4, CorrI2P [39] exhibits large overall error. This is mainly attributed to the significant perspective difference between the vehicle's camera and the infrastructure's LiDAR. Consequently, few feature correspondences are matched between the vehicle camera images and the infrastructure point cloud, leading to lower registration accuracy and poor robustness. P2P approaches, on the other hand, utilize the entire local point cloud map from the vehicle side, efficiently improving the number of corresponding features and enhancing the completeness of the perception perspective compared to the I2P method. However, as described in Sec. 4.4, the local map exhibits deformations due to cumula-

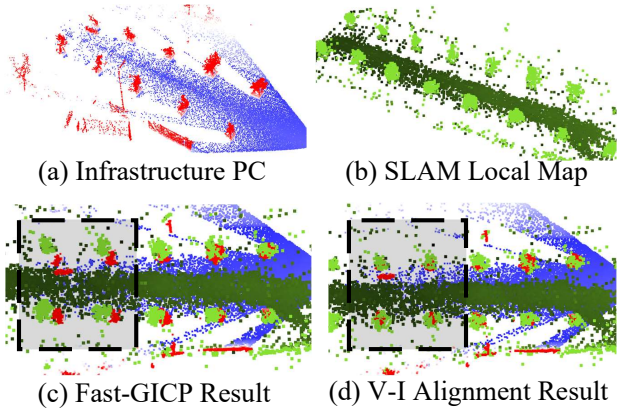


Figure 14: Visualization of the registration between the vehicle local map and the infrastructure measurements.

Table 4: Performance of the proposed elastic alignment method and conventional registration algorithms. "VILAM VIA" denotes the V-I Alignemnt module of VILAM, "I2P" represents image-to-point cloud registration algorithm, and "P2P" represents point cloud-to-point cloud registration algorithm.

Method	Type	APE (m)	ARE (°)	Time (s)
CorrI2P [39]	I2P	1.74 ± 0.65	10.31 ± 2.29	2.98
MAC [51]	P2P	1.57 ± 0.27	8.71 ± 2.36	0.65
Fast-GICP [26]		1.36 ± 0.34	9.07 ± 2.19	**0.13**
S-ICP [9]		1.07 ± 0.39	7.39 ± 1.85	0.61
VILAM		$\mathbf{0.31 \pm 0.08}$	$\mathbf{2.29 \pm 0.52}$	0.24

tive errors. Rigid body-based registration methods, such as MAC [51] and Fast-GICP [26], struggle to accurately align most regions of the local map to the infrastructure measurement. Taking Fast-GICP as an example, as depicted in Fig. 14 (c), only the starting region of the local map matches well with the infrastructure point cloud. Yet, there is a significant deviation towards the end of the local map (highlighted by the grey rectangle).

S-ICP [9] introduces a scale factor into the rigid-body registration model, which helps alleviate errors caused by the rigid-body assumption and achieve lower APE. The proposed *V-I alignment* further divides the local map into fine-grained fragments, allowing each region of the local map to align better with the infrastructure measurement, as shown in Fig. 14 (d). This significantly reduces APE and ARE, with only a slight increase in calculation delay. Furthermore, the concurrent design of VILAM ensures that it does not impact the real-time performance of the overall system, making this latency negligible.

Performance of Global Map Correction. In this section, we evaluate the influence of the *global map correction* module on mapping consistency. We perform ORB-SLAM3, VILAM without *global map correction*, and the full VILAM setup on the data trace and assessed the resultant trajectory accuracy.

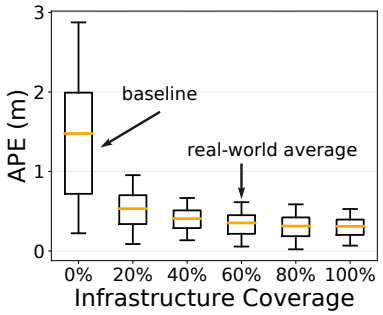


Figure 15: Performance of VILAM under different infrastructure density.

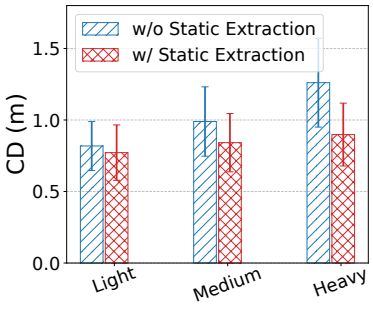


Figure 16: Performance of VILAM under different traffic conditions.

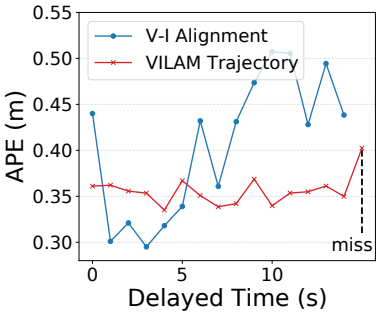


Figure 17: Performance of VILAM under different I-V Transmission latency.

The result is presented in Fig. 13. As the vehicle's driving time increases, the trajectory error of ORB-SLAM3 demonstrates a continuous growth trend due to the absence of global constraints. In the case of VILAM without *global map correction*, only the trajectories near the infrastructure are corrected by *V-I alignment*. The trajectory errors persistently escalate for the gap areas between the infrastructures, resulting in abrupt error shifts at the subsequent infrastructure points (e.g., the APE at the 2nd minute in Fig. 13). Although the APE without *global map correction* is substantially reduced compared to the baseline SLAM methods, such inconsistencies seriously degrade the map quality. Upon integrating the *global map correction* module, the overall APE remains within a low range.

6.6 Robustness Analysis

Infrastructure Coverage. As observed in Sec. 6.2, the accuracy of VILAM can be affected by the coverage of infrastructure, i.e., the proportion of the area that infrastructure LiDARs can perceive. We evaluate VILAM under varying infrastructure coverage in an underground parking garage. Specifically, we place mobile poles equipped with LiDARs at different intervals to simulate varied infrastructure coverage. The results are depicted in Fig. 15. Although the error escalates as the coverage diminishes, VILAM significantly outperforms the baseline even with extremely low infrastructure coverage (i.e., the average APE is kept around $0.5\,m$ at 20% infrastructure coverage). This demonstrates VILAM's robustness to diverse infrastructure setups in the real world.

Environmental Dynamics. We compare the performance of VILAM with and without static scene extraction under different traffic conditions. The results in Fig. 16 show that the error of VILAM without static scene extraction increases significantly under heavier traffic, while the CD of VILAM remains under $1\,m$ across all the traffic conditions. This evaluation confirms VILAM's robustness to environmental dynamics.

Delayed I-V Transmission. During the transmission of the extracted Static Scene Representation from the infrastructure to the vehicle, connectivity issues or transmission errors may delay VILAM from timely accessing the infrastructure's information for map correction. To evaluate the influence of the delayed time on VILAM's performance, we manually set latency at the infrastructure side to simulate transmission delays. The result is shown in Fig. 17. The delay time is measured from the moment when the vehicle is closest to the infrastructure. Interestingly, for *V-I alignment*, moderate transmission delays can actually enhance alignment accuracy. This benefit arises as the vehicle constructs a more extensive local map during the delayed time, leading to a larger overlap with the infrastructure point cloud. However, as the delay time further increases, the cumulative error of the local map intensifies, leading to a reduction in *V-I alignment* accuracy. Meanwhile, the overall trajectory precision of VILAM is relatively impervious to delayed times. A noticeable trajectory error increase occurs only when the vehicle entirely misses the current infrastructure's measurement. The *global map correction V-I alignment* can leverage this information to maintain a consistent trajectory as long as the vehicle eventually receives the infrastructure measurement and undergoes *V-I alignment*.

7 Conclusion

In this paper, we present VILAM, the first system that leverages distributed roadside infrastructures to accomplish high-precision and globally consistent localization and mapping for autonomous vehicles. We implement VILAM end-to-end and evaluate its performance across various challenging driving scenarios. The experiment results show that VILAM effectively enhances the performance of existing SLAM methods in terms of localization accuracy, map consistency, and system robustness.

Acknowledgements. We would like to thank our shepherd Dongsu Han and the anonymous reviewers for their valuable feedback. This work was supported in part by National Key R&D Program of China under Grant No. 2023YFB4503700, Research Grants Council (RGC) of Hong Kong under General Research Fund under Grant 14222222, the Centre for Perceptual and Interactive Intelligence (CPII) under Grant EW01610 (RP4-3), NSFC under Grant No. LY23F020026, 62303036, 62372027, 62372028 and U23B2025.

References

[1] Pwrpak7-e1, [2023] url=https://novatel.com/products/gnss-inertial-navigation-systems/combined-systems/pwrpak7d-e1, journal=Novatel.

[2] Lidar market size, [2023] url=https://www.mordorintelligence.com/industry-reports/global-lidar-market, journal=LiDAR Market Size, Overview.

[3] Fawad Ahmad, Hang Qiu, Ray Eells, Fan Bai, and Ramesh Govindan. CarMap: Fast 3d feature map updates for automobiles. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1063–1081, Santa Clara, CA, February 2020. USENIX Association.

[4] Carlos Campos, Richard Elvira, Juan J Gómez Rodríguez, José MM Montiel, and Juan D Tardós. Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam. *IEEE Transactions on Robotics*, 37(6):1874–1890, 2021.

[5] Shaozu Cao, Xiuyuan Lu, and Shaojie Shen. Gvins: Tightly coupled gnss–visual–inertial fusion for smooth and consistent state estimation. *IEEE Transactions on Robotics*, 38(4):2004–2021, 2022.

[6] Jun Cheng, Liyan Zhang, Qihong Chen, Xinrong Hu, and Jingcao Cai. A review of visual slam methods for autonomous driving vehicles. *Engineering Applications of Artificial Intelligence*, 114:104992, 2022.

[7] Kai-Wei Chiang, Guang-Je Tsai, Hone-Jay Chu, and Naser El-Sheimy. Performance enhancement of ins/gnss/refreshed-slam integration for acceptable lane-level navigation accuracy. *IEEE Transactions on Vehicular Technology*, 69(3):2463–2476, 2020.

[8] Christian Creß, Zhenshan Bing, and Alois C Knoll. Intelligent transportation systems using external infrastructure: A literature survey. *arXiv preprint arXiv:2112.05615*, 2021.

[9] Shaoyi Du, Nanning Zheng, Shihui Ying, Qubo You, and Yang Wu. An extension of the icp algorithm considering scale factor. In *2007 IEEE International Conference on Image Processing*, volume 5, pages V–193. IEEE, 2007.

[10] Richard Elvira, Juan D Tardós, and Jose MM Montiel. Orbslam-atlas: a robust and accurate multi-map system. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6253–6259. IEEE, 2019.

[11] Abrar Fahim, Mehedi Hasan, and Muhtasim Alam Chowdhury. Smart parking systems: comprehensive review based on various aspects. *Heliyon*, 7(5), 2021.

[12] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3354–3361, 2012.

[13] Patrick Geneva, Kevin Eckenhoff, Woosik Lee, Yulin Yang, and Guoquan Huang. Openvins: A research platform for visual-inertial estimation. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4666–4672. IEEE, 2020.

[14] Yulan Guo, Hanyun Wang, Qingyong Hu, Hao Liu, Li Liu, and Mohammed Bennamoun. Deep learning for 3d point clouds: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 43(12):4338–4364, 2020.

[15] O Hassan, I Adly, and KA Shehata. Vehicle localization system based on ir-uwb for v2i applications. In *2013 8th International Conference on Computer Engineering & Systems (ICCES)*, pages 133–137. IEEE, 2013.

[16] Yuze He, Li Ma, Zhehao Jiang, Yi Tang, and Guoliang Xing. Vi-eye: semantic-based 3d point cloud registration for infrastructure-assisted autonomous driving. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 573–586, 2021.

[17] F Hidayat, BR Trilaksono, and H Hindersah. Distributed multi robot simultaneous localization and mapping with consensus particle filtering. In *Journal of physics: conference series*, volume 801, page 012003. IOP Publishing, 2017.

[18] Md Hossain, Ibrahim Elshafiey, Abdulhameed Al-Sanie, et al. Cooperative vehicle positioning with multi-sensor data fusion and vehicular communications. *Wireless Networks*, 25(3):1403–1413, 2019.

[19] Mohamed Ibrahim, Ali Rostami, Bo Yu, Hansi Liu, Minitha Jawahar, Viet Nguyen, Marco Gruteser, Fan Bai, and Richard Howard. Wi-go: accurate and scalable vehicle positioning using wifi fine timing measurement. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 312–324, 2020.

[20] Zhehao Jiang, Neiwen Ling, Xuan Huang, Shuyao Shi, Chenhao Wu, Xiaoguang Zhao, Zhenyu Yan, and Guoliang Xing. Coedge: A cooperative edge system for distributed real-time deep learning tasks. In *Proceedings of the 22nd International Conference on Information Processing in Sensor Networks*, pages 53–66, 2023.

[21] Michael Kaess, Hordur Johannsson, Richard Roberts, Viorela Ila, John J Leonard, and Frank Dellaert. isam2:

Incremental smoothing and mapping using the bayes tree. *The International Journal of Robotics Research*, 31(2):216–235, 2012.

[22] Jan Kallwies, Bianca Forkel, and Hans-Joachim Wuensche. Determining and improving the localization accuracy of apriltag detection. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 8288–8294. IEEE, 2020.

[23] Iman Abaspur Kazerouni, Luke Fitzgerald, Gerard Dooly, and Daniel Toal. A survey of state-of-the-art on visual slam. *Expert Systems with Applications*, 205:117734, 2022.

[24] Giseop Kim and Ayoung Kim. Remove, then revert: Static point cloud map construction using multiresolution range images. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 10758–10765. IEEE, 2020.

[25] Dániel Kiss-Illés, Cristina Barrado, and Esther Salamí. Gps-slam: an augmentation of the orb-slam algorithm. *Sensors*, 19(22):4973, 2019.

[26] Kenji Koide, Masashi Yokozuka, Shuji Oishi, and Atsuhiko Banno. Voxelized gicp for fast and accurate 3d point cloud registration. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11054–11059. IEEE, 2021.

[27] Juichung Kuo, Manasi Muglikar, Zichao Zhang, and Davide Scaramuzza. Redesigning slam for arbitrary multi-camera systems. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2116–2122. IEEE, 2020.

[28] Mathieu Labbé and François Michaud. Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation. *Journal of Field Robotics*, 36(2):416–446, 2019.

[29] Pierre-Yves Lajoie, Benjamin Ramtoula, Yun Chang, Luca Carlone, and Giovanni Beltrame. Door-slam: Distributed, online, and outlier resilient slam for robotic teams. *IEEE Robotics and Automation Letters*, 5(2):1656–1663, 2020.

[30] Jiaxin Li and Gim Hee Lee. Deepi2p: Image-to-point cloud registration via deep classification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15960–15969, 2021.

[31] Zheqi Li and Jidong Huang. Study on the use of qr codes as landmarks for indoor positioning: Preliminary results. In *2018 IEEE/ION position, location and navigation symposium (PLANS)*, pages 1270–1276. IEEE, 2018.

[32] Jiarong Lin, Chunran Zheng, Wei Xu, and Fu Zhang. R^2 live: A robust, real-time, lidar-inertial-visual tightly-coupled state estimator and mapping. *IEEE Robotics and Automation Letters*, 6(4):7469–7476, 2021.

[33] Zongwei Liu, Hao Jiang, Hong Tan, and Fuquan Zhao. An overview of the latest progress and core challenge of autonomous vehicle technologies. In *MATEC Web of Conferences*, volume 308, page 06002. EDP Sciences, 2020.

[34] Zhenchao Ouyang, Jiahe Cui, Xiaoyun Dong, Yanqi Li, and Jianwei Niu. Saccadefork: A lightweight multi-sensor fusion-based target detector. *Information Fusion*, 77:172–183, 2022.

[35] T Perković, P Šolić, H Zargariasl, D Čoko, and Joel JPC Rodrigues. Smart parking sensors: State of the art and performance evaluation. *Journal of Cleaner Production*, 262:121181, 2020.

[36] David Prokhorov, Dmitry Zhukov, Olga Barinova, Konushin Anton, and Anna Vorontsova. Measuring robustness of visual slam. In *2019 16th International Conference on Machine Vision Applications (MVA)*, pages 1–6. IEEE, 2019.

[37] Tong Qin, Shaozu Cao, Jie Pan, and Shaojie Shen. A general optimization-based framework for global pose estimation with multiple sensors. *arXiv preprint arXiv:1901.03642*, 2019.

[38] Tong Qin, Peiliang Li, and Shaojie Shen. Vins-mono: A robust and versatile monocular visual-inertial state estimator. *IEEE Transactions on Robotics*, 34(4):1004–1020, 2018.

[39] Siyu Ren, Yiming Zeng, Junhui Hou, and Xiaodong Chen. Corri2p: Deep image-to-point cloud registration via dense correspondence. *IEEE Transactions on Circuits and Systems for Video Technology*, 2022.

[40] Jose Jesus De Rubio. Stability analysis of the modified levenberg-marquardt algorithm for the artificial neural network training. *IEEE Transactions on Neural Networks and Learning Systems*, PP(99):1–15, 2021.

[41] Tixiao Shan, Brendan Englot, Drew Meyers, Wei Wang, Carlo Ratti, and Daniela Rus. Lio-sam: Tightly-coupled lidar inertial odometry via smoothing and mapping. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5135–5142. IEEE, 2020.

[42] Tixiao Shan, Brendan Englot, Carlo Ratti, and Daniela Rus. Lvi-sam: Tightly-coupled lidar-visual-inertial odometry via smoothing and mapping. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5692–5698. IEEE, 2021.

[43] Shuyao Shi, Jiahe Cui, Zhehao Jiang, Zhenyu Yan, Guoliang Xing, Jianwei Niu, and Zhenchao Ouyang. Vips: Real-time perception fusion for infrastructure-assisted autonomous driving. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, pages 133–146, 2022.

[44] Ashutosh Singandhupe and Hung Manh La. A review of slam techniques and security in autonomous driving. In *2019 third IEEE international conference on robotic computing (IRC)*, pages 602–607. IEEE, 2019.

[45] Lukas von Stumberg and Daniel Cremers. Dm-vio: Delayed marginalization visual-inertial odometry. *IEEE Robotics and Automation Letters*, 7(2):1408–1415, 2022.

[46] Konstantinos A Tsintotas, Loukas Bampis, and Antonios Gasteratos. The revisiting problem in simultaneous localization and mapping: A survey on visual loop closure detection. *IEEE Transactions on Intelligent Transportation Systems*, 23(11):19929–19953, 2022.

[47] Jessica Van Brummelen, Marie O'Brien, Dominique Gruyer, and Homayoun Najjaran. Autonomous vehicle perception: The technology of today and tomorrow. *Transportation research part C: emerging technologies*, 89:384–406, 2018.

[48] Srihaarika Vijjappu. Distributed decentralised visual slam for multi-agent systems, 2020.

[49] Jingao Xu, Hao Cao, Zheng Yang, Longfei Shangguan, Jialin Zhang, Xiaowu He, and Yunhao Liu. SwarmMap: Scaling up real-time collaborative visual SLAM at the edge. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 977–993, Renton, WA, April 2022. USENIX Association.

[50] Georges Younes, Daniel Asmar, Elie Shammas, and John Zelek. Keyframe-based monocular slam: design, survey, and future directions. *Robotics and Autonomous Systems*, 98:67–88, 2017.

[51] Xiyu Zhang, Jiaqi Yang, Shikun Zhang, and Yanning Zhang. 3d registration with maximal cliques. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 17745–17754, 2023.

[52] Shibo Zhao, Hengrui Zhang, Peng Wang, Lucas Nogueira, and Sebastian Scherer. Super odometry: Imu-centric lidar-visual-inertial estimator for challenging environments. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 8729–8736. IEEE, 2021.

[53] Zihan Zhu, Songyou Peng, Viktor Larsson, Weiwei Xu, Hujun Bao, Zhaopeng Cui, Martin R Oswald, and Marc Pollefeys. Nice-slam: Neural implicit scalable encoding for slam. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12786–12796, 2022.

Catch Me If You Can: Laser Tethering with Highly Mobile Targets

Charles J. Carver[†1], Hadleigh Schwartz[†1], Qijia Shao[1], Nicholas Shade[2], Joseph Lazzaro[2], Xiaoxin Wang[2], Jifeng Liu[2], Eric Fossum[2], and Xia Zhou[1]

[†]*Co-primary authors*

[1]*Department of Computer Science, Columbia University,* [2]*Thayer School of Engineering, Dartmouth College*

{cjc, hadleigh, qijia, xia}@cs.columbia.edu,

{nicholas.r.shade.th, joseph.p.lazzaro.th, xiaoxin.wang, jifeng.liu, eric.r.fossum}@dartmouth.edu

Abstract

Conventional wisdom holds that laser-based systems cannot handle high mobility due to the strong directionality of laser light. We challenge this belief by presenting Lasertag, a *generic framework* that tightly integrates laser steering with optical tracking to maintain laser connectivity with high-velocity targets. Lasertag creates a constantly connected, laser-based tether between the Lasertag core unit and a remote target, irrespective of the target's movement. Key elements of Lasertag include (1) a novel optical design that superimposes the optical paths of a steerable laser beam and image sensor, (2) a lightweight optical tracking mechanism for passive retroreflective markers, (3) an automated mapping method to translate scene points to laser steering commands, and (4) a predictive steering algorithm that overcomes limited image sensor frame rates and laser steering delays to quadruple the steering rate up to 151 Hz. Experiments with the Lasertag prototype demonstrate that, on average, Lasertag transmits a median 0.97 of laser energy with a median alignment offset of only 1.03 cm for mobile targets accelerating up to 49 m/s^2, with speeds up to 6.5 m/s, and distances up to 6 m ($\approx$47°/s). Additional experiments translate the above performance to a 10^{-8} median bit error rate across trials when transmitting a 1 Gbps, on-off keying signal. Lasertag paves the way for various laser applications (e.g., communication, sensing, power delivery) in mobile settings. A demonstration video of Lasertag is available at:

mobilex.cs.columbia.edu/lasertag

1 Introduction

The physical properties of laser light make it an excellent medium for numerous applications. Examples include high-bandwidth communication due to its fast modulation speeds [25, 39, 53], efficient power delivery thanks to its high energy density [27, 54, 55, 58], and fine-grained sensing given its nanometer-level spectral widths [19, 43, 112, 149].

Despite its potential, the inherent directionality of laser light has precluded its use in highly-mobile settings. Although existing free-space optics (FSO) systems can track and steer to fast-moving objects with *predetermined* trajectories (e.g., satellites) at *kilometer-level* distances, these systems are unsuitable for emerging mobile applications – such as virtual reality (VR) streaming and mobile power delivery – that exhibit *arbitrary* trajectories at *meter-level* distances. Fundamentally, this discrepancy is due to the higher angular tracking rate required for near-range targets. For example, a target 1m away traveling at 5m/s requires an angular tracking rate of approximately 79°/s, in contrast with the 1.5°/s [141] rate required to maintain laser connectivity with a low earth orbit satellite moving at 7.8 km/s [96]. Compounding FSO's insufficient tracking rate for emerging applications, these systems are often bulky and unsuitable for mobile environments.

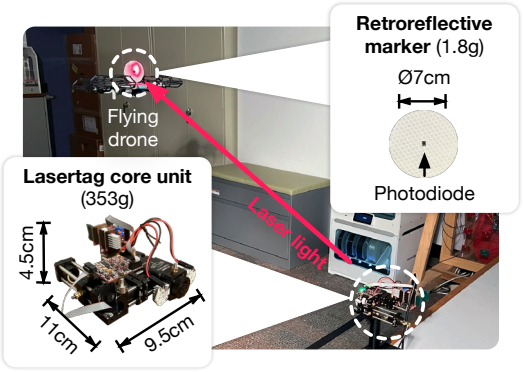


Figure 1: *Lasertag maintains constant laser alignment with a flying drone equipped with a passive retroreflective marker.*

In this work, we bridge the gap between traditional FSO systems and emerging mobile applications by enabling fast tracking and laser steering using portable, off-the-shelf hardware. To accomplish this task, we first consider various optical designs and their corresponding characteristics. While diffusing the beam can mitigate the alignment issue, it inevitably sacrifices supporting range and requires high-power laser diodes, resulting in low energy efficiency. An alternative is to scan the narrow laser beam in search of the target [16, 69, 114], which entails delays of hundreds of milliseconds [18]. Such delays are problematic for tracking fast motions such as consumer drone and human head movements (both multiple m/s [28, 118]). Additionally, once the target is acquired, additional scans may be required to maintain tracking [16, 59, 83, 86, 116] resulting in breaks in the laser link, which is unsuitable for applications requiring constant connections (e.g., data transmission).

A more efficient approach is to *separately* track the target's movement and then steer the laser beam directly to the target's location. While object tracking and laser steering are well-explored endeavours on their own, their integration is nontrivial on multiple fronts. First, the narrow-beam nature of laser light leads to a low tolerance for positioning errors, rendering existing localization technologies insufficient for accurate laser steering. Second, even if the target is perfectly positioned, it is challenging to translate the object's 3D location to the 2D reference frame used by the laser-steering device: any offset between the positioning and laser-steering device causes depth

ambiguity (i.e., different object depths necessitate different steering angles). This offset often cannot be measured due to the unknown origin points within the positioning and steering devices. Finally, mapping the steering device's input to an outgoing beam angle requires exhaustive measurements which are susceptible to human errors. This is further complicated by additional optics (e.g., wide-angle lenses) used in portable laser-steering systems [16–18, 75, 84], which affects both the tracking unit's perception of the scene and outgoing laser angle.

We present Lasertag to address the above practical issues. Lasertag is a *reconfigurable, application-agnostic platform* that tightly integrates laser steering and optical tracking in support of numerous high mobility applications. As shown in Figure 1, Lasertag provides a laser-based tether between a laser diode and an arbitrary object, irrespective of the object's movement. Leveraging a novel optical design that superimposes the optical paths of a steerable laser beam and single image sensor, we efficiently fuse the power of computer vision with the communication and sensing benefits of laser light. The optical path sharing addresses the issue of depth ambiguity and optical-element complications. Furthermore, we exploit the physics of fluorescence to enable an automated, precise mapping between any point in the scene and the laser steering's input drive signal. Finally, to overcome the limited frame rate of low-cost image sensors and non-negligible delays of laser steering devices, we propose a predictive steering algorithm that forecasts the target's future location, interpolates its intermediate locations, and proactively steers to the interpolated points until a new sensor reading is ready.

We fabricate a Lasertag prototype using off-the-shelf hardware, and evaluate its performance in the context of three applications where unidirectional,[1] high-mobility laser tethering is a necessity: ground-to-drone communication, laser-based streaming of VR content, and mobile power delivery. Since application-specific performance fundamentally relies on the ability to maintain a laser connection with a moving target, we focus on evaluating Lasertag's *tethering efficacy* by measuring the offset between the laser beam's center and the marker, as well as the received laser power at the target. We then provide additional experiments and analyses to translate the measured tethering performance to application-specific performance. Finally, we examine Lasertag's individual components, its robustness, and power consumption. In summary:

- Lasertag, on average, tethers a laser beam to a mobile target's light sensor with a median offset of only 1.03 cm, delivering a median normalized laser power of 0.97 for targets accelerating up to 49 m/s^2, traveling with speeds up to 6.5 m/s, and at distances up to 6 m ($\approx$47°/s).
- The predictive steering algorithm boosts the tracking and steering rate from 36 Hz to 151 Hz, delivering 15% higher laser power and reducing alignment offset by 12% on average across all velocity and trajectory patterns.
- Translating tethering efficacy to communication performance, Lasertag maintains a median 10^{-8} bit error rate (BER) for a 1 Gbps on-off keying (OOK) signal using predictive steering, a 54 times smaller BER than baseline steering.
- Lasertag's optical tracking reliably detects a passive retroreflective marker up to 6 m in the presence of confounding background objects (e.g., LEDs, reflective objects) and strong ambient light (up to 200 kLx).
- Lasertag supports a $\pm$95° steering range with $\leq$3 dB loss.

[1] Bidirectional applications can be supported with either a second Lasertag unit on the mobile node or alternative communication techniques.

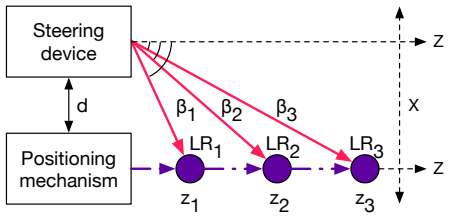


Figure 2: *Challenge of depth ambiguity, where changes in an object's depth (z_1,z_2,z_3) require different steering angles (β_1,β_2,β_3).*

2 System Challenges

The integration of laser steering and object tracking faces the following challenges.

Small Beam Divergence. Laser-based systems typically utilize narrowly collimated beams with small divergences (0.0005° – 0.005°) [66] for enhanced range and energy efficiency. This results in low tolerance of localization/tracking errors. Existing localization methodologies using radio frequencies (e.g., GPS [8], Wi-Fi [68, 120], and Bluetooth [3, 148]) have localization errors ranging from tens of centimeters [3, 68, 120, 148] to multiple meters (outdoor GPS [135]). Similarly, for systems relying on inertial measurement units, positioning errors can accumulate to meters in only a short duration [36, 72]. These localization errors can easily cause a laser beam to miss a target at meter-level distances, even with perfect steering control. Diffused laser beams may mitigate this issue, but at the cost of significantly reduced energy efficiency.

Depth Ambiguity. Even with perfect localization, it is challenging to translate the localization's 3D reference frame to the 2D reference frame of the outgoing laser beam. All common laser-steering devices, from bulky mechanical gimbals to micro-electromechanical (MEMS) mirrors, convert an input signal to a mechanical deflection around a mirror's center point. This 2D mechanical deflection steers the laser beam in 3D space so that it reaches the first object lying in its path.

As shown in Figure 2, the offset d between the positioning mechanism and laser-steering device can cause depth ambiguity when determining the 2D angle for steering the laser. This offset leads to different views for the two units. Hence, even locations that differ only in their depths (i.e., z_0,z_1,z_2) require different steering angles (β_0,β_1,β_2) for the laser beam to reach them. Therefore, to calculate the correct outgoing 2D angle that will reach a target object in 3D space, an accurate measurement of the offset d is crucial. Obtaining this measurement

is often impractical, as the positioning and steering devices typically have unknown origin points somewhere within their packaging, and even mm-level measurement inaccuracies would cause the steered laser beam to miss its target.

Mapping Overhead. Compounding the above challenges, the laser steering mechanism must be precisely characterized to map input drive signals to outgoing steering angles, which becomes even more difficult when using additional lenses.

Traditional free-space optics systems utilize bulky, slow, and expensive gimbals to steer laser light to the target [115]. A major benefit of these systems is their simple mapping between input signal and output angle [132], often characterized by the manufacturer during fabrication. High mobility applications require smaller and faster steering devices, typically in the form of MEMS mirrors. Unfortunately, MEMS mirrors exhibit model-dependent nonlinear relationships between input signal and output signal [89]. Mapping MEMS mirror drive signals to deflection angles thus requires an independent characterization, which is prone to human error, time-consuming, and reduces the accuracy of the system.

Worsening the above issue, fast-steering MEMS mirrors have an extremely limited steering range (e.g., $\pm 7°$ mechanical tilt) and therefore require additional optics to expand the steering range to a desirable field-of-view (FOV) [16, 18]. These secondary optics not only increase the mapping complexity and introduce more opportunities for error, but also change the outgoing ray geometry so they no longer exit the optical system from a single point. Instead, these rays exit the optical system at different spatial positions along the exit lens depending on steering angle, such that the offset d (Figure 2) between the laser steering system and object positioning system becomes angle-dependent. This significantly increases the colocation complexity and requires an additional mapping.

3 Lasertag Design

To address the above challenges, we present Lasertag, a generic framework that fuses laser steering with optical tracking. We choose optical tracking because it provides the most flexibility and ease of colocation compared to alternatives. We now elaborate on the four key design elements of Lasertag.

3.1 Efficient Optical Path Sharing

The first design element tackles the issue of translating the localization reference frame to the laser steering reference frame. We propose a novel optical design that intrinsically fuses the two together, efficiently sharing the optical path between the outgoing laser light and a single image sensor. This optical arrangement eliminates manual and time-consuming measurements to enable 2D laser steering in 3D space.

The proposed optical design requires only a single image sensor for tracking. To understand the rationale, we envision a hypothetical optical circuit consisting of an image sensor and single focusing lens. The lens focuses light from the scene onto the image sensor, creating a 2D projection of the 3D world. For each patch of focused light, a chief ray (CR) exists, originating

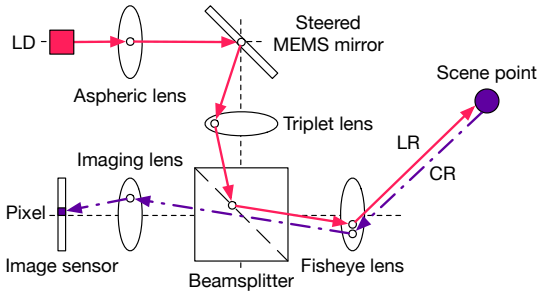


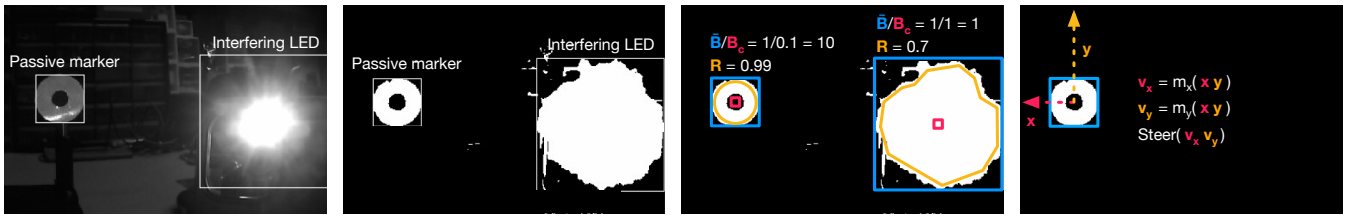
Figure 3: *The optical design of Lasertag, enabling outgoing laser light (red) to be mapped 1:1 with incoming scene light (purple).*

from a 3D scene point and passing through the lens to project onto a 2D pixel patch. With this optical circuit in mind, we can then consider adding an imaginary laser steering mechanism that is perfectly colocated with the image sensor. This laser steering device is capable of steering a narrow laser beam (LR) out through the same focusing lens, retracing the steps of the incoming CR to reach the same object as imaged by the sensor. In this way, a 2D pixel is mapped to a 2D angle relative to the image sensor, thereby eliminating depth ambiguity.

To realize this optical design, we exploit two intrinsic properties of laser light: strong linear polarization and narrow emission wavelength. Specifically, we overlay the laser's outgoing optical path with an image sensor's incoming optical path ($CR+LR$) using a beamsplitter, which reflects laser light out of the system and transmits incoming scene light onto the image sensor. Our methodology – inspired by the field of microscopy [97, 109] – maximizes the steered laser energy without sacrificing imaging quality, thus enabling efficient laser tethering. Figure 3 illustrates Lasertag's optical design, where additional lenses (e.g., triplet lens) create the proper ray orientation for the optical circuit. The final fisheye lens enables full-hemisphere steering and imaging of the scene.

To support our goal, we consider beamsplitters that redirect light using either fixed ratios, wavelength (i.e., dichroic beamsplitters), or polarization. Fixed-ratio beamsplitters allow only a fixed ratio of transmission to reflection (e.g., 10:90, 50:50), sacrificing transmitted laser power for imaging brightness (or vice versa). Given the narrow spectral width of laser light, dichroic beamsplitters, which reflect light above/below a certain wavelength and transmit the remaining, seem an obvious choice. Unfortunately, dichroic beamsplitters cause the imaged scene to be extremely skewed in color (since only light above or below the beamsplitter threshold would reach the image sensor), and are often manufactured on thin plates resulting in poor image quality due to ghosting [127].

To solve these challenges, we design our system around a laser-line polarizing beamsplitter. These beamsplitters reflect linearly polarized light lying within a narrow wavelength range, and transmit light that is either (a) polarized in the other direction, or (b) at a wavelength above the polarizing range. This design leverages the fact that laser light is intrinsically highly polarized and narrow in wavelength [108]. As a result, our design can operate in two modes: one in which true



(a) Captured scene image. *(b) Otsu binarization [9].* *(c) Computing blob statistics.* *(d) Steering to optimal blob.*

Figure 4: *Workflow for isolating the passive marker from cluttered backgrounds and steering the laser to the marker's center.*

scene color is desired, and one in which maximum scene brightness is required. In the first case, since ambient light is typically unpolarized, the image sensor sees all wavelengths with approximately 50% attenuation in overall brightness. In the second case, artificial light is injected into the scene at a wavelength above the beamsplitter's polarizing wavelength range, at which it transmits all light, regardless of polarization orientation. This enables the bulk of sensing light to be transmitted back to the image sensor. Our prototype (§4) implements the second mode. Notably, this requires the use of *separate laser and sensing wavelengths*: a laser wavelength *inside* the beamsplitter's specified polarizing wavelength range, and a sensing wavelength *outside* it. Lasertag can be deployed with any pair of wavelengths that satisfy this basic constraint.[2] Since commercial beamsplitters are available in a variety of polarizing ranges, the appropriate choice of laser and sensing wavelengths primarily depends on application requirements and channel characteristics. For instance, Lasertag should be equipped with blue-green light sources for underwater scenarios, given the lower attenuation of these wavelengths in water. In contrast, for aerial power delivery applications, it would be favorable to use an IR laser to match the responsitivity of typical solar cells. Additionally, our design imposes no restrictions on the light sources' optical powers, so that safety, power consumption, and signal-to-noise ratio (SNR) requirements can also be considered. Finally, since the outgoing light is completely separable from the incoming light, the input laser parameters have no impact on Lasertag's sensing performance.

While several elements of the design are explored in existing works (e.g., MEMS mirrors for laser beam steering [16, 75], beamsplitting optics for colocation of laser steering and detection components [6, 23, 32], fisheye lenses for expanding laser steering range [16, 18, 22, 151]), the integration of these elements for tracking and steering is nontrivial. The optical characteristics must be carefully designed to create the appropriate ray geometry for imaging and steering.

3.2 Fast Tracking with Retroreflective Imaging

The second design element identifies the target object from each image frame. Object detection is well studied in computer vision (CV). The proposed marker design and algorithm are built upon prior CV techniques and inspired by existing works leveraging retroreflectors for tracking [83, 137, 147]. The algorithm incorporates simple heuristics that differentiate the passive marker from interfering scene objects and run real-time tracking on resource-constrained hardware.

Retroreflective Marker. The target object is equipped with a passive marker made of retroreflective tape, cut into a circle, with a small hole in the center to host a light sensor. After adding a wide-angle LED to Lasertag, the scene is illuminated with a specific wavelength that retroreflects off the marker and back to the image sensor. In general, these retroreflections will be significantly brighter than other objects in the scene, so the target will appear as a blob that can be segmented from background objects. The laser light is then steered to the center of the blob, so that it can pass through the hole to the sensor (e.g., photodiode for receiving data, solar-cell for harvesting energy). Lasertag's optical design ensures that the laser beam is sufficiently narrow so the majority of light passes through.

Although retroreflections are generally brighter than arbitrary scene objects, active luminaries emitting at the same wavelength have the potential to be brighter. To overcome this challenge, we leverage the fact that since light is not retroreflected by the center of the marker (due to the cutout), the marker's center pixels will be dimmer than the remainder of the marker. Confounding luminaries in the scene, however, will have a roughly uniform illumination within the blob (e.g., LEDs will emit light from the center of their package). Consequently, we identify the marker by taking the ratio of the blob's overall brightness to the blob's center brightness, favoring blobs with higher ratios. We add an additional weight for the roundness of the blob, since the circular marker will remain elliptical even under rotation. The entire image processing algorithm is described in the following section.

Marker Detection and Tracking. To achieve high marker tracking rates, we develop a lightweight marker detection algorithm based on traditional CV blob detection [26, 45] and implement *trajectory-based region of interest (ROI) windowing* to increase both frame capture and detection speeds. The key insight of our detection algorithm is that the passive marker registers in the image as a distinctly bright region with a known shape. Based on these characteristics, we isolate the marker in each frame and determine the pixel corresponding to its center. This process is visually described in Figure 4. We note that regardless of the marker's angular orientation, its center remains constant in the 2D frames. In addition to localizing the marker, the output of our detection algorithm informs our trajectory-based ROI windowing algorithm (Algorithm 1) to increase Lasertag's tracking rate. ROI windowing is a camera

[2] Advanced Lasertag designs can consider multispectral beam splitters to support multiple input laser wavelengths.

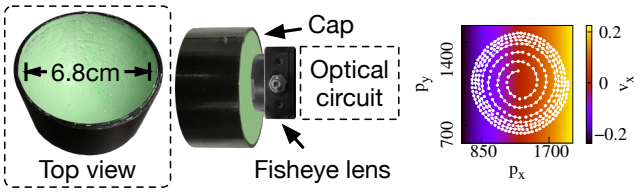


Figure 5: *(Left) Fluorescent cap design enabling automatic pixel-to-voltage mapping. (Right) After steering the laser to each white dot, we fit a polynomial surface to obtain our mapping functions.*

feature that enables reading a smaller window of pixel values, effectively zooming in to a region of interest. This reduces the number of pixel values the camera must process and thus decreases image capture time [92], which can facilitate real-time tracking of objects moving at high speeds [111]. We set an ROI window that hones in on the marker and follows its trajectory.

As shown in Figure 4, after capturing each frame, we binarize the image using a threshold automatically determined with Otsu's method [9]. The retroreflective marker and any confounding luminaries then appear in the image as clusters of connected white pixels (i.e., blobs). We run blob detection on the image to determine the center pixel, bounding box, and roundness for each region. To isolate our passive marker, we assign each blob a score based on our marker heuristics (favoring round regions with high ratio of overall brightness to center brightness). We then save the blob with the highest score and return its center pixel coordinate. Finally, we update the image sensor registers to move the ROI window based on the updated marker location. If no marker is detected, the ROI is expanded to encompass the whole image sensor so that the target can be re-acquired. Our algorithm's simplicity, afforded by its tight coupling with the physical design of our passive marker, enables localization and tracking within milliseconds.

Notably, Lasertag's framework is inherently flexible and works with any conventional CV detection technique. While we demonstrate our system with a passive circular retroreflective marker, Lasertag is compatible with markers of any shape and size (e.g., a 3D sphere). Furthermore, Lasertag can support markerless tracking and active feedback as detailed in §6.

3.3 Fluorescence-based Optical Mapping

The third design element maps the marker's position on the image sensor to the steering command required to steer the laser to the object. Lasertag performs beam steering with a MEMS mirror – a compact, electromagnetically driven device in which deflection angles are determined by input voltages. We enable automatic pixel-to-voltage mapping via a short, one-time calibration process that is independent of the environment. The calibration process leverages fluorescence – a physical phenomenon in which a substance is stimulated by the light it absorbs, causing it to emit light of a different color – to shift the outgoing laser's wavelength to one always visible to the image sensor. With our optical design, this wavelength shift allows us to leverage the embedded image sensor to *automatically* map all pixel locations to steering voltages,

Input: μ: Sensor mode, α or τ.
Output: (x,y): Absolute center pixel of the marker.

Function Track(μ):
  while *True* **do**
    $I \leftarrow$ Capture new image
    $(x',y') \leftarrow$ **Detect**(I)
    if $(x',y') = None$ **then**
      $\mu \leftarrow \alpha$
      Set sensor resolution to $W_\alpha \times H_\alpha$
      Set sensor readout offset to $(0,0)$
      Set sensor readout area to (W,H)
      Skip to next iteration
    if $\mu = \alpha$ **then**
      $\mu \leftarrow \tau$
      $t_x \leftarrow (x' - W_\alpha/2) * W/W_\alpha$
      $t_y \leftarrow (y' - H_\alpha/2) * H/H_\alpha$
      Set sensor resolution to $W_\tau \times H_\tau$
      Set sensor readout offset to (t_x,t_y)
      Set sensor readout area to $(W/z,H/z)$
    else
      $o_x \leftarrow (x' - W_\tau/2) * (W/z)/W_\tau$
      $o_y \leftarrow (y' - H_\tau/2) * (H/z)/H_\tau$
      $(t_x,t_y) \leftarrow$ Updated sensor readout offset
      $(x,y) \leftarrow (o_x + t_x, o_y + t_y)$
      Set sensor readout offset to $(t_x + o_x, t_y + o_y)$
      return (x,y)

Algorithm 1: Marker tracking algorithm for an image sensor with a $W \times H$ active area. After marker acquisition ($\mu = \alpha$), the $W_\alpha \times H_\alpha$ resolution is scaled to $W_\tau \times H_\tau$ for tracking ($\mu = \tau$) and the active pixel readout window is reduced to $W/z \times H/z$.

bypassing the requirement of accurate angular measurements.

Specifically, we borrow from the field of fluorescence microscopy [97, 109] and use fluorescent powder to shift the laser wavelength *after* it has been steered to one visible to the image sensor. We design a small plastic cap that is placed over the optical circuit's exit lens during the calibration process (Figure 5). It is coated with either an up-converting [40] or down-converting phosphor [82][3]. The cap is a hollow hemisphere whose interior is a minimum distance of 2 cm from the fisheye lens at all points. This design enforces the minimum required distance between the fluorescent surface and fisheye lens to mitigate comatic aberrations (which result in comet-like tails around the fluorescing spot, leading to mapping errors) [56]. We empirically determine this distance in our system to be 2 cm. To do so, we fix the steering unit's outgoing laser angle and place a fluorescent surface in the beam's path in the far field. We then shift the surface towards the unit along the beam until the center coordinate of the fluorescent spot's blob changes, indicating the minimum distance.

Finally, to generalize the mapping to any pixel on the image, we fit two 3D surfaces with a subset of laser steering voltages and pixel positions. With the fluorescent cap installed, we steer the laser to the center of the fisheye lens, then scan

[3] For up-converting phosphors, higher wavelength photons are absorbed causing lower wavelength photons to be emitted. Down-converting phosphors absorb lower wavelength photons and emit higher wavelengths.

the beam in an Archimedian spiral [81] to cover N spots on the cap (Figure 5). Steering via the MEMS mirror requires applying two voltages, v_x, and v_y, corresponding to the two orthogonal scanning axes. Given the fisheye lens's large distortion on its edges, we decrease the step size between adjacent steering voltages at larger spiral radii, enabling an accurate mapping despite the spatial compression. Since the sensed pixel position is influenced by both v_x and v_y, we feed (x,y,v_x) and (x,y,v_y) through a multivariate, polynomial regression algorithm to obtain our mapping functions:

$$m_x(x,y)=\sum_{i=0}^{3}\sum_{j=0}^{3}a_{ij}x^iy^j, \quad m_y(x,y)=\sum_{i=0}^{3}\sum_{j=0}^{3}b_{ij}x^iy^j,$$

where the coefficients a_{ij} and b_{ij} are determined by the respective regressions. We choose a third-degree polynomial to account for the nonlinear relationship between steering voltage and outgoing beam angle. Overall the calibration takes roughly ten minutes, creating the functions later used by Lasertag to map an identified marker to a steering command in real time. Notably, this streamlined calibration can be automatically run across environments and can be repeated at any point throughout deployment to maintain steering accuracy. This makes it highly flexible compared to alternative systems requiring hard-coded system parameters [4, 126], gimbal positions [20], uplink feedback [1, 64], or complex learning strategies [39].

3.4 Predictive Steering for High Mobility

The fourth design element overcomes the low frame rate of conventional image sensors and delays associated with laser steering devices. Conventional image sensors have a low frame rate (30-60 FPS), which results in a considerable delay between consecutive frames. Additionally, laser steering takes a non-negligible amount of time (on the order of milliseconds). These delays can cause a steering gap between the current and subsequent target locations, particularly when tracking fast-moving objects (e.g., flying drones which accelerate to over $20\,\mathrm{m/s^2}$ when making sharp turns). As a consequence, laser alignment continuity can easily be disrupted.

To address this problem, we exploit movement continuity on a micro timescale and propose *predictive steering*, wherein we (1) predict the target's expected location in the next frame, (2) interpolate locations from now to the next frame, and (3) steer to interpolated locations until a new frame is ready. This upsampling strategy achieves smoother laser steering and handles fast movement more effectively. While upsampling techniques are common in the domain of audio and image processing [34, 71, 140], they are usually performed offline, requiring the signal to be captured in advance. In the scenario of real-time laser tracking and steering, we must predict the next steering location using only historical data.

The rationale of mobility prediction stems from the continuous nature of motion at micro timescales. Although motion is typically non-deterministic, it is still subject to certain constraints. For example, speed and acceleration are bounded and cannot be changed instantaneously. Therefore, by utilizing historical data, it is feasible to learn motion dynamics and predict the object's next location. Numerous techniques for motion prediction have been proposed, including double exponential filters [48, 74], Kalman filters [7, 78, 143], and neural networks [44, 113]. In our implementation (§4), we utilize a double exponential predictor due to its low computational overhead. Our framework is generic, however, and can work with any motion prediction model as outlined in §6.

Input: M: Prediction buffer size, $\bar{\varepsilon}$: Nominal steering delay.
Output: None.

Function Steer(M,$\bar{\varepsilon}$)**:**
  Let B,s,b be an empty array of length M
  while *True* **do**
    $T \leftarrow$ Time since last iteration
    if *first iteration* **then**
      $(x_0,y_0) \leftarrow$ **Track**(α)
    else
      $(x_0,y_0) \leftarrow$ **Track**(τ)
    Append (x_0,y_0) to B
    Skip to next iteration if length(B) $\neq M$
    $(s_x,s_y)^{(0)} \leftarrow B[0]$
    $(b_x,b_y)^{(0)} \leftarrow B[1]-B[0]$
    for $i \leftarrow 1$ *to* $M-1$ **do**
      $(s_x,s_y)^{(i)} \leftarrow$ Smooth $B[i]$ with Eq. (1)
      $(b_x,b_y)^{(i)} \leftarrow$ Estimate trend with Eq. (2)
    $(x_T,y_T) \leftarrow (s_x+b_x,s_y+b_y)^{(M-1)}$
    $\Delta \leftarrow$ Time since call to **Track**(...)
    $t \leftarrow \Delta+\bar{\varepsilon}$
    while $t \leq T-\bar{\varepsilon}$ **do**
      $\delta=(y_T-y_0)/(x_T-x_0)$
      $x_t=x_0+t*(x_T-x_0)/T$
      $y_t=\delta*(x_T-x_0)+y_0$
      $(v_x,v_y) \leftarrow (m_x(x_t,y_t),m_y(x_t,y_t))$
      Set mirror voltage to (v_x,v_y)
      $\varepsilon_i \leftarrow$ Time since previous nested iteration
      $t \leftarrow t+\varepsilon_i$
    Delete $B[0]$

Algorithm 2: Predictive steering algorithm to continuously steer the laser light to pixels (x_t,y_t) lying between the captured marker position (x_0,y_0) and predicted final location (x_T,y_T) expected at time T. The ith iteration of the double exponential prediction is denoted by a superscript (i).

The double exponential predictor is a time-series forecasting method that utilizes a double exponential filter to smooth the input data and estimate future trends [48]. At a high level, the predictor first applies a double exponential filter to the time series data and obtains a smoothed estimate of the underlying trend [98]. Then, it predicts future values by adding the estimated slope of the trend to the current smoothed value. We define the smoothing and trend estimation at iteration i as:

$$s^{(i)}=\alpha r^{(i)}+(1-\alpha)\left(s^{(i-1)}+b^{(i-1)}\right), \tag{1}$$

$$b^{(i)}=\beta\left(s^{(i)}-s^{(i-1)}\right)+(1-\beta)b^{(i-1)}, \tag{2}$$

where the raw input is represented by $r^{(i)}$, the smoothed value is $s^{(i)}$, and $b^{(i)}$ is the best estimate of the trend. The parameters

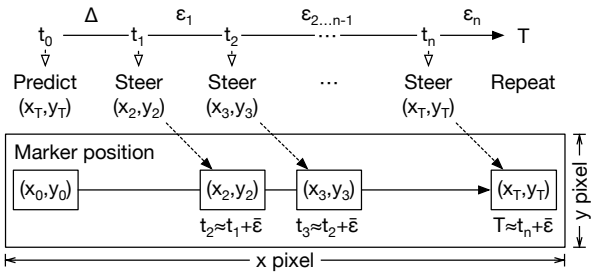


Figure 6: *Predictive steering methodology that maximizes the number of steers within a time period T, taking into account the estimated initial steering time $\bar{\varepsilon}$, image processing and prediction overhead Δ, and subsequent steering overheads $\varepsilon_{\{1\ldots n-1\}}$.*

$\alpha, \beta \in [0,1]$ are the smoothing parameters that control the weight given to current and past values. In general, α and β values can be chosen with empirical guidance[4], where larger values assign stronger weights to recent inputs and are suitable for faster motions. Given the smoothed value $s^{(i)}$ and estimated trend $b^{(i)}$, the *next* prediction is defined as $s^{(i+1)} = s^{(i)} + b^{(i)}$ occurring at iteration $i+1$.

Figure 6 and Algorithm 2 describe the detailed steps. Once an image frame is captured at time t_0, the system predicts the target location (x_T, y_T) at the next frame. The predicted location is produced at time t_1 after image processing delay Δ. The system then utilizes linear interpolation to infer locations between the captured pixel coordinate (x_0, y_0) and predicted coordinate (x_T, y_T). It consecutively steers laser beam to these locations, taking into account the steering delay. Specifically, a steering command sent at t_i steers the laser beam to the interpolated location expected at time $t_i + \varepsilon_i$. Since ε_i is not known before the command at t_i completes, we steer to the *expected* location based on the average steering delay $\bar{\varepsilon} \approx \varepsilon_i$. Since Lasertag's mirror moves only a small amount between subsequent steers, $\bar{\varepsilon}$ is a suitable approximation for ε (see §5.3).

4 Prototype Implementation

The Lasertag prototype consists of the core unit, containing the imaging and laser-steering subsystems, and a separate marker. The marker is a 7 cm diameter circle made of retroreflective tape [107]. We cut a 2 cm diameter hole in the center to place a photodiode (PD) [50] for performance evaluation.

Imaging Subsystem. The imaging subsystem images the scene to identify the marker's location. Two 520 nm LEDs [93] are placed next to the system's fisheye lens to flood the scene with green light. The green light reflects off the retroreflective marker and passes through the fisheye lens. The light then traverses the optical beamsplitter [130] and a 633 nm notch filter [105], which removes any stray reflections from the laser steering subsystem. Next, light passes through a 520 nm bandpass filter [125] with a 55 nm pass region bandwidth. The monochrome light is focused using an $f = 16$mm focal length lens [106], before finally arriving at a 5 MP image sensor [102]. The processing pipeline is built upon the OpenMV H7 Plus platform. We perform adaptive windowing by sending IOCTL commands directly to the image sensor.

[4] Other prediction algorithms (see §5.3) can eliminate parameter tuning.

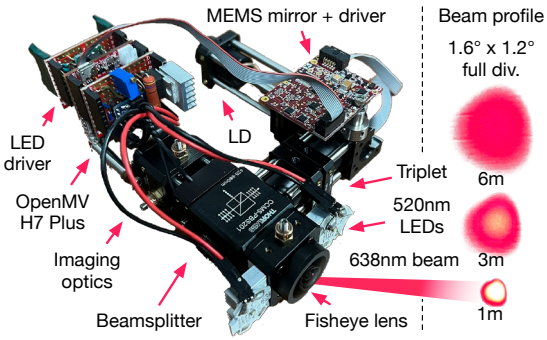


Figure 7: *The Lasertag prototype and resulting beam profile measured at different distances.*

Laser-Steering Subsystem. The laser steering subsystem takes the marker location from the imaging subsystem and feeds the appropriate steering voltages to a digital-input MEMS driver [91] operating with an SPI bus frequency of 25 MHz. We utilize a 638 nm laser diode [129] with the driving current capped to 120 mA, resulting in 50 mW of optical power. The outgoing laser light is focused using an $f = 2.75$mm aspheric lens [128]. After the aspheric lens, the laser light reflects off a 2.4 mm MEMS mirror [88] with a steering bandwidth of 300 Hz. Finally, the laser light passes through an $f = 20$mm triplet lens [133]. The laser light then reflects off the beamsplitter and through the fisheye lens [103], achieving a 3 dB half-angle (x, y) divergence of (0.6, 0.8) degrees.

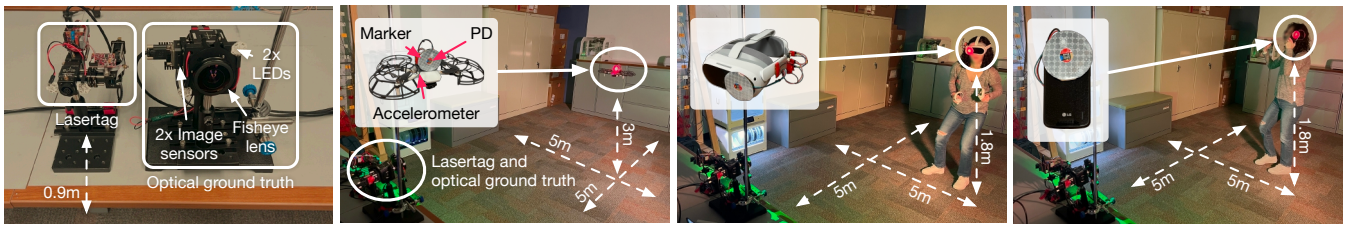
5 Evaluation

We evaluate Lasertag's tethering efficacy in the context of three practical applications, followed by micro-benchmarks and the impact of practical factors. A demonstration is available at [15].

5.1 Experimental Setup

Since application-specific performance (e.g., communication, power delivery) fundamentally relies on the ability to maintain a constant laser connection, we primarily focus on evaluating Lasertag's *tethering efficacy*. We then translate these results to application specific performance with additional experiments.

Ground Truth. We assemble a separate optical unit to collect ground truth marker and laser beam positions (Figure 8a). We combine two 2.4 MP cameras [138] with a 50:50 beamsplitter and place a fisheye lens [104] on the beamsplitter's output for imaging. Each camera has a 520 nm or 633 nm bandpass filter for isolating the marker or laser spot. We add two 520 nm LEDs to generate retroreflective light. Each camera streams images to a host computer, in parallel, at an average 315 FPS. Frames from both cameras are synced by minimizing the difference between frame timestamps, and aligned by applying a homographic transform. We perform binarization, blob detection, and ellipse contouring to localize the laser spot and marker in each frame. The marker velocity is then estimated by taking the Euclidean distance between the marker's center in adjacent frames and dividing by the frame interval. Notably, we mea-



(a) Lasertag and ground truth. *(b) Drone communication.* *(c) Laser VR.* *(d) Power delivery.*

Figure 8: *Experimental setup for evaluating Lasertag in the context of three different practical scenarios.*

sure *pixel velocity* instead of physical velocity since Lasertag's performance is only dependent on pixel changes. Since pixel velocities scale with distance (e.g., a fast-moving object close to Lasertag has a large pixel velocity, and a slower pixel velocity when it is farther away), we can generalize Lasertag's performance to any distance. We also add an accelerometer [99] to the target to complement the velocity data.

Evaluation Metrics. We measure tethering efficacy via: (1) the physical offset between the center of the laser beam and marker; (2) the normalized laser power received by a PD colocated with the marker.

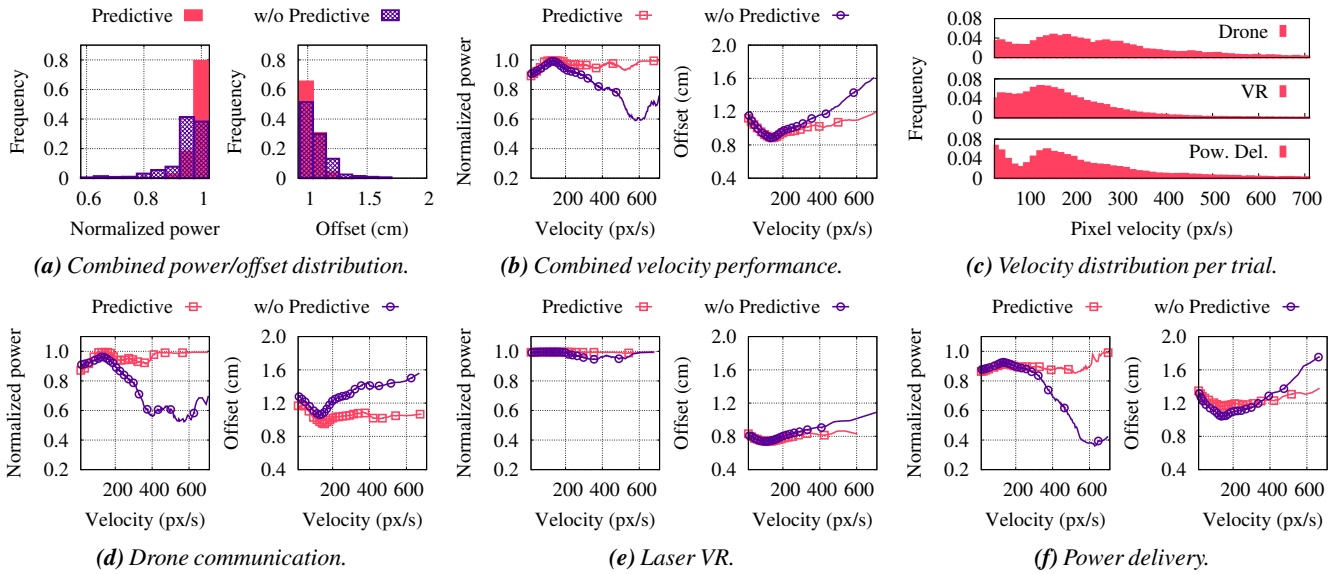
We compute the first metric from ground-truth videos. For each ground truth frame, we compute the pixel offset as the Euclidean distance between the laser and marker blobs in the image. To convert pixel offset to physical offset, we approximate the focal length f of the optical ground-truth camera using the MATLAB and a conventional checkerboard calibration pattern. Assuming minimal lens distortion and a fixed marker depth d relative to the image plane, the physical offset o (in meters) corresponding to a pixel offset p (in pixels) is $o = p * d * 1/f$ [79], where f is experimentally measured to be 322.5 pixels. Notably, this equation can also be used to relate pixel velocity to physical velocity, where o is in m/s and p is in px/s. Since transient marker occlusions and image processing noise impact our measurements, we extract the most *likely* offset by applying a rolling median window of width 100 px/s over the velocity-offset pairs. We also ignore velocities higher than the 98th percentile (710 px/s) and lower than the 2nd percentile (13 px/s) due to lack of samples.

The second metric is measured by a PD placed at the center of the marker on the target. Throughout target motion, accelerometer data and PD voltage are sampled at 1 ksps by a micro-controller [5]. PD voltage and velocity data are then synced together to associate each PD reading with a velocity. We then perform a series of signal processing techniques to remove the confounding factors of depth, angular response, laser power, and occlusion – all of which detrimentally affect the PD's response but do not indicate poor tethering performance. First, we observe that signal fluctuations due to target motion occur with a frequency between 3 Hz and 500 Hz. Thus we perform a 0.5 Hz rolling average of the signal's 99th percentile, removing signal spikes caused by PD/ADC noise. We then rescale the readings between the 99th percentile rolling average and the PD's noise floor (83 mV), partially removing the impact of the PD's angular response, experiment depth, and laser power. To account for temporary PD occlusion, we apply a rolling median window of width 100 px/s, and ignore velocities above 710 px/s and below 13 px/s.

Evaluation Procedure. We evaluate Lasertag in an enclosed indoor space. To understand the contribution of predictive steering, we also evaluate Lasertag without predictive steering as our baseline. First, we place the Lasertag core unit and ground truth measurement device on a 0.9 m height table (Figure 8a). Second, we place the marker and sensing components on the target (Figure 8). Third, we turn on a 1100 lx floor lamp (measured 1 m away) to provide ambient light and guidance for the drone's optical positioning. Finally, we place the target at a fixed depth [5] of either 0.8 m, 2.4 m, or 4 m from the Lasertag core unit, and record 3-8 minute trials for both steering methods at each depth. We synthesize over 1.6 million velocity measurements with associated PD readings. Laser safety requirements are maintained for all experiments, and since the laser's reflections are mainly retroreflective, authors were only exposed to diffuse, eye-safe reflections.

Translating Tethering Efficacy. We translate tethering results to communication performance by mapping the marker PD's voltage to the BER of a 1 Gbps stationary link between Lasertag and an avalanche photodiode (APD) [124]. First, we generate a random 1024-bit OOK, non-return-to-zero packet with an arbitrary waveform generator [62] connected to the Lasertag laser diode. Second, we colocate the APD and marker PD using a 50:50 beamsplitter, ensuring both sensors receive an identical copy of the modulated laser light. Third, we simultaneously capture the APD's received signals using an oscilloscope [63] and the corresponding voltage of the marker PD. For each packet, we decode the signal in MATLAB, compute the BER, and measure the marker PD's voltage. We then vary the link's SNR by programmatically decreasing laser power. To extrapolate to all possible voltages, we fit all recorded samples with a sigmoid function (R^2 of 0.992). We then apply this function to our experimental PD values to obtain a BER over time. This methodology avoids placing the wired APD on our experimental targets which would restrict their motion. Translating tethering results to power-delivery performance is straightforward, as harvested power is directly proportional to connection time. We avoid reporting charging efficacy as it highly depends on the specific laser and solar cell.

[5] The max distance is greater than this depth, since path length increases at extreme steering angles. This is accounted for with our adaptive normalization.



(a) Combined power/offset distribution. *(b) Combined velocity performance.* *(c) Velocity distribution per trial.*

(d) Drone communication. *(e) Laser VR.* *(f) Power delivery.*

Figure 9: *(a-b) Distribution and velocity trends of received normalized power and physical offset across all scenarios. (c) Velocity distribution from individual scenarios. (d-f) Velocity breakdowns for each scenario.*

5.2 Tethering Efficacy

To examine tethering efficacy, we consider object motions in three applications well-primed for the benefits of laser light: (1) ground-to-drone laser communication, (2) laser communication for VR, and (3) mobile power delivery using lasers. Across scenarios, our targets underwent realistic motions including unpredictable movements, jitter, velocity swings, and rotations.

Performance Overview. Across all applications, predictive steering consistently aligns the laser beam center with the PD on the target, delivering a median normalized laser power of 0.97 on average, compared to only 0.85 without predictive steering. Similarly, the median offset seen with predictive steering is 1.03 cm on average, compared to 1.17 cm for without predictive steering. To maintain the laser tether in the face of steering errors, the laser beam divergence could be expanded to increase the beam spot size and compensate for steering offsets, at the expense of energy efficiency. The required half-angle beam divergence to maintain the link in the face of a 1.17 cm offset is $\approx 0.67°$ for a marker depth of 1 m. This low beam divergence corresponds to a relatively well-collimated beam, and thus Lasertag can efficiently deliver the majority of outputted laser energy to the marker's receiver. Histograms of received normalized power and offsets across all scenarios are shown in Figure 9a, demonstrating predictive steering's performance gain. We attribute this gain to the boosted steering rate of our predictive steering algorithm, achieving average rates up to 151 Hz from the baseline 36 Hz. As shown in Figure 9b, while predictive steering maintains relatively constant performance across velocities, baseline steering performance considerably declines. At the maximum observed marker speed of 710 px/s, corresponding to a physical speed of 8.8 m/s at a depth of 4 m, predictive steering achieves, on average, a 32% higher median power and 26% lower median offset than baseline steering.

In the context of power delivery, these results indicate that Lasertag is able to deliver 97% of its laser power to mobile device, compared to 85% with baseline steering. Translating these results to communication performance, we observe a median BER of 1.15×10^{-8} ($p25 = 8.19\times10^{-9}$, $p75 = 7.97\times10^{-5}$) for predictive steering and 6.25×10^{-7} ($p25 = 9.26\times10^{-9}$, $p75 = 3.97\times10^{-4}$) for baseline steering. Notably, this corresponds to a 54-fold improvement in BER utilizing predictive steering. Including periods when the link was fully occluded, we observe *mean* BERs of 2.62×10^{-2} and 5.03×10^{-2} for predictive and baseline steering, respectively. To improve these results, the laser power can be increased to improve the overall SNR (thereby decreasing BER), or lower throughputs (e.g., 100s of Mbps) can be considered to relax the SNR requirements.

Diving into each application, we observe that Lasertag's performance is highly dependent on not only the target's velocity, but also its trajectory. We identify two broad classes of object motion: (1) continuous linear movement; (2) short periods of linear or angular movement, which register as bursts of small, seemingly random displacements (later referred to as micro-movements). Importantly, both continuous and micro-movements can occur at all velocities. For instance, although the drone predominately flies in a continuous, linear fashion at a variety of speeds, VR games generate a combination of slow and rapid angular head movements, generating micro-movements of varying velocities. The presence of continuous, linear motion vs. micro-movements affects the received power and offset achieved by both steering methods.

Lastly, we observe that the distribution of velocities across applications is heavily positively skewed (Figure 9c). Since there are fewer samples recorded at higher velocities, we observe a larger variance in received power and offset at higher velocities. Despite this, predictive steering maintains a tighter confidence interval around the median than baseline steering. Additionally, a slightly lower PD reading and higher

offset exists at the lowest pixel velocities which we attribute to occlusion not accounted for by our adaptive normalization. Below we break down our results by application and examine the impact of specific motions on Lasertag's tethering efficacy.

Drone Communication. We consider maintaining a communication link with a moving aerial drone for our first application scenario. We track, in real-time, a DJI Mini 2 drone flying with a maximum acceleration of 29 m/s^2 in the z-direction and 19 m/s^2 in the x-y plane. At each starting depth, the drone flew left-to-right in repeated sweeps across the room, each measuring roughly 5 m horizontally. With the exception of take-off and landing, the drone maintained a relatively constant vertical position. As shown in Figure 9c, a significant number of samples at low velocities were collected, corresponding to instances in which the drone was changing direction, such that the gradient of pixel movement was close to zero. The second peak and tail of the distribution correspond to samples collected while the drone cruised across the room.

On average across all measured velocities, predictive steering delivers 27% higher median-normalized laser power than baseline steering and maintains a 22% smaller median offset. As the marker velocity increases beyond 230 px/s, predictive steering maintains median normalized powers above 0.86 and median offsets less than 1.11 cm. In contrast, baseline steering performance significantly declines, such that at the maximum recorded velocity of 710 px/s it achieves a median normalized power of 0.71 and median 1.61 cm offset. Translating these results to communication performance, we observe a median BER of 9.26×10^{-9} for predictive steering and 2.73×10^{-6} for baseline steering. Notably, a large portion of the drone's motion consists of continuous, linear motion, which is ideal for double exponential prediction. As such, we observe increased tethering performance with predictive steering enabled.

Laser VR. For our second scenario, we consider laser communication to a VR headset for high-quality content streaming. We track a Pico4 VR headset while the participant plays a game, moving left/right up to 5 m, up/down up to 1.8 m, and back/forth up to 5 m. Throughout all trials, the VR headset experienced a maximum acceleration of 37 m/s^2 in the z-direction and 15 m/s^2 in the x-y plane.

As shown in Figure 9e, predictive steering and baseline steering achieve median normalized laser powers of 0.99 and 0.98 and offsets of 0.81 cm and 0.89 cm, respectively, with relatively constant performance across velocities. The noticeably smaller performance gain of predictive steering is due to the prominence of micro-movements throughout VR gameplay. While such movements are typically not amenable to predictive steering, here they correspond to small (albeit fast) *physical* displacements of the marker such that the PD remains within the laser beam's diameter at all times. Steering without prediction similarly benefits from this motion pattern. Translating these results to communication performance, we observe a median BER of 1.11×10^{-8} for predictive steering and 1.18×10^{-8} for baseline steering. For high-resolution VR requiring multiple Gbps of throughput, the laser power can be increased to improve the overall SNR and decrease the resulting BER.

Mobile Power Delivery. For our final scenario, we consider the potential of continuously delivering power to mobile devices. We track the back of a smartphone while a participant performs various actions, including (1) talking on the phone while walking, (2) pausing to answer a video call, and (3) bending over while on the phone. Throughout trials, the participant moved left/right up to 5 m, up/down up to 1.8 m, and back/forth up to 5 m. The phone experienced a maximum acceleration of 49 m/s^2 in the z-direction and 27 m/s^2 in the x-y plane.

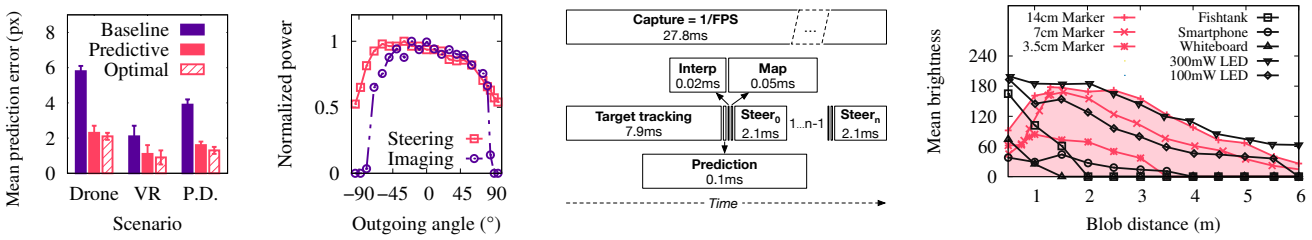
As shown in Figure 9f, predictive steering on average delivers 19% higher median normalized laser power. Predictive steering consistently achieves median normalized laser powers above 0.90. In the context of power delivery, this translates to a mobile device receiving over 90% of laser energy from the core unit. The median offset achieved by baseline steering on average is 1.33 cm, compared to 1.25 cm with predictive steering. As in the case of drone motion, higher velocities see increased performance differences between predictive and baseline steering. Additionally, during slower motions, the user occasionally occluded the PD, causing a consistent decline in power at velocities below 300 px/s that negatively skewed the median.

5.3 Tethering Micro-benchmarks

Motion Prediction. To assess the accuracy of motion prediction, we compare the predicted marker pixel locations with ground truth values. We first downsample the ground truth values (recorded at 315 FPS) to the average frame rate of Lasertag's image sensor (36 FPS). We then compare our prediction algorithm to two reference points: (1) *baseline without prediction*, which treats the target's current location as its next location; (2) *optimal case*, which uses a brute-force search to find the optimal hyperparameters α,β of the double exponential filter for each application. Although the optimal case is not practical for real-time applications due to the time-consuming brute-force search, we include it as an upper bound for comparison. We quantify prediction error as the Euclidean distance between the predicted and ground truth marker pixel locations and report the mean/STDV of prediction error.

As shown in Figure 10a, compared to the baseline, our predictive steering algorithm reduces prediction error by roughly half in all applications, achieving an average error of 1.67 px, corresponding to a 1.04 cm error at a distance of 2 m. The difference between our prediction algorithm to the optimal case is, on average, only 0.3 px (1.8 mm at 2 m). Across prediction methods, we observe that applications with higher marker velocities (i.e., drone motion) see higher mean prediction errors. This relationship arises because higher marker velocities correspond to larger displacements of the marker between consecutive frames. Inaccurate predicted locations are then increasingly far from the actual location. Tuning of the double exponential parameters or more sophisticated motion prediction models (§6) could improve accuracy at higher speeds.

Angular Range. Next we measure the angular range of the



(a) Prediction accuracy. *(b) Angular range.* *(c) Timing breakdown.* *(d) Background noise.*

Figure 10: *Lasertag micro-benchmarks and practical considerations including (a) prediction accuracy, (b) angular range of the steering and imaging subsystems, (c) the timing breakdown, and (d) impact of background noise.*

laser subsystem and the imaging subsystem. To quantify the imaging range, we place a power meter [131] at the center of the retroreflective marker to measure optical power. We then rotate the marker in 10° increments when 18 cm away from Lasertag and measure the power at each angle. To measure the laser subsystem's range, we measure the *maximum* received power at each angle, indicating how efficient the optical circuit is at delivering laser light to any angle. In both scenarios, we measure the received power relative to the maximum power.

As shown in Figure 10b, the laser subsystem is capable of delivering over 56% of its laser power up to ±95°, covering 10° more than a full hemisphere before dropping below its −3 dB point. Notably, the steering is slightly asymmetrical around the 0° mark. We hypothesize this is due to reduced MEMS mirror area at extreme positive angles, which lowers reflection efficiency [90]. The imaging subsystem delivers ≥66% of its power to the marker center over a ±81° range. This is because at extreme angles, the circular retroreflective marker becomes distorted, resulting in an incorrect center measurement.

System Speed. Finally we evaluate the delay of each step in Lasertag and plot the timing breakdown in Figure 10c. Overall, image capture takes the most time (27.8 ms), limited by the frame rate (36 FPS). Lasertag runs other steps in parallel with image capture, including target tracking (7.9 ms), motion prediction and interpolation (≈ 0.1 ms), and consecutive steering (2.1 ms each on average). On average, these timing constraints support up to four predictive steers before the next frame is requested. For a tag moving with an angular speed of 1 °/s (roughly corresponding to 10 px/s), the MEMS mirror takes approximately 0.6 ms to rotate to its updated physical deflection angle. The angular response is nonlinear, maxing out at ≈6 ms for angles above 30°. Notably, all these durations are extremely variable and dependent on hardware characteristics, image sensor configuration, and marker movement. Consequently, we observe a variable frame rate from 238 FPS (when the marker is stationary) to 14 FPS (when the marker is causing frequent ROI window updates). As a result, the number of predictive steers varies from an observed minimum of one to a maximum of eighteen. Faster MCUs and a streamlined MEMS communication protocol would decrease the overall time spent in image processing and communicating with the mirror, thereby enabling more steering between frames.

5.4 Practical Considerations

Finally, we investigate practical aspects of Lasertag.

Power Consumption. Using a Monsoon FTA22D power meter, we measure the power consumption of Lasertag (Table 1). Overall, the power consumption of the MCU, image sensor, and MEMS mirror is less than 1.5 W, 70% smaller than the *minimum* power consumption of common computer vision processors, e.g., the 5 W Jetson Nano. Notably, our chosen laser consumes 353 mW of input power to generate 50 mW of optical power. Higher efficiency/lower-power diodes can fit seamlessly within the system to lower the power consumption.

Noisy Environments. To examine Lasertag's robustness against confounding scene objects, we directly measure the perceived brightness of the retoreflective marker (cut to three diameters) and common scene objects (e.g. reflective screens, glass, whiteboards). Figure 10d plots the mean brightness of the marker as function of distance. In general, larger markers retroreflect more light than smaller markers, resulting in brighter blobs on the image sensor.[6]. In general, fairly low-power LEDs have comparable brightness to the retroreflective markers, supporting our claim that active markers would improve Lasertag's overall power efficiency at the cost of system complexity. Additionally, these results validate the importance of the marker detection algorithm in §3.2 which enables Lasertag to operate with bright light sources in the scene.

Ambient Light. Similar to supported imaging range, ambient light rejection is tightly coupled with the chosen optical filters. To reject more ambient light, a narrow bandpass filter can be utilized to only pass a small subset of wavelengths. In our implementation, we opted for a wider bandwidth (55 nm) filter to match the broad emission spectrum of the 520 nm LEDs. Despite the wide bandwidth, it still required over 200.000 lx of light to saturate the image sensor using a white LED.

6 Discussion

Additional Tracking Modes. Our framework is compatible with any image-based detection technique and is thus application-agnostic. The tracking system may be configured to support two additional modes: (1) tracking active markers, and (2) markerless object tracking. An active marker can be

[6] We observe a decrease in blob brightness within 1 m of Lasertag due to the majority of retroreflective light returning to their source instead of overlapping with the colocated fisheye lens.

Table 1: *Average Lasertag power consumption breakdown.*

Imaging	Power (mW)	Steering	Power (mW)
MCU	596	50mW LD	353
Image sensor	692	MEMS mirror	159
2x LEDs	305		
Subtotal	1593 mW	**Subtotal**	512 mW

realized by adding an active light emitter (e.g., IR LED) to the target, potentially enabling alignment feedback or uplink communication (e.g., using frequency modulation). This would also increase sensing ranges and robustness to background objects or targeted attacks (e.g., use of "spoofing" markers to fool Lasertag). The second mode requires no marker on the object, and relies fully on either standard feature detection algorithms or machine learning models to identify the target. Prior work has examined the tracking of people [10], traffic [24], and arbitrary objects [122], which can be plugged into the Lasertag framework. We leave these explorations to future work.

Motion Prediction. More sophisticated prediction models can predict the target's next position with higher accuracy. Recent work has proposed learning models that can be trained to learn complex patterns and relationships between an object's motion and its surrounding environment [13, 38, 117]. To test the potential benefits of deep learning prediction for Lasertag, we apply the N-euro Predictor [117] to the data collected during our experiments without further fine-tuning. This model reduced the prediction error from an average of 1.67 px to 0.8 px, a 52% decrease. This performance boost requires more powerful hardware for neural network inference, which Lasertag can support depending on the application requirements.

Path Blockage. Lasertag can support non-line of sight applications by leveraging its image sensor to find reflections in the scene and peer around obstructions. For example, imagine a scenario where the Lasertag core unit and a target are separated by a wall, but to one end of the wall is a mirror that is visible by both. The core unit will see the target's reflection in the mirror, and can steer the laser beam to the reflection. Because of path symmetry between the image sensor and laser, the beam will reflect off the mirror and onto the target.

7 Related Work

Colocating Tracking/Positioning with Laser Steering. Prior work has studied the integration of object acquisition, tracking, and laser steering to maintain connection of a laser beam with a target object. To the best of our knowledge, no existing systems integrate target-agnostic computer vision based tracking with laser steering. Existing systems often equip the target with retroreflectors [14, 59, 77, 83, 86, 87], photodiodes [87], or active emitters like LEDs [83, 87, 95, 134] or laser beacons [12, 20, 61, 119, 123, 139, 144]. Lasertag imposes no restrictions on the target's design, as it can be reconfigured to support tracking and steering to any object visible to the embedded camera. [4, 126] require a multi-camera system to determine the mirror tilt needed to steer to a 3D position. These systems must be carefully calibrated by hard-coding the camera system's parameters [4, 126]. Lasertag requires no *a priori* knowledge of camera characteristics, instead automatically determining the correct steering parameters during the mapping stage. [59, 83, 86, 87] perform iterative scans to track and steer to a passive mobile target. In contrast, Lasertag continuously steers directly to the target and thus does not suffer from tether downtime resulting from scanning. While [59, 83, 86, 87] perform several demonstrations of their system and describe its design, quantified performance results are not provided. [1, 2, 14, 39, 51, 64] rely on uplink feedback between the transmitter and receiver, either to enable initial beam alignment [1, 64, 141], continuously communicate link quality [39, 51] or receive GPS locations [2, 14]. We employ a purely optical approach that does not require uplink feedback.

Laser Steering. Laser beam steering mechanisms fall into two categories: non-mechanical and mechanical. Mechanical beam steering mechanisms involving rotating mirrors have large steering ranges but tend to be bulky, sensitive to mechanical stress, and limited in steering resolution [60, 142]. Non-mechanical beam steering mechanisms include acousto-optic deflectors [121, 136], optical gratings [11, 41, 67, 80, 100], tunable liquid lenses [22, 101, 150], phased arrays [29, 30, 35], and MEMS mirrors [33, 47, 85, 146]. Overall, MEMS mirror are the most appealing due to their small size, low cost, and robustness. Other solutions lead to significant loss of optical power during steering [121, 136], suffer from limited FoVs and switching speeds [11, 22, 41, 67, 80, 100, 101, 150], or have limited commercial availability [29, 30, 35].

3D Object Positioning and Tracking. [94] and [37] explore the use of feature matching combined with reprojection to perform 3D tracking. A large body of work explores light-based positioning and tracking techniques that do not employ cameras [42, 65, 76], including those requiring [46, 70, 110] active emitters or photodiodes [31, 52, 145] on the target. Other 3D object localization methodologies include ultrasonic [21], LiDAR [49], MEMS [73, 87], IMU [36, 72], and RF [3, 8, 57, 68, 120, 148] sensing. As discussed in §2, these methodologies are nontrivial to integrate with laser steering devices.

8 Conclusion

We present Lasertag, a reconfigurable framework for laser tethering with highly mobile targets. Extensive evaluation shows that Lasertag can maintain laser alignment with rapidly moving targets equipped with retroreflective markers. Lasertag paves the way for a myriad of laser-based mobile applications, including laser communication and wireless power delivery.

9 Acknowledgements

We sincerely thank our reviewers and shepherd for their insightful feedback. This work is supported in part by the National Science Foundation (GRFP-1840344, CNS-1552924). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the funding agencies or others.

References

[1] Mojtaba Mansour Abadi, Mitchell A Cox, Rakan E Alsaigh, Shaun Viola, Andrew Forbes, and Martin PJ Lavery. A space division multiplexed free-space-optical communication system that can auto-locate and fully self align with a remote transceiver. *Scientific Reports*, 9(1):1–8, 2019.

[2] Mouhammad Al Akkoumi, Robert Huck, and James Sluss. High-speed communications enabling real-time video for battlefield commanders using tracked FSO. *Proc. of SPIE - The International Society for Optical Engineering*, 2007.

[3] Marco Altini, Davide Brunelli, Elisabetta Farella, and Luca Benini. Bluetooth indoor localization with multiple neural networks. In *IEEE 5th International Symposium on Wireless Pervasive Computing*, pages 295–300, 2010.

[4] Nicolas Andreff, Brahim Tamadazte, Sounkalo Dembélé, and Zill E Hussnain. Preliminary variation on multiview geometry for vision-guided laser surgery. In *Workshop on Multi-View Geometry in Robotics*, pages 1–10, 2013.

[5] Arduino. Arduino MKR1000 WiFi. https://store-usa.arduino.cc/products/arduino-mkr1000-wifi, 2023.

[6] Charles K Asawa. Boresighting system for infrared optical receiver and transmitter, May 2 1978. US Patent 4,087,689.

[7] Ronald Azuma and Gary Bishop. Improving static and dynamic registration in an optical see-through HMD. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 197–204, 1994.

[8] R. Bajaj, S.L. Ranaweera, and D.P. Agrawal. Gps: location-tracking technology. *Computer*, 35(4):92–94, 2002.

[9] Sunil L Bangare, Amruta Dubal, Pallavi S Bangare, and Suhas Patil. Reviewing otsu's method for image thresholding. *International Journal of Applied Engineering Research*, 10(9):21777–21783, 2015.

[10] Antonio Brunetti, Domenico Buongiorno, Gianpaolo Francesco Trotta, and Vitoantonio Bevilacqua. Computer vision and deep learning techniques for pedestrian detection and tracking: A survey. *Neurocomputing*, 300:17–33, 2018.

[11] J Buck, S Serati, R Serati, H Masterson, M Escuti, J Kim, and M Miskiewicz. Polarization gratings for non-mechanical beam steering applications. In *Acquisition, Tracking, Pointing, and Laser Systems Technologies XXVI*, volume 8395, page 83950F, 2012.

[12] Jamie W Burnside, Stephen D Conrad, Allen D Pillsbury, and Catherine E DeVoe. Design of an inertially stabilized telescope for the llcd. In *Free-Space Laser Communication Technologies XXIII*, volume 7923, pages 133–140, 2011.

[13] Judith Butepage, Michael J Black, Danica Kragic, and Hedvig Kjellstrom. Deep representation learning for human motion prediction and classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6158–6166, 2017.

[14] Alberto Carrasco-Casado, Ricardo Vergaz, José M Sánchez-Pena, Eva Otón, Morten A Geday, and José M Otén. Low-impact air-to-ground free-space optical communication system design and first results. In *2011 International Conference on Space Optical Systems and Applications (ICSOS)*, pages 109–112, 2011.

[15] Charles J Carver, Hadleigh Schwartz, Qijia Shao, Nicholas Shade, Joseph P Lazzaro, Xiaoxin Wang, Jifeng Liu, Eric R Fossum, and Xia Zhou. Catch me if you can: Demonstrating laser tethering with highly mobile targets. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, pages 1–3, 2023.

[16] Charles J Carver, Qijia Shao, Samuel Lensgraf, Amy Sniffen, Maxine Perroni-Scharf, Hunter Gallant, Alberto Quattrini Li, and Xia Zhou. Sunflower: Locating underwater robots from the air. In *Proc. of MobiSys*, page 14–27, 2022.

[17] Charles J Carver, Zhao Tian, Qijia Shao, Hongyong Zhang, Kofi M Odame, Alberto Quattrini Li, and Xia Zhou. Air-water communication and sensing with light. In *2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 371–374, 2022.

[18] Charles J Carver, Zhao Tian, Hongyong Zhang, Kofi M. Odame, Alberto Quattrini Li, and Xia Zhou. AmphiLight: Direct air-water communication with laser light. In *Proc. of NSDI*, pages 373–388, 2020.

[19] Justin Chan, Anandithan Raghunath, Kelly E. Michaelsen, and Shyamnath Gollakota. Testing a drop of liquid using smartphone lidar. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 6(1), 2022.

[20] C Chen, A Grier, M Malfa, E Booen, H Harding, C Xia, M Hunwardsen, J Demers, K Kudinov, G Mak, B Smith, A Sahasrabudhe, F Patawaran, T Wang, A Wang, C Zhao, D Leang, J Gin, M Lewis, B Zhang, D Nguyen,

D Jandrain, F Haque, and K Quirk. Demonstration of a bidirectional coherent air-to-ground optical link. In *Free-Space Laser Communication and Atmospheric Propagation*, volume 10524, page 105240G, 2018.

[21] Moi Tin Chew, Fakhrul Alam, Mathew Legg, and Gourab Sen Gupta. Accurate ultrasound indoor localization using spring-relaxation technique. *Electronics*, 10(11):1290, 2021.

[22] Hyun Choi and Wan-Chin Kim. Design of mechaless LiDAR optical system with large FOV using liquid lens and fisheye lens. In *ASME-JSME 2018 Joint International Conference on Information Storage and Processing Systems and Micromechatronics for Information and Precision Equipment*, page V001T10A001, 2018.

[23] Hyun Choi, No-Cheol Park, and Wan-Chin Kim. Optical system design for light detection and ranging with ultra-wide field-of-view using liquid lenses. *Microsystem Technologies*, 26(1):121–131, 2020.

[24] Benjamin Coifman, David Beymer, Philip McLauchlan, and Jitendra Malik. A real-time computer vision system for vehicle tracking and traffic surveillance. *Transportation Research Part C: Emerging Technologies*, 6(4):271–288, 1998.

[25] Max Curran, Md Shaifur Rahman, Himanshu Gupta, Kai Zheng, Jon Longtin, Samir R. Das, and Thanvir Mohamed. FSONet: A wireless backhaul for multi-gigabit picocells using steerable free space optics. In *Proc. of MobiCom*, page 154–166, 2017.

[26] Alan J. Danker and Azriel Rosenfeld. Blob detection by relaxation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-3(1):79–92, 1981.

[27] Carlo De Santi, Matteo Meneghini, Alessandro Caria, Ezgi Dogmus, Malek Zegaoui, Farid Medjdoub, Boris Kalinic, Tiziana Cesca, Gaudenzio Meneghesso, and Enrico Zanoni. Gan-based laser wireless power transfer system. *Materials*, 11(1):153, 2018.

[28] DJI. DJI Mini 3 Specs. https://www.dji.com/mini-3/specs, 2023.

[29] Jonathan Doylend, Martijn R Heck, Jock Bovington, Jon Peters, Larry Coldren, and John Bowers. Free-space beam steering in two dimensions using a silicon optical phased array. In *Optical Fiber Communication Conference*, pages OM2J–1, 2012.

[30] Jonathan K. Doylend, M.J.R. Heck, Jock T. Bovington, Jonathan D Peters, L.A. Coldren, and J.E. Bowers. Two-dimensional free-space beam steering with an optical phased array on silicon-on-insulator. *Optics Express*, 19(22):21595–21604, 2011.

[31] Minzhen Du. *Assessment of a Low Cost IR Laser Local Tracking Solution for Robotic Operations*. PhD thesis, Virginia Tech, 2021.

[32] Josef Eisenring, Klaus W Hildebrand, and Jakob Tanner. Coaxial transmitting and receiving optics for an electro-optic range finder, August 28 1979. US Patent 4,165,936.

[33] Yusuf Said Eroglu, Ismail Guvenc, Alphan Sahin, Nezih Pala, and Murat Yuksel. Diversity combining and piezoelectric beam steering for multi-element VLC networks. In *Proceedings of the 3rd Workshop on Visible Light Communication Systems*, pages 25–30, 2016.

[34] Zhufeng Fan, Jinyu Zhan, and Wei Jiang. Detecting deepfake videos by visual-audio synchronism: work-in-progress. In *Proceedings of the 2021 International Conference on Embedded Software*, pages 31–32, 2021.

[35] Reza Fatemi, Aroutin Khachaturian, and Ali Hajimiri. A low power PWM optical phased array transmitter with 16° field-of-view and 0.8° beamwidth. In *2018 IEEE Radio Frequency Integrated Circuits Symposium (RFIC)*, pages 28–31, 2018.

[36] Ling feng Shi, Yu-Le Zhao, Gong xu Liu, Sen Chen, Yue Wang, and Yi-Fan Shi. A robust pedestrian dead reckoning system using low-cost magnetic and inertial sensors. *IEEE Transactions on Instrumentation and Measurement*, 68:2996–3003, 2019.

[37] Dmitry Gorodnichy, S Malik, and Gerhard Roth. Affordable 3D face tracking using projective vision. In *Proc. of Int. Conf. on Vision Interface*, pages 383–390, 2002.

[38] Mahir Gulzar, Yar Muhammad, and Naveed Muhammad. A survey on motion prediction of pedestrians and vehicles for autonomous driving. *IEEE Access*, 9:137957–137969, 2021.

[39] Himanshu Gupta, Max Curran, Jon Longtin, Torin Rockwell, Kai Zheng, and Mallesham Dasari. Cyclops: An FSO-Based Wireless Link for VR Headsets. In *Proc. of SIGCOMM*, page 601–614, 2022.

[40] Markus Haase and Helmut Schäfer. Upconverting nanoparticles. *Angewandte Chemie International Edition*, 50(26):5808–5829, 2011.

[41] Michael Hall, Qing Chao, Byron Taylor, and Xinqiao Liu. Non-mechanical beam steering for depth sensing, February 25 2020. US Patent 10,574,973.

[42] Anum Hameed and Hafiza Anisa Ahmed. Survey on indoor positioning applications based on different technologies. In *2018 12th International Conference on Mathematics, Actuarial Science, Computer Science and Statistics (MACS)*, pages 1–5, 2018.

[43] Yuze He, Li Ma, Zhehao Jiang, Yi Tang, and Guoliang Xing. Vi-eye: Semantic-based 3D point cloud registration for infrastructure-assisted autonomous driving. In *Proc. of MobiCom*, page 573–586, 2021.

[44] Niels Henze, Sven Mayer, Huy Viet Le, and Valentin Schwind. Improving software-reduced touchscreen latency. In *Proceedings of the 19th International Conference on Human-Computer Interaction with Mobile Devices and Services*, 2017.

[45] S Hinz. Fast and subpixel precise blob detection and attribution. In *IEEE International Conference on Image Processing*, volume 3, pages III–457, 2005.

[46] Andrew Hogue, Matt Robinson, MR Jenkin, and Robert S Allison. A vision-based head tracking system for fully immersive displays. In *Proceedings of the workshop on Virtual environments*, pages 179–187, 2003.

[47] Sven TS Holmstrom, Utku Baran, and Hakan Urey. MEMS laser scanners: a review. *Journal of Microelectromechanical Systems*, 23(2):259–275, 2014.

[48] Jinhui Huang, Chunlin Li, and Jie Yu. Resource prediction based on double exponential smoothing in cloud computing. In *2012 2nd International Conference on Consumer Electronics, Communications and Networks*, pages 2056–2060, 2012.

[49] Rui Huang, Wanyue Zhang, Abhijit Kundu, Caroline Rebecca Pantofaru, David Alexander Ross, Thomas Funkhouser, and Alireza Fathi. An LSTM approach to temporal 3d object detection in lidar point clouds. In *ECCV*, pages 266–282, 2020.

[50] Texas Instruments. OPT101. https://www.ti.com/lit/ds/symlink/opt101.pdf, 2023.

[51] Femi Ishola and Mengu Cho. Experimental study on photodiode array sensor aided mems fine steering mirror control for laser communication platforms. *IEEE Access*, 9:100197–100207, 2021.

[52] Shahidul Islam, Bogdan Ionescu, Cristian Gadea, and Dan Ionescu. Indoor positional tracking using dual-axis rotating laser sweeps. In *IEEE International Instrumentation and Measurement Technology Conference Proceedings*, pages 1–6, 2016.

[53] David J Israel, Bernard L Edwards, Richard L Butler, John D Moores, Sabino Piazzolla, Nic du Toit, and Lena Braatz. Early results from NASA's laser communications relay demonstration (LCRD) experiment program. In *Free-Space Laser Communications XXXV*, volume 12413, page 1241303, 2023.

[54] Vikram Iyer, Elyas Bayati, Rajalakshmi Nandakumar, Arka Majumdar, and Shyamnath Gollakota. Charging a smartphone across a room using lasers. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 1(4), January 2018.

[55] Johannes James, Vikram Iyer, Yogesh Chukewad, Shyamnath Gollakota, and Sawyer B. Fuller. Liftoff of a 190 mg laser-powered aerial vehicle: The lightest wireless robot to fly. In *Proc. of ICRA*, 2018.

[56] Tim P Johnson and Jose Sasian. Image distortion, pupil coma, and relative illumination. *Appl. Opt.*, 59(22):G19–G23, 2020.

[57] Kiran Joshi, Dinesh Bharadia, Manikanta Kotaru, and Sachin Katti. WiDeo: Fine-grained Device-free Motion Tracing using RF Backscatter. In *Proc. of NSDI*, page 189–204, 2015.

[58] Jordin Kare and Thomas Nugent. Laser power beaming on a shoestring. *AIP Conference Proceedings*, 997:97–108, 04 2008.

[59] Abhishek Kasturi, Veljko Milanovic, Bryan H Atwood, and James Yang. UAV-borne lidar with mems mirror-based scanning capability. In *Laser Radar Technology and Applications XXI*, volume 9832, pages 206–215, 2016.

[60] Yagiz Kaymak, Roberto Rojas-Cessa, Jianghua Feng, Nirwan Ansari, MengChu Zhou, and Tairan Zhang. A survey on acquisition, tracking, and pointing mechanisms for mobile free-space optical communications. *IEEE Communications Surveys and Tutorials*, 20(2):1104–1123, 2018.

[61] Kamugisha Kazaura, Kazunori Omae, Toshiji Suzuki, Mitsuji Matsumoto, Edward Mutafungwa, Tadaaki Murakami, Koichi Takahashi, Hideki Matsumoto, Kazuhiko Wakamori, and Yoshinori Arimoto. Performance evaluation of next generation free-space optical communication system. *IEICE Transactions on Electronics*, E90-C, 2007.

[62] Keysight. M8190A. https://www.keysight.com/us/en/product/M8190A/12-gsa-s-arbitrary-waveform-generator.html, 2023.

[63] Keysight. MSOS254A. https://www.keysight.com/us/en/product/MSOS254A/high-definition-oscilloscope-2-5ghz-4-analog-16-digital-channels.html, 2023.

[64] Mahmudur Khan and Murat Yuksel. Autonomous alignment of free-space-optical links between UAVs. In *Proc. of HotWireless*, 2015.

[65] Hiam M Khoury and Vineet R Kamat. Evaluation of position tracking technologies for user localization in indoor construction environments. *Automation in construction*, 18(4):444–457, 2009.

[66] Dennis K. Killinger. Free space optics for laser communication through the air. *Optics & Photonics News*, 13:36–42, 2002.

[67] AMJ Koonen, CW Oh, and E Tangdiongga. Reconfigurable free-space optical indoor network using multiple pencil beam steering. In *2014 OptoElectronics and Communication Conference and Australian Conference on Optical Fibre Technology*, pages 204–206, 2014.

[68] Manikanta Kotaru, Kiran Joshi, Dinesh Bharadia, and Sachin Katti. Spotfi: Decimeter level localization using wifi. *Proc. of SIGCOMM*, 45(4):269–282, 2015.

[69] A. Kotsopoulos, Pantazi A., and Antonakopoulos T. Control for high-speed archimedean spiral nanopositioning. In *2010 17th IEEE International Conference on Electronics, Circuits and Systems*, page 3, 2010.

[70] Anil Kumar and Pinhas Ben-Tzvi. Spatial object tracking system based on linear optical sensor arrays. *IEEE Sensors Journal*, 16(22):7933–7940, 2016.

[71] Nina Siu-Ngan Lam. Spatial interpolation methods: a review. *The American Cartographer*, 10(2):129–150, 1983.

[72] Ariel Larey, Eliel Aknin, and Itzik Klein. Multiple inertial measurement units–an empirical study. *IEEE Access*, PP:1–1, 04 2020.

[73] Steven M LaValle, Anna Yershova, Max Katsev, and Michael Antonov. Head tracking for the Oculus Rift. In *Proc. of ICRA*, pages 187–194, 2014.

[74] Joseph J. LaViola. Double Exponential Smoothing: An Alternative to Kalman Filter-Based Predictive Tracking. In *Proceedings of the workshop on Virtual environments*, page 199–206, 2003.

[75] Xiaobao Lee and Chunhui Wang. Optical design for uniform scanning in MEMS-based 3D imaging LiDAR. *Appl. Opt.*, 54(9):2219–2223, 2015.

[76] Angus Leigh, Joelle Pineau, Nicolas Olmedo, and Hong Zhang. Person tracking and following with 2D laser scanners. In *Proc. of ICRA*, pages 726–733, 2015.

[77] Long Li, Runzhou Zhang, Zhe Zhao, Guodong Xie, Peicheng Liao, Kai Pang, Haoqian Song, Cong Liu, Yongxiong Ren, Guillaume Labroille, Pu Jian, Dmitry Starodubov, Brittany Lynn, Robert Bock, Moshe Tur, and Alan E. Willner. High-capacity free-space optical communications between a ground transmitter and a ground receiver via a uav using multiplexing of multiple orbital-angular-momentum beams. *Scientific Reports*, 7(1):17427, 2017.

[78] Jiandong Liang, Chris Shaw, and Mark Green. On temporal-spatial realism in the virtual reality environment. In *Proc. of UIST*, page 19–25, 1991.

[79] Kuan-Ying Lin, Yi-Hsing Tseng, and Kai-Wei Chiang. Interpretation and transformation of intrinsic camera parameters used in photogrammetry and computer vision. *Sensors*, 22(24), 2022.

[80] Yu-Hua Lin, Milind Mahajan, Donald Taber, Bing Wen, and Bruce Winker. Compact 4 cm aperture transmissive liquid crystal optical phased array for free-space optical communications. In *Free-Space Laser Communications V*, volume 5892, page 58920C, 2005.

[81] Wolfram MathWorld. Archimedes' spiral. https://mathworld.wolfram.com/ArchimedesSpiral.html, 2021.

[82] Joanna McKittrick and Lauren E Shea-Rohwer. Down conversion materials for solid-state lighting. *Journal of the American Ceramic Society*, 97(5):1327–1352, 2014.

[83] V Milanović, N Siu, A Kasturi, M Radojičić, and Y Su. MEMSEye for optical 3D position and orientation measurement. In *MOEMS and Miniaturized Systems X*, volume 7930, page 79300U, 2011.

[84] Veljko Milanovic, Kenneth Castelino, and Daniel T. McCormick. Highly adaptable MEMS-based display with wide projection angle. In *IEEE 20th International Conference on Micro Electro Mechanical Systems (MEMS)*, pages 143–146, 2007.

[85] Veljko Milanovic, Kenneth Castelino, and Daniel T. McCormick. Highly adaptable MEMS-based display with wide projection angle. In *IEEE 20th International Conference on Micro Electro Mechanical Systems*, pages 143–146, 2007.

[86] Veljko Milanović, Abhishek Kasturi, James Yang, and Frank Hu. A fast single-pixel laser imager for vr/ar headset tracking. In *MOEMS and Miniaturized Systems XVI*, volume 10116, page 101160E, 2017.

[87] Veljko Milanovic and Wing Kin Lo. Fast and high-precision 3D tracking and position measurement with mems micromirrors. In *IEEE/LEOS International Conference on Optical MEMs and Nanophotonics*, pages 72–73, 2008.

[88] MirrorcleTech. A5M24.3-2400AL. https://www.mirrorcletech.com/pdf/DSI/

MirrorcleTech_Datasheet_A5M24.3-2400AL.pdf, 2023.

[89] MirrorcleTech. AN002 - approximate linear models for mems mirrors. https://www.mirrorcletech.com/pdf/AN/AN002_-_Approximate_Linear_Models_for_MEMS_Mirrors.pdf, 2023.

[90] MirrorcleTech. Angle of incidence and field distortions. https://www.mirrorcletech.com/pdf/AN/AN004_-_Angle_of_Incidence_and_Field_Distortions.pdf, 2023.

[91] MirrorcleTech. DR-10-056-00. https://www.mirrorcletech.com/wp/products/hardware/drivers/digital/, 2023.

[92] Steve P Monacos, Angel A Portillo, William Liu, James W Alexander, and Gerardo G Ortiz. A high frame rate CCD camera with region-of-interest capability. In *2001 IEEE Aerospace Conference Proceedings (Cat. No. 01TH8542)*, volume 3, pages 3–1513, 2001.

[93] Mouser. LST1-01H06-GRN1-01. https://mou.sr/45Lh6r1, 2023.

[94] Jurriaan D Mulder, Jack Jansen, and Arjen Van Rhijn. An affordable optical head tracking system for desktop vr/ar systems. In *Proceedings of the workshop on Virtual environments*, pages 215–223, 2003.

[95] Kazuki Nakamura, Shingo Nakagawa, Hiroshi Matsubara, Daisuke Tatsui, Kiyotaka Seki, Shinichiro Haruyama, and Fumio Teraoka. Development of broadband telecommunications system for railways using laser technology. *Electrical Engineering in Japan*, 190(3):45–56, 2015.

[96] S Narayana, R V Prasad, V Rao, L Mottola, and TV Prabhakar. Hummingbird: Energy efficient gps receiver for small satellites. In *Proc. of MobiCom*, pages 1–13, 2020.

[97] T Nielsen, M Fricke, D Hellweg, and P Andresen. High efficiency beam splitter for multifocal multiphoton microscopy. *Journal of microscopy*, 201(3):368–376, 2001.

[98] NIST. Single exponential smoothing. https://www.itl.nist.gov/div89x8/handbook/pmc/section4/pmc431.html, 2023.

[99] NXP. MMA8452Q. https://www.nxp.com/docs/en/data-sheet/MMA8452Q.pdf, 2023.

[100] Chin Wan Oh, Zizheng Cao, Eduward Tangdiongga, and Ton Koonen. Free-space transmission with passive 2D beam steering for multi-gigabit-per-second per-beam indoor optical wireless networks. *Optics Express*, 24(17):19211–19227, 2016.

[101] Hiromasa Oku and Masatoshi Ishikawa. High-speed liquid lens with 2 ms response and 80.3 nm root-mean-square wavefront error. *Applied Physics Letters*, 94(22):221108, 2009.

[102] OpenMV. OV5640. https://openmv.io/products/ov5640-fpc-camera-module, 2023.

[103] Edmund Optics. 22-827. https://www.edmundoptics.com/p/132mm-fl-m12-mount-180-deg-fisheye-lens/49590/, 2023.

[104] Edmund Optics. 62-274. https://www.edmundoptics.com/p/23quot-format-c-mount-fisheye-lens-18mm-fl/16922/, 2023.

[105] Edmund Optics. 67-119. https://www.edmundoptics.com/p/532nm-25mm-diameter-od-4-notch-filter/21656/, 2023.

[106] Edmund Optics. 83-107. https://www.edmundoptics.com/p/160mm-fl-no-ir-cut-filter-f4-micro-video-lens/26288/, 2023.

[107] Oralite. M82. https://www.orafol.com/en/americas/products/oralite-m82-aids-to-navigation-tape, 2023.

[108] Dr. Rüdiger Paschotta. Polarization of laser emission, Oct 2018.

[109] JA Picazo-Bueno, M Trusiak, and V Micó. Single-shot slightly off-axis digital holographic microscopy with add-on module based on beamsplitter cube. *Optics Express*, 27(4):5655–5669, 2019.

[110] Ling Qin, Ben Niu, Bao-Shan Li, Xiao-Li Hu, and Yong-Xing Du. High precision indoor positioning algorithm of single led lamp based on a-bayes. *Optik*, 241:167190, 2021.

[111] Bharath Ramesh, Hong Yang, Garrick Orchard, Ngoc Anh Le Thi, Shihao Zhang, and Cheng Xiang. DART: distribution aware retinal transform for event-based cameras. *IEEE transactions on pattern analysis and machine intelligence*, 42(11):2767–2780, 2019.

[112] Hamada Rizk, Yuma Okochi, and Hirozumi Yamaguchi. Demonstrating hitonavi-μ: A novel wearable lidar for human activity recognition. In *Proc. of MobiCom*, 2022.

[113] Miguel Fabián Romero Rondón, Lucile Sassatelli, Ramón Aparicio Pardo, and Frédéric Precioso. Track: a multi-modal deep architecture for head motion prediction in 360° videos. In *IEEE International Conference on Image Processing*, pages 2586–2590, 2020.

[114] John Rzasa. *Pointing, acquisition, and tracking for directional wireless communications networks*. PhD thesis, The University of Maryland, 2012.

[115] Wee-Leong Saw, Hazem H Refai, and James J Sluss Jr. Free space optical alignment system using gps. In *Free-Space Laser Communication Technologies XVII*, volume 5712, pages 101–109, 2005.

[116] Qijia Shao, Amy Sniffen, Julien Blanchet, Megan E. Hillis, Xinyu Shi, Themistoklis K. Haris, Jason Liu, Jason Lamberton, Melissa Malzkuhn, Lorna C. Quandt, James Mahoney, David J. M. Kraemer, Xia Zhou, and Devin Balkcom. Teaching american sign language in mixed reality. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 4(4), dec 2020.

[117] Qijia Shao, Jian Wang, Bing Zhou, Vu An Tran, Gurunandan Krishnan, and Shree Nayar. N-euro predictor: A neural network approach for smoothing and predicting motion trajectory. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 7(3), 2023.

[118] K P Shaw and S Y Hsu. Horizontal distance and height determining falling pattern. *Journal of Forensic Sciences*, 43(4):765–771, July 1998.

[119] Josef Sofka, Vladimir V Nikulin, Victor A Skormin, David H Hughes, and David J Legare. Laser communication between mobile platforms. *IEEE transactions on Aerospace and Electronic Systems*, 45(1):336–346, 2009.

[120] Elahe Soltanaghaei, Avinash Kalyanaraman, and Kamin Whitehouse. Multipath Triangulation: Decimeter-Level WiFi Localization and Orientation with a Single Unaided Receiver. In *Proc. of MobiSys*, page 376–388, 2018.

[121] In Keun Son and Shiwen Mao. A survey of free space optical networks. *Digital communications and networks*, 3(2):67–77, 2017.

[122] Thad Starner, Bastian Leibe, David Minnen, Tracy Westyn, Amy Hurst, and Justin Weeks. The perceptive workbench: Computer-vision-based gesture tracking, object tracking, and 3D reconstruction for augmented desks. *Machine Vision and Applications*, 14:59–71, 2003.

[123] Daria Stepanova, Valentin Pryanichnikov, Sergey Khandorin, Andrey Kulchitsky, and Alexey Kuznetsov. Developing a highly accurate pointing system for free space optical communications. *Journal of Communications*, 14(12), 2019.

[124] Menlo Systems. APD210. https://www.menlosystems.com/products/photodetectors/apd210/, 2023.

[125] Midwest Optical Systems. BN520. https://midopt.com/filters/bn520/, 2023.

[126] Brahim Tamadazte, Rupert Renevier, Jean-Antoine Séon, Andrey V. Kudryavtsev, and Nicolas Andreff. Laser beam steering along three-dimensional paths. *IEEE/ASME Transactions on Mechatronics*, 23(3):1148–1158, 2018.

[127] Thorlabs. Beamsplitter guide. https://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=9028, 2023.

[128] Thorlabs. C392TME-A. https://www.thorlabs.com/thorproduct.cfm?partnumber=C392TME-A, 2023.

[129] Thorlabs. L638P. https://www.thorlabs.com/thorproduct.cfm?partnumber=L638P200, 2023.

[130] Thorlabs. PBS25-633-HP. https://www.thorlabs.com/thorproduct.cfm?partnumber=PBS25-633-HP, 2023.

[131] Thorlabs. S121C. https://www.thorlabs.com/thorproduct.cfm?partnumber=S121C, 2023.

[132] Thorlabs. Small beam diameter scanning galvo mirror systems. https://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=3770, 2023.

[133] Thorlabs. TRS127-020-A. https://www.thorlabs.com/thorproduct.cfm?partnumber=TRS127-020-A, 2023.

[134] Hideki Urabe, Shinichiro Haruyama, Tomohiro Shogenji, Shoichi Ishikawa, Masato Hiruta, Fumio Teraoka, Tetsuya Arita, Hiroshi Matsubara, and Shingo Nakagawa. High data rate ground-to-train free-space optical communication system. *Optical Engineering*, 51(3):031204, 2012.

[135] Frank Van Diggelen and Per Enge. The world's first GPS MOOC and worldwide laboratory using smartphones. In *Proceedings of the 28th International Technical Meeting of the Satellite division of the Institute of Navigation*, pages 361–369, 2015.

[136] D Yu Velikovskiy, VE Pozhar, and MM Mazur. Acousto-optics devices for high-power laser beam. In *WDS*, volume 12, pages 65–68, 2012.

[137] Vicon. Vicon documentation. https://docs.vicon.com/pages/viewpage.action?pageId=107483982, 2023.

[138] Allied Vision. Alvium 1800 U240. https://www.alliedvision.com/en/products/alvium-configurator/alvium-1800-u/240/, 2023.

[139] Anjitha Viswanath, Shailesh Singh, Vk Jain, and Subrat Kar. Design and implementation of MOEMS based ground to satellite free space optical link under turbulence condition. *Procedia Computer Science*, 46:1216–1222, 12 2015.

[140] Michele Volpi and Devis Tuia. Dense semantic labeling of subdecimeter resolution images with convolutional neural networks. *IEEE Transactions on Geoscience and Remote Sensing*, 55(2):881–893, 2017.

[141] Shane M. Walsh, Skevos F. E. Karpathakis, Ayden S. McCann, Benjamin P. Dix-Matthews, Alex M. Frost, David R. Gozzard, Charles T. Gravestock, and Sascha W. Schediwy. Demonstration of 100 Gbps coherent free-space optical communications at LEO tracking rates. *Scientific Reports*, 12(1):18345, 2022.

[142] Dingkang Wang, Connor Watkins, and Huikai Xie. MEMS Mirrors for LiDAR: A Review. *Micromachines*, 11(5), 2020.

[143] Greg Welch and Gary Bishop. An introduction to the kalman filter. *Proc. Siggraph Course*, 8, 01 2006.

[144] Thomas Weyrauch and Mikhail Vorontsov. Free-space laser communications with adaptive optics: Atmospheric compensation experiments. *Journal of Optical and Fiber Communications Reports*, 1:355–379, 05 2010.

[145] Linghui Yang, Yuanlin Pan, Jiarui Lin, Yang Liu, Yue Shang, Shuo Yang, and Hanwen Cao. Automatic guidance method for laser tracker based on rotary-laser scanning angle measurement. *Sensors*, 20(15):4168, 2020.

[146] Liangchen Ye, Gaofei Zhang, Zhen You, and Chi Zhang. A 2D resonant MEMS scanner with an ultra-compact wedge-like multiplied angle amplification for miniature LIDAR application. In *Sensors, 2016 IEEE*, pages 1–3, 2016.

[147] Lin Yongbing, Zhang Guoxiong, and Li Zhen. An improved cat's-eye retroreflector used in a laser tracking interferometer system. *Measurement Science and Technology*, 14(6):N36, May 2003.

[148] Yuan Zhuang, Jun Yang, You Li, Longning Qi, and Naser El-Sheimy. Smartphone-based indoor localization with bluetooth low energy beacons. *Sensors*, 16(5), 2016.

[149] Jan Zizka, Alex Olwal, and Ramesh Raskar. SpeckleSense: Fast, Precise, Low-Cost and Compact Motion Sensing Using Laser Speckle. In *Proc. of UIST*, page 489–498, 2011.

[150] Mo Zohrabi, Robert H Cormack, and Juliet T Gopinath. Nonmechanical beam steering using tunable lenses. In *2017 Conference on Lasers and Electro-Optics (CLEO)*, pages 1–2, 2017.

[151] Mo Zohrabi, Wei Yang Lim, Robert H Cormack, Omkar D Supekar, Victor M Bright, and Juliet T Gopinath. Lidar system with nonmechanical electrowetting-based wide-angle beam steering. *Optics Express*, 27(4):4404–4415, 2019.

MobileConfig: Remote Configuration Management for Mobile Apps at Hyperscale

Matt Guo[1], Soteris Demetriou[2], Joey Yang[1], Michael Leighton[1], Diedi Hu[1], Tong Bao[1], Amit Adhikari[1], Thawan Kooburat[1], Annie Kim[1], and Chunqiang Tang[1]

[1] Meta Platforms
[2] Imperial College London

Abstract

While software configuration management is a ubiquitous practice in the industry and has been extensively studied, prior research has focused solely on desktop or server applications. This paper presents MobileConfig, perhaps the world's largest configuration management system for mobile apps. It has been in production since 2015 and manages apps running on billions of devices, including Facebook, Instagram, Messenger, and AR/VR/glasses apps. Every day, Meta's developers make a staggering number of live configuration changes, often in the thousands, to remotely control mobile apps, driving them to change runtime behaviors without requiring app code updates. These configuration changes serve diverse purposes such as A/B testing, feature rollout, and app personalization. We discuss how MobileConfig addresses several challenges unique to mobile environments, including (1) the lack of data consistency models that can simultaneously ensure both fast app startup and configuration data freshness; (2) the risk of misconfiguration impacting billions of app users; and (3) the proliferation of mobile client SDKs needed to support diverse mobile platforms, programming languages, and configuration use cases.

1 Introduction

At Meta, we develop tens of mobile apps used by billions of people. Each year, more than 1,000 developers contribute to the codebase of our most popular app. Each week, the app is updated with over 300 code changes, and a new version is released to the app store. This fast-paced development and release of the app, especially when it is collectively done by a large number of developers, pose significant challenges to the reliability of the app. Unlike server-side software, where bug fixes can be deployed instantly under our control, mobile apps lack a transparent method to upgrade from a buggy version to a newer one without user involvement.

Contributions: Chunqiang and Thawan initiated the MobileConfig project in 2014. In terms of paper writing, Chunqiang and Soteris drafted the paper and contributed equally. In terms of coding, Thawan led the project's development from 2014 to 2017, succeeded by Matt until 2023. All other co-authors also made major contributions to the project's development.

We work around the constraint of having no control over app upgrades by providing the ability to remotely modify an app's configuration (*config* for short) and trigger it to change behaviors without a code upgrade. Consider the example of an experimental app feature. Initially, it is gated by a remote config that is enabled for only 0.01% of users. If a bug is detected, the feature can be instantly disabled via a remote config change without user involvement, ensuring that the bug will not be exposed to any user. Besides feature gating, remote configs enable rich functions (§2.1), such as A/B testing [20–22, 39] and personalized user experiences.

Remote configs are extensively used in all our apps, as evidenced by the fact that the frequency of config commits is about 15 times that of code commits. In this paper, we present MobileConfig, our configuration management system for mobile apps, and share our experience in addressing several challenges unique to mobile environments.

Slow app startup. When a developer modifies the value of a remote config on the server side, she typically wants app devices to fetch and apply the new config value soon. For instance, if the config is used to run an A/B test, the developer can obtain useful A/B test results only after a sufficient number of devices have applied the new value. For this reason, Google's Firebase RemoteConfig [16], one of the most widely used mobile configuration systems, suggests that,

> "*If you are loading values for an A/B testing experiment, this strategy (of showing a loading screen to block the app startup until it finishes fetching all the latest configs from the server side) is very strongly recommended [17].*"

Unfortunately, this simple strategy of blocking app startup increases startup time, an important app quality metric [4, 11, 19, 27, 30, 37, 41]. Our evaluation shows that this approach would unacceptably add 2,499 ms to the startup time of our largest app. To put this into perspective, our app performance team often invests months in optimizing an app to reduce its startup time by just tens of milliseconds, and the effort is considered worthwhile as it impacts the engagement of billions of users.

Slow startup plays an outsized role in users' negative perception of an app and discourages users from opening the app,

especially for short interactions such as typing a brief message on a social-network app. In the industry, AppDynamics reported that 60% of users abandon an app after their first usage if they encounter performance problems [5]. At Meta, based on extensive user studies, the product teams set the priority that "*startup is the single most important factor for app performance, and anything not absolutely necessary will be ruthlessly removed from the startup path.*"

At startup, an app wants the latest configs, but fetching them from the server delays startup. This tension has not been addressed by existing data consistency models, as they do not take advantage of the unique characteristics of configs, specifically the timing and granularity of config consumption.

To resolve this tension, we propose *progressive consistency*. Our key insight is that, out of our app's as many as 4,300 configs, only a small fraction is accessed during the app's early startup phase. App startup can proceed by consuming those configs' locally cached values, even if they are stale. In parallel, the app asynchronously fetches updates for all configs from the server. Once the updates arrive, it switches to using the new values for the vast majority of configs that have not yet been accessed. Our production data shows that progressive consistency achieves 99.7% config data freshness across billions of devices without slowing down app startups.

Despite mixed consumption of old and new configs, progressive consistency ensures app correctness by guaranteeing repeatable reads [7], monotonic reads [40], and intra-config consistency. Note that, due to the nature of config usage, it is unnecessary to enforce cross-config consistency (§3.3). To mitigate the downside of consuming some stale configs, MobileConfig offers an emergency-push mechanism that can swiftly purge harmful stale configs from app devices. Finally, it can cap config staleness below an app-specified threshold.

Config error prevention. Every day, our app developers make thousands of live config changes in production, with each change carrying the risk of causing app malfunctions. Moreover, recovering from a config error on mobile devices is much harder than that on servers because unlike servers, we have no access to users' mobile devices to directly rectify the error. MobileConfig uses a defense-in-depth approach to prevent config errors. This includes enforcing compile-time validation, conducting multi-stage canary tests [15], and continuously comparing configs on billions of devices with their values on the server to catch inconsistency.

One surprising finding is that traditional small-scale canary tests are ineffective for mobile apps. This differs from not only the general industry practice [12] but also our own experience in datacenter environments where small-scale canary tests are effective. We find that one cause of this difference is the heterogeneity of mobile environments, including many wild device types and legacy OS versions. As a result, many bugs appear only on a small fraction of user devices, making them hard to detect through small-scale canary tests. Another cause is that, out of a large app's many features, only a small fraction of users may use a specific feature during the canary time window and trigger the bug. Because of these difficulties, MobileConfig uses unconventionally large canary tests.

Proliferation of mobile SDK. MobileConfig supports apps on various platforms (Android, iOS, Windows, Linux, MacOS, and custom OSes for AR/VR/glasses/display) and multiple programming languages (Java, Objective-C, Kotlin, Swift, C++, JavaScript, and ReactNative). The expansion of platforms and languages would naturally lead to an increase in the number of mobile Software Development Kits (SDKs). Moreover, historically, different config use cases such as A/B testing and personalization were supported by separate SDKs and backend systems, leading to further SDK proliferation.

Two key insights help eliminate SDK proliferation. First, to support diverse config use cases at the scale of billions of devices, it is far easier to implement sophisticated capabilities and deploy code changes on the server side than on the mobile client side. On the client side, we simplify the SDK by offering a uniform config API and data schema, agnostic of various config use cases. On the server side, we use a translation layer to dynamically map mobile config parameters to various backends that support different use cases. By introducing this one level of remapping on the server side, a single client SDK can seamlessly work with various backends.

Second, to support diverse platforms and languages, we use a proper mix of cross-platform C++ code and platform-specific code. While C++ is known for cross-platform development, we have discovered that the overhead of using Java Native Interface (JNI) to bridge C++ is too high for latency-sensitive config-read operations. Consequently, on Android, we employ native Java code for these operations, while employing cross-platform C++ code for all other operations.

Contributions. We summarize our contributions below.

- To our knowledge, this is the first systematic study on configuration management for mobile apps. We compare design alternatives and identify pitfalls in some widely used mobile config systems. We also report lessons learned from operating MobileConfig on billions of devices in the wild.
- We demonstrate the practicality of employing remote configs to push the limits of agile app development—our developers make thousands of live config changes daily. We hope that our experience will inspire others.
- We propose *progressive consistency* to meet mobile apps' requirements for both fast app startup and fresh config data. This is not possible with existing consistency models.
- We prevent config errors using a defense-in-depth approach. In particular, we demonstrate that traditional small-scale canary tests are ineffective for mobile apps.
- We avoid SDK proliferation by using cross-platform code and server-side config parameter remapping.

2 Config Usage

In this section, we use production data and examples to demonstrate the usage of configs and the challenges that MobileConfig needs to address. Figure 1 shows an example of how an app uses MobileConfig. In this example, `ButtonCfg` is a *config*, and `ButtonCfg.color` is a *parameter*. The new code path for the experimental "*Button*" feature is gated behind the `ButtonCfg.isEnabled` parameter, which is personalized and can return different values for different users.

```
class ButtonCfg {bool isEnabled; String color; int size;}
class MusicCfg {int volume; String list; bool shuffle;}

if(MobileConfig.getBool(ButtonCfg.isEnabled /* Personalized */)){
    // New code path for the experimental "Button" feature.
    color = MobileConfig.getString(ButtonCfg.color);
    ...
} else {
    // Old code path without the experimental "Button" feature.
    ...
}
```

Figure 1: An app uses a simple API to access remote configs.

2.1 Config Use Cases

Remote configs enable many powerful use cases, as illustrated in the examples below.

- Developers can update a config to incrementally enable an experimental app feature, starting with our employees and gradually expanding to the general population.
- Developers can use a config to set up an A/B test on different users to assess the impact of a new product feature on key business metrics.
- Configs can be used along with machine learning (ML) to personalize user experience. For example, when a user login fails, if ML predicts that the user is unlikely to succeed with password retries, it sends a one-time passcode via SMS to the user. We use a config to store the ML-personalized per-user login retry setting.
- Some parameters that control an app's behavior, such as the amount of data to prefetch from the server, depend on the execution context, such as battery level and network performance. We apply contextual Bayesian optimization to tune these parameters and manage them via a config.
- When datacenters face capacity shortages, specific config changes can be promptly distributed to mobile devices to disable less-essential app features, accordingly alleviating the load they impose on datacenter backend services [25].

Backend	Feature Rollout	A/B Testing	Mutable Parameters	700+ Custom Functions	Dev
Parameters%	26.0%	40.8%	23.0%	2.7%	7.4%

Table 1: Breakdown of config parameters by backends.

To support diverse use cases, many different config *backend* systems have been developed, as summarized in Table 1. The *feature rollout* backend allows hundreds of teams to independently enable or disable different features for users without interfering with one another. The *A/B testing* backend enables developers to study the effectiveness of product features via A/B testing [21]. With the *mutable parameters* backend, instead of hardcoding constants in code, developers can set them as parameters that can be updated remotely without upgrading the installed app. The *custom-function* framework allows developers to easily introduce a new config backend. More than 700 custom backends have been implemented, mostly for personalization and ML model automation. Finally, the *Dev* backend is for local testing only.

Historically, the team that developed a new config backend for a new config use case must also develop a corresponding mobile SDK. Now, different config backends are all supported by a single mobile SDK provided by MobileConfig. This SDK exposes a uniform config API and data schema that are agnostic to different config backends and use cases. On the server side, we use a translation layer to dynamically map mobile config parameters to different backends (§4.1.2).

2.2 Statistics of Mobile Environments

Next, we report some statistics to motivate the problem.

Very old app versions. Users' infrequent app updates necessitate the reliance on remote configs to change app behavior. Table 2 shows the cumulative age distribution of our largest Android app. The (56, 14%) column means that 14% of the app's installations were released in the past 56 days. Notably, some extremely old app versions remain in active use. For instance, 1% of the app's installations are older than 987 days, with the oldest one dating back almost 7 years (2,499 days).

App age (days)	49	56	63	70	77	217	504	987	2,499
Cumulative distribution	1%	14%	34%	56%	73%	90%	95%	99%	100%

Table 2: Very old versions of our Android app are still in use.

Very old OS versions. Our apps run on thousands of different Android device types, with OS versions spanning over a decade (Table 3). This complex environment makes it hard to prevent config errors just by development-time testing. Hence, MobileConfig relies on large-scale multi-stage canary tests in live production as the last line of defense (§5).

Android Version	4	5	6	7	8	9	10	11	12
Release year	2011	2014	2015	2016	2017	2018	2019	2020	2021
Percent of devices	0.1%	1.2%	2.3%	2.8%	9%	12%	26%	38%	8%

Table 3: Very old versions of Android are still in active use.

Low-end mobile devices. The majority of our app users are on low-end devices. We categorized the Android devices on

which our apps run based on the *year-class* metric, which corresponds to the year when the device would have been considered as a flagship device. For example, Samsung Galaxy S6 was released in 2015 as a flagship smartphone. If a low-end device was released in 2023 but its performance is comparable to Galaxy S6, it would be classified as the 2015 class. Table 4 shows that 75% of the devices are comparable to 2015 or older flagship devices. The combination of low-end devices and many configs require us to heavily optimize MobileConfig for performance and efficiency (§6).

Year Class	2010	2011	2012	2013	2014	2015	2016-2023
Percentage	0.04%	0.32%	3.53%	18%	20%	33%	25%

Table 4: Breakdown of Android devices by Year Class.

Many configs. We measured the number of config parameters in our most popular apps, MM and VH (Table 5). They are available on both Android (MM_a, VH_a) and iOS (MM_i, VH_i). The largest app uses more than 4,300 configs and 26,000 parameters. The existence of a large number of independent configs has made progressive consistency possible.

App Name	MM_a	MM_i	VH_a	VH_i
# Configs	4,344	3,546	3,178	3,050
# Parameters	26,770	18,057	8,563	7,828

Table 5: Config usage for our most popular apps.

Frequent config changes. Configs are updated frequently by many authors. The configs of our largest app are edited by more than 3,000 different authors over the app's lifetime. On an average workday, our developers make more than 2,700 config parameter value changes in production, and introduce more than 110 config schema changes. MobileConfig relies on defense in depth to mitigate the risk of config errors introduced by frequent config changes (§5).

Many languages. To show the usage of languages in our apps, we counted the number of static call sites in each language's source code that read remote configs (Table 6). As expected, Java for Android and Objective-C for iOS are most popular, but call sites in other languages still account for 37%. To support multiple languages and OSes while avoiding the development costs of multiple client SDKs, we have implemented MobileConfig's core functions in a portable cross-platform C++ runtime and exposed them to different languages through language-specific bindings (§4.1.1).

Language	Java	Objective-C	Java Script	Kotlin	React Native	Swift	C++
Call site%	35%	28%	18%	11%	6%	0.6%	0.6%

Table 6: Usage of MobileConfig by different languages.

Parameter types. Table 7 shows the breakdown of config parameter types for MM_a and MM_i. As Boolean dominates, MobileConfig implements special optimizations for it (§6.3).

	Boolean	Integer	String	Double	Other
MM_a	69%	23%	4.9%	2.4%	1.3%
MM_i	66%	22%	4.8%	5.1%	1.5%

Table 7: Breakdown of config parameter types.

3 Agile Development with MobileConfig

This section describes how developers use MobileConfig in an agile development process and how it ensures app correctness while enabling both fast app startup and fresh configs.

3.1 Agile Development Process

To illustrate how MobileConfig enables agile development, we describe the workflow of a developer named Alice working on the experimental "*Button*" feature shown in Figure 1.

After sufficient local testing, Alice includes the code for the Button feature in the app's new release and uploads it to the app store. Although some users quickly install the new release, the new feature is not yet exposed to anyone, as `ButtonCfg.isEnabled` is still set to false for all users.

Alice initiates testing for the new feature, initially enabling `ButtonCfg.isEnabled` for only 0.01% of users. Uncertain about the ideal `ButtonCfg.color` and `ButtonCfg.size` values for the best user experience, she sets up an A/B test and divides the test population into multiple groups, each receiving distinct parameter values. By comparing metrics, like user engagement, across the test groups, Alice identifies optimal parameter values. Importantly, the entire A/B test process is driven by Alice making config changes on the server side using a web interface. These config changes are automatically fetched by mobile devices, determining which users will participate in the A/B test and what parameter values they will get for `ButtonCfg`. As Alice makes remote config changes, it does not require participating users to upgrade their installed app or perform any manual operations.

If the app crashes on an old Android version when using the new feature, Alice remotely disables `ButtonCfg.isEnabled` for those users, without requiring an app upgrade. After fixing the bug, she releases a new version of the app to the app store. While still keeping `ButtonCfg.isEnabled` disabled for the affected users on the old version, Alice enables it for a subset of users on the new version to continue testing.

After months of A/B testing and numerous releases, Alice determines that the new feature consistently harms business metrics rather than enhancing them. She abandons the feature, removing the experimental code and releasing a new version to the app store. Notably, the majority of users were never exposed to the feature from its introduction to its removal.

3.2 Stale Configs Hinder Agile Development

While remote configs are supposed to enable agile development, straightforward solutions are ineffective. To highlight the challenges, we analyze the three solutions provided by Google's Firebase [16], which are summarized in Table 8.

	Fast app startup	Repeatable reads	Intra-config consistency	Fresh configs in normal state	Bound staleness if required	Purge misconfig quickly
Firebase (1)	✗	✓	✓	✓	✓	✗
Firebase (2)	✓	✗	✗	Vast majority	✗	✗
Firebase (3)	✓	✓	✓	✗	✗	✗
MobileConfig	✓	✓	✓	Vast majority	✓	✓

Table 8: Comparison of config frameworks. Firebase [16] supports three config consistency models [17].

In the table, Firebase (1) is Firebase's default approach as described in §1, which blocks app startup to fetch configs. This approach would unacceptably increase our largest app's startup time by 2,499 ms.

Firebase (2) uses cached configs to boot the app while asynchronously fetching the latest configs from the server. Once the new configs arrive, the app immediately switches to using them, without ensuring repeatable reads [7] or intra-config consistency. This approach is unacceptable as altering an app's configs during its live execution can result in user-visible anomalies, such as a sudden change in a button's color.

Firebase (3) also uses cached configs to boot the app and asynchronously fetches the latest configs from the server. However, when the new configs arrive, the app saves them to storage and will not utilize them until the next cold start of the app. This approach increases the staleness of configs and slows down the agile development process described in §3.1. Suppose Alice updates an A/B test parameter in the morning with the intention of collecting and analyzing the testing results in the afternoon; she faces a challenge. Despite many user devices fetching the updated parameter in the morning, they will not apply it until the next cold app restart, which may not occur soon. Even if the user switches out of the app, it may remain paused in the background for hours or even days without termination, preventing the new parameter from taking effect when the app is brought to the foreground again. Consequently, Alice cannot gather enough A/B test results in the afternoon or even the following day, significantly hindering agile development.

At Meta, for mobile apps, config commits occur about 15 times more frequently than code commits. This indicates a highly iterative development process where developers invest a significant portion of their time making config changes, collecting and analyzing test results to inform their next steps. Delays in collecting test results caused by stale configs would thus greatly hinder developer productivity.

Rather than delaying the use of new configs until the next cold restart, a potential improvement for Firebase (3) is to apply new configs when the user switches to other apps and puts the app in the background. However, this may still lead to user-visible anomalies. For instance, if a user is midway through reading a news article in the app and switches to other apps, altering the app's configs in the background could lead to the article's text appearing in a different color upon the user's return, as the color is controlled by a config.

Between Firebase (1)'s drawback of slow app startup and Firebase (3)'s drawback of config staleness, Firebase considers the latter to be a bigger problem and hence *"very strongly recommends"* Firebase (1) for apps using A/B testing [17]. In the literature, while the importance of fast app startup is widely recognized [4, 19, 27, 30, 37, 41], the importance of config freshness is often overlooked. Firebase's recognition of this issue based on its experiences with mobile developers is commendable.

3.3 Progressive Consistency

To enable agile development, progressive consistency solves both the problems of slow app startup and stale configs. During an app's startup, it uses cached configs to unblock the app while asynchronously fetching updates for all configs from the server. Once the updates arrive, the app switches to using the new values for the configs that it has not read yet but will stick to the old values for the configs that it has already read.

Our production data shows that progressive consistency achieves 99.7% config data freshness across billions of devices without slowing down app startup. A key reason for the high config freshness is that most configs are not accessed during an app's early startup phase. To aggressively minimize startup time, any code that is not absolutely necessary during startup, such as the initialization of app features that will not be shown on the first user interaction screen, is postponed to later stages. Since the vast majority of app features will not be initialized on the startup path, the configs used by those features will not be accessed during startup either.

Below, we describe how progressive consistency ensures app correctness and discuss the ease of use of its API.

App correctness. Despite mixed consumption of old and new configs, progressive consistency ensures app correctness by guaranteeing intra-config consistency, monotonic reads [40] across app restarts, and repeatable reads [7] within a user session. A session is the time duration between two cold restarts of the app, or it ends early if the user explicitly logs out of the app. A key difference between MobileConfig and Firebase (2), as summarized in Table 8, lies in MobileConfig's support for intra-config consistency and repeatable reads. If the app consumes a parameter value during a session, subsequent reads of that parameter will retrieve the same value, even if it has already been updated on the server. This prevents unexpected app behavior, such as sudden UI button color changes due to parameter updates. Moreover, MobileConfig guarantees intra-config consistency—if the app reads two parameters of the same config in one user session, such as `ButtonCfg.color` and `ButtonCfg.size`, those parameters' values always come from a single atomic update on the server side.

MobileConfig does not guarantee cross-config consistency as configs are intended to be independent, serving different code modules. If dependencies arise among configs, they should be merged into a single config. While merging could

theoretically result in an excessively large config, this has never occurred in the nine years of MobileConfig's production usage. On average, a config contains only 4.3 parameters. Although we have a design for MobileConfig to support cross-config consistency by maintaining metadata about matching versions of interdependent configs, it remains unimplemented due to the absence of a genuine need for such complexity.

Moreover, the lack of support for cross-config consistency is not merely a workaround to expedite startup in mobile environments but extends to datacenter (DC) environments as well, where startup time is not an important consideration. In contrast to MobileConfig's progressive consistency, our DC configuration management system, Configerator [38], adopts Firebase (1)'s approach, blocking a DC application's startup to fetch configs synchronously, as startup time is less important in DCs. However, Configerator still does not guarantee cross-config consistency. It may push real-time updates for different configs to a DC application's running instances in different orders, as configs are independent. Finally, despite config errors being a primary cause of production outages [38], we do not recall that either Configerator or MobileConfig, with 12 and 9 years of production usage respectively, has experienced outages due to the lack of cross-config consistency.

API simplicity. While implementing progressive consistency, ensuring advanced features do not complicate the developer API is a challenge. A transaction-like API to ensure config consistency seems straightforward but could hinder usability and adoption. Instead, MobileConfig offers a straightforward API to apps (see Figure 1) while handling advanced features internally. For example, when the app calls `MobileConfig.getString(ButtonCfg.color)`, MobileConfig must detect if `ButtonCfg.isEnabled` was already read and return the corresponding version of `ButtonCfg.color` for intra-config consistency. This streamlined API has successfully facilitated the migration of about a dozen legacy config frameworks at Meta to MobileConfig.

4 MobileConfig Design

In this section, we describe the design of MobileConfig and compare it to Firebase [16] and Configerator [38].

4.1 MobileConfig Architecture

MobileConfig's architecture is depicted in Figure 2, which encompasses the client library and the server-side components.

4.1.1 Client-side Library

MobileConfig's client-side runtime library (*runtime* for short) needs to support various platforms and languages. To avoid the development costs of multiple client SDKs, the runtime is implemented in portable cross-platform C++. The API layer exposes the C++ runtime to multiple languages.

The Java API uses JNI to bridge to C++, incurring higher

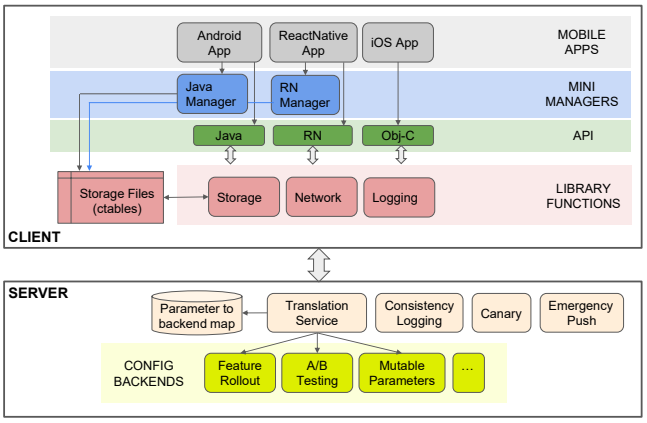


Figure 2: MobileConfig high-level architecture.

overhead. Due to the aggressive optimizations described in §6.2, the C++ API only needs two memory accesses to read a parameter value, which makes the JNI overhead prohibitively high in comparison, especially because config reads can be on the critical path of app execution. Reimplementing the entire runtime in Java would alleviate the problem but at the expense of additional development costs. To strike a balance between code reuse and performance, MobileConfig introduces a *Mini Manager* that implements a minimal read path in Java to efficiently read cached configs. This read path is on the critical path of app performance. All other functions that are not on the critical path, such as fetching configs from the server and updating the cache, are bridged through JNI to C++.

The support for other languages is simpler. The Objective-C API natively interfaces with C++. As ReactNative can cross-compile JavaScript to either Java or Objective-C, MobileConfig offers a JavaScript API that bridges JavaScript functions to either Java or Objective-C functions.

The runtime provides *networking*, *storage*, *exposure logging* and other functions. For *networking*, it efficiently synchronizes config values between client and server (§6.3). For *storage*, it stores configs in a highly optimized read-only binary format for fast access (§6.2). For *exposure logging*, the events of an app's exposure to config parameter values are logged, queued locally, and at opportune moments sent to the server in batches. Finally, the runtime supports operations related to config testing, debugging, canary, monitoring, and quick rectification of erroneous configs (§5).

4.1.2 Server-side Components

On the server side, different backend services support diverse config use cases, such as A/B testing and feature rollout. The *translation service* consults the *parameter-to-backend map* stored in a key-value store to map each config parameter requested by the client to its corresponding backend. This server-side translation drastically simplifies the client runtime, as the client can use a uniform config API and data schema that are agnostic to different config backends and use cases.

The translation service provides an extension point that allows developers to easily add a new custom-function backend, by implementing the following interface: "*generateParamValue (requestContext, userID)*." Based on the *userID* and *requestContext*, a custom backend can personalize parameter values. More than 700 custom backends have been implemented, mostly for different kinds of personalization.

MobileConfig also encompasses tools for detecting, preventing and rectifying config errors. *Consistency logging* samples client and server parameter values to detect config divergence. To prevent config errors, MobileConfig performs multi-stage canary tests when rolling out a config parameter change. Finally, MobileConfig's *emergency push* mechanism can quickly push a corrected config to billions of devices.

4.1.3 Client-server Protocol

Push vs. pull is an important design decision in the client-server protocol. With a push model, the server immediately pushes new data to clients. With a pull model, the client periodically pulls new data from the server in batches.

Table 9 compares push and pull for different systems. Configerator [38], our configuration system for datacenter applications, uses a push model (column 2 in Table 9). Our mobile messaging app uses a push model to deliver chat messages (column 3 in Table 9). No known system uses a push model for mobile configs (column 4 in Table 9). Firebase RemoteConfig [16] uses a pull model for mobile configs (column 5 in Table 9). MobileConfig (column 6 in Table 9) uses a pull model complemented by emergency push, which only happens several times per year to rectify severe config errors.

In datacenter environments, many config use cases require instant delivery for real-time automation, like global load balancing [32]. In contrast, mobile config use cases rarely need instant updates of configs because apps often do not consume them immediately. Apps may not be running or prefer to stick with the old config until the next restart to prevent abrupt user experience changes, such as UI alterations.

To minimize resource consumption, we opt for a pull model. However, in emergency situations such as rectifying config errors, it is necessary to push a config update to mobile devices quickly and restart the app to consume it. Hence, we complement our pull model with occasional emergency push.

		Config Mgmt in Datacenters (push)	Mobile Messaging App (push)	Config mgmt in mobile environ.		
				Push (no real use)	Pull (Firebase)	Pull + Emergency Push (MobileConfig)
# Endpoints		millions	billions	billions		
Endpoint availability		high	low	low		
Message fanout		millions	<100	billions		
Message rate		high	low	medium		
Push notif. reliability		high	medium	medium		
Hardware resources		abundant	scarce	scarce		
Outcome: infra cost relative to requirements		low	low	high ✗	low	low
Reliability	Requirement	high	medium	high		
	Outcome	high	medium	medium ✗	high	high
Delivery Speed	Requirement	fast	fast	usually slow but fast in emergency		
	Outcome	fast	fast	fast	always slow ✗	usually slow but fast in emergency

Table 9: Comparison of push vs. pull.

4.2 Support for Progressive Consistency

In this section, we present how progressive consistency works during the app lifecycle, and describe the implementation of repeatable reads and intra-config consistency.

4.2.1 Config Consistency during App Lifecycle

When a user installs a new app and logs in for the first time, MobileConfig blocks app startup to fetch the latest config values from the server. This is done to ensure a better user experience than using up to 26,000 unoptimized default parameter values. It is important to note that this blocking does not occur on subsequent app restarts. Moreover, MobileConfig can use *partial fetch* (§6.3) to retrieve only the essential configs and reduce blocking time. If the initial synchronous config fetch fails or times out (e.g., due to no network connectivity), the app proceeds with startup using the default parameter values statically compiled into the app's executable.

During the app's steady-state execution, whenever it enters the foreground, it checks whether H hours have passed since the previous config fetch. If so, it asynchronously fetches configs from the server, and caches configs on disk. We empirically found that H=4 hours strike a good balance between resource consumption and config freshness.

When the app cold-starts next time, it `mmap()` cached binary-format configs into memory, and can immediately read individual config parameters without loading or parsing configs (§6.2). While the app boots normally with cached configs, it immediately issues an asynchronous config-fetch request to the server. Suppose the response comes back with new config values in S seconds (e.g., S=2). The app's future reads to configs that are not read yet will consume the newly fetched config values. Overall, during the first S seconds of app startup, it consumes cached configs; after that, it consumes newly fetched configs. This approach strikes a balance between app start time and config freshness.

To bound config staleness, an app can block its startup to fetch configs if and only if the cached configs are fetched more than T hours ago, where T is configurable per app. If T=48 hours, a person who uses the app daily would never experience the blocking. If T=0, the app always blocks on startup, which is the Firebase-recommended approach. By default, our apps use T=∞ (i.e., never block) because 1) it already provides our apps with the latest version of the vast majority of configs, and 2) our apps can quickly recover from harmful stale configs via emergency push, and hence there is no need to be overly conservative.

After an app executable upgrade (not config update), the app consumes cached configs just like a normal cold start.

Some new config parameters might be added to the app's new version and hence do not exist in the local cache. The app can boot using those parameters' default values until the first asynchronous config fetch finishes. Alternatively, the app can block on a *partial fetch* (§6.3) to retrieve a minimal subset of those missing parameters that are important to the app startup.

4.2.2 Repeatable Reads and Intra-config Consistency

Progressive consistency guarantees that 1) during a user session, an app's multiple reads to the same parameter always return the same value, and 2) parameters in the same config are consistent with each other when consumed by the app.

Once every H hours, the MobileConfig runtime fetches the latest config values from the server, and saves configs in a so-called `ctable` file, resulting in a sequence of `ctables` over time: `0.ctable`, `1.ctable`, `2.ctable`, and so on. Old versions are garbage collected when they are no longer used.

On app startup, suppose `0.ctable` exists and it contains the configs in Figure 1. When the app reads `ButtonCfg.color`, the runtime will retrieve it from `0.ctable`. Later, after the runtime fetches new configs and creates a new version, `1.ctable`, the app reads `ButtonCfg.color` again and the runtime will still retrieve it from `0.ctable` (instead of `1.ctable`) in order to ensure repeatable reads. Later, when the app reads parameter `ButtonCfg.size`, the runtime will also retrieve it from `0.ctable` in order to ensure that the app reads `ButtonCfg`'s parameters from the same version.

Later, the app reads `MusicCfg.volume` and the runtime will retrieve it from `1.ctable` since that is the latest cached version. Note that the app reads `ButtonCfg` and `MusicCfg` from different versions of `ctables`. Treating each config independently allows the app to consume each config's latest cached version on its first access to the config.

5 Config Reliability

Multiple factors make it challenging to prevent config errors at scale. First, at Meta, hundreds of developers modify one app's code concurrently and release a new version every week. Second, they also make config changes in live production thousands of times per day. Finally, our apps run on billions of devices in the wild—various OSes and unmaintained old versions, plus thousands of wild Android device types. Hence, we have to rely on defense in depth for reliability.

MobileConfig automatically runs multi-stage canary tests on config changes. As soon as a config change is code-reviewed and accepted, MobileConfig tests the change in production by randomly selecting 0.5% of users as the canary group and another 0.5% as the control group. The canary test spans 30 minutes, during which the MobileConfig runtime uses exposure logging to report the app's consumption of the new parameter values. Once the canary time expires, our tool checks whether the canary parameters are associated with regressions in key metrics. If so, the change is reverted.

After the first-stage canary, MobileConfig initiates a more thorough second-stage canary to catch harder-to-detect config errors. It partitions the entire population into a 50% canary group and a 50% control group. This longer, four-hour canary primarily monitors key metrics such as app crashes.

The second-stage canary (50% of users) is much larger than the common practice [12]. For datacenter applications, we also do not run canary tests at this large scale. However, the mobile environment is diverse, with various device types and older OS versions. Consequently, bugs may only manifest on a small subset of devices, making them hard to detect in small-scale canary tests. Moreover, in a large app, only a fraction, or even a very small fraction, of users may use a specific feature during the canary window, potentially triggering a bug. Due to these challenges, MobileConfig employs unusually large and unusually long canary tests in the second stage.

While more intermediate stages could be added to MobileConfig's multi-stage canary, doing so would further extend the rollout time for config changes, already at 4.5 hours, significantly longer than in datacenter environments. Given the noisy mobile environments, it requires a longer canary time compared to datacenter environments. To reduce the rollout time, we opt for the minimum of two stages. The first stage promptly identifies obvious problems within a small user population, preventing widespread impact, while the second stage, with a large population, catches difficult-to-detect issues, serving as a robust last line of defense.

Emergency push. Severe config errors have to be fixed quickly by emergency push, as the normal process of updating a config can take a long time to reach most devices and even longer for apps to restart and consume the new config.

Emergency push works as follows. When a severe config error is either detected by multi-stage canary or manually reported, the developer can start an emergency push by specifying the target devices to be notified, e.g., based on device model, app version, country, user's spoken language, etc. The server maintains for each config an emergency version number (EVN), which is incremented whenever an emergency push happens to the config. The config's name and latest EVN are pushed to the target devices via our own implementation of an MQTT-based push notification mechanism.

The device compares the received EVN with its local EVN. If the former is higher, the device requests the latest parameter values and an action from the server. The action options are: *do nothing* (new values take effect after next app restart), *force refresh* (the next parameter read will consume the new value), *background restart* (the app will restart when switched to background), and *foreground restart* (the app restarts immediately, even if in foreground). *Foreground restart* is the most disruptive to users and is reserved for severe issues.

Config consistency checking. MobileConfig continuously monitors config consistency between clients and the server. The MobileConfig runtime periodically captures snapshots

of a device's local configs and transmits them to the server. The server retrieves parameter values from config backends, compares them to the client-reported values, and logs results in a database. Numerous health monitoring tools scan the database to detect issues promptly.

6 Performance Optimizations

In this section, we elaborate on several optimizations that allow MobileConfig to operate at scale efficiently.

6.1 Optimizing Strongly Typed Parameters

MobileConfig uses a strongly typed config API to prevent type errors. Given a config schema, our tool generates strongly typed parameter definitions for multiple languages. Apps use the strongly typed API to access parameters, for example, *MobileConfig.getString(ButtonCfg.color)*. By contrast, apps using Firebase [16] access parameters through *untyped* string identifiers, which can result in runtime type errors.

Unfortunately, the benefits of strong types come with a high cost. The symbols of thousands of Java config classes and parameters would inflate an Android app's binary size and slow down app startup. To mitigate this, our compilation tool transparently replaces all Java config class and parameter symbols with encoded integer IDs, which serve as indices to efficiently locate parameter values in `ctable` files at runtime.

6.2 Optimizing Config Storage

Config storage also affects the app startup time. Our evaluation shows that using Firebase [16]'s approach of storing configs in JSON files on Android would unacceptably prolong our largest app's cold start time by 558ms. Hence we heavily optimize config storage for high performance.

Config storage format. To enable fast config reads, MobileConfig uses flatbuffers [18] to encode hierarchical config data into an efficient byte-array representation. For each parameter type, it creates a byte array that contains two sections: 1) the values of parameters of the given type, and 2) metadata for each parameter, e.g., a `loggingEnabled` field indicating whether an exposure event should be logged. The per data-type byte arrays are then concatenated, with header and tail sections added. The header contains the offsets of the per-type subarrays. The tail contains the LoggingIDs for each parameter. Finally, the entire byte array is persisted on disk as the so-called `ctable` file.

Fast parameter read. On app startup, the MobileConfig runtime `mmap()` the `ctable` file so that the app can immediately read specific parameters without loading all configs into memory or parsing configs. On a parameter read operation, the runtime extracts from the encoded 64-bit parameter ID the following metadata: the config's rank among all configs, the parameter's rank, and the parameter's type. Then it uses them as indices to efficiently locate the parameter's value, metadata, and LoggingID in the `ctable` file.

6.3 Optimizing Client-Server Protocol

A straightforward implementation of the protocol described in §4.1.3 would be inefficient. This section describes the optimizations we have made to the baseline protocol.

Partial fetch. Usually, an app asynchronously fetches configs from the server without blocking app startup. However, there are two exceptions: the first login after app installation and the first startup after an app upgrade (§4.2.1). One important insight is that even if occasional synchronous config fetches are necessary, it is unnecessary to fetch all config in a single batch because most configs are not used in the first few seconds of app startup. Hence, it is likely that an asynchronous fetch can finish before those parameters are used.

To minimize the delay, MobileConfig uses *partial fetch*, which only retrieves the minimum subset of configs needed in the early phase of app startup. These parameters are identified through tests in our lab environment by tracing the parameters read during app startup until it is ready for user interaction.

Config schema hash. Apps of different versions use different config schemas and the server does not know the exact list of configs and parameters that a client wants to fetch. Since each app version is associated with a fixed set of configs and parameters, the server only needs to know the client's app version. At compilation time, our tool generates a SHA-256 hash of the list of configs and parameters, and stores it along with the app binary. At runtime, the client sends to the server the SHA-256 hash, and the server consults a key-value store to map the hash back to a list of configs and parameters.

Parameter value hash. Sometimes the server may wastefully send a parameter's latest value to the client while the client's cached version is already up-to-date. To avoid this overhead, the client partitions its configs into sets, and produces a hash for parameter values in each set. The client includes these hashes in its request to the server, which are used by the server to identify and skip unchanged config sets.

Boolean encoding. Most parameters are booleans (Table 7). We reduce the server response size by using two bits to represent a boolean (null/valid and true/false) and then concatenating all boolean bits into an efficient byte array.

7 Evaluation

Unless otherwise noted, all experiments described in this section are conducted in production at hyperscale.

7.1 Usage and Adoption

MobileConfig has been in production since 2015 and has become the only mobile config solution at Meta after consolidating about a dozen different legacy solutions. Currently, it manages tens of apps running on billions of devices. Table 10 summarizes the usage statistics for several apps.

The largest app, MM_a, uses more than 26,000 config pa-

App	MM_a	$MM_{a,L}$	MM_i	VH_a	VH_i	TH_a	TH_i	BA_a	BA_i	WP_a	WP_i
# Configs	4344	46	3546	3178	3050	2780	1177	4680	3254	4381	2864
# Parameters	26770	362	18057	8563	7828	17662	4793	28189	17223	27192	15885
# Projects	5797	92	2437	1607	1642	4650	645	6727	2644	6596	2385
# Teams	754	19	607	336	315	532	185	725	539	718	505
Config fetches per week (billion)	64	8	26	52	38	34	46	0.15	0.2	0.02	0.02

Table 10: Config usage statistics. *Projects* are makefile-like compilation targets. The subscripts a and i mean Android and iOS, respectively, e.g., MM_a and MM_i. The business apps (BA and WP) have less users than the other consumer apps.

rameters and is jointly developed by more than 700 teams. The smallest app, $MM_{a,L}$, is a lightweight version of the MM app, optimized for minimal resource consumption on low-end Android devices. $MM_{a,L}$ uses 362 config parameters, which is about two orders of magnitude smaller than that of MM_a.

Overall, Table 10 shows that MobileConfig is capable of supporting both small and large apps. Moreover, the large number of teams and projects in Table 10 highlights that the agile development process (§3.1) enabled by remote configs can scale to many people jointly working on one large app.

7.2 Impact on App Startup Time

Both our apps and MobileConfig are aggressively optimized for fast startup because it directly impacts user engagement.

7.2.1 Consistency Model's Impact on App Startup Time

Figure 3 shows the average config-fetch time measured in production for different apps. Blocking app startup while fetching configs, as Firebase does [17], would significantly prolong the startup times of our apps by anywhere from 1091ms to 2866ms, causing a detrimental impact on user engagement. To put this into perspective, our app performance team often invests months in optimizing an app to reduce its start time by just tens of milliseconds. The config-fetch time is longer than the network round-trip time because it includes time for the config backends to generate personalized values for as many as 27,000 config parameters. Overall, the long config-fetch times emphasize the importance of progressive consistency's approach of using cached configs to unblock app startup.

MobileConfig performs synchronous config fetches in two cases: the first login after app installation and the initial startup following an app upgrade. In these cases, it employs partial fetch (§6.3) to retrieve only the most important configs. For our popular app, MM_a, compared to a full fetch, partial fetch reduces fetch time from 2,499ms to 1,571ms and decreases transferred data from 401KB to 31KB. The smaller reduction in fetch time is due to the non-linear relationship between config size and config-fetch time. The fetch time includes computation time for the config backends to generate personalized values for many parameters in parallel, which does not decrease linearly as the config data size reduces.

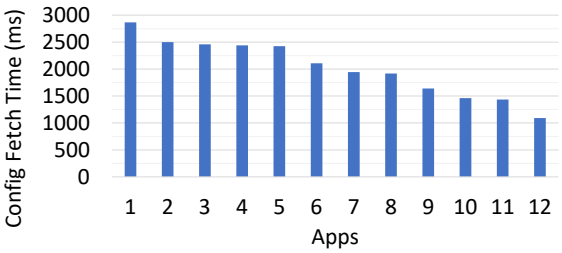


Figure 3: Config-fetch time for different apps.

7.2.2 Config Storage's Impact on App Startup Time

On app startup, the MobileConfig runtime directly `mmap()` a binary-format `ctable` file into memory and immediately read individual config parameters, without the delay of parsing configs or loading all configs into memory. By contrast, Firebase [16] on Android stores configs in JSON files. To do a direct comparison, we modified our VH_a app to use a similar approach to store configs in JSON, and we call it $VH_{a,json}$. On startup, both VH_a and $VH_{a,json}$ use cached configs, and hence the impact of config fetch is excluded from the comparison. We measured app startup time in production. On average, VH_a boots 558ms faster than $VH_{a,json}$. This significant win underscores the importance of optimizing storage format and access method. Unlike $VH_{a,json}$, which parses the JSON file to extract all parameters before accessing even a single one, MobileConfig utilizes `mmap()` to access configs in an optimized binary format. This enables it to selectively page in the specific page containing the needed parameter and read it directly, without being concerned about other parameters.

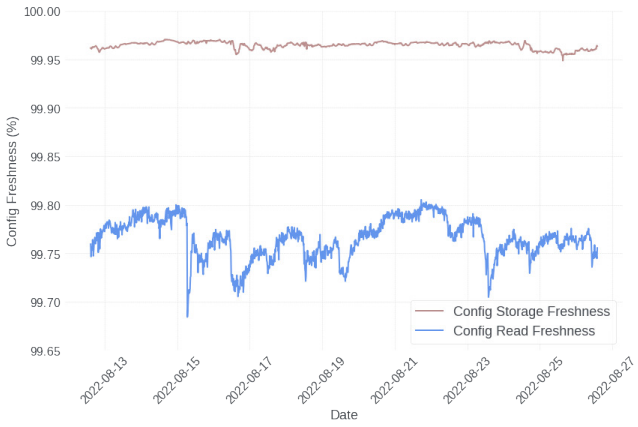


Figure 4: Config freshness for the MM_a and MM_i apps.

7.3 Config Data Freshness

Progressive consistency not only enables fast app startup but also provides apps with highly fresh configs. We use MobileConfig's *config consistency checker* (§5) to measure config freshness in production. Config *read freshness* measures whether a parameter value consumed by an app is identical to the parameter's value on the server side. Config *storage freshness* measures whether a parameter's value stored in a device's cache is identical to the parameter's value on the server side. Read freshness and storage freshness may diverge because even if a client has fetched and stored a parameter's latest value in cache, the app may keep using an older cached version in order to guarantee repeatable reads.

Figure 4 shows config freshness reported by billions of devices for our biggest app *MM*. The P50 values of config read freshness are 99.70% and 99.84% for MM_a and MM_i, respectively. These demonstrate that progressive consistency is able to provide apps with highly fresh configs while enabling fast app startup.

The config storage freshness is close to 100% but never reaches 100% due to how it is calculated. A config storage sample is taken at time T_0 right after a MobileConfig client fetches configs from the server. Later, the sample is compared with the config data on the server side at time T_1. Between T_0 and T_1, some parameter values might have changed on the server side, causing the sample to be considered "not fresh."

7.4 Emergency Push

Emergency push (§5) accelerates the process of purging harmful stale config data from app devices to rectify config errors. We designed an experiment to measure in production how quickly an update on a specific parameter is disseminated and consumed by app devices. We compare different setups: 1) *baseline*—no use of emergency push, 2) *EP w/o restart*—emergency push without app restart, 3) *EP w/ restart*—emergency push plus forced app foreground restart.

The results are shown in Figure 5. The *x*-axis is the time since the parameter value is updated. The *freshness coverage* metric on the *y* axis measures the percentage of parameter reads that return the new parameter value out of all app devices' reads to the parameter. The app is set up to read the specific parameter immediately after it boots. Freshness coverage is calculated in a 30-minute moving time window.

By design, "*EP w/ restart*" guarantees near 100% freshness coverage. If a device's app is running at time 0 when the parameter value is updated, soon the app will receive the new parameter value via emergency push and then immediately restart to consume it. If the app is not running at time 0, when it is opened later, it will boot with the cached old parameter value, but will quickly receive the asynchronously fetched new configs and notice that an emergency push has happened. It will immediately restart the app to consume the new parameter value. If we exclude the old parameter value temporarily consumed by the app during the very short period of time between the app's first startup and its immediate restart, by design "*EP w/ restart*" guarantees near 100% freshness coverage. This is shown as the top curve in Figure 5. Note that in this experiment, the top curve is inferred instead of measured in production because we cannot afford to force-restart the app in production just for an experiment, which would cause a disruptive experience to many real users. Local tests on our devices confirm that "*EP w/ restart*" indeed restarts the app in seconds to consume the new parameter value.

The "*baseline*" curve in Figure 5 shows that without emergency push, it takes a long time for freshness coverage to reach a high value. Specifically, after 4 hours, the coverage reaches 26%; after 24 hours, it reaches 85%. The long tail is caused by users who do not use the app for a long time.

"*EP w/o restart*" improves freshness coverage. Specifically, after 4 hours, the coverage reaches 40% (vs. 26% in "*baseline*"); after 24 hours, it reaches 92% (vs. 85% in "*baseline*"). However, the wide gap in freshness coverage between "*EP w/o restart*" and the ideal setup of "*EP w/ restart*" shows that it is insufficient to just quickly push the new parameter value to devices because the app will not consume it until the app's next restart. Therefore, forced app restart, though disruptive, is a necessary step to quickly purge stale configs.

In production, EP is used approximately once per quarter to rectify config errors and almost all those cases use "*EP w/ restart*." In addition, "*EP w/o restart*" is frequently used for Defcon drills [25], as explained in §8.1, for disaster readiness drills rather than handling real production outages.

7.5 Multi-stage Canary Tests

Multi-stage canary tests help catch code or config bugs early. Over a one-month period, MobileConfig conducted 81,014 canaries and caught 15 bugs, all in the second-stage canary. All these bugs slipped through the small-scale first-stage canary, because the regression in app health metrics was too subtle to be reliably detected in very noisy mobile environments. This highlights the difficulty of config error prevention in mobile environments. Despite a large body of research on config error prevention [9, 14, 24, 29, 44, 46, 47, 49], we found that large-scale canary tests in production are still the most robust and widely applicable method.

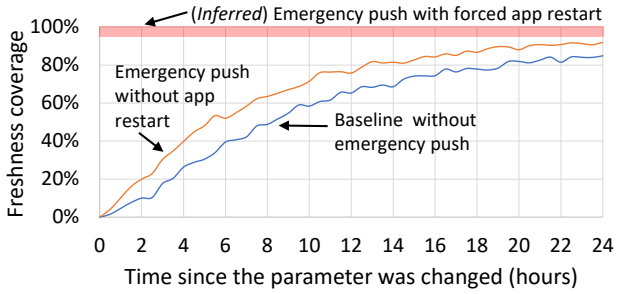


Figure 5: Impact of emergency push on freshness coverage.

App	All OFF		QH OFF		VH OFF		BE OFF		VH+BE OFF		All ON (production version)			Boolean Count (%)
	Request (KB)	Response (KB)	Request (KB)	Response (KB)	Request (KB)	Response (KB)	Request (KB)	Response (KB)	Request (KB)	Response (KB)	Request (KB)	Response (KB)	Total savings (%)	
MM_a	148	880	181	6.4	0.85	698	33	802	0.85	880	33	5.7	96	70
WP_a	145	808	175	6.3	0.82	642	31	738	0.82	808	31	6.7	96	69
MM_i	102	473	125	3.4	0.74	354	24	533	0.75	476	24	3.9	95	66
WP_i	84	374	101	1.5	0.64	287	19	417	0.64	374	19	4.7	95	65

Table 11: Impact of client-server protocol optimizations: Value Hashing (VH), Boolean Encodings (BE) and Query Hashing (QH).

We describe two prevented bugs below to give some intuition on why the second-stage canary caught the bugs but not the first-stage canary. In the first example, a developer used remote config to drive an A/B test that targeted a very small population *P* of Android users. Because MobileConfig's first-stage canary only samples 0.5% of those *P* users, i.e., an even smaller population, it did not catch any problem. The second-stage canary was conducted on 100x more users and identified about 3,000 app crashes that only happened to users exposed to the A/B test. It turned out that a bad config parameter value used for one of the A/B test groups caused `IndexOutOfBoundsException`. The spike and recovery of the app crash is shown in Figure 6.

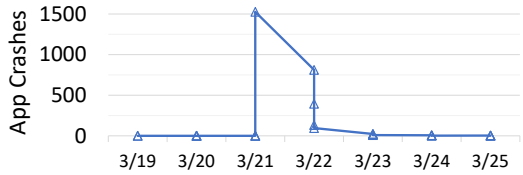


Figure 6: A canary test caught app crashes.

In the second example, a developer set up an A/B test to enable a new code path in an Android app, which increased memory consumption. Because the increase in memory consumption was quite moderate, it caused out-of-memory crashes on only a small fraction of very low-end devices with limited memory. The second-stage canary caught this subtle problem and prevented a silent regression in memory consumption.

7.6 Client-server Protocol Optimization

§6.3 describes MobileConfig's various network optimizations: query hashing (QH), value hashing (VH), and boolean encoding (BE). To evaluate their effectiveness, we measured the request and response size when fetching all configs for several apps under different settings. The results are summarized in Table 11, where *"All ON"* means enabling all optimizations, which is the setting used in production. Other settings mean disabling certain optimizations from the production baseline. These experiments are performed in a local testbed because disabling optimizations in production would cause a poor experience to real users.

Compared with *"All OFF"*, *"All ON"* reduces the total size of request and response by ≈96%. Value hashing and boolean encoding reduce the response size by about two orders of magnitude, while query hashing reduces the request size by ≈80%. The effectiveness of these optimizations is an important reason why we prefer the simple soft-state client-server protocol over a more complex hard-state protocol.

8 Operational Experiences

We use several production incidents to highlight the challenges and then share the lessons learned.

8.1 Production Incidents

Incident 1: lack of emergency push (EP). In 2016, a developer mistakenly changed the config parameter that controls the maximum number of comments to fetch in the *MM* app from 25 to 0, causing users to see no comments. The parameter value was quickly fixed on the server side, but as EP was not implemented in MobileConfig at the time, we had to painfully wait for over a day for most devices to gradually restart the app and apply the corrected parameter value. This incident expedited our development of EP.

Incident 2: EP malfunction. Early on, we observed that some devices had corrupted storage, leading to incorrect appearances of many config files. Initially, each use of EP fixed a single broken config parameter, and it was never the case that many configs needed simultaneous correction through EP. Given this, we adjusted the MobileConfig runtime to treat the situation of too many EP-delivered configs on storage as a sign of storage corruption, causing it to revert to default values for all configs. This precaution served us well for years until we began conducting large-scale Defcon drills [25]. These drills involved using EP to instruct apps to rapidly disable less-essential features to reduce the load on backend systems. In 2021, as Defcon drills expanded to disable more app features simultaneously, the number of configs pushed out by EP exceeded a threshold, triggering the MobileConfig runtime to mistakenly conclude that the storage was corrupted on many devices and fall back to default values for all configs. This incident demonstrates that as the operating environment shifts over time, an initial defensive mechanism can unexpectedly transform into a destructive force. Therefore, a robust solution needs to stand the test of time.

Incident 3: mishandling of file names. We once noticed network connection anomalies from about 0.1% of users due

to MobileConfig randomly flipping some config parameters. As the problem was not reproducible in our lab, initially, we made no progress after 27 days of laborious investigation. Eventually, a breakthrough occurred when the problem manifested on an employee's device, allowing us to directly attach a debugger. It turned out that when the app starts, if the latest config file's sequence number happens to end with 0, such as `10.ctable` or `20.ctable`, the MobileConfig runtime code, *"if (latestConfigFile.endsWith("0.ctable")),"* mistakenly treated it as `0.ctable`. The fix was simple—just changing the code to *"if (latestConfigFile.endsWith("/0.ctable"))"*—yet the investigation process was extremely difficult. This incident underscores the difficulty of developing low-level mobile systems like MobileConfig, exacerbated by the lack of direct access to user devices.

Incident 4: configs on VR devices. On our VR products, since we own the operating system, we run the MobileConfig runtime in a daemon to manage configs for all apps, which is more efficient than each app managing its own configs. When an app starts, it subscribes to the daemon for certain configs. While the solution overall worked well, it was reported that the daemon occasionally returned incorrect parameter values. As the problem was not reproducible in our lab, much of the investigation involved reading source code, changing code through trial and error, and waiting for logging data from user devices for confirmation. The whole process lasted 60 days. Eventually, it was discovered that when two apps concurrently subscribe to two sets of overlapping configs, the ordering of configs on storage depends on the timing of the subscription calls. This ordering issue was overlooked in some cases, resulting in retrieving wrong parameter values. This incident, once again, underscores the difficulties of developing low-level mobile systems.

8.2 Lessons Learned

We draw several lessons from our experience above.

- Following from Incident 1, we recommend every config framework to support emergency push. Although it is not supported by existing solutions in the public domain [1, 16, 28, 42] and the dozen legacy config frameworks predating MobileConfig at Meta, we found it important for enabling a safe and agile development process.
- Incidents 2, 3, and 4 all demonstrate the difficulty of developing a robust and feature-rich mobile config framework. Therefore, it should be done only once and then reused across all platforms, programming languages, and config use cases. This principle drives our design of the single, universal SDK, as opposed to Firebase's approach of using different implementations for different platforms.
- Incidents 3 and 4 also demonstrate that debugging subtle issues in the wild for low-level mobile systems like MobileConfig is very difficult due to a lack of access to user devices. Over time, we have enhanced consistency checking to detect various issues early (§5) and also upload snapshots of devices' config files to help us more easily reproduce problems.
- Finally, as shown in §7.5, traditional small-scale canary tests fall short for mobile apps due to noisy mobile environments. Consequently, MobileConfig employs unusually large and unusually long canary tests.

9 Related Work

Configuration management for mobile apps. Out of the few existing mobile config systems [1, 16, 28, 42], Google's Firebase RemoteConfig [16] is the closest to MobileConfig. A detailed comparison is shown in Table 8.

Configuration management for datacenter applications. Past studies on configuration management [10, 33–36, 38, 45, 50] have been mostly focused on datacenter applications. Configerator [38] is a representative system in the industry and a comparison is shown in Table 9.

Configuration error prevention. A large body of work studies misconfiguration [6, 9, 14, 24, 29, 43, 44, 46, 47, 49]. Like Configerator [38], MobileConfig primarily uses large-scale canary tests in production to prevent misconfiguration due to its robustness in complex environments, but MobileConfig has to address additional challenges in mobile environments such as ineffectiveness of small-scale canaries.

Consistency models. Out of many consistency models [2, 3, 23, 26, 31, 48], TACT [48] is most related to MobileConfig's progressive consistency. Both MobileConfig and TACT allow consumption of stale data and can bound the level of staleness. Config consumption on mobile devices can be viewed as read-only transactions, but neither related database work [8, 13] nor config consumption in datacenters [38] is optimized for accessing up-to-date data without blocking on remote reads.

10 Conclusion

We presented MobileConfig, a configuration management framework that manages tens of mobile apps on billions of devices. Its progressive consistency balances fast app startup with fresh config data. To prevent config errors, it uses a defense-in-depth approach that employs multi-stage canary tests at scale, compile-time validation, and config consistency checking. Its novel use of cross-platform code and server-side config parameter remapping prevents the proliferation of mobile SDKs while supporting diverse platforms, programming languages, and config use cases. Additionally, we reported our lessons learned from operating MobileConfig at hyperscale. Finally, we hope that our experience—for example, our developers making thousands of config changes daily in live production—will inspire others to also employ remote configs to push the limits of agile app development.

References

[1] Adobe Target: A/B Test, Personalize & Automate, 2024. https://business.adobe.com/products/target/adobe-target.html.

[2] Sarita V Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[3] Atul Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1999.

[4] App startup time, 2024. https://developer.android.com/topic/performance/vitals/launch-time.

[5] AppDynamics. Mobile app performance explained, 2014. https://www.appdynamics.com/media/uploaded-files/mobileapp.pdf.

[6] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the Ninth USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[7] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[8] Arvola Chan and Robert Gray. Implementing distributed read-only transactions. *IEEE Transactions on Software Engineering*, (2):205–212, 1985.

[9] Runxiang Cheng, Lingming Zhang, Darko Marinov, and Tianyin Xu. Test-case prioritization for configuration testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 452–465, 2021.

[10] Azure App Config, 2024. https://docs.microsoft.com/en-us/azure/azure-app-configuration/overview.

[11] Colin Contreary. Why should you care about your mobile app's startup time?, 2023. https://blog.embrace.io/why-should-you-care-about-your-mobile-apps-startup-time/.

[12] Feature toggle, 2024. https://en.wikipedia.org/wiki/Feature_toggle.

[13] Hector Garcia-Molina and Gio Wiederhold. Read-only transactions in a distributed database. *ACM Transactions on Database Systems (TODS)*, 7(2):209–234, 1982.

[14] Peng Huang, William J Bolosky, Abhishek Singh, and Yuanyuan Zhou. ConfValley: A systematic configuration validation framework for cloud services. In *Proceedings of the 10th European Conference on Computer Systems*, page 19, 2015.

[15] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[16] Google Inc. Firebase Remote Config, 2024. https://firebase.google.com/docs/remote-config.

[17] Google Inc. Firebase Remote Config Loading Strategies, 2024. https://firebase.google.com/docs/remote-config/loading.

[18] Google Inc. Flatbuffers, 2024. https://google.github.io/flatbuffers/.

[19] Tyler Kieft. Building a better Instagram app for Android, 2014. https://instagram-engineering.com/building-a-better-instagram-app-for-android-c08f973662b.

[20] Ron Kohavi, Alex Deng, Brian Frasca, Toby Walker, Ya Xu, and Nils Pohlmann. Online controlled experiments at large scale. In *Proceedings of the 19th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1168–1176, 2013.

[21] Ron Kohavi, Diane Tang, and Ya Xu. *Trustworthy online controlled experiments: A practical guide to A/B testing*. Cambridge University Press, 2020.

[22] Ronny Kohavi, Thomas Crook, Roger Longbotham, Brian Frasca, Randy Henne, Juan Lavista Ferres, and Tamir Melamed. Online experimentation at Microsoft. *Data Mining Case Studies*, 11(2009):39, 2009.

[23] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416, 2011.

[24] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding BGP misconfiguration. In *Proceedings of the ACM SIGCOMM 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 3–16, 2002.

[25] Justin J Meza, Thote Gowda, Ahmed Eid, Tomiwa Ijaware, Dmitry Chernyshev, Yi Yu, Md Nazim Uddin, Rohan Das, Chad Nachiappan, Sari Tran, et al. Defcon: Preventing Overload with Graceful Feature Degradation. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, pages 607–622, 2023.

[26] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 174–187, 2001.

[27] Mike Nakhimovich. Improving Startup Time in the NYTimes Android App, 2016. https://archive.nytimes.com/open.blogs.nytimes.com/2016/02/11/improving-startup-time-in-the-nytimes-android-app/.

[28] Optimizely, 2024. https://www.optimizely.com/.

[29] Vasileios Pappas, Zhiguo Xu, Songwu Lu, Daniel Massey, Andreas Terzis, and Lixia Zhang. Impact of configuration errors on DNS robustness. In *Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 319–330, 2004.

[30] Anshu Rustagi. How We Improved Our Android App "Cold Start" Time by 28%, 2018. https://redfin.engineering/how-we-improved-our-android-app-cold-start-time-by-28-a722e231314a.

[31] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys (CSUR)*, 37(1):42–81, 2005.

[32] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. ServiceRouter: a Scalable and Minimal Cost Service Mesh. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[33] Gerald Schermann, Jürgen Cito, and Philipp Leitner. Continuous experimentation: challenges, implementation techniques, and current research. *IEEE Software*, 35(2):26–31, 2018.

[34] Gerald Schermann, Dominik Schöni, Philipp Leitner, and Harald C Gall. Bifrost: Supporting continuous deployment with automated enactment of multi-phase live testing strategies. In *Proceedings of the 17th International Middleware Conference*, pages 1–14, 2016.

[35] Rian Shambaugh, Aaron Weiss, and Arjun Guha. Rehearsal: A configuration verification tool for puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 416–430, 2016.

[36] Alex Sherman, Philip A Lisiecki, Andy Berkheimer, and Joel Wein. ACMS: The Akamai Configuration Management System. In *Proceedings of the Second USENIX Symposium on Networked Systems Design and Implementation*, pages 245–258, 2005.

[37] Snap Inc. Measuring 'Time to Camera ready', 2021. https://eng.snap.com/time_to_camera_ready.

[38] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[39] Diane Tang, Ashish Agarwal, Deirdre O'Brien, and Mike Meyer. Overlapping experiment infrastructure: More, better, faster experimentation. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 17–26, 2010.

[40] Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149. IEEE, 1994.

[41] Natansh Verma. Optimizing Facebook for iOS start time, 2015. https://engineering.fb.com/2015/11/20/ios/optimizing-facebook-for-ios-start-time/.

[42] VWO, 2024. https://vwo.com/.

[43] Helen J Wang, John C Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, pages 245–257, 2004.

[44] Avishai Wool. A quantitative study of firewall configuration errors. *IEEE Computer*, 37(6):62–67, 2004.

[45] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early detection of configuration errors to reduce failure damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 619–634, 2016.

[46] Tianyin Xu and Yuanyuan Zhou. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Computing Surveys*, 47(4):70, 2015.

[47] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 159–172, 2011.

[48] Haifeng Yu. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, 2000.

[49] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. EnCore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th Architectural Support for Programming Languages and Operating Systems*, pages 687–700, 2014.

[50] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 338–350, 2017.

Passengers' Safety Matters: Experiences of Deploying a Large-Scale Indoor Delivery Monitoring System

Xiubin Fan[1], Zhongming Lin[2], Yuming Hu[3], Tianrui Jiang[2], Feng Qian[4],
Zhimeng Yin[1]*, S.-H. Gary Chan[2]*, Dapeng Wu[1]

[1]*City University of Hong Kong* [2]*The Hong Kong University of Science and Technology* [3]*University of Minnesota – Twin Cities*
[4]*University of Southern California*

Abstract

Delivering goods to many indoor stores poses significant safety issues, as heavy, high-stacked packages carried on delivery trolleys may fall and hurt passersby. This paper reports our experiences of developing and operating DeMo, a practical system for real-time monitoring of indoor delivery. DeMo attaches sensors to trolleys and analyzes the Inertial Measurement Unit (IMU) and Bluetooth Low Energy (BLE) readings to detect delivery violations such as speeding and using non-designated delivery paths. Differing from typical indoor localization applications, DeMo overcomes unique challenges such as unique sensor placement and complex electromagnetic characteristics underground. In particular, DeMo adapts the classical logarithmic radio signal model to support fingerprint-free localization, drastically lowering the deployment and maintenance cost. DeMo has been operating since May 2020, covering more than 200 shops with 42,248 deliveries (3521.4 km) across 12 subway stations in Hong Kong. DeMo's 3-year operation witnessed a significant violation rate drop, from 19% (May 2020) to 2.7% (Mar 2023).

1 Introduction

Indoor localization has been extensively studied by the research community [4, 5, 19, 39, 41, 43, 64, 66]. Very recently, there emerged commercial, large-scale indoor localization systems that leverage the two-decade research to benefit end customers (*e.g.,* mall navigation [30, 43] and presence detection [18, 19]) and even offer monetization opportunities [46].

In this paper, we investigate a unique and important application that falls into the broad topic of indoor localization: *indoor delivery monitoring*. Many public places such as airports, subway stations, malls, and office buildings feature dense retail stores and crowded visitors. Delivering goods to the stores (Fig. 1) poses significant safety issues, as heavy, high-stacked packages carried on delivery trolleys may fall and hurt passersby. The particular indoor environments may exacerbate such a risk: there are usually no corridors reserved exclusively for delivery; the uneven road surfaces such as slopes, tactile pavings, and contraction joints make packages less stable; delivery personnel may even use passenger lifts, whose acceleration and deceleration may make packages fall. Reckless indoor delivery has resulted in severe injury or even deaths [1, 14, 21, 45, 57, 62]. For example, in a recent accident, multiple packages fell and paralyzed a nearby passenger when a delivery worker was using a passenger lift to transport a trolley full of goods [45]. When indoor environments are crowded, delivery accidents may be more severe compared to their outdoor counterparts [6, 38], and they may trigger cascading accidents such as crowd collapse and even stampedes where the death toll can reach hundreds.

Fig. 1: Indoor delivery in an MTR station.

As an advocator of indoor delivery safety, Hong Kong Electrical and Mechanical Services Department (HK EMSD) has been strictly monitoring delivery violations in Mass Transit Railway (MTR) stations in HK. MTR has a daily ridership of more than 5 million (Feb 2023). Henceforth, its stations are dense with passengers, stores, and indoor deliveries. EMSD has established four delivery violations in MTR stations: (1) speeding (trolley moving speed $\geq$ 1.5 m/s), (2) using non-designated delivery paths, (3) using passenger lifts without prior permission, and (4) performing delivery in peak hours. To ensure the rules are being properly followed, since 2010, MTR has been hiring safety staff to manually monitor the delivery behavior. The staff needs to physically follow the delivery worker and manually document observed violations. Clearly, this approach is inaccurate (in particular for speed estimation), unscalable, and labor-intense.

To address the disadvantages of the manual efforts, we collaborated with HK EMSD and MTR on devising a fully automated indoor delivery monitoring solution, referred to as DeMo. In this paper, we report our multi-year experiences of developing, deploying, and maintaining DeMo. Our system was commercially deployed in 12 MTR stations for monitoring 40K+ deliveries to 200+ stores since May 2020.

At a first glance, it appears that we can trivially apply an existing indoor localization solution (Table 1) as is: by tracking the delivery worker's location, ideally one can find out the speed and path of the delivery in real time. However, we face several unique challenges and practical constraints that

*Zhimeng Yin and S.-H. Gary Chan are co-corresponding authors.

Table 1: Large-scale indoor operational systems. "B+G" means BLE and geomagnetic fields; "det." represents detection; "prop." is for propagation; "mon." means monitoring.

System	Technique	Signal	Application
MLoc [30]	Fingerprinting	B+G, IMU	Navigation
Tencent [43]	Fingerprinting	WiFi, IMU	Navigation
myCoex [27]	Fingerprinting	WiFi	Navigation
aBeacon [18]	Proximity det.	BLE	Presence det.
VALID [19]	Proximity det.	BLE	Presence det.
DeMo	Prop. model	BLE, IMU	Delivery mon.

render off-the-shelf indoor localization solutions not applicable. First, *the sensor placement is different*. Unlike prior solutions that assume users carry hand-held smartphones for localization, for indoor delivery, workers usually attach sensors to the trolley. Therefore, motion sensor readings do not exhibit periodical footstep-incurred patterns that are widely leveraged for online location tracking [34, 53]. Again due to the sensor placement, sensor readings are significantly disturbed when the trolley moves over special ground surfaces such as tactile pavings and contraction joints. Second, compared to prior localization systems' target environments (*e.g.,* shopping malls), HK's MTR stations are usually *underground and bear much more complex electromagnetic characteristics* due to operating trains, as demonstrated in Appendix A.1. Consequently, we cannot adopt commonly used features, *e.g.,* geomagnetic field (GMF) strength. Due to the same reason, we are not even able to obtain the accurate moving direction that is essential for position tracking. Third, due to *privacy concern and loud noises* in MTR stations, we cannot use vision-based or acoustic-based localization; due to *constraints of device form factor and energy usage*, we are not allowed to adopt WiFi-based localization either, which was heavily researched [13, 50, 65] and commercially deployed [27, 43]. Fourth, HK EMSD and MTR also hope to *minimize the preparation and maintenance overhead*. We thus decide to not use a fingerprint-based approach that was adopted by almost all prior commercial indoor navigation solutions [27, 30], because training and maintaining the fingerprint database requires significant labor in crowded MTR stations.

To tackle the above challenges, DeMo only uses cheap, lightweight Bluetooth Low Energy (BLE) beacons as the infrastructure. To further lower the deployment bar, instead of using BLE beacons' RSSI readings as location fingerprints, we adopt a simple RSSI-distance model as the core localization mechanism. The model is generic across all MTR stations; it thus eliminates the need for training and updating the per-site fingerprint database. While RSSI propagation modeling has been extensively studied [5, 7, 13, 31, 42], our contribution lies in *adapting the classical logarithmic model [42] to complex indoor environments, and for the first time, demonstrate its efficacy in supporting fingerprint-free localization through large-scale commercial deployment*. Specifically, we note that in MTR stations where occlusion, interference, and dynamic crowds are common, abrupt RSSI change and weak RSSI values can cause significant ranging errors and fluctuations. Therefore, we adjust the classical free-space propagation model to accommodate these unique challenges in complex MTR environments. Meanwhile, we properly calibrate our model through one-time training and then adopt it across MTR stations. Experiments indicate that our adjusted model significantly outperforms the literature [5, 7, 13, 31, 42], many requiring sophisticated tuning such as ray tracing.

On the trolley side, we engineer a lightweight sensor with an inertial measurement unit (IMU) and a BLE RSSI receiver. Our sensing algorithm can work with diverse sensor placement: hand-holding, in-pocket, and most importantly, sensor attaching to the trolley. For trolley-attached placement, we develop robust algorithms that identify three types of road surfaces appearing in MTR stations: normal road, tactile paving, and contraction joints (Fig. 7). The detected surface type is then utilized to improve the speed estimation. To overcome the aforementioned challenge of missing moving direction, we design a customized particle filter (PF) that leverages the RSSI-distance model and estimated speed to accurately localize the trolley, without requiring explicit direction reading.

Last but not least, we integrate the above components (RSSI model, speed measurement, surface detection, PF-based localization), together with other essential modules (floor plan processing, violation detection/alarming, store classification, delivery recording, *etc.*), into the holistic system of DeMo. The overall development/testing took 6 months. We then worked with HK EMSD and MTR to commercially deploy DeMo in 12 MTR stations in May 2020. We conduct thorough evaluations using two complementary sources: (1) data collected from our 3-year deployment (42K+ deliveries to 200+ shops, with 3521.4 km total travel distance), and (2) 15-day controlled experiments (900 deliveries with 54 km travel distance, with ground truth). Our key results are as follows.

- DeMo's 3-year operation witnessed a significant violation rate drop, from 19% (May 2020) to 2.7% (Mar 2023). This demonstrates DeMo's influence on delivery behaviors.
- We conducted an A/B test to confirm that the improvement of delivery behaviors is indeed due to DeMo's violation detection/warning capability instead of delivery workers' perception of our sensing devices.
- In contrast to the common belief that a propagation model suffers from large errors, our integrated design yields a mean positioning error of 2.17 m in MTR stations without the need for labor-intensive site surveys.
- DeMo achieves accurate road surface detection, which further facilitates trolley speed estimation (mean error 0.31 m/s). Both only use IMU sensors.
- Compared to manual delivery monitoring used before 2020, DeMo improved the monitoring coverage (*i.e.,* the fraction of detected delivery events) from 53% to 88%, and meanwhile reduced the operational cost by 8X.
- DeMo achieves perfect detection for all violation types with

speeding being the only exception. For speeding violations, DeMo reported 9.3% FP and 2.4% FN [1]; the real-time warning is even less accurate. Therefore, DeMo is not intended for law enforcement actions, similar to prior systems (*e.g.,* detecting reckless driving [69]). Despite this limitation, DeMo was endorsed by MTR safety staff: we invited 20 staff to participate in a questionnaire survey; 95% of the participants agree or strongly agree that DeMo can improve delivery safety.

To summarize, our contribution consists of the following. (1) We develop DeMo, a first commercially deployed indoor delivery monitoring system. (2) Through the 3-year deployment of DeMo, we learn several key lessons and operational experiences. (3) Most importantly, DeMo indeed makes HK MTR safer, benefiting millions of riders every day. Note that DeMo can be potentially extended to broader indoor services such as property management [73] for monitoring shuttle vehicles in shopping malls and luggage carts in airports.

Ethical Consideration. Our IRB-approved study complies with the agreement between us and HK MTR. We did not collect any personally identifiable information (PII) of delivery workers or passengers. Neither was MTR willing to release data for the actual incidents because of privacy concerns.

Data Release. To facilitate further research, we have released our collected IMU and BLE readings on GitHub [15].

2 Motivation

Delivery Violations. The MTR is a major public transportation network that transports around 5 million daily passengers in HK. To facilitate commuter needs, MTR stations offer diverse shops (including food/beverage, health/beauty, books, banking, and convenience stores), similar to typical small malls in many countries. Because of the crowded MTR stations, indoor delivery has a potential risk and EMSD in HK requires strict monitoring of delivery behaviors in MTR stations. Targeting the specific scenario of delivery in MTR stations, the EMSD defines the violations as follows.

- Violation 1: speeding (trolley moving speed $\geq$ 1.5 m/s for more than 3 seconds).
- Violation 2: non-designated delivery path (entry/exit/path).
- Violation 3: usage of passenger lift without prior permission from the station.
- Violation 4: deliver during peak hours (07:00-10:00 am and 4:00-8:00 pm).

Manual Monitoring and Limitations. To ensure safe deliveries following government guidelines, MTR stations hire additional safety staff for manual monitoring. These safety staff need to follow each delivery, warn delivery workers once they observe a violation, and file delivery reports for future action. These delivery reports include the delivery time, path, and violation type (if there exists a violation) [2].

[1] FP is less of a concern since MTR stations hope to slow down delivery speeds for passenger safety.

[2] One delivery record example: station code KXX, entry/exit A, start time 5/10/21 10:05, shop ID K001, violation type 2.

Not surprisingly, the current manual system has substantial limitations. (1) It is difficult to visually monitor delivery workers' moving speed, which leads to potential violations and safety risks. (2) Manual recording requires substantial human resources - a safety staff is required for each delivery. It is impossible for one safety staff to monitor multiple deliveries concurrently, leading to monitoring failures. (3) High labor cost. Each station requires multiple safety staff depending on its scale, which directly adds up to a high expense.

The above limitations motivate us to design a fully automated system with reliable monitoring services and reduced costs - DeMo. During DeMo's design and prototyping, we discovered a few unique challenges, such as IMU readings' large fluctuation due to special road surfaces and the severe electromagnetic environment, which need to be addressed in practice for reliable monitoring.

3 System Design

3.1 An Overview of System

DeMo consists of two phases: *offline preparatory phase* and *online operational phase*.

Offline Preparatory Phase. In this phase, we design customized sensors that will be attached to trolleys for delivery monitoring. Then we strategically deploy BLE beacons in MTR stations to balance the monitoring reliability and deployment/maintenance cost. To completely remove the intensive radio fingerprinting cost in many wireless localization systems, we adopt a Received Signal Strength Indication (RSSI) to distance model that is quickly applied across different scenarios with reliable accuracy. In addition, we further process the MTR station floor maps. The preparatory phase's details are discussed in Sec. 3.2.

Online Operational Phase. Fig. 2 shows the operational DeMo. When reaching an MTR station entrance, delivery workers receive DeMo's sensors from MTR staff and then attach sensors to workers' own trolleys that satisfy different shops' specific supply requirements.

During delivery, workers manually drive trolleys that carry supply goods while DeMo monitors the whole procedure. Specifically, DeMo calculates the moving speed (Sec. 3.3) using IMU readings. For the trolley trajectory generation (Sec. 3.4), DeMo adopts a particle filter to leverage received BLE packets and estimated speed. Although the idea seems straightforward, DeMo needs to address a few unique challenges in MTR stations, *e.g.,* large errors due to MTR's special road surfaces and missing directions in a severe electromagnetic environment. Based on the estimated speed and trajectory, DeMo analyzes them for violation detection (Sec. 3.5).

When arriving at the target store, delivery workers unload their goods which usually lasts 5-20 minutes. DeMo utilizes this opportunity to identify the specific store and alleviates the possible false alarms by analyzing the historical IMU+BLE readings. Then, DeMo uploads the delivery record to the

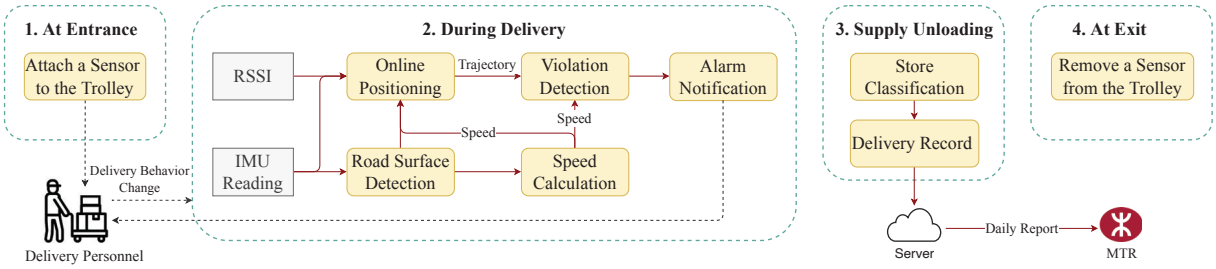


Fig. 2: DeMo system overview (operational phase).

server which analyzes the daily violation rate to identify abnormally high violations. When delivery workers reach an MTR exit, they return DeMo's sensors to MTR staff.

3.2 Offline Preparatory Phase

Our offline phase includes signal choice, sensor design, floor plan processing, BLE beacon deployment, and RSSI-distance model verification.

Signal Choice. Various signals, such as acoustics, WiFi, camera images, and visible light, have been used for indoor localization. DeMo leverages the RSSI of BLE packets and IMU readings because of the following reasons:

- The privacy issue restricts us from installing privacy-intrusive devices, such as cameras or depth cameras.
- As required by MTR stations, we are only allowed to install small-sized battery-powered devices and cannot place power cables. As a result, it is infeasible to install power-hungry devices such as WiFi access points, while MTR stations have limited WiFi coverage and require dedicated WiFi access points for WiFi-based localization.
- Acoustic and light-based indoor localization systems are not practical due to challenges such as noise, reverberation, limited visibility, and complexity.
- Our DeMo achieves desired violation detection based on BLE and IMU with low deployment and maintenance costs.

Sensor Design. In addition to commodity BLE beacons deployed in MTR stations, we customize on-trolley sensors with two major components, Raspberry PI 4B and customized hardware attached on top (HAT) in Fig. 4. Our HAT contains an inertial measurement unit (IMU) MPU9250, a speaker for alarm, 6 indicator lights to indicate the operation statuses and a fan for cooling.

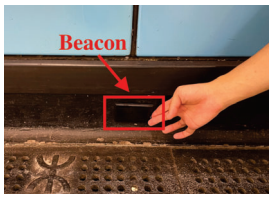


Fig. 3: Sender: a beacon deployed on the skirting board.

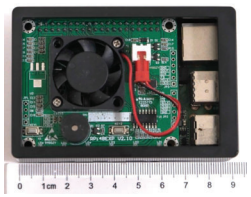


Fig. 4: Receiver: Raspberry PI 4B and customized HAT.

Our sensor captures BLE packets via a Bluetooth chipset and detects violations in real-time. Once detecting a violation, it alarms through a speaker and also reports to the server through a 4G dongle. Since this report contains only essential information (*e.g.,* sensor id, station, entry/exit, delivery time, destination shop, and violation type), it requires moderate energy consumption. The overall energy consumption for DeMo's sensing, computation, and uploading is around 400 mW. We chose the Raspberry PI to reduce our hardware design workload, while its operating system accounts for the majority of the energy consumption (2000 mW). Our sensor is connected to a small-capacity portable power bank that is charged roughly every 2 days.

BLE Beacon Deployment. MTR stations impose strict aesthetic constraints on BLE beacon deployment. We adopt small-size, battery-powered, and black-coated beacons and deploy them at skirting board locations (Fig. 3) near the ground. The beacon distance is 6 to 8 meters in the majority of areas. During DeMo's initial trial, we noticed several areas that often lead to large errors or are critical in DeMo's operation. This inspires us to adopt dedicated deployment strategies for these areas. For large pillars, we deploy beacons on all four sides to alleviate signal obstruction. For in-station stores, we deploy 3 beacons (two outside and one inside the store) to improve the store classification accuracy.

During DeMo's operation, we also recognize the beacon types affect our maintenance. Table 2 shows two types of beacons primarily used for DeMo. In our initial few stations, we deployed beacon type 1 but noticed a significant loss rate, reaching 9% after 5 months (Sec. 4.4). This loss is caused by the high chances of collisions with passengers. To alleviate maintenance costs, we turned to beacon type 2, which is more expensive with a card shape. This card shape offers larger contact areas for gluing and reduces the chances of colliding with walking passengers with better reliability, around 2.3% after 5 months. This motivates us to only deploy beacon type 2 in later deployment/maintenance. Overall, we have deployed DeMo in 12 MTR stations with more than 1,500 beacons.

RSSI-Distance Model Verification. Many localization systems [11, 23, 24, 30, 63] rely on radio fingerprinting for accurate localization and require intensive deployment costs. Although RSSI-distance models have been extensively discussed in the literature for localization in small-scale indoor scenarios for alleviating deployment efforts [5, 7, 13, 31, 42], its verification is fundamentally missing in large-scale settings. In contrast, DeMo adopts a low-cost and accurate RSSI-

Comparison	Type 1	Type 2
Appearance		
Size	39x39x15mm	86x54x6mm
Cost	6.3 USD	9 USD

Table 2: Beacon type comparison.

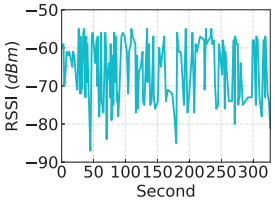


Fig. 5: Abrupt RSSI change with time at a static sensor.

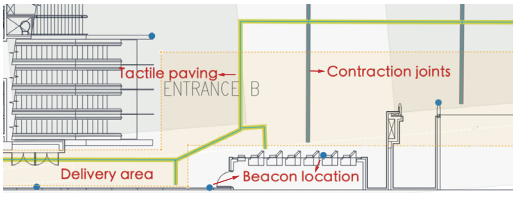


Fig. 6: Pre-processed floor plan. Yellow regions represent designated delivery areas.

distance model for large-scale operation.

We started with classic free-space propagation models, *e.g.,* the well-known logarithmic model $r = r_0 + 10N\log d + X_\sigma$ [42]. Our initial trials brought up multiple unique challenges in complex MTR stations. (1) Crowd obstruction and reflection. These factors lead to signal losses that deviate from the logarithmic model, which is especially inaccurate under weak RSSI. (2) Highly dynamic crowd movement. The complex MTR environment results in abrupt RSSI changes even when our sensor is static. Fig. 5 shows an example of the RSSI change with time at a static sensor - directly adopting a propagation model will lead to significant localization errors. In addition to the logarithmic model, we also tried multiple fine-tuned RSSI-distance models (*e.g.,* [5, 7, 13]) but noticed more severe localization errors caused by the complex and dynamic MTR environment.

These preliminary tests motivate us with the following reliable model applicable to complex and dynamic settings. First, we transfer the logarithmic model to a probability distribution model to be used in a particle filter: $p(r|d) = \frac{1}{\sigma\sqrt{2\pi}}\exp(-\frac{1}{2\sigma^2}(r-\hat{r})^2)$, which represents the conditional probability of receiving an RSSI value r given a distance d, where $\hat{r} = r_0 + 10N\log d$. This probability follows a Gaussian distribution with mean $\hat{r}$ and variance σ^2. Next, to combat crowd reflection and obstruction, we ignore weak RSSI values under a threshold R_{th} (empirically set to -80 dBm) to only leverage strong signals with good reliability. We also modify the probability model as follows: $p(r|d) = k\exp(-\frac{1}{2\sigma^2}(r-L(\hat{r}))^2)$, where $L(\hat{r}) = \gamma_0 + \gamma_1\hat{r} + \gamma_2\hat{r}^2 + \gamma_3\hat{r}^3$ is a cubic polynomial function to model crowds' impacts and k is adjusted normalization. We have tried multiple polynomial functions and observed that our current cubic function provides good accuracy while maintaining reliable generalization. Higher or lower-order polynomials commonly encounter larger errors when applied across different MTR stations. A detailed comparison is included in the Appendix A.2. Coefficients (*e.g.,* γ_0, γ_1, γ_2, and γ_3) are determined through simulation experiments, and our detailed training procedure is open-sourced at [16]. To mitigate the highly dynamic crowds, DeMo adopts a sliding window t_{sw} of 4 seconds and leverages the maximum RSSI within this period, which outperforms other statistics, *e.g.,* the average, median, or the most recent RSSI value. This sliding window enhances DeMo's reliability against lost BLE beacons while its duration is empirically set to 4 seconds to ensure the balance between reliability and delay. A short window might suffer from insufficient BLE packets, while a long window leads to a larger delay.

Our training data is collected in three representative areas (*e.g.,* MTR's entrance, store area, and corridors) and consists of RSSI values collected at different distances from our beacons to sensors. There is no significant accuracy difference between these areas. The overall mean localization error solely based on BLE without IMU (Sec. 5.3) is 2.81 m. Then, we further validate our model across the remaining stations without parameter retraining. Detailed results are included in Appendix A.3. In most stations, the positioning error is similar to our initial station used for model training, ranging between 2.42 m and 3.39 m. Nevertheless, two subway stations suffer from more significant positioning errors at 3.68 m and 3.83 m. These two stations boast unique layouts featuring larger open spaces, unlike most subway stations that have long, narrow delivery areas. In addition, we observed more severe beacon damages in these two stations. These factors result in diminished positioning accuracy. These experiments verify the feasibility of low-cost and accurate localization based on a simple RSSI propagation model - one-time training is adequate for accurate localization in complex environments like HK's MTR stations. Transferring our model to other scenarios (*e.g.,* subway stations located in other cities) involves parameter reconfiguration, while we have open-sourced our parameter configuration procedures at [16].

■ **Finding 1** *Without labor-intensive site surveys, a good RSSI model with necessary customization offers accurate localization in complex and dynamic indoor environments like MTR stations. The deployment cost could be reduced by one-time parameter tuning and adoption to all stations with similar accuracy.*

Floor Plan Processing. We enhance the floor plans of MTR stations with detailed information, including geo-fencing [51] and surface statuses. Geo-fencing involves adding polygons to the floor map to demarcate the delivery area in compliance with MTR regulations. These polygons are then utilized during the online phase for violation detection. We also examine MTR stations' ground surface conditions. During our preliminary deployment, we noticed that certain road surfaces (tactile paving and contraction joints) lead to special patterns on the IMU readings, which could be leveraged for improving detection performance (further details in

Sec. 3.3). Note that DeMo avoids tedious examination of each station since MTR stations' surfaces follow specific construction regulations and exhibit very similar IMU patterns. Fig. 6 shows an example of the pre-processed floor plan.

3.3 Online Speed Detection

DeMo estimates the trolley's real-time speed by analyzing IMU readings with two major components: road surface detection and speed calculation. Special road surfaces cause substantial IMU fluctuations, resulting in significant integral errors. Consequently, in the speed calculation process, we exclude IMU readings caused by these surfaces to maintain speed accuracy.

Road Surface Detection. Special road surfaces (tactile paving [48] and contraction joints [33]) lead to large IMU fluctuations and speed estimation errors when a trolley passes these surfaces. Specifically, tactile pavings (Fig. 7(a)) consist of 4 parallel bars and are widely used to assist pedestrians with vision impairment. Contraction joints (Fig. 7(b)) have a narrow width and are often used to avoid cracking damage caused by thermal expansion. This motivates us to detect these special surfaces to improve speed detection accuracy.

(a) Tactile paving (b) Contraction joints (c) Normal road

Fig. 7: Examples of tactile paving, contraction joints, and a normal road.

Fig. 8 compares the IMU readings when a trolley passes different surfaces. These large IMU fluctuations inspire us to adopt peak detection [47] to recognize road surfaces. Specifically, we calculate the standard deviation of the IMU denoted as a_{std} and extract peak values that are at least 4 times the standard deviation. The time interval between sequential peaks is denoted as $t_1 \ldots t_n$, with n number of peaks. Considering that tactile paving contains 4 parallel bars and a trolley's typical movement speed, DeMo's tries to detect $n = 8$ peaks within a short interval threshold (*e.g.,* 0.04 s to 0.2 s). If the above two conditions hold, the current surface is recognized as tactile paving. DeMo adopts a similar strategy to detect a contraction joint, if n is 2 and each of $t_1 \ldots t_n$ is between 0.2 and 1 second. The value of n considers the characteristics of the specific road surface (*e.g.,* 4 parallel bars) and the number of times the delivery wheel has passed.

Speed Calculation. DeMo analyzes IMU readings for accurate speed estimation with the following procedures.

• *Sensor status detection.* When the a_{std} is under 0.01 g [50], the sensor is considered to be static; it is moving otherwise.

• *Sensor placement detection.* In practice, delivery workers adopt the following placement: 1) Our sensor is in a pocket or hand. Traditional pedestrian dead reckoning [25, 34, 50] methods could be adopted to recognize human walking patterns like step detection for speed inference, as widely discussed in literature [30, 49, 55]. 2) Our sensor is placed on a trolley. DeMo recognizes this placement if IMU readings lack human walking patterns [8]. Based on DeMo's operational results (Sec. 4.3), our sensor is placed on trolleys for 95% of deliveries.

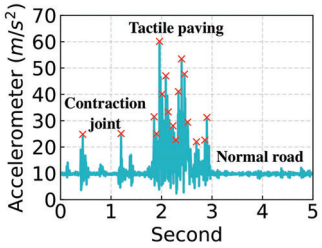


Fig. 8: IMU readings at different surfaces.

• *Adaptive integral of IMU readings.* For a on-trolley sensor[3], DeMo estimates this trolley's speed by analyzing IMU readings. First, DeMo removes noises and outliers through a low pass filter, excluding IMU readings that resulted from special road surfaces. Second, DeMo detects acceleration and deceleration by analyzing the IMU distribution. According to our practical experience, the accelerometer readings are dominated (>80%) by either positive or negative values when the trolley accelerates or decelerates. Once identifying these phenomena, DeMo integrates IMU readings with the previous speed to re-estimate the current speed. Otherwise, when a sensor is considered to have uniform movement, DeMo does not re-estimate the current speed.

■ **Finding 2** *Identification of road surfaces (e.g., tactile paving and contraction joints) via IMU processing benefits speed estimation accuracy.*

3.4 Online Positioning

DeMo leverages a particle filter (PF) [20] to recursively update the probability distribution for tracking the trolley position in real-time. We choose PF DeMo because it is good at fusing diverse signal sources (*e.g.,* BLE, IMU) and also offers good reliability with less training data in dynamic situations.

For initialization, DeMo generates N particles that have the same weight and are evenly distributed over the delivery area. In each iteration, DeMo moves its particles through the transition model and updates particle weights accordingly. To tackle the missing movement direction, DeMo leverages IMU readings and the trolley's historical trajectory to estimate the moving direction. Specifically, we combine the accelerometer and gyroscope reading to calculate the Euler angle [17] and move each particle to a new position based on the variance of the yaw angle and the speed produced from Sec. 3.3. In practice, we observe a high variance of the yaw angle when

[3] For in-hand or in-pocket sensor, DeMo leverages existing techniques for speed estimation [34].

the trolley turns, while the variance is small when moving in a straight line. As a result, when the yaw angle's variance is small, most particles maintain the previous motion direction, while a small number of particles follow a uniform distribution to simulate random movement. Otherwise, the number of particles with random movement increases to estimate the new movement direction. When receiving a BLE packet, DeMo updates its particle weights via our RSSI-distance model (Sec. 3.2). After the above updates, DeMo finds a center point of the top p_w weighted particles and repeats this iteration with new BLE+IMU readings. p_w is empirically set to 60%.

3.5 Operational Model

Targeting violation detection and safe delivery, DeMo has the following designs.

Real-Time Violation Detection. To satisfy the requirement of real-time operation, DeMo's computation is placed on the sensor (Raspberry PI) side. This also avoids significant delay and energy consumption for data uploaded to the cloud server. For speed violations, DeMo compares the speed calculated in Sec. 3.3 with the regulation (1.5 m/s) to detect if there is a speed violation. Then DeMo leverages the positioning results calculated in Sec. 3.4 for delivery trajectory generation. By checking allowed delivery areas, DeMo is able to detect delivery involving non-designated path/entrance/exit. For passenger lift violation, DeMo first utilizes IMU and BLE readings to recognize the floor change via lifts and then checks the current position with the map to identify passenger lifts. As for the detection of peak-hour delivery, our on-trolley sensors record the delivery start time and end time for checking with peak hours (7:00-10:00 am and 4:00-8:00 pm). If a violation is detected, DeMo alarms the delivery worker through its speaker immediately to correct his/her delivery behavior.

Daily Report. When a delivery worker arrives at the target shop, this delivery worker stops his/her trolley for supply unloading, which will usually last 5-20 minutes. DeMo leverages this opportunity to process historical IMU+BLE data to mitigate false alarms and identify the destination shop based on positioning results and IMU patterns. Then DeMo generates the current delivery record that logs the DeMo sensor ID, entry/exit, destination shop, delivery time, and the delivery violation type. After this, DeMo uploads the current delivery record to our cloud server via a 4G dongle. Due to a delivery record's limited size, this record uploading requires tiny energy consumption. DeMo's server processes these records to generate daily reports for MTR.

4 Large-Scale Operation

This section offers DeMo's large-scale in-the-wild operation results. DeMo covers more than 200 shops at 12 MTR stations with a delivery area of 19,433 m^2. Since its debut in May 2020, DeMo has monitored 42,248 deliveries with a 3521.4 km delivery length. By default, the beacons have a broadcast interval of 200 ms, and the sample rate of IMU is 500 Hz.

4.1 Violation Behavior Analysis

Violation Reduction. For each violation type, its violation rate is calculated as the number of violations divided by the total number of deliveries for all stations. According to DeMo, the violation rate for wrong delivery path, delivery in peak hours, and using passenger lifts was 1% in 2020 and dropped to 0.5% in 2023. Fig. 9 shows DeMo's detection results of speeding violations[4]. At first, DeMo detected a high violation rate of around 19%, meaning that almost one-fifth of the delivery is speeding with potential safety risks to commuters. Targeting safe delivery, DeMo offers accurate violation detection (more details in Sec. 5.2) and generates alarm warnings in real time. These real-time alarms effectively correct workers' delivery behaviors, *e.g.,* slowing down the movement speed. By doing so, DeMo gradually reduced the violation rate with time, which reached 2.7% in March 2023.

Prior to DeMo, MTR stations also adopted a manual monitoring system for many years without such an achievement. Given that DeMo effectively monitors more than 88% of the total delivery events [5], DeMo offers effective correction on delivery behaviors with enhanced safety protection.

A/B Testing. To exclude the placebo effect, we launched additional A/B testing to analyze DeMo's influence on delivery behaviors. This test was conducted in two MTR stations for two months. Version A was the operational DeMo discussed in Sec. 3.1 and was tested in the first month. In the second month, we tested Version B, which had all the same components (*e.g.,* speeding violation threshold and daily report generation) as Version A, while the only exception was that the real-time alarm notification was disabled. Delivery workers were not informed of this change. This also rules out delivery workers' perception of our devices since the *alarming function* is the only difference between Version A and B. Table 3 shows the comparison of violation rates. Note that we have only detected the speeding violations, and no other types of violations were detected during these two months. The violation rate increased in both of these two stations, demonstrating the critical role of real-time alarms in notifying delivery workers of behavior changes.

Table 3: A/B testing.

Station	Version A Violation Rate	Version B Violation Rate
Station 1	3.61%	9.22%
Station 2	5.19%	11.51%

[4]Due to the COVID-19 outbreak, DeMo was suspended at MTR stations from July 2021 to November 2021.

[5]MTR corporation has a record of all delivery activities since shops are mandatory to submit their delivery applications. This record is used as the ground truth to evaluate DeMo's monitoring efficiency. 88% is the number of deliveries detected by DeMo divided by the total number of delivery records according to MTR. The previous manual monitoring only covers 53% of all deliveries due to its intensive human resource requirement and limited staff.

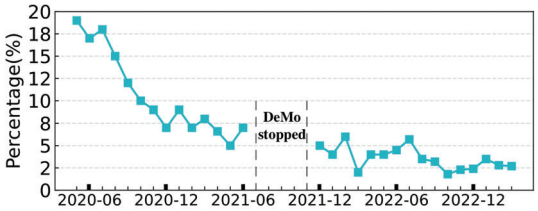


Fig. 9: Violation (speeding) rate.

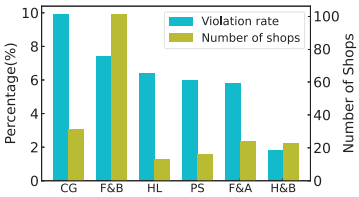


Fig. 10: Distribution of violation shops.

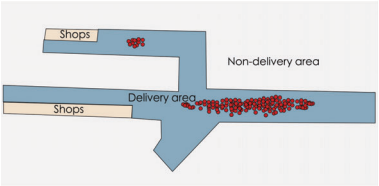


Fig. 11: A field study of the speeding area.

■ **Lesson 1** *DeMo's 3-year large-scale operation effectively reduces the violation rate from 19% to 2.7%. As indicated by our A/B test, accurate violation detection and real-time warning are necessary prerequisites for such a positive delivery behavior change.*

Shop Category's Impacts. We classify more than 200 shops in MTR stations as convenience goods (CG), food and beverage (F&B), fashion and accessories (F&A), health and beauty (H&B), home living (HL), and passenger services (PS). For each shop category, the violation ratio is counted as the number of violations divided by the total delivery, as demonstrated in Fig. 10. Overall, the CG category has the highest violation rate, followed by the F&B category. This is likely because these stores have a high demand for replenishment that naturally leads to more incentives for delivery workers to expedite deliveries.

Geographic Distribution. We observed that certain areas, such as long (more than 20 m) and wide (5 m or more) corridors, exhibit much higher chances of speeding. Generally, speeding violations often occur in the middle of straight roads to the shops. This suggests MTR stations take specific countermeasures to improve safety. Fig. 11 shows one example of the clustered violation distribution, where each dot represents a speeding violation.

4.2 DeMo vs. Manual Monitoring

Compared with manual monitoring/recording via hired MTR staff[6], DeMo has the following unique advantages.

• **Full coverage of violation detection.** DeMo reliably detects 4 types of violations, while manual monitoring is not well suited for speeding detection.

• **Delivery behavior change.** DeMo regulates delivery behaviors with corrections and reduces the violation rate from 19% to 2.7% as demonstrated in Sec. 4.1, which is fundamentally missing in the manual system.

• **Efficiency.** Manual monitoring requires a safety staff for each delivery with intensive human resource requirements. In contrast, DeMo successfully monitors 42,248 deliveries, covering 88% of the total delivery activities on average (verified with the total number of delivery activities), much better than the 53% monitoring rate offered in manual monitoring. The missing cases are largely the result of insufficient sensors at some subway stations and sensor hardware failures due to rough handling.

• **Cost saving.** DeMo's cost consists of one-time deployment and maintenance costs. Its deployment cost includes hardware cost, floor plan processing, beacon installation, and program development. For the hardware cost, DeMo requires Raspberry PI with HAT (90 USD each) and beacons (9 USD each)[7]. Each station needs 5-15 Raspberry PI and 50-250 beacons, depending on delivery area size. Overall, the one-time deployment cost is about 15K-20K USD per station. DeMo's maintenance cost to replace failed beacons is about 40 USD per station/month, thanks to DeMo's low maintenance requirement. As for manual monitoring, each station hires multiple safety staff (*e.g.,* 2) depending on the station size. Due to privacy issues, we do not know the exact wage and use the median monthly wage (2,383 USD) in the statistical reports [44]. After DeMo's 3-year operation, the cost of manual monitoring is at least 8X higher than DeMo.

■ **Finding 3** *DeMo outperforms manual services in terms of violation detection coverage (especially speeding detection), delivery behavior change (violation rate drops from 19% to 2.7%), monitoring efficiency (DeMo detects 88% of the total delivery events in contrast to the prior 53%), and cost reduction ($> 8X$).*

4.3 System Performance

Approximated Positioning Accuracy. In DeMo's large-scale operation, we lack the trolley's ground-truth position. To alleviate this limitation, we utilize the events when trolleys pass tactile pavings to *approximately* evaluate the positioning accuracy since these events could be reliably detected. We adopt the following methodology. IMU reading analysis offers the time T when a trolley reaches tactile paving. We compute the shortest distance from DeMo's current positioning result to the nearest segment of tactile paving. This shortest distance is leveraged as an approximated error, not the *exact* localization error.

Fig. 12 plots the approximated positioning error distribution in March 2022, with a median error of 1.45 m. Furthermore, we evaluate DeMo's stability with time by comparing

[6] Manual records include violation type, start time, station code, entry/exit, and the destination shop. Note that the records are taken for both violation and non-violation deliveries.

[7] The prices were in 2020.

the approximated error in March and July 2022. After four months, the median positioning error slightly increases to 1.49 m. Note that this approximated error differs from the actual localization error offered from our controlled experiments in Sec. 5.3.

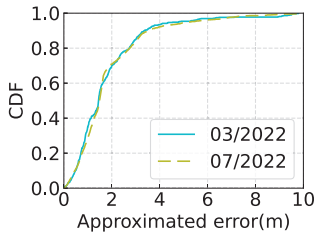


Fig. 12: Approximated positioning error in 2022.

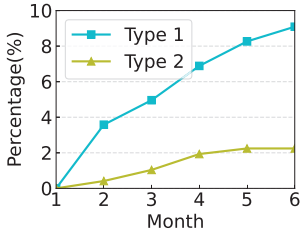


Fig. 13: Beacon failure rate with time in 2020.

Statistics of Sensor Placement. We observe that a small percentage of delivery personnel (5%) hold the sensor in their hands or pockets, while the other 95% place sensors on trolleys. DeMo is compatible with both placements.

4.4 System Maintenance

This section analyzes DeMo's maintenance cost.

Impact of Beacon Types. Our initial deployment adopted beacon type 1 but noticed a high loss rate caused by various reasons, such as natural falls and collisions with passengers. This high loss rate motivated us to switch to beacon type 2 in our latter deployment or maintenance, as discussed in Sec. 3.2. Fig. 13 shows the beacon failure rate with time. After 5 months, the loss rate of type 1 reached 9%, while type 2 was only around 2.3%. This experiment validates that appropriate beacon selection could effectively reduce maintenance costs.

Failed Beacon Location. Table 4 offers the beacon loss rate at different locations. These statistics only include beacon type 2 and were collected 5 months after the deployment. Areas such as stores and entry/exit have the highest beacon loss rates, which is likely to be caused by the dense crowd. For locations with high loss rates, we increase the beacon broadcast frequency (*e.g.,* 100 ms) to compensate for lost beacons and leverage the deployed beacons before they fail.

Table 4: Beacon failure with the location.

Location	Store	Entry/Exit	Corridor	Others
Failure rate (%)	5.3	3	1	0.6

■ **Finding 4** *Strategic beacon deployment could alleviate system maintenance costs. For example, card shape beacon (type 2) significantly reduces the beacon failure rate compared with common box-shaped beacons. As a result, beacon shape and deployment areas should be thoroughly considered, while beacon broadcast frequency in high loss-rate areas could be increased for better utilization before device failures.*

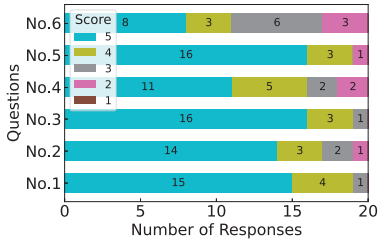


Fig. 14: Feedback from 20 MTR staff.

4.5 MTR Feedback

We designed a questionnaire to evaluate DeMo's performance and Fig. 14 depicts the feedback from 20 safety staff in 12 MTR stations. Our survey contains six questions, each employing a 5-point Likert scale ranging from 5 (strongly agree) to 1 (strongly disagree). The six questions are: 1. satisfaction with our system, 2. low complexity of device usage, 3. effect on violation reduction, 4. speed detection accuracy, 5. decrease of workload, and 6. frequency of sensor damage. The detailed questionnaire is included in the Appendix A.5. We summarize these questions with 4 takeaway messages.

Overall, DeMo receives positive feedback from the majority of MTR staff. Taking question 1 (satisfaction with our system) for example, more than 95% of the interviewees highly rate DeMo with a score of 4 (agree) or 5 (strongly agree). The major takeaway messages are as follows.

(1) User friendly. The operation of DeMo is easy and convenient without complicated knowledge.

(2) Violation rate decrease. DeMo effectively decreases the violation rate by correcting delivery behaviors.

(3) Workload reduction. DeMo effectively reduces MTR staff's workload.

(4) Sensor failure. Some interviewees complained about sensor failure caused by delivery workers' rough handling (about one device/month in some stations). Our device was not designed with industry-level reliability and could be damaged by collision, which needs to be improved in the future.

■ **Finding 5** *DeMo is highly rated by the majority (95%) of MTR operators. Its positive feedback covers various aspects, such as easy accessibility, effective delivery behavior change, and workload reduction. On the other hand, we plan to address device failures by designing more reliable hardware.*

5 Evaluation via Controlled Experiments

This section evaluates DeMo via controlled experiments with collected ground truths, in contrast to our large-scale in-the-wild evaluation in Sec. 4. Our controlled experiments last 15 days and cover 0.9k delivery cases with a total length of approximately 54 km.

5.1 Evaluation Methodology

Without loss of generality, we chose three stations of different scales (552, 1105, and 3,003 m^2 respectively) to represent

small, medium, and large stations for our controlled experiments. We hired several delivery personnel to perform test deliveries in each station, while DeMo operated in real-time to monitor all the delivery procedures. To evaluate DeMo, we adopt the following ground truth collection mechanisms instead of leveraging in-the-wild deployment data. In this evaluation, our testing device has an embedded screen that displays the MTR floor map with special *checkpoints* that could be easily found (*e.g.,* pavement contraction joint, pillar, lift). The distance between the two checkpoints is about 5 m. When passing through a checkpoint, hired delivery personnel will click on the screen to record the ground-truth location and time. The ground-truth speed of two adjacent checkpoints can be calculated accordingly.

For violation behaviors, hired delivery personnel purposefully violate the delivery rules, such as speeding, delivering during peak hours, using non-designated paths, and using passenger lifts. We have obtained approval from MTR stations and adopted additional safety measures for passenger safety.

5.2 Violation Detection Reliability

We evaluate DeMo's detection reliability for four violations mentioned in Sec. 2. Overall, our data set contains 900 deliveries. Among them, the number of violations for speeding, wrong delivery paths, taking passenger lifts, and delivery in peak hours is 41, 16, 18, and 11 respectively, with a total number of 86 violations. Each delivery only includes one violation. To ensure a balanced data set, we randomly selected a subset of normal deliveries (without any violation), with the number of 43, 18, 19, and 12, respectively. Table 5 demonstrates the detection performance with the following metrics. True Positive (TP) represents that DeMo correctly identifies a delivery violation. True Negative (TN) indicates the accurate detection of a normal delivery. False Positive (FP) shows DeMo falsely recognizes a normal delivery as a violation. False Negative (FN) means DeMo recognizes a violation as non-violation. Note that only speeding (daily report) in Table 5 is based on the historical IMU+BLE data during the delivery procedure, while all other evaluations are real-time.

For real-time speeding detection, DeMo achieves a TP rate of 95.1% and TN rate of 83.7%, while it suffers from 16.3% FP errors and 4.9% FN errors. When the current speed exceeds 1.5 m/s, the speeding criteria defined by MTR stations, DeMo could reliably detect it as a speeding violation. However, when the current speed is smaller than but close to 1.5 m/s, DeMo might classify this case as speeding, leading to FP results. This is the reason that DeMo's FP is significantly worse than FN, which is deliberately adjusted by us since MTR stations hope to slow down delivery workers to ensure safety. Leveraging historical IMU+BLE data, DeMo manages to further reduce the FP and FN performance.

Besides speeding detection, DeMo offers reliable performance for the remaining 3 violations. Thanks to DeMo's accurate localization performance (a median positioning error of 1.89 m), it is able to generate accurate delivery trajectory by connecting individual positioning results. By checking the current trajectory with allowed areas, DeMo can identify non-designated delivery paths. Since freight lifts and passenger lifts are far away from each other (more than 6 m), DeMo can accurately detect the use of passenger lifts. For each delivery, our on-trolley sensors record the delivery start and end times for checking with peak hours (7:00-10:00 am and 4:00-8:00 pm). Given a long delivery duration, *e.g.,* 20 minutes, DeMo ensures accurate detection of peak-hour delivery. Overall, DeMo offers accurate detection for different types of violations in Table 5, demonstrating its reliability in practice.

Table 5: Violation detection accuracy.

Violation Type	TP (%)	TN (%)	FP (%)	FN (%)
Speeding (real-time)	95.1	83.7	16.3	4.9
Speeding (daily report)	97.6	90.7	9.3	2.4
Wrong delivery path	100	100	0	0
Using passenger lift	100	100	0	0
In peak hour	100	100	0	0

■ **Lesson 2** *A monitoring system based on low-cost hardware (BLE and IMU) is adequate for reliable (TP $>$ 95% for all required scenarios) and real-time violation detection toward safe delivery.*

5.3 System Performance

Speed Detection Accuracy. We compare three speed calculation methods: the algorithm used in DeMo, traditional integral without road surface detection, and positioning-based approach (the speed is calculated from the localization results). The ground truth speed is calculated from the hired delivery personnel's ground-truth locations and the time passing them.

Fig. 15 shows the overall speed detection error. Not surprisingly, the position-based approach leads to the highest mean speed error of 0.52 m/s. This is because the localization includes multiple sources of errors, such as wireless fading, failed BLE beacons, and IMU fluctuations. In contrast, directly calculating speed via IMU readings is only affected by the IMU errors. For the speed detection based on direct integral, the mean error is 0.43 m/s. For DeMo, it detects the special road surfaces and further excludes them to improve its speed detection accuracy, leading to a mean error of 0.31 m/s. The tail improvement is much more significant: the third quartile error decreases from 0.62 m/s to 0.44 m/s, meaning that the speeding violation detection is much more reliable.

Impact of Delivery Path Length. Table 6 shows the distribution of the delivery length from the station entry to the destination shop. Fig. 16 demonstrates trolley speed estimation errors under the different path lengths of 50m, 100m, 200m. In general, we observe a smaller error (median error of 0.18 m/s) for short paths (<50 m), while long paths (100-200 m) exhibit larger errors (median error of 0.23 m/s). This difference is because the integral error accumulates with time.

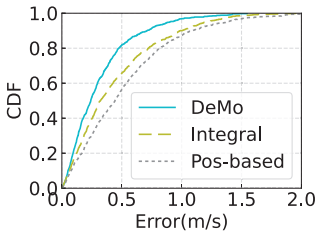


Fig. 15: Trolley speed estimation accuracy.

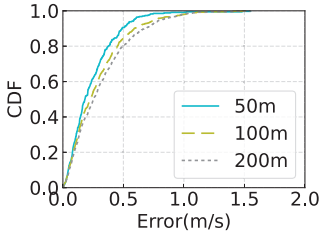


Fig. 16: Delivery path length with speed estimation error.

Table 6: Delivery path length distribution.

Delivery Path	< 50 m	50 - 100 m	100 - 200 m
Fraction (%)	10	75	15

■ **Finding 6** *DeMo detects special road surfaces (*e.g., *tactile paving and contraction joints) solely based on IMU readings for accurate speed estimation, which reduces 28% of the average speed error compared with conventional integral computation.*

Positioning Accuracy. Fig. 17 compares positioning accuracy of using traditional logarithmic model [42], DeMo, DeMo which only relies on BLE without IMU (denoted as DeMo: BLE only), and the ideal localization based on BLE and ground-truth velocity calculated from passing two checkpoints (denoted as BLE+GTV). When only using BLE with the traditional logarithmic model, the mean positioning error is 3.22 m, while our new RSSI-distance model effectively reduces the mean positioning error to 2.86 m. By combining the IMU data analysis, DeMo significantly reduces the localization error to 2.17 m, demonstrating the effectiveness of utilizing IMU data for improving tracking accuracy. The 90th and 99th positioning errors are 4.29 m and 6.44 m, respectively. Additionally, we notice that for 2% of the checkpoints, DeMo's positioning error exceeds 6 m. This large error is primarily due to failures in receiving the broadcast BLE beacon packets and does not affect DeMo's operation in practice. Not surprisingly, BLE+GTV offers the best accuracy with a 1.70 m mean error, while GTV is not available in practice.

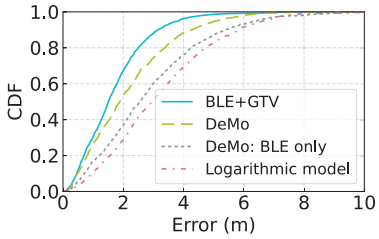


Fig. 17: Positioning accuracy.

■ **Lesson 3** *Without labor-intensive radio fingerprinting, an RSSI-distance model with customization is feasible to achieve accurate localization in complex environments. DeMo's integrated analysis of BLE and IMU readings yields an average positioning error of 2.17 m in dynamic and highly crowded MTR stations. This accuracy is adequate for common indoor applications [30, 43].*

BLE Beacon Density. The beacon density affects the positioning accuracy. (1) For the evaluation of beacon density in typical areas, we randomly select one station that has a beacon density of 6m and adjust the beacon density to 12 m by temporarily disabling some beacons. As in Fig. 18, the mean error increases from 2.17 m to 3.23 m. This is because a higher beacon density leads to more BLE packets received by DeMo and thus better positioning accuracy. A higher beacon density with an interval smaller than 6 m will further improve the localization accuracy, which is more than enough since the existing system already satisfies the violation detection required by MTR stations. At the same time, a higher density significantly increases the deployment and maintenance costs, so it is not adopted in DeMo. (2) In addition to typical areas, several key areas are critical for DeMo's operation. For example, beacons deployed in shops (details in Sec. 3.2) are leveraged in DeMo's shop classification. The identified shop is essential in DeMo's delivery record and used to find shops with abnormally high violations. With three beacons, DeMo's store classification accuracy is 97%, which drops to 86% when there is one beacon. The classification errors largely result from trolleys' parking positions, which could be very close to nearby shops. To ensure reliable operation, DeMo adopts three beacons for shop classification.

BLE Beacon Broadcast Frequency. Another factor to affect localization accuracy is beacon broadcast frequency. Fig. 19 compares the positioning error with beacon broadcast intervals of 200 ms and 500 ms at the same station, while all other parameters are the same. The mean localization error increases from 2.17 m to 2.78 m, since DeMo's sensor receives more BLE packets with a smaller broadcast interval. Based on our 3-year operational experience, the average battery life of a BLE beacon with a 200 ms setting is 22 months. This leads to a maintenance cost (replacing failed beacons) of around 40 USD for each station per month.

IMU Sample Rate. In this experiment, we set the IMU sample rate to 500 Hz for data collection and then downsampled it to generate the IMU data at different frequencies. Fig. 20 shows the detection accuracy of road bump events, including tactile paving and contraction joints. Generally, DeMo's detection rate increases with the IMU sample rate and offers a stable performance at 200 Hz. Similarly in Fig. 21, the speed estimation error is significant when the IMU sample rate is less than 100 Hz. As for energy consumption, increasing the sampling rate from 50 Hz to 500 Hz contributes to less than 3% of the total energy consumption (including sensing and computation). This experiment suggests that a relatively high IMU sample rate is desirable for reliable monitoring.

■ **Lesson 4** *DeMo aims at reliable monitoring under practical costs. It deploys sparse beacons for localization in most*

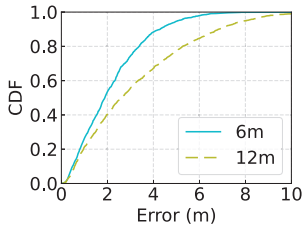


Fig. 18: BLE beacon density vs. positioning accuracy.

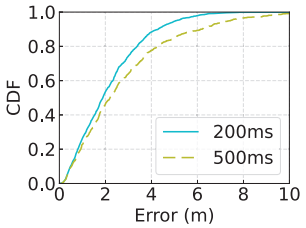


Fig. 19: BLE beacon broadcast frequency vs. positioning accuracy.

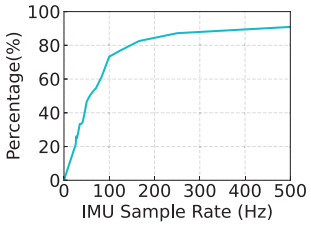


Fig. 20: IMU sample rate vs. road bump detection rate.

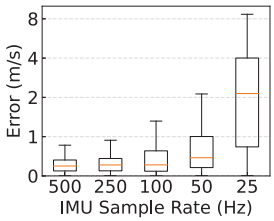


Fig. 21: IMU sample rate vs. speed error.

areas but adopts dense beacons for accurate store classification and reliable operation. Additional considerations such as maintenance cost and energy consumption also contribute to DeMo's trade-off.

6 Related Work

Prior Deployment Experiences. A few BLE-based [22, 30, 40, 70] and WiFi-based [27] localization systems were deployed in public indoor sites to offer localization and navigation services for clients. Recent studies [18, 19] reported deploying arrival detection for efficient delivery in large-scale instant delivery systems. In contrast, DeMo provides fine-grained monitoring of delivery violations toward better safety. A prior study [30] integrates BLE beacons and geomagnetic fields as fingerprints to offer localization services for shopping malls, while DeMo utilizes a labor-free RSSI-distance model to achieve similar localization accuracy. Other real-world existing BLE systems conducted presence detection [59, 60]. To our knowledge, there is very limited work on indoor delivery violation detection and regulation. During the operation of DeMo, we have learned valuable lessons that will contribute to future safety delivery.

Wireless Indoor Localization. Researchers have proposed various wireless indoor localization techniques for navigation [4, 54], positioning [56, 65], and assets tracking [35, 72]. Their design principles could be classified as RSSI propagation model [10, 13, 26, 29, 31], fingerprinting [5, 11, 23, 28, 43, 55, 61], Angle-of-Arrival model [3, 12, 36, 37, 67], and Time-of-Flight [4, 58]. Although existing studies offer good positioning accuracy, they are commonly evaluated in small-scale environments. For example, Spotfi [37], ArrayTrack [67] and ToneTrack [68] achieve sub-meter localization accuracy by leveraging WiFi Channel State Information (CSI). Nevertheless, these systems are not applicable in real-world applications due to the lack of CSI support in most commercial WiFi Access points and limited WiFi coverage in MTR stations. In contrast, DeMo's in-the-wild operation validates the possibility of accurate localization via simple RSSI models with low deployment and maintenance costs, in addition to several unique lessons and insights.

IMU-Assisted Sensing System. These works [9, 71] analyze IMU's signal characteristics and extract the motion feature of humans to infer their posture, especially for Pedestrian Dead Reckoning [32]. This study [52] extracts step events from various types of periodic human behaviors by carrying a smartphone with IMU through CNN. Another study [2] improves positioning accuracy by inferring the posture direction of IMU readings. DeMo leverages IMU reading to detect special road surfaces and further improves speed accuracy, and provides reliable positioning service.

7 Discussion

Limitations. (1) On-trolley sensor failure. We plan to design more reliable hardware with additional protection methods to ensure good device reliability. (2) Bypassing DeMo. DeMo covers 88% of the total deliveries. We will analyze delivery behaviors to understand possible ways of bypassing DeMo and further adopt corresponding mitigation to offer seamless monitoring coverage. (3) Delivery worker dissatisfaction. Although DeMo offers accurate violation detection as demonstrated by controlled experiments and feedback from MTR staff (Sec. 4.5), our small-scale interviews suggest that delivery workers perceived DeMo as a surveillance system, and so it was not received favorably.

Future Improvement and Deployment. We plan to improve DeMo with the following aspects. (1) Automatic parameter tuning. Delivery trolleys are generally still in front of the destination shops for 5 to 20 minutes when offloading products. This opportunity could be leveraged for fine-tuning our RSSI-distance model. (2) Future hardware design and deployment. We will improve our hardware reliability under rough handling and deploy DeMo in 10 more MTR stations.

8 Conclusion

Our experiences with DeMo demonstrate the feasibility of monitoring indoor delivery. Dedicated designs are essential to combat unique challenges and ensure reliable violation detection. DeMo outperforms the prior manual system for better coverage, effective change of human behaviors, better efficiency, and cost reduction. We hope that these lessons will contribute to future delivery monitoring systems.

9 Acknowledgements

We thank our shepherd and the anonymous reviewers for their insightful comments. This work was substantially supported by NSF China 62102332, CityU 21216822, CityU APRC 9610491, and CityU 11206023.

References

[1] P. accident on escalator. Here's why you really shouldn't wheel strollers and trolleys onto escalators. https://forums.hardwarezone.com.sg/threads/heres-why-you-really-shouldnt-wheel-strollers-trolleys-onto-escalators.6745840/, 2022.

[2] M. Atashi, P. Malekzadeh, M. Salimibeni, Z. Hajiakhondi-Meybodi, K. N. Plataniotis, and A. Mohammadi. Orientation-matched multiple modeling for rssi-based indoor localization via ble sensors. In *2020 28th European Signal Processing Conference (EUSIPCO)*, pages 1702–1706. IEEE, 2021.

[3] R. Ayyalasomayajula, A. Arun, C. Wu, A. Shaikh, S. Rajagopalan, Y. Hu, S. Ganesaraman, C. J. Rossbach, A. Seetharaman, E. Witchel, et al. LocAP: Autonomous millimeter accurate mapping of WiFi infrastructure. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1115–1129, 2020.

[4] R. Ayyalasomayajula, A. Arun, C. Wu, S. Sharma, A. R. Sethi, D. Vasisht, and D. Bharadia. Deep learning based wireless localization for indoor navigation. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.

[5] P. Bahl and V. N. Padmanabhan. Radar: An in-building rf-based user location and tracking system. In *Proceedings IEEE INFOCOM 2000. Conference on computer communications. Nineteenth annual joint conference of the IEEE computer and communications societies (Cat. No. 00CH37064)*, volume 2, pages 775–784. Ieee, 2000.

[6] T. A. Bentley. Slip, trip and fall accidents occurring during the delivery of mail. *Ergonomics*, 41(12):1859–1872, 1998.

[7] A. Bose and C. H. Foh. A practical path loss model for indoor wifi positioning enhancement. In *2007 6th International Conference on Information, Communications & Signal Processing*, pages 1–5. IEEE, 2007.

[8] A. Brajdic and R. Harle. Walk detection and step counting on unconstrained smartphones. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*, pages 225–234, 2013.

[9] V. Chandel, N. Ahmed, S. Arora, and A. Ghose. Inloc: An end-to-end robust indoor localization and routing solution using mobile phones and ble beacons. In *2016 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, pages 1–8. IEEE, 2016.

[10] D. Chen, K. G. Shin, Y. Jiang, and K.-H. Kim. Locating and tracking ble beacons with smartphones. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 263–275, 2017.

[11] Y. Chen, D. Lymberopoulos, J. Liu, and B. Priyantha. Fm-based indoor localization. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 169–182, 2012.

[12] Z. Chen, Z. Li, X. Zhang, G. Zhu, Y. Xu, J. Xiong, and X. Wang. Awl: Turning spatial aliasing from foe to friend for accurate wifi localization. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 238–250, 2017.

[13] K. Chintalapudi, A. Padmanabha Iyer, and V. N. Padmanabhan. Indoor localization without the pain. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking*, pages 173–184, 2010.

[14] I. delivery accident. Courier smashes passerby into paraplegia with escalator delivery. https://www.sohu.com/a/524951955_120823584, 2022.

[15] DeMo. IMU and BLE readings. https://github.com/Starry102/DeMo, 2023.

[16] DeMo. Parameters configuration. https://github.com/Starry102/DeMo/tree/main/train, 2023.

[17] J. Diebel. Representing attitude: Euler angles, unit quaternions, and rotation vectors. *Matrix*, 58(15-16):1–35, 2006.

[18] Y. Ding, L. Liu, Y. Yang, Y. Liu, D. Zhang, and T. He. From conception to retirement: a lifetime story of a 3-year-old wireless beacon system in the wild. *IEEE/ACM Transactions on Networking*, 30(1):47–61, 2021.

[19] Y. Ding, Y. Yang, W. Jiang, Y. Liu, T. He, and D. Zhang. Nationwide deployment and operation of a virtual arrival detection system in the wild. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 705–717, 2021.

[20] P. M. Djuric, J. H. Kotecha, J. Zhang, Y. Huang, T. Ghirmai, M. F. Bugallo, and J. Miguez. Particle filtering. *IEEE signal processing magazine*, 20(5):19–38, 2003.

[21] L. D. Driver. Liquor delivery driver recovers $ 445,000 for fall on strip mall macadam. https://www.clarklawnj.com/liquor-delivery-driver-recovers-445000-for-fall-on-strip-mall-macadam/, 2020.

[22] F. T. Experience. Gatwick's beacon installation provides augmented reality wayfinding. https://www.futuretravelexperience.com/on-the-ground/wayfinding-and-passenger-services/, 2017.

[23] R. Faragher and R. Harle. Location fingerprinting with bluetooth low energy beacons. *IEEE journal on Selected Areas in Communications*, 33(11):2418–2428, 2015.

[24] C. Feng, W. S. A. Au, S. Valaee, and Z. Tan. Received-signal-strength-based indoor positioning using compressive sensing. *IEEE Transactions on mobile computing*, 11(12):1983–1993, 2011.

[25] Y. Gao, Q. Yang, G. Li, E. Y. Chang, D. Wang, C. Wang, H. Qu, P. Dong, and F. Zhang. Xins: The anatomy of an indoor positioning and navigation architecture. In *Proceedings of the 1st international workshop on Mobile location-based service*, pages 41–50, 2011.

[26] Y. Gu and F. Ren. Energy-efficient indoor localization of smart hand-held devices using bluetooth. *IEEE Access*, 3:1450–1461, 2015.

[27] D. Han, S. Jung, M. Lee, and G. Yoon. Building a practical wi-fi-based indoor navigation system. *IEEE Pervasive Computing*, 13(2):72–79, 2014.

[28] A. M. Hossain and W.-S. Soh. Cramer-rao bound analysis of localization using signal strength difference as location fingerprint. In *2010 Proceedings IEEE INFOCOM*, pages 1–9. IEEE, 2010.

[29] A. M. Hossain and W.-S. Soh. A survey of calibration-free indoor positioning systems. *Computer Communications*, 66:1–13, 2015.

[30] Y. Hu, F. Qian, Z. Yin, Z. Li, Z. Ji, Y. Han, Q. Xu, and W. Jiang. Experience: Practical indoor localization for malls. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, pages 82–93, 2022.

[31] Y. Ji, S. Biaz, S. Pandey, and P. Agrawal. Ariadne: A dynamic indoor signal map construction and localization system. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, pages 151–164, 2006.

[32] A. R. Jimenez, F. Seco, C. Prieto, and J. Guevara. A comparison of pedestrian dead-reckoning algorithms using a low-cost mems imu. In *2009 IEEE International Symposium on Intelligent Signal Processing*, pages 37–42. IEEE, 2009.

[33] C. joints. Guidance notes on panelling design and joint construction of concrete slabs. https://www.hyd.gov.hk/en/technical_references/technical_document/guidance_notes/pdf/gn020a.pdf, 2021.

[34] W. Kang and Y. Han. Smartpdr: Smartphone-based pedestrian dead reckoning for indoor localization. *IEEE Sensors journal*, 15(5):2906–2916, 2014.

[35] C.-H. Kao, R.-S. Hsiao, T.-X. Chen, P.-S. Chen, and M.-J. Pan. A hybrid indoor positioning for asset tracking using bluetooth low energy and wi-fi. In *2017 IEEE international conference on consumer electronics-Taiwan (ICCE-TW)*, pages 63–64. IEEE, 2017.

[36] A. Kludze, R. Shrestha, C. Miftah, E. Knightly, D. Mittleman, and Y. Ghasempour. Quasi-optical 3d localization using asymmetric signatures above 100 ghz. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, pages 120–132, 2022.

[37] M. Kotaru, K. Joshi, D. Bharadia, and S. Katti. Spotfi: Decimeter level localization using wifi. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 269–282, 2015.

[38] C. Li, M. Mirosa, and P. Bremer. Review of online food delivery platforms and their impacts on sustainability. *Sustainability*, 12(14):5528, 2020.

[39] M. Li, N. Liu, Q. Niu, C. Liu, S.-H. G. Chan, and C. Gao. Sweeploc: Automatic video-based indoor localization by camera sweeping. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(3):1–25, 2018.

[40] Locatify. Intuitively designed beacon based museum audio guide. https://locatify.com/blog/eldheimar-museum/, 2020.

[41] A. Mackey, P. Spachos, L. Song, and K. N. Plataniotis. Improving ble beacon proximity estimation accuracy through bayesian filtering. *IEEE Internet of Things Journal*, 7(4):3160–3169, 2020.

[42] H. A. Nguyen, H. Guo, and K.-S. Low. Real-time estimation of sensor node's position using particle swarm optimization with log-barrier constraint. *IEEE Transactions on Instrumentation and Measurement*, 60(11):3619–3628, 2011.

[43] J. Ni, F. Zhang, J. Xiong, Q. Huang, Z. Chang, J. Ma, B. Xie, P. Wang, G. Bian, X. Li, et al. Experience: pushing indoor localization from laboratory to the wild. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, pages 147–157, 2022.

[44] T. G. of the Hong Kong Special Administrative Region. Wages and labour earnings. https://www.censtatd.gov.hk/tc/scode210.html, 2022.

[45] A. on escalator. Transport goods by escalator. https://orientaldaily.on.cc/cnt/news/20160223/00176_125.html, 2022.

[46] A. Opher, A. Chou, A. Onda, and K. Sounderrajan. The rise of the data economy: driving value through internet of things data monetization. *IBM Corporation: Somers, NY, USA*, 2016.

[47] G. Palshikar et al. Simple algorithms for peak detection in time-series. In *Proc. 1st Int. Conf. Advanced Data Analysis, Business Analytics and Intelligence*, volume 122, 2009.

[48] T. paving. Design manual:barrier free access. https://www.bd.gov.hk/doc/en/resources/codes-and-references/code-and-design-manuals/BFA2008_e.pdf, 2021.

[49] J. Racko, P. Brida, A. Perttula, J. Parviainen, and J. Collin. Pedestrian dead reckoning with particle filter for handheld smartphone. In *2016 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, pages 1–7. IEEE, 2016.

[50] A. Rai, K. K. Chintalapudi, V. N. Padmanabhan, and R. Sen. Zee: Zero-effort crowdsourcing for indoor localization. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 293–304, 2012.

[51] S. Rodriguez Garzon and B. Deva. Geofencing 2.0: taking location-based notifications to the next level. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 921–932, 2014.

[52] W. Shao, H. Luo, F. Zhao, C. Wang, A. Crivello, and M. Z. Tunio. Depedo: Anti periodic negative-step movement pedometer with deep convolutional neural networks. In *2018 IEEE international conference on communications (ICC)*, pages 1–6. IEEE, 2018.

[53] G. Shen, Z. Chen, P. Zhang, T. Moscibroda, and Y. Zhang. Walkie-markie: Indoor pathway mapping made easy. In *NSDI*, volume 13, pages 85–98, 2013.

[54] S. Shen, N. Michael, and V. Kumar. Autonomous multi-floor indoor navigation with a computationally constrained mav. In *2011 IEEE International Conference on Robotics and Automation*, pages 20–25. IEEE, 2011.

[55] Y. Shu, K. G. Shin, T. He, and J. Chen. Last-mile navigation using smartphones. In *Proceedings of the 21st annual international conference on mobile computing and networking*, pages 512–524, 2015.

[56] M. L. Sichitiu and V. Ramadurai. Localization of wireless sensor networks with a mobile beacon. In *2004 IEEE international conference on mobile Ad-hoc and sensor systems (IEEE Cat. No. 04EX975)*, pages 174–183. IEEE, 2004.

[57] SINGAPORE. Death of delivery man who fell from platform at century square mall ruled a workplace misadventure. https://www.straitstimes.com/singapore/courts-crime/death-of-delivery-man-at-century-square-mall-ruled-a-workplace-misadventure, 2022.

[58] E. Soltanaghaei, A. Kalyanaraman, and K. Whitehouse. Multipath triangulation: Decimeter-level wifi localization and orientation with a single unaided receiver. In *Proceedings of the 16th annual international conference on mobile systems, applications, and services*, pages 376–388, 2018.

[59] F. Technologies. Eddystone beacon installation at indian railway stations by google. https://www.fabliantechnologies.com/eddystone-beacon-installation-at-indian-railway-stations-by-google/, 2018.

[60] THINKPROXI. Thinkproxi announces famous beale street implemented beacon technology. https://www.thinkproxi.com/thinkproxi-announces-famous-beale-street-implemented-beacon-technology/, 2017.

[61] X. Tian, R. Shen, D. Liu, Y. Wen, and X. Wang. Performance analysis of rss fingerprinting based indoor localization. *IEEE Transactions on Mobile Computing*, 16(10):2847–2861, 2016.

[62] R. v. Department of Water and Power. Making a delivery via a service elevator. https://caselaw.findlaw.com/ca-court-of-appeal/1764404.html, 1962.

[63] X. Wang, L. Gao, S. Mao, and S. Pandey. Csi-based fingerprinting for indoor localization: A deep learning approach. *IEEE transactions on vehicular technology*, 66(1):763–776, 2016.

[64] M. Werner, M. Kessel, and C. Marouane. Indoor positioning using smartphone camera. In *2011 international conference on indoor positioning and indoor navigation*, pages 1–6. IEEE, 2011.

[65] C. Wu, Z. Yang, Y. Liu, and W. Xi. Will: Wireless indoor localization without site survey. *IEEE Transactions on Parallel and Distributed systems*, 24(4):839–848, 2012.

[66] H. Wu, S. He, and S.-H. G. Chan. Efficient sequence matching and path construction for geomagnetic indoor localization. In *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks*, pages 156–167, 2017.

[67] J. Xiong and K. Jamieson. ArrayTrack: A fine-grained indoor location system. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 71–84, 2013.

[68] J. Xiong, K. Sundaresan, and K. Jamieson. Tonetrack: Leveraging frequency-agile radios for time-based indoor wireless localization. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 537–549, 2015.

[69] J. Yang, S. Sidhom, G. Chandrasekaran, T. Vu, H. Liu, N. Cecan, Y. Chen, M. Gruteser, and R. P. Martin. Detecting driver phone use leveraging car speakers. In *Proceedings of the 17th annual international conference on Mobile computing and networking*, pages 97–108, 2011.

[70] Y. Yang, Y. Ding, D. Yuan, G. Wang, X. Xie, Y. Liu, T. He, and D. Zhang. Transloc: transparent indoor localization with uncertain human participation for instant delivery. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.

[71] P. K. Yoon, S. Zihajehzadeh, B.-S. Kang, and E. J. Park. Adaptive kalman filter for indoor localization using bluetooth low energy and inertial measurement unit. In *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 825–828. IEEE, 2015.

[72] J.-H. Youn, H. Ali, H. Sharif, J. Deogun, J. Uher, and S. H. Hinrichs. Wlan-based real-time asset tracking system in healthcare environments. In *Third IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob 2007)*, pages 71–71. IEEE, 2007.

[73] M. Zhao, T. Chang, A. Arun, R. Ayyalasomayajula, C. Zhang, and D. Bharadia. Uloc: Low-power, scalable and cm-accurate uwb-tag localization and tracking for indoor applications. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 5(3):1–31, 2021.

A Appendix

A.1 Geomagnetic Field

When designing DeMo, we collected geomagnetic field (GMF) strength at different locations, including entry, exit, and open areas, within three stations. Our preliminary experiments validated significant GMF strength changes caused by operating trains. These changes are difficult to predict and eliminate. In normal indoor settings, *e.g.,* shopping malls, the geomagnetic field values are relatively stable without significant disruptions. Fig. 22 compares the collected GMF strength values at a fixed location (*e.g.,* entrance) in an MTR station and a shopping mall, respectively. As a result, we decided not to leverage geomagnetic field sensing in DeMo.

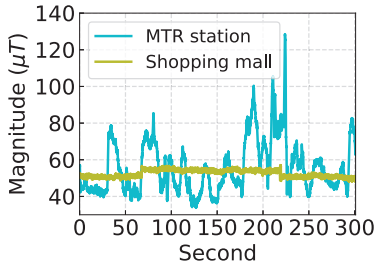


Fig. 22: Examples of GMF strength change with time at a shopping mall and MTR station.

A.2 RSSI-distance Models Comparison

Crowd obstruction and reflection cause severe signal strength degradation (or even signal loss), deviating from the traditional logarithmic model. Therefore, we instead use a polynomial function to model the effect of crowds on signal strength. Table 7 lists the average positioning accuracy of 4 different polynomial models with degrees ranging from 1 to 4. The parameters of the RSSI-distance model were trained using data from one station and applied to other stations without retraining. Models with a higher polynomial degree are prone to overfitting, while those with a lower polynomial degree exhibit unstable accuracy due to the influence of subway crowds. Consequently, we have opted for a cubic polynomial function as it delivers superior accuracy compared to other polynomial orders.

Table 7: Mean localization accuracy under different polynomial orders at two MTR stations.

Polynomial Orders	1	2	3	4
Station 1	3.16 m	3.09 m	2.81 m	3.22 m
Station 2	3.61 m	3.38 m	2.93 m	3.97 m

A.3 Positioning Accuracy Validation

DeMo's training data for its RSSI-distance model is collected in one MTR station, followed by a one-time parameter tuning process. Then we validate this RSSI-distance model across the remaining stations. To demonstrate the robustness of our model, we present the overall mean localization errors (solely relying on BLE without IMU integration) in Table 8. The validation process is conducted via control experiments, following Sec. 5.1. In most stations, the positioning accuracy closely matches the initial station used for model training. However, two subway stations exhibit poorer positioning accuracy (3.68 m and 3.83 m). Compared to most subway stations, which typically feature long, narrow delivery areas, these two stations have unique layouts with larger open spaces. This distinct layout can result in reduced positioning accuracy within those open areas. Besides, we noticed more severe beacon losses in these two stations which inevitably affect the positioning accuracy.

Table 8: Positioning accuracy in 12 stations.

Stations	Mean Accuracy (m)
Station 1	2.81
Station 2	2.55
Station 3	2.93
Station 4	2.62
Station 5	2.99
Station 6	3.39
Station 7	2.82
Station 8	2.42
Station 9	3.68
Station 10	3.83
Station 11	2.45
Station 12	2.51

A.4 Glossary of Main Parameters

Table 9 lists DeMo's main parameters, which are crucial for the system's functionality. t_{sw}, R_{th}, n and p_w are empirical parameters. Coefficients γ_0 through γ_3 are trained via simulation experiments, with the detailed training method available at the link [16]. Besides, a_{std} refers to the study [50].

A.5 Questionnaire

The questionnaire consists of six multiple-choice questions, each utilizing a 5-point scale ranging from 5 (strongly agree) to 1 (strongly disagree). The detailed questions are: 1. please select the option that best represents your level of satisfaction with the DeMo system, 2. please select the option that best describes your perception of the ease of use of the DeMo device, 3. please select the option that best represents your

Table 9: Glossary of main parameters.

Parameters	Values	Description
t_{sw}	4 s	A sliding window
R_{th}	-80 dBm	RSSI threshold
γ_0	94.8951	Coefficient of the cubic function
γ_1	5.8301	Coefficient of the cubic function
γ_2	0.0811	Coefficient of the cubic function
γ_3	0.0005	Coefficient of the cubic function
n	2, 8	# of IMU reading peaks
a_{std}	0.01 g	Standard deviation of the IMU
p_w	60%	# of weighted particles

assessment of the DeMo system's effectiveness in reducing safety violations, 4. please select the option that best reflects your perception of the DeMo system's accuracy in detecting speeds, 5. please select the option that best describes the extent to which the DeMo system has reduced your workload, and 6. please select the option that best represents how frequently you have encountered sensor damage issues with the DeMo system. The feedback from 20 safety staff in 12 MTR stations is demonstrated in Fig. 14.

AUGUR: Practical Mobile Multipath Transport Service for Low Tail Latency in Real-Time Streaming

Yuhan Zhou[*ℙ𝕋], *Tingfeng Wang*[*𝕋], *Liying Wang*[ℙ], *Nian Wen*[𝕋], *Rui Han*[𝕋], *Jing Wang*[𝕋]
Chenglei Wu[𝕋], *Jiafeng Chen*[𝕋], *Longwei Jiang*[𝕋], *Shibo Wang*[𝕏𝕋], *Honghao Liu*[𝕋†], *Chenren Xu*[ℙℤ𝕂†]
[ℙ]*School of Computer Science, Peking University*
[𝕋]*Tencent Inc.* [𝕏] *Xi'an Jiaotong University* [ℤ]*Zhongguancun Laboratory*
[𝕂]*Key Laboratory of High Confidence Software Technologies, Ministry of Education (PKU)*

Abstract – Real-time streaming applications like cloud gaming require consistently low latency, even at the tail. Our large-scale measurement based on a major cloud gaming service provider reveals that in Wi-Fi networks, the delay of the wireless hop can inflate due to its fluctuating nature, making it difficult to achieve consistently low tail latency. While cellular paths can be leveraged to alleviate the impact of wireless fluctuation of Wi-Fi paths, our user study reveals that it is crucial to constrain cellular data usage while using multipath transport. In this paper, we present AUGUR, a multipath transport service designed to reduce long tail latency and video frame stall rates in mobile real-time streaming. To address the challenge of reducing long tail latency by utilizing cellular paths while minimizing cellular data usage, AUGUR captures user characteristics by deriving state probability models and formulates the equilibrium into Integer Linear Programming (ILP) problems for each user session to determine the opportunity of frame retransmission and path selection. Our trace-driven emulation and large-scale real-world deployment in Tencent Start cloud gaming platform demonstrate that AUGUR achieves up to 66.0% reduction in tail latency and 99.5% reduction in frame stall rate with 88.1% decrease in cellular data usage compared to other multipath transport schemes.

1 Introduction

Emerging real-time streaming applications like cloud gaming [1, 2], video conferencing [3, 4], and AR/VR [5, 6] provide users with interactive experiences for both entertainment and business. Such applications have grown rapidly worldwide and made a large market (*e.g.,* The global cloud gaming market reached $1.28 billion in 2022 and expects to reach $13.6 billion by 2028 [7]). However, to provide users with a seamless interactive experience, service providers must achieve *consistently* low tail latency [8]. Based on experience and statistics from our real-time streaming service platform, we observe that a tail latency (*i.e.,* 99.9th percentile) of 200 ms can lead to frequent video frame stalls, and even a 0.5% increase in stall rate results in a 33% drop in user retention time (§2.2). Therefore, it is essential to reduce long tail latency to improve user experience.

However, for mobile devices that access wireless networks, existing solutions for latency-intensive applications fail to meet the consistently low tail latency requirements imposed by interactive real-time streaming or are impractical to be widely deployed. The most intuitive and practical method to eliminate high network latency is to deploy a dedicated congestion control algorithm (CCA). However, we observe that with Wi-Fi networks, path RTT inflation caused by random wireless fluctuations can occur. Therefore, while existing CCAs designed for real-time streaming [9, 10, 11, 12] and wireless networks [13, 14] can provide sufficient bandwidth and low median latency, they fail to consistently achieve low tail latency (§2.2). Even with a low sending bitrate (*i.e.,* 512 Kbps), the tail latency can dramatically exceed 200 ms. Besides, RTT inflation is intrinsic in wireless links. Thus, it is difficult to achieve low tail latency with only one network path. Therefore, leveraging multipath is a straightforward approach to alleviate the impact of an RTT-fluctuating path.

Although many works on mobile multipath transport have been proposed to reduce latency by leveraging both Wi-Fi and cellular paths, they are not practical to be widely deployed to reduce long tail latency for real-time streaming applications in terms of performance, cellular cost, and deployability: *i)* Most multipath schedulers only make decisions on packet departure, leaving packet retransmission to the underlying loss recovery mechanism, thus the packets would be inevitably delayed in the presence of random RTT inflation [15, 16, 17, 18, 19, 20, 21]. Therefore they cannot meet the performance requirement of consistently low tail latency; *ii)* While most multipath transport schemes only focus on the performance heterogeneity of multiple paths, they ignore the data budget heterogeneity of Wi-Fi and cellular paths [17, 22]. Since Internet service providers (ISPs) provide cellular connections with limited data budget [23], we find that users of our mobile real-time applications express strong concerns about cellular data usage (§2.3). Therefore, a multipath transport system that consumes a large amount of cellular data is not practical to be widely deployed; *iii)* Many existing multipath transport systems require modification to

*equal contribution.
†corresponding authors.
✉: coreyliu@tencent.com ✉: chenren@pku.edu.cn

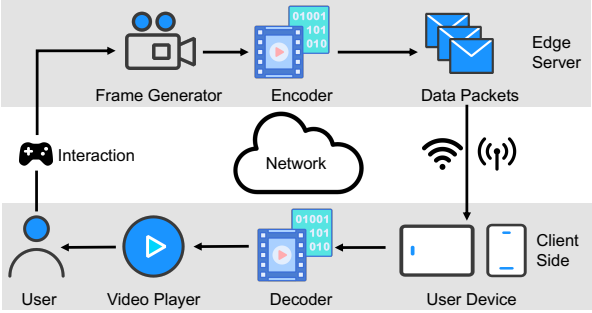


Figure 1: Overview of the real-time streaming pipeline.

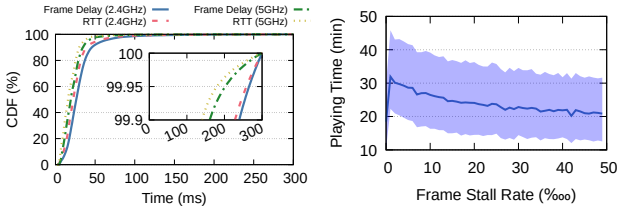


Figure 2: Long tail latency in real-time streaming.

Figure 3: Correlation between video frame stall rate and user retention time.

the kernel of user devices [16, 17, 18, 19] or rely on values that do not exist in real-time streaming (*e.g.,* volume video size [21] and playback buffer occupancy [24]) for decision-making. Consequently, they are infeasible to be widely deployed for real-time streaming applications.

In this work, we propose a multipath transport service AUGUR in mobile real-time streaming that meet the requirement of reducing long tail latency, minimizing cellular data usage, and enabling large-scale deployment. AUGUR uses the Wi-Fi path as the primary path for frame transmission and leverages the cellular path by introducing *application-level frame retransmission* and *primary path switch scheduling*. It tackles two challenges when leveraging multipath to deal with an RTT-fluctuating path: *i)* the RTT inflation caused by wireless fluctuation of the Wi-Fi path is highly unpredictable and *ii)* the usage of cellular paths is strictly limited and should be minimized as much as possible. These two challenges make it difficult to determine when to use the cellular path to alleviate the impact of RTT inflation.

To address these challenges, we observe that while user characteristics such as frame stall rate and the impact of prolonged latency vary greatly among different users, they remain stable for an individual user over a certain period of time (§2.4). Based on this observation, AUGUR divides the statistics of a user session into states and creates *per-session probability models* for each user to quantify the various user characteristics. It then formulates the equilibrium of reducing tail latency by leveraging cellular path and constraining cellular data usage as Integer Linear Programming (ILP) problems and uses the state probability models as inputs. By solving the ILP problems, AUGUR determines the appropriate moments to utilize cellular paths to achieve both the performance requirement of consistently low latency and the cost requirement of minimizing cellular usage. To achieve the deployability requirement, AUGUR is fully implemented in edge servers and requires no modification to user devices, customed hardware, or network middleboxes, making it immediately deployable.

We have deployed AUGUR in Tencent Start [25] cloud gaming platform for over six months with a wide range of millions of users, resulting in tens of millions of hours of user retention time. Our trace-driven emulation and large-scale production experiments show that AUGUR can reduce up to 66.0% tail latency and 99.5% frame stall rate with 2.7% average data usage on the cellular path, compared to other multipath transport schemes.

Contributions.

- We conduct a large-scale and in-depth statistical analysis to demonstrate the characteristics of long tail latency in mobile real-time streaming applications within a production environment;
- We propose AUGUR, a multipath transport service designed to reduce the long tail latency and frame stall while maintaining a strictly limited cellular data usage;
- We deploy AUGUR in Tencent Start cloud gaming platform and demonstrate that it can significantly reduce tail latency and stall rate while incurring negligible cellular costs.

Ethical claim. All user feedback and data statistics collected in this work are obtained with explicit permission from the users and are anonymized to protect their privacy. This work does not raise any ethical concerns and conforms to the IRB policies of the authors' institutions.

2 Background and Motivation

In this section, we introduce the background of interactive real-time streaming (§2.1). We then collect real-world statistics to present the long tail latency caused by RTT inflation in Wi-Fi path transmission (§2.2) and the limitation of existing mobile multipath transport schemes (§2.3). Finally, we discuss the user characteristics observed from our online measurement (§2.4).

2.1 Interactive Real-Time Streaming

An interactive real-time streaming service like cloud gaming provides users with an interactive video experience. As shown in Fig. 1, a typical real-time streaming pipeline consists of four components: frame generator, codec (encoder and decoder), transport system, and video player. The encoder encodes the video frames produced by the frame generator, and each frame is typically partitioned into multiple data packets for network transport. After being delivered to the user devices through the network transport system, the frame is immediately decoded and displayed by

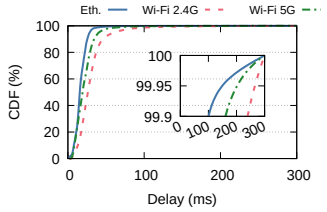
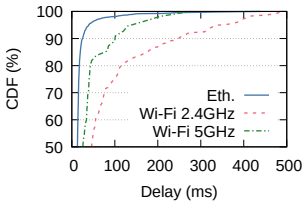
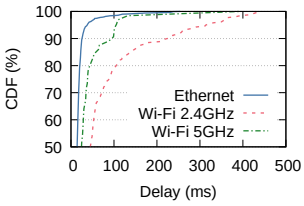


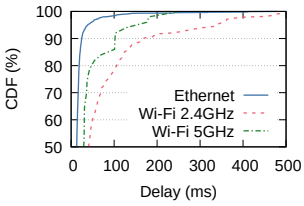
Figure 4: Frame delivery latency of wired and wireless networks.



(a) 512 Kbps.



(b) 1 Mbps.



(c) 2 Mbps.

Figure 5: Frame delivery latency over 99th percentile under low sending bitrate.

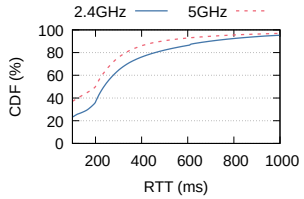


Figure 6: RTT inflation of Wi-Fi link before frame stall occurs.

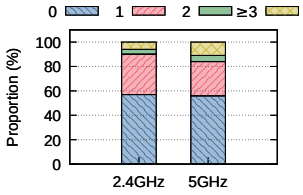


Figure 7: Packet loss rate on frame stalls with Wi-Fi link.

the video player without buffering. According to the inter-dependencies of video frames [26], all packets in a frame should be sent to the decoder to prevent video quality degradation. Furthermore, the user continuously interacts with the server, providing instructions or feedback to generate new video frames. This pipeline contrasts with buffer-based streaming applications such as video-on-demand (VoD), where the entire video content is pre-recorded, and a playback buffer in the video player can be utilized to absorb frame delivery latency [27] or improve user Quality of Experience (QoE) [24]. Therefore, to ensure a smooth and seamless interactive experience, it is crucial for the network transport system to guarantee consistently low latency for the timely delivery of all video frames, especially under fluctuating wireless network environments.

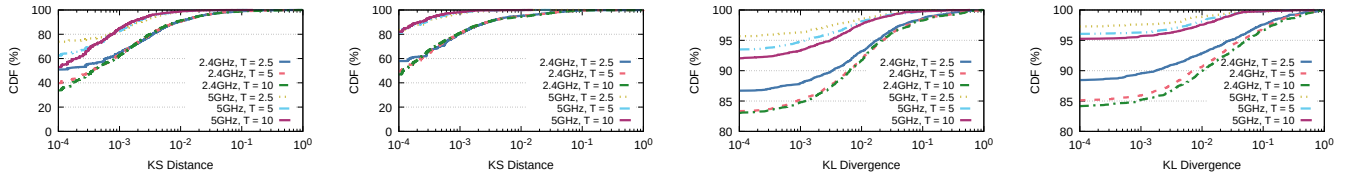
2.2 Long Tail Latency in Real-time Streaming

Long tail latency severely degrades user experience. The low latency requirement for frame delivery in real-time streaming means that even a slight increase in tail latency can have a significant impact on user experience. To investigate the impact of long tail latency, we conduct a measurement on our online cloud gaming service. As shown in Fig. 2, while the median RTT and frame delivery latency is kept below 30 ms, the tail latency (*i.e.,* 99.9th percentile) can reach over 200 ms, and such a severely delayed frame is highly likely to cause a video stall [8]. Moreover, as shown in Fig. 3, we observe that an increased video frame stall rate results in a significant decrease in user retention time, even an 0.5% increased stall rate would result in 33% drop in retention time, and there still exists an increase of user churn even the stall rate is kept below 0.1%, indicating user dissatisfaction with the service. Therefore, long tail latency severely degrades user experience, and achieving *consistently low latency* necessitates the attainment of at least a 99.9% in-time delivery of frames.

RTT inflation contributes to long tail latency instead of network congestion. Intuitively, packet loss and retransmission caused by network congestion significantly contribute to packet delivery latency [28]. However, with our private CCA designed for real-time streaming and edge servers deployed, we find that long tail latency is mainly induced by inflated path RTT due to wireless fluctuation for mobile devices. As shown in Fig. 4, compared to the wired network environment, when streaming through the wireless network connection from a Wi-Fi access point (AP) to mobile devices, the frame delivery latency at the 99th percentile is increased by up to 290%. Additionally, we plot the RTT of the wireless path *before* a frame stall occurs, Fig. 6 shows that the path RTT is already inflated to more than 200 ms for 65% of stalled frames on Wi-Fi 2.4G networks and 50% of stalled frames on Wi-Fi 5G, respectively. Furthermore, we observe that packet loss is infrequent on frame stalls. To demonstrate this, we plot the packet loss rate when frame stall happens and show the result in Fig. 7, nearly 60% of packets are delivered without loss during a frame stall. This indicates that the long tail latency is induced by the fluctuating intrinsic of the wireless path instead of network congestion.

CCAs fail to reduce latency at the tail. The most widely used method to reduce frame delivery latency in real-time streaming is to employ sophisticated CCAs. However, with inflated RTT caused by wireless fluctuation, while state-of-the-art CCAs designed for real-time streaming [9, 10, 11, 12] or wireless networks [13, 14] can reach a low median latency, they fail to ensure a consistently low tail latency. The primary goal of CCAs is to adapt to link capacity variations and promptly reduce the sending rate to eliminate queuing delay or packet loss to maintain low transport latency [29, 30, 31, 32]. Nonetheless, due to RTT inflation caused by Wi-Fi link fluctuation, the link is unable to deliver data in time. Consequently, the long tail latency cannot be addressed by deploying CCAs.

To demonstrate the limitations of CCAs under wireless link fluctuations, we conduct an experiment by enforcing



(a) P(frame stall) KS distance. (b) P(frame stall | RTT $> X$) KS distance (c) P(frame stall) KL divergence. (d) P(frame stall | RTT $> X$) KL divergence.

Figure 8: Kolmogorov-Smirnov distance and Kullback-Leibler divergence between the characteristics probability distributions of the current minute and the past T minutes during each individual user's playing time.

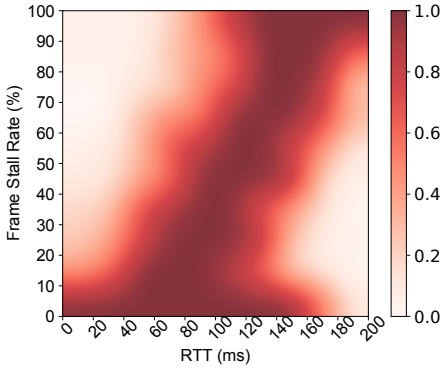


Figure 9: Overlayed curves of the relationship between frame stall rate and path RTT. Each point in each curve contributes to a point in the heatmap and the color indicates the number of points normalized by maximum.

our CCA to maintain extremely low sending bitrates that are below our video quality requirement of service (*i.e.,* 2 Mbps) and access our platform from both wired and wireless networks. As shown in Fig. 5, even with low sending bitrates, frame delivery latency at the tail remains high, and the latency distributions remain consistent across all low bitrates. This highlights that under fluctuating wireless networks, CCAs are insufficient to reduce latency at the tail for real-time streaming applications. Therefore, it is valuable to leverage cellular paths to alleviate the impact of wireless fluctuation of the Wi-Fi path.

2.3 Limitation of Existing Multipath Transport in Real-time Streaming

Multipath transport is a promising and practical solution to reduce long tail latency in mobile real-time streaming because: *i)* Wireless fluctuations are hardly correlated between heterogeneous wireless networks like Wi-Fi and cellular networks, indicating that prolonged latency induced by one path could be eliminated by another [17]; *ii)* Contemporary mobile devices are typically equipped with multiple network interfaces, enabling the large-scale deployment of a multipath transport service [24]. However, existing multipath transport schemes are not practical to be widely deployed to reduce tail latency for mobile real-time streaming in terms of performance, cellular cost, and deployability.

Multipath schedulers fail to achieve low tail latency. Although many works on multipath transport have been proposed, they mainly focus on maximizing throughput [15] or minimizing request completion time (RCT) [19, 20, 24]. Some multipath transport systems designed for latency-sensitive applications like BLEST [18] and RAVEN [17] schedule packet transmission on multiple paths but leave retransmission to the underlying loss recovery mechanism. However, packets scheduled on the Wi-Fi path could already be severely delayed before retransmission timeout (RTO) due to highly unpredictable RTT inflation, thus they cannot meet the performance requirement of low tail latency. XLINK [24] uses packet re-injection to reduce frame delivery latency, but it assumes a known video chunk size and requires playback buffer occupancy level as QoE signal, which cannot be used in real-time streaming.

Minimizing cellular data usage is crucial for practical deployment. Multipath schedulers designed for interactive applications like RAVEN [17] and ReMP [33] utilize cellular paths for redundant frame transmission but are unaware of cellular data cost. However, video frame delivery consumes a large amount of bandwidth and can lead to significant cellular data usage. To investigate users' concerns about cellular data cost, we explicitly deliver a questionnaire within our cloud gaming application to 1,251,420 users. All user feedback and data statistics collected are obtained with explicit permission and are anonymized to protect their privacy. Our analysis of online users' feedback reveals: *i)* While accessing our gaming platform with mobile devices, 89.8% of users prefer Wi-Fi connection due to the lower cost compared to cellular networks; *ii)* 57.0% of users are not willing to consume cellular data for streaming service; *iii)* For the remaining 43.0% of users willing to use cellular networks for better performance, 62.9% of them express a strong desire to reduce cellular data costs Therefore, we argue that for a practical multipath transport service, the cellular cost constraint should not be an incidental concern, but rather a primary design consideration. Although some multipath transport schemes like MP-DASH [21] and COM [34] reduce cellular data usage for VoD applications, they require the size of the pre-recorded video content, thus cannot be

used in real-time streaming.

Kernel-based multipath transport schemes are infeasible to widely deploy. Many existing multipath transport schemes require modifications to the kernel of user devices [16, 17, 18, 19, 21]. As a cloud gaming service provider, it is not feasible to make modifications to user devices, network middleboxes, or customed hardware. Therefore, they are impractical to be widely deployed.

2.4 User Characteristics in Real-time Streaming

Network characteristics vary among different users. Our real-time streaming platform is utilized by millions of users across a wide range of regions with different user device models and network environments. According to our measurements, there is a significant characteristics variation among users. We study the correlation between the frame stall rate and path RTT from different user sessions in our measurement. We plot the conditional probability $P(\text{frame stall} \mid \text{RTT} > X)$ curves and overlay all curves to formulate a heatmap. As shown in Fig. 9, while higher latency does increase the severely-tail rate overall, the correlation between them varies among different users. For example, when path RTT reaches 100 ms, the probability of a frame stall can range from 0% to 100% for different users. This indicates that RTT inflation has different impacts on different users and such variation in user characteristics should be considered for a widely deployed system.

Network characteristics remain stable for individual users. Although there is significant variation among user characteristics, we observe that there is some stability of statistics for an individual user over a certain period of time. While characteristics such as frame stall rate and network latency pattern can vary significantly among different users, they tend to remain relatively stable for an individual user over a time window. To demonstrate the stability, we computed the Kolmogorov-Smirnov (KS) distance [35] and Kullback-Leibler (KL) divergence [36] between the probability distributions of two characteristics, namely $P(\text{frame stall})$ and $P(\text{frame stall} \mid \text{RTT} > X)$, for both the current minute and the previous T minutes of each individual user's playing time and plot the CDFs of both the KS distance and KL divergence values for over 3000 users in Fig. 8. The results indicate that nearly 99% of the current minute's characteristics probability distributions are similar to those of the previous time window (*i.e.,* KS distance and KL divergence values are less than 0.1), suggesting a strong correlation between statistics calculated from a previous time window and the current minute. We leverage such stability to capture each user's characteristics in our design.

3 AUGUR Design

In this section, we propose a multipath transport service AUGUR to reduce the long tail latency induced by wireless link RTT inflation while minimizing cellular data usage. We first present the design goals of AUGUR.

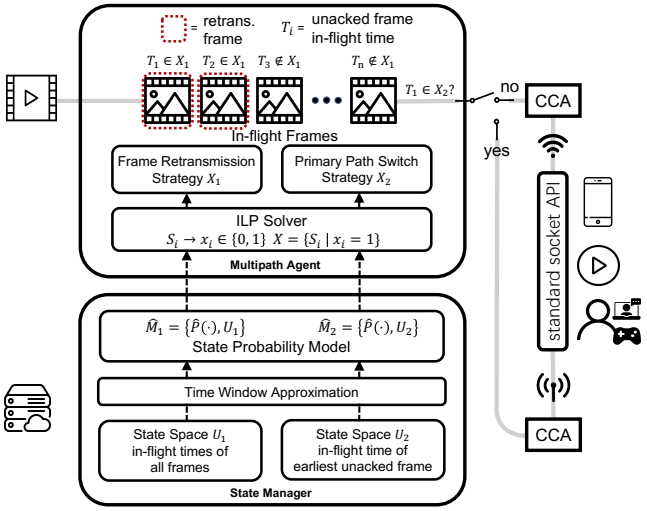


Figure 10: System overview of AUGUR.

3.1 Design Goals

The main idea behind AUGUR is to use the Wi-Fi path as the primary path to stream video frames and leverage the cellular network as a backup path when the Wi-Fi connection suffers wireless fluctuations. In light of the observation presented in §2, we design AUGUR to achieve three goals.

G1: Reduce long tail latency induced by wireless fluctuation. As presented in §2.2, the long tail latency resulting from Wi-Fi wireless fluctuation like RTT inflation is an inherent issue of the Wi-Fi path, thus it is valuable to also utilize the cellular path for data streaming. However, it is challenging to determine the opportunity of using another path because the RTT inflation of the Wi-Fi path is highly unpredictable.

G2: Constrain cellular data usage. One straightforward approach to leveraging a redundant path would be to replicate all frames on cellular paths, but this would incur a significant consumption of cellular data and violates the user concerns in §2.3. Moreover, with the significant imbalance in link capacity [37], the cellular link may be unable to handle the additional traffic from replicating all frames (*e.g.,* cellular bufferbloat [38]). Therefore AUGUR must carefully constrain the usage of the cellular path.

G3: Enable large-scale deployment. Our real-time streaming platform provides service to millions of users with various device models, operating systems, and Wi-Fi AP models. As a streaming service provider, it is not feasible to make modifications to user device kernels, network middleboxes, or customized hardware. To be practically deployed on a large scale, AUGUR should only require modifications to game servers and user applications.

3.2 System Overview

To address the challenge of reducing long tail latency by utilizing cellular paths while adhering to a strict cellular data usage constraint, AUGUR first leverages our observation on user characteristics (§2.4) and derives *per-session probability models* to capture the characteristics of each user. It then formulates the equilibrium of tail latency reduction and cellular cost minimization into Integer Linear Programming (ILP) problems. Using the measured probability models as inputs, AUGUR solves ILP problems to determine the opportunity to use the cellular path. As a multipath transport service, AUGUR leverages the cellular path through two approaches: *i) application-level frame retransmission.* AUGUR monitors in-flight frames sent on the Wi-Fi path and promptly retransmits frames that are likely to be severely delayed on the cellular path; *ii) primary path switch scheduling.* AUGUR transiently switches the primary path to cellular and streams all newly generated video frames through it in the presence of severe Wi-Fi path capacity degradation. As illustrated in Fig. 10, AUGUR consists of two main components: the *state manager* (§3.5) monitors the capacity of the Wi-Fi and cellular path and approximates probability models for each user. Based on the models provided by the state manager, the *multipath agent* (§3.6) continuously solves the formulated ILP problems to obtain cellular path utilization strategies, and performs frame retransmission and primary path switch.

In the rest of this section, we first introduce our state probability model for real-time streaming in §3.3 and our strategy for utilizing the backup path in §3.4. We then demonstrate how we apply our theoretical analysis into practice to derive the model (§3.5) and make decisions on frame retransmission and primary path switch(§3.6).

3.3 State Probability Model

In a real-time streaming session, a sender and a receiver communicate over a network path along a timeline. The sender continuously sends data, such as video frames, to the receiver. Concurrently, the receiver provides feedback (ACKs) to the sender regarding the data's arrival or at specific time intervals. The sender can then derive some statistics based on this feedback (*e.g.,* RTT, acknowledged bytes). To effectively utilize these statistics and extract vital information for the sender to determine the transmission pattern, we divide them into several *states*. We let $U = \bigcup_i \{S_i\}, \forall i \neq j, S_i \cap S_j = \emptyset$ donate all possible states. For example, if the sender chooses to use `(RTT, Bandwidth)` as states, U would be a two-dimensional space and each point in the space represents a state. For each session, the probability distribution of states $P(S_i)$ could be significantly different due to the great variety of user characteristics.

When the receiver experiences a frame stall, the next feedback can carry information about the stall event, allowing the sender to record it. If we define $stall$ to be the stall events across all states of a session, the overall frame stall probability for a user session would be:

$$P(stall) = P(stall, U) = \sum_{S_i \in U} P(stall, S_i)$$
$$= \sum_{S_i \in U} P(stall \mid S_i) P(S_i) \quad (1)$$

The distributions of $P(stall)$, $P(stall \mid S_i)$, and $P(S_i)$ can provide insight into the condition of the receiver and the network for a given user session. We define the state probability model for a user session as $M = \{P(stall), P(stall \mid S_i), P(S_i)\}, S_i \in U$, and the relationship between these distributions is given in equation Eqn. 1. By obtaining M, we can capture and quantify the user characteristics of a session.

3.4 Backup Path Utilization Strategy

To reduce long tail latency and frame stalls of the primary path, the sender can utilize a backup path for frame transmission. A backup path utilization strategy decides when the backup path should be used. Ideally, a backup path should only be used when frames are likely to be delayed on the primary path. However, it is difficult to accurately predict if a frame would be delayed and cause a stall before it is acknowledged. To effectively use the backup path, we break down the problem of deriving an optimal utilization strategy into two tasks:

T1: Covering delayed frames caused by wireless fluctuation. The backup path utilization strategy should minimize the false negative (FN) rate of the decisions by ensuring that it utilizes the backup path to cover delayed frames to the greatest extent possible.

T2: Reducing the utilization of backup paths. The backup path should only be used when necessary, as excessive utilization of the backup path can result in additional data usage and negative impacts (*e.g.,* increased cellular data charges in our case). The strategy should aim to minimize the false positive (FP) rate of the decisions.

As explained in §3.3, we divide the sender statistics of a real-time streaming process into states. We use these states as decision points. For each state S_i, we employ x_i to indicate whether the backup path should be utilized for frame transmission to mitigate delivery delay, where $x_i = 1$ denotes a positive decision and $x_i = 0$ denotes a negative decision. A *strategy* refers to a selection of several states from U, where the chosen states are represented as $X = \{S_i \in U \mid x_i = 1\}$, and the remaining states are represented as $\bar{X} = \{S_i \in U \mid x_i = 0\}$. We next derive the optimal backup path utilization strategy based on the state probability model M.

To achieve **T1**, we present the false negative rate of strat-

egy X to be:

$$P(FN) = P(\bar{X} \mid stall) = \frac{P(stall, \bar{X})}{P(stall)} = \frac{P(stall) - P(stall, X)}{P(stall)} = \frac{P(stall) - \sum_{S_i \in X} P(stall, S_i)}{P(stall)} = \frac{P(stall) - \sum_{S_i \in X} P(stall \mid S_i) P(S_i)}{P(stall)} \quad (2)$$

As discussed in §2.4, the stall rate of a single user session tends to remain stable in a time window. Therefore, we assume that in a certain period of time, the overall frame stall probability $P(stall)$ is a constant value. In this case, the minimization of $P(FN)$ turns into the maximization of function:

$$f(X) = \sum_{S_i \in X} P(stall \mid S_i) P(S_i) \quad (3)$$

This maximization function implicitly assumes that the capacity of the backup path is adequate for mitigating prolonged latency, which may not always be the case. Therefore, we introduce $P_{backup}(S_i)$ to represent the probability that the backup path is sufficient to perform frame transmission at state S_i. The maximization function in Eqn. 3 is modified as follows:

$$F(X) = \sum_{S_i \in X} P(stall \mid S_i) P(S_i) P_{backup}(S_i) \quad (4)$$

To achieve **T2**, we present the false positive rate of strategy X to be:

$$P(FP) = P(X \mid !stall) = \frac{P(X, !stall)}{P(!stall)} = \frac{P(X) - P(X, stall)}{1 - P(stall)} = \frac{P(X) - f(X)}{1 - P(stall)} \quad (5)$$

Given that the primary objective of **T1** is to maximize $f(X)$ while assuming a constant value of $P(stall)$, our aim is to minimize $P(FP)$ by restricting the value of $P(X)$ below a specific threshold. The threshold represents the data usage limit of the backup path and should be adaptable to different specified budgets. Thus, we limit the backup path utilization rate to be less than or equal to the frame stall rate multiplied by a parameter δ:

$$P(X) = \sum_{S_i \in X} P(S_i) \leq \sum_{S_i \in U} P(stall \mid S_i) P(S_i) \cdot \delta \quad (6)$$

With the maximization target (Eqn. 4), the constraint (Eqn. 6), and the probability model M, we can formulate the backup path utilization strategy derivation as an Integer Linear Programming problem:

$$\begin{aligned} &\text{Maximize} \sum_{S_i \in U} P(stall \mid S_i) P(S_i) P_{backup}(S_i) \cdot x_i \\ &x_i = 0, 1 \\ &\text{Subject to} \sum_{S_i \in U} P(S_i) \cdot x_i \leq \sum_{S_i \in U} P(stall \mid S_i) P(S_i) \cdot \delta \end{aligned} \quad (7)$$

In Eqn. 7, the maximization term aims to achieve objective **T1**, which is to cover delayed frames, and the constraint term aims to achieve objective **T2**, which is to minimize the data usage of the backup path. By solving the ILP problem, the sender can obtain a strategy X and determine when to use the backup path.

3.5 State Manager

The state manager continuously monitors the capacity of the Wi-Fi and cellular path for each user session. It creates and manages two state probability models for the multipath agent, one for the decision on frame retransmission and the other for the primary path switch. Nonetheless, we encounter two challenges in practice: *i)* What information should be extracted from sender statistics to form a state space; *ii)* While the overall frame stall rate is assumed to be a constant value, how can we obtain the distribution of $P(stall \mid S_i)$ and $P(S_i)$.

3.5.1 *State space formulation.*

In principle, it is possible to extract all available information from sender statistics to form a high-dimensional state space. However, we take two factors into consideration when deciding on the state space formulation: *i)* A multi-dimensional state space increases the difficulty and complexity of maintaining a probability model and deriving a backup path utilization strategy based on it; *ii)* Some information, such as RTT and acknowledged bytes, is already utilized by CCAs to maintain low median latency, making it unnecessary to include them in the state space. Therefore, we choose to use a single-dimension state space for both probability models.

Frame retransmission state space. We utilize the in-flight time of all frames to create a state space U_1 for deciding when to retransmit frames on the cellular path. Specifically, we calculate the in-flight time of a video frame once it is acknowledged, and this value is used to represent a point in the state space. We choose to use the per-frame in-flight time as the input state because it allows us to derive a probability model that can indicate the correlation between the stall rate and the frame in-flight time. This, in turn, provides us with the necessary information to promptly retransmit frames on the backup path.

Primary path switch state space. To derive the state probability model for the primary path switch decision, we use the in-flight time of the earliest unacknowledged frame to form a state space U_2. This choice reflects the frame delivery delay

of the primary path, providing clues on the degradation of the primary path's capacity and the chance for a switch. In practice, the state manager periodically inspects the in-flight time of all unacknowledged frames and sets the in-flight time of the earliest dispatched frame as the current state.

Cellular path capacity monitoring. Since AUGUR uses the cellular as the backup path, which can also be susceptible to fluctuations [39, 40, 41] and lead to prolonged latency, the state manager needs to monitor the characteristics of the cellular path to estimate its capacity and determine whether it is sufficient to perform frame transmission. As we use the in-flight time of frames as input states, at state S_i, the frame has been sent for S_i time. Therefore, to perform effective retransmission or primary path switch to reduce long tail latency, the RTT of the cellular path should not exceed $T_{thresh} - S_i$. Since a frame delivery latency greater than 200 ms is highly likely to cause a video stall [8], we set T_{thresh} to be 200 ms. Hence, we evaluate the RTT as a metric of the backup path capacity using the following equation:

$$P_{backup}(S_i) = P(\text{RTT}_{cell} \leq T_{thresh} - S_i) \quad (8)$$

However, we cannot obtain $P(\text{RTT}_{cell} \leq T_{thresh} - S_i)$ passively from receiver feedback because the cellular path acts as a backup, and the feedback is intermittent. Therefore, the state manager actively probes the cellular path by periodically sending an 8-byte PING message (with an interval of 50 ms) to monitor the RTT. Since the backup cellular path is infrequently used and other applications using cellular interfaces (*e.g.,* voice call, video conferencing) are typically inactive during cloud gaming sessions, small-sized packets are efficient for probing the path RTT. The overhead of the probe messages is less than 0.06% in theory, compared to the high frame rate at which large-sized video frames (typically over 4KB) are transmitted (*e.g.,* 60 fps).

3.5.2 Probability distribution derivation

The state probability model M contains the distributions of $P(stall \mid S_i)$ and $P(S_i)$. However, they are difficult to accurately describe. Firstly, these distributions could be arbitrary over time for different users. Secondly, the state manager cannot predict future states or stalls, and can only deduce an approximate distribution based on recorded states. To address this challenge, we leverage our observation discussed in §2.4 that the user characteristics remain stable for an individual user over a period of time. Based on this, we assume that the probability distributions of states remain fixed for a time window. Within a time window Δ, we can obtain the frequency of each state S_i and the corresponding stall event, denoted as $\hat{P}(S_i)$ and $\hat{P}(stall \mid S_i)$. We use these frequencies to approximate the probability distribution of the states in this time window. The approximated stall rate $\hat{P}(stall)$ can be derived by Eqn. 1.

In this way, the state manager continuously updates the states explained in §3.5.1. For each time window Δ, it creates and maintains two approximated state probability models $\hat{M}_{1,2} = \{\hat{P}(stall), \hat{P}(stall \mid S_i), \hat{P}(S_i)\}, S_i \in U_{1,2}$ with two different state spaces. The value of Δ reflects the level of fluctuation in a single user's characteristics, and its effect on AUGUR is evaluated in §5.4.

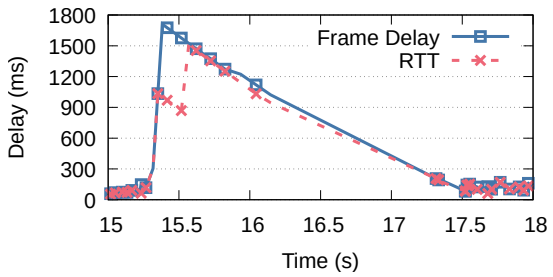


Figure 11: Wi-Fi capacity degradation causes bursty frame stalls lasting for hundreds of milliseconds.

3.6 Multipath Agent

The multipath agent plays a critical role in deciding whether to retransmit in-flight frames on the cellular path and whether to switch the primary path from Wi-Fi to cellular. To make these decisions, it utilizes the state probability models $\hat{M}_1$ and $\hat{M}_2$ provided by the state manager and plugs them into the ILP problem described in Eqn. 7. It is typically time-consuming and computation-intensive to solve an ILP problem. Nevertheless, when considering our single-dimensional state spaces $U_{1,2}$, and utilizing RTT-based backup path capacity specified in Eqn. 8, instead of looping each combination of states, we only need to determine a threshold in the continuous state space to find an optimal strategy. Therefore, the ILP problem in Eqn. 7 transforms into a linear-time solution, facilitating quick resolution. Based on the output of the simplified ILP problems, the multipath agent obtains the necessary decisions to reduce long tail latency and minimize cellular cost.

3.6.1 Application Level Frame Retransmission

Existing multipath schedulers usually only decide which path to use for delivering incoming new data at packet departure time. As presented in §2.2, unpredictable RTT inflation of wireless links could severely delay the scheduled packets. Therefore, to reduce long tail latency, it is important to track the in-flight data and actively retransmit it when it is at risk of being delayed.

In order to eliminate as much tail latency as possible, AUGUR introduces application-level frame retransmission by promptly retransmitting in-flight frames on the cellular path to prevent delays. We choose to retransmit the entire frame instead of individual data packets because: *i)* An RTT inflation at the wireless last hop would typically result in delays for the majority of packets within a frame (*e.g.,* we observe that when the frame delivery delay surpasses 200 ms, the delay of the initial packet in the frame exceeds 200 ms in 71.8% of cases), making it reasonable to actively retransmit the

entire frame through an alternative path; *ii)* Implementing frame retransmission in userspace is straightforward, rendering it a portable and easily deployable solution across a wide range of scenarios.

Naturally, there is a strong relationship between the in-flight time of an unacknowledged frame and the possibility the frame would be severely delayed, and such a relationship is quantified by our state probability model $\hat{M}_1$, which uses frame in-flight time as its state space. Consequently, as shown in Fig. 10, while the state manager keeps updating the probability model $\hat{M}_1$ during the streaming process, the multipath agent continuously solves the simplified ILP problem using $\hat{M}_1$ as input and obtains a decision X_1. It constantly monitors all unacknowledged frames and retransmits those whose in-flight time T matches with a state in X_1 (*i.e.,* $T = S_i \in X_1$). By doing so, AUGUR can proactively rescue frames that are at risk of stalling while also adhering to limited usage of the backup path. We evaluate the effectiveness of application-level frame retransmission in §5.4.

3.6.2 Primary Path Switch Scheduling

Unlike existing multipath schedulers that schedule paths for each data packet, AUGUR chooses the Wi-Fi path as the primary path and schedules frames on it by default, while the cellular path should only be when necessary to rescue delayed frames. However, in practice, we observe that severe Wi-Fi capacity degradation can occur occasionally. As demonstrated in Fig. 11, such degradation results in bursty frame stalls that last for hundreds of milliseconds, which can have a significant impact on user experience.

To address this issue, the multipath agent can temporarily switch the primary path to cellular and stop sending new frames on the degraded Wi-Fi path. Similar to the frame retransmission decision procedure, the multipath agent continuously solves the simplified ILP problem using $\hat{M}_2$ provided by the state manager and obtains a decision X_2. It initiates a primary path switch when two conditions are met: *i)* the in-flight time of the earliest unacknowledged frame T_1 matches with a state in X_2 (*i.e.,* $T_1 = S_i \in X_2$) and *ii)* the RTT of the cellular path is lower than that of the Wi-Fi path. In this scenario, AUGUR sends all newly generated frames through the cellular path and stops deriving backup path utilization strategies. In addition, the multipath agent promptly retransmits all in-flight frames of the Wi-Fi path through the cellular path since they are likely to experience prolonged delays upon detection of capacity degradation. However, directly using the cellular path as the primary path violates our cellular data limit rule, and since the capacity degradations of the Wi-Fi path are transient, it is necessary to switch back to Wi-Fi as soon as possible. While new frames are being sent through the cellular path, the sending queue of the Wi-Fi path can be drained, and we start sending probe frames on the Wi-Fi path when there are fewer than two in-flight frames on the Wi-Fi path to detect any capacity improvements. Once the capacity of the Wi-Fi path recovers (*i.e.,* the Wi-Fi path RTT is not in X_2), the multipath agent switches the primary path back to Wi-Fi and restores the simplified ILP problem-solving process. The effectiveness of primary path transition is evaluated in §5.4.

4 Trace-driven Emulation

In this section, we evaluate AUGUR in an emulation environment with real-world traces to compare it with other multipath transport schemes. Based on the results of our emulation, we further conduct large-scale experiments on our cloud gaming platform (§5).

4.1 Evaluation Methodology

Testbed. We use *mpshell* [42], a multipath extension of Mahimahi [43] for network emulation. We develop a testbed framework to implement the streaming pipeline in Fig. 1 with approximately 6000 lines of Python code. To demonstrate the cooperation between AUGUR and CCAs, we also implement two CCAs designed for real-time streaming including Salsify [11] and SQP [12].

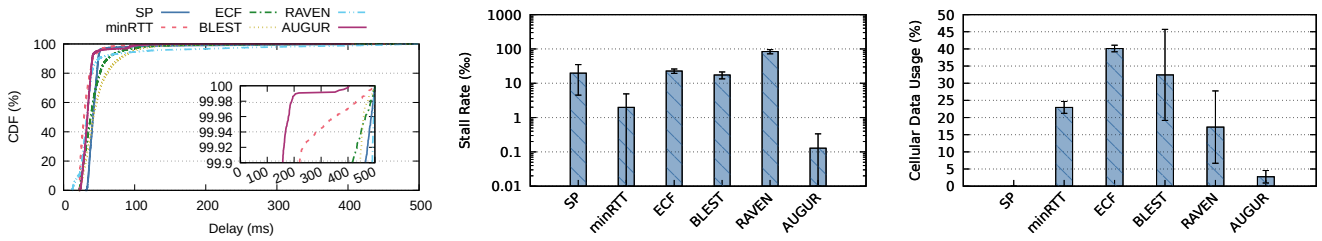
Trace collection. We collect Wi-Fi and cellular link traces based on the running logs of our user sessions. The logs contain the *user-perceived* wireless network bandwidth, and RTT inflation caused by wireless fluctuation would lead to a sudden decrease in user-perceived bandwidth. To evaluate the effectiveness of AUGUR in the presence of wireless fluctuations, we filter out traces that were either too brief (less than 10 minutes) or too consistent (with no RTT inflation). Each trace is replayed for more than 10 minutes.

Baseline. We compare AUGUR with the following multipath transport schemes as baselines:

- Single Path (SP): all video frames are streamed exclusively through the Wi-Fi path;
- minRTT [16]: the default multipath scheduler of MPTCP, which schedules packets through the path with the lowest estimated RTT;
- ECF [19]: it utilizes some relevant information (*e.g.,* `cwnd` value) of a path besides RTT to provide available aggregate bandwidth of all paths. It assumes that the underlying CCAs are congestion-window-based and require `cwnd` values for decision-making. For the rate-based CCAs we use, we derive `cwnd` value by multiplying the sending rate and the estimated RTT.
- BLEST [18]: it aims to avoid HoL-blocking and spurious retransmissions by controlling the buffer blocking. We provide it with the `cwnd` value in the same way as described above.
- RAVEN [17]: it replicates packets on multiple paths when confidence about network latency predictions is low to reduce latency.

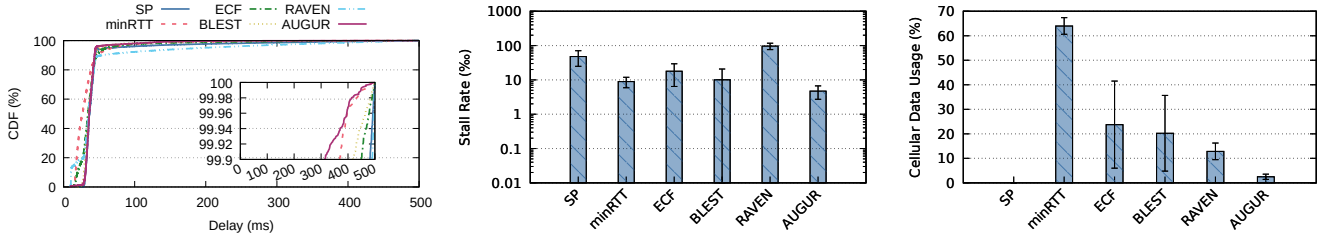
4.2 Performance

Frame delivery delay and stall rate. We measured the frame delivery delay for all multipath transport schemes with



(a) Frame delivery delay. (b) Frame stall rate. (c) Cellular data usage.

Figure 12: AUGUR performance cooperating with Salsify.



(a) Frame delivery delay. (b) Frame stall rate. (c) Cellular data usage.

Figure 13: AUGUR performance cooperating with SQP.

different CCAs. As shown in Fig. 12a and Fig. 13a, while the median frame delivery delays are similar for all multipath schemes, AUGUR effectively reduces tail latency and outperforms other multipath schemes for both CCAs. Compared to only using Wi-Fi for frame transmission, AUGUR reduces the 99.9th percentile latency by 66.0% and 35.4% for Salsify and SQP, respectively. As a result, AUGUR effectively reduces the frame stall rate, as shown in Fig. 12b and Fig. 13b. Among all multipath schemes, AUGUR achieves the lowest average frame stall rate, and compared to using only the Wi-Fi path, AUGUR reduces the stall rate by 99.5% and 91.5% for Salsify and SQP, respectively. This demonstrates that AUGUR cooperates with different CCAs and effectively reduces long tail latency in real-time streaming.

Cellular data usage. To evaluate AUGUR's ability to reduce cellular data usage, we calculate the ratio of bytes transmitted on the cellular path to the total transmitted bytes for all multipath schemes. Our results, as shown in Fig. 12c and Fig. 13c, indicate that AUGUR achieves the lowest cellular data usage among all multipath schemes, with only 2.7% and 2.3% cellular data usage for Salsify and SQP, respectively. It is worth noting that compared to minRTT, which can also effectively reduce the frame stall rate, AUGUR reduces cellular data usage by 88.1% and 96.4% for Salsify and SQP, respectively. These findings demonstrate that AUGUR can effectively reduce cellular data usage without compromising performance, making it a viable option for deployment in commercial platforms.

Multipath scheme	SP	minRTT	AUGUR
Num. of user sessions	3974	4167	3699
Total		11840	

Table 1: Large-scale experiment setup.

5 Large-scale Deployment in the Wild

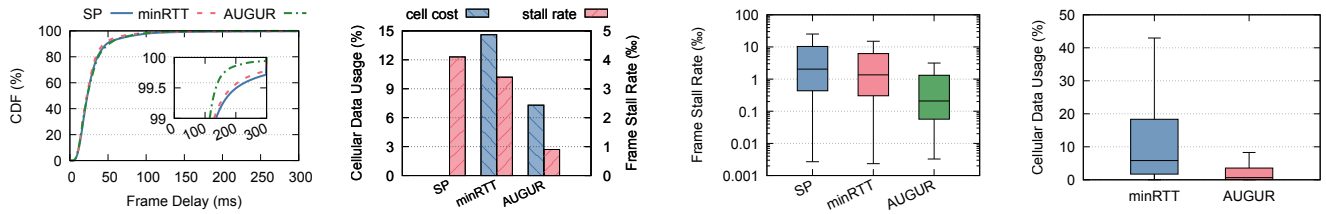
We further deploy AUGUR in Tencent Start cloud gaming server clusters to evaluate its effectiveness in reducing long tail latency and constraining cellular data usage.

5.1 Implementation and Deployment

We implement AUGUR in our cloud gaming platform fully in the userspace. The streaming application on user devices initiates separate connections to our servers with two sockets to leverage the Wi-Fi and the cellular path. Regarding the cellular budget, users are required to set a per-month limit in MB, and AUGUR calculates the corresponding value of δ based on the average stall rate and playing time of all users. For instance, a limit of 500 MB per month would result in a δ value of 3.5. We set the default value for the time window Δ to be 5 min. We have deployed AUGUR for over six months and served millions of users with over ten million hours of playing time.

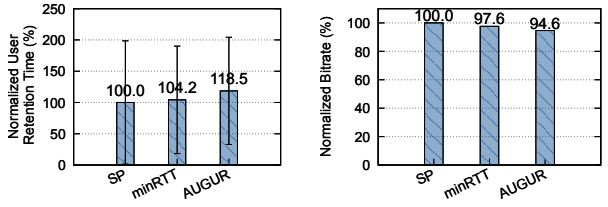
5.2 Experiment Setup

We conduct large-scale A/B tests to evaluate AUGUR in our platform. In addition to AUGUR and single-path Wi-Fi (SP), we also deployed minRTT [16] based on our emulation results in §4.2. We initiate SP, minRTT, and AUGUR for all users who are willing to use cellular interfaces for better performance, and each user is served by each with equal possi-



(a) Frame delivery delay. (b) Average frame stall rate and cellular data usage. (c) Frame stall rate. (d) Cellular data usage.

Figure 14: Online performance of multipath transport schemes.



(a) User retention time. (b) Average bitrate.

Figure 15: Online user QoE of multipath transport schemes.

bility. All other settings and implementations, such as CCA, remain the same. The A/B tests are conducted for two weeks and result in a total of 11840 user sessions. The numbers of user sessions for each scheme are shown in Tab. 1.

5.3 Performance

Frame delivery delay and stall rate. Achieving consistent low latency is a key requirement for real-time services that require interactivity. To evaluate the performance of frame transport delay in AUGUR, we plot the CDFs of frame delivery delays and frame stall rate for all user sessions using three multipath schemes in Fig. 14a, Fig. 14b, and Fig. 14c. While the median delay is similar across all schemes, AUGUR outperforms SP and minRTT in reducing long tail delay at the 99th percentile, achieving reductions of 14.2% and 7.7% respectively. More importantly, for severely prolonged tail delays (*i.e.,* frame delay $\geq$ 200 ms) that significantly impact user experience, AUGUR is capable of maintaining the 99.9th percentile latency below 200 ms, while SP and minRTT leave the 99.9th percentile latency over 300 ms.

Consequently, AUGUR reduces the average stall rate of all user sessions to below 0.1%, improving by 78.0% and 73.5% compared to SP and minRTT, respectively. In addition, AUGUR reduces the median stall rate of all user sessions to only 0.02%, and reduces the median stall rate by 90.0% and 85.7% compared to SP and minRTT, respectively. These results demonstrate that in the presence of RTT inflation caused by wireless fluctuation, AUGUR is able to reduce the long tail latency and frame stall rate to enhance the service experience for the majority of mobile users.

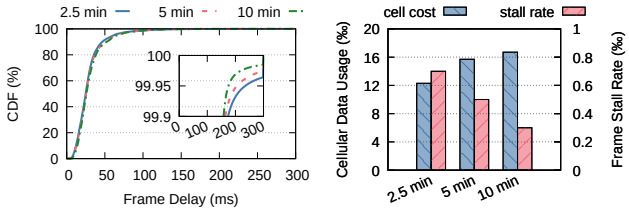
Cellular cost. In order to compare the cellular data usage of AUGUR and the baseline minRTT multipath scheduler, we calculate the ratio of bytes transmitted on the cellular path to the total transmitted bytes for each user session. As shown in Fig. 14b and Fig. 14d, on average, AUGUR sends 7.3% of the video data streaming through the cellular path, which is 2$\times$ less than minRTT. Additionally, for all user sessions, AUGUR incurs 0.65% median cellular data usage, which is 8.9$\times$ less than minRTT. This reduction in cellular data usage can be attributed to two factors. First, AUGUR retransmits only those frames that are likely to be delayed, instead of scheduling all frames on the path with a lower RTT. Second, AUGUR strictly constrains the cellular budget according to Eqn. 6, resulting in a controlled cellular usage pattern. Note that this constraint allows AUGUR to bound the maximum cellular cost and reduce it by 4.7$\times$ compared to minRTT scheduler.

User QoE. We use user retention time and average video bitrate as our QoE metrics. As discussed in §2.1, a higher frame stall rate leads to a decrease in user retention time, which represents a degradation in the user experience of our service. To illustrate the effect of AUGUR on user experience, we show the normalized average user retention time of three transport schemes in Fig. 15a, which demonstrates that with AUGUR deployed, user retention time can be improved by 18.5% and 13.7% on average compared to SP and minRTT, respectively. Note that user retention time has a large deviation because it is also affected by other factors that are uncorrelated to network performance, such as personal preference.

As AUGUR actively retransmits frames on other paths and switches the primary path, it may interfere with the bandwidth estimation mechanism of the underlying transport (*e.g.,* CCA), thus affects the user QoE. We show the normalized average video bitrate of three transport schemes in Fig. 15b, which demonstrates that AUGUR has little impact on video bitrate, with 5.4% and 3.1% decrease compared to SP and minRTT, respectively.

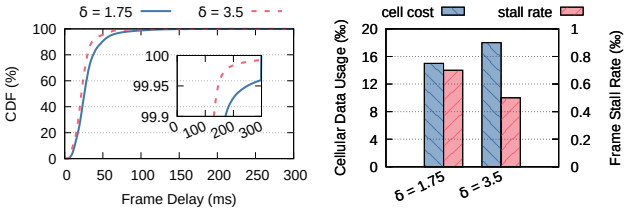
5.4 Micro-benchmark

We further perform some micro-benchmarks to investigate the impact of different design choices made in the development of AUGUR, including parameter settings and cellular utilization approaches.



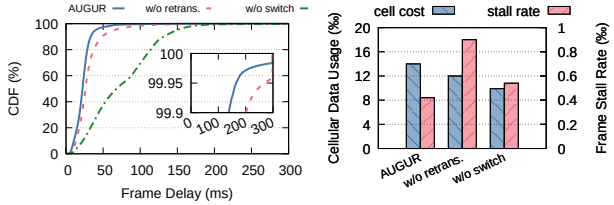
(a) Frame delivery delay. (b) Stall rate and cellular cost.

Figure 16: Impact of time window value Δ.



(a) Frame delivery delay. (b) Stall rate and cellular cost.

Figure 17: Impact of cellular constraint factor δ.



(a) Frame delivery delay. (b) Stall rate and cellular cost.

Figure 18: Impact of frame retransmission and primary path switch.

5.4.1 Parameter Setting

Time window value. AUGUR uses a time window Δ to approximate the state probability models. The value of Δ reflects the fluctuation level of a single user's characteristics. To investigate the impact of Δ, we conduct an experiment involving 2,186 users, with Δ set to 2.5 min, 5 min, and 10 min. As illustrated in Fig. 16, we observe that AUGUR is not sensitive to the choice of Δ, and the frame delivery delays remain similar across all three settings. Additionally, a larger time window results in slightly higher cellular costs and a lower frame stall rate. This is because a larger time window can detect more stall events and adopt a more aggressive backup path utilization strategy. In practice, we set Δ to 5 min based on our deployment experience to reach a balance between stall rate and cellular cost.

Cellular constraint factor. AUGUR uses a factor δ to constrain the cellular cost in Eqn. 6 and its value adapts to the cellular budget specified by the user. To demonstrate the effect of different δ values, we show the performance of AUGUR with two most frequently specified settings: $\delta = 1.75$ corresponding to 250 MB per month and $\delta = 3.5$ corresponding to 500 MB per month. As shown in Fig. 17, a larger constraint factor results in a more effective reduction of long tail latency and stall rate with higher cellular data usage. This demonstrates that δ controls the trade-off between better network performance and lower cellular data cost. We leave the decision of choosing δ to each user based on their data budget.

5.4.2 Cellular Path Utilization

AUGUR utilizes the cellular path in two approaches: frame retransmission and primary path switch. To investigate the effectiveness of both approaches, we conduct an experiment with 7,266 invited users accessing our real-time streaming platform. We randomly assign one-third of users with frame retransmission disabled, one-third of users with primary path switch disabled, and the rest with full AUGUR functionality.

Frame retransmission. AUGUR uses application-level frame retransmission to rescue delayed frames on the Wi-Fi path to reduce long tail latency. As shown in Fig. 18, compared to only performing the primary path switch, enabling frame retransmission reduced the 99th percentile latency and stall rate by 45.8% and 55.5%, respectively, with 0.2% higher cellular data usage. This demonstrates that leveraging the cellular path to perform application-level frame retransmission is necessary to reduce long tail latency caused by RTT inflation.

Primary path switch. To cope with bursty frame stalls caused by severe Wi-Fi capacity degradation, AUGUR schedules the primary path to cellular. As depicted in Fig. 18, the decision to switch the primary path avoids streaming frames on a degraded path and can significantly reduce long tail latency. Compared to only retransmitting frames, AUGUR reduces the 99th percentile latency and stall rate by 25% and 31.7%, respectively, with only a 0.4% increase in cellular cost.

6 Discussion

In this section, we discuss some potential limitations and future work of AUGUR.

Deployment scenario. As a cloud gaming service provider, we primarily evaluated the performance of AUGUR in our cloud gaming clusters and demonstrated its ability to reduce long tail latency. Meanwhile, the scenario of using both Wi-Fi and cellular paths for real-time streaming can be applied to many other applications with consistently low latency requirements, such as video conferencing and live streaming. With the development of Wi-Fi 7 [44] and 5G [41], which provide higher network bandwidth, AUGUR can also be deployed for AR/VR applications across a wide range.

Multiple cellular paths. AUGUR currently leverages only one cellular path. However, as multi-carrier phones become increasingly popular and the benefits of a multipath system design that supports more than two paths are being acknowledged [45], AUGUR can be extended to support multiple cel-

lular paths. This would allow AUGUR to enable multiple cellular backup paths and constrain their usage by introducing different δ values for each path. We leave the extension of AUGUR to support multiple cellular paths as our future work.

Cross-layer optimization. Recent years have seen the proposal of various approaches to improve the performance and user QoE of mobile real-time streaming. These approaches include deploying dedicated CCAs [9, 11, 12], using the adaptive bitrate (ABR) to meet network conditions [46, 47, 48, 49], leveraging forward error correction (FEC) to recover lost frames [50, 51, 28], encoding video frames in a QoE-aware manner [52, 53, 54], allocating and scheduling streaming resources [55, 56, 57], utilizing timely feedback from Wi-Fi APs [58], and adjusting frame rate based on network conditions [8, 59]. As a multipath transport service, AUGUR is orthogonal to these approaches and can be used jointly with them. Furthermore, AUGUR can be extended to perform cross-layer optimization with these approaches (*e.g.,* retransmit frames with FEC). We leave cross-layer optimization of AUGUR as our future work.

7 Related Work

Multipath transport. Multipath transport utilizes multiple paths simultaneously for data delivery in a single connection. MPTCP [60] and MPUDP [61] integrate multipath transport into the OS kernel and provide a single-connection abstraction to applications. Based on this framework, many multipath schedulers have been proposed, such as ECF [19], BLEST [18], RAVEN [17], and DEMS [20]. However, the large-scale deployment of such a framework has been slow due to its modification of user device kernels [16] and network middleboxes [62]. Therefore, some multipath transport systems based on QUIC have been proposed, such as MPQUIC [63], PQUIC [64], and XLINK [24]. Additionally, there are some multipath transport schemes using TCP splitting (POLYCORN [65]) or implemented in the application layer (MP-H2 [66], MSPlayer [67], MP-DASH [21], mHTTP [68]). Like these approaches, AUGUR can be deployed on a large scale and is specifically designed for real-time streaming with low tail latency requirements.

Cellular cost reduction. Several approaches have been proposed to reduce the cellular data cost for network traffics. By leveraging the delay-tolerant nature of VoD applications, MP-DASH [21] schedules the cellular path based on a user-specified preference and a known video chunk size, and Obilgir *et al.* schedules data transmissions to reduce cellular data usage. TrafficGuard [69] adopts a proxy-based method to reduce cellular traffic using a network-layer VPN. Yanyuan *et al.* [70] uses a chunk filter and a rate adaptation algorithm to reduce cellular usage in video streaming. In contrast to all approaches above, AUGUR is designed for real-time streaming applications and reduces cellular cost by directly avoiding unnecessary data transmission.

8 Conclusion

In this paper, we demonstrate that long tail latency can severely degrade the user experience in real-time streaming. We further find that in wireless networks like Wi-Fi, long tail latency is induced by RTT inflation caused by wireless fluctuation, and thus cannot be reduced by deploying dedicated CCAs. While it is straightforward to leverage cellular paths to alleviate the impact of Wi-Fi RTT inflation, we reveal that users express strong concern about cellular cost, thus it is crucial to minimize cellular data usage for a widely deployed multipath transport service. We design AUGUR, a practical multipath transport service for mobile real-time streaming that meets the requirement of reducing long tail latency, minimizing cellular data usage, and enabling large-scale deployment. We deploy AUGUR on our cloud gaming platform with millions of users across a wide area and demonstrate its effectiveness in reducing long tail latency and frame stall rate with minimal cellular data usage. We believe that AUGUR can be widely deployed at any scale to provide universal low-latency mobile streaming access.

Acknowledgements

We are grateful to the NSDI reviewers for their constructive critique, and to our shepherd Akshay Narayan in particular, for his valuable comments, all of which have helped us greatly improve this paper. This work is supported by National Key Research and Development Plan, China (Grant No. 2020YFB1710900), National Natural Science Foundation of China (Grant No. 62022005 and 62172008).

References

[1] Samsung gaming hub. https://www.samsung.com/us/televisions-home-theater/tvs/gaming-hub/, 2022.

[2] Google cloud gaming. https://cloud.google.com/solutions/games, 2023.

[3] Zoom: One platform to connect. https://zoom.us, 2023.

[4] Microsoft teams. https://www.microsoft.com/en-us/microsoft-teams/group-chat-software, 2023.

[5] Google map live view support. https://support.google.com/maps/answer/9332056?hl=en&co=GENIE.Platform%3DiOS, 2023.

[6] Youtube vr - home. https://vr.youtube.com/, 2023.

[7] Cloud gaming market: Global industry trends, share, size, growth, opportunity and forecast 2023-2028. Market report 5732901, IMARC Group, 2023.

[8] Zili Meng, Tingfeng Wang, Yixin Shen, Bo Wang, Mingwei Xu, Rui Han, Honghao Liu, Venkat Arun, Hongxin Hu, and Xue Wei. Enabling high quality Real-Time communications with adaptive Frame-Rate. In *USENIX NSDI*, 2023.

[9] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Analysis and design of the google congestion control for web real-time communication (webrtc). In *ACM MMSys*, 2016.

[10] Xiaoqing Zhu and Rong Pan. Nada: A unified congestion control scheme for low-latency interactive video. In *20th IEEE International Packet Video Workshop*, 2013.

[11] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *USENIX NSDI*, 2018.

[12] Devdeep Ray, Connor Smith, Teng Wei, David Chu, and Srinivasan Seshan. Sqp: Congestion control for low-latency interactive video streaming. *arXiv preprint arXiv:2207.11857*, 2022.

[13] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive congestion control for unpredictable cellular networks. In *ACM SIGCOMM*, 2015.

[14] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *USENIX NSDI*, April 2013.

[15] Christoph Paasch, Simone Ferlin, Ozgu Alay, and Olivier Bonaventure. Experimental evaluation of multipath tcp schedulers. In *ACM CSWS*, 2014.

[16] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *USENIX NSDI*, 2012.

[17] HyunJong Lee, Jason Flinn, and Basavaraj Tonshal. Raven: Improving interactive latency for the connected car. In *ACM MobiCom*, 2018.

[18] Per Hurtig, Karl-Johan Grinnemo, Anna Brunstrom, Simone Ferlin, Özgü Alay, and Nicolas Kuhn. Low-latency scheduling in mptcp. *IEEE/ACM Transactions on Networking*, 2019.

[19] Yeon-sup Lim, Erich M. Nahum, Don Towsley, and Richard J. Gibbens. Ecf: An mptcp path scheduler to manage heterogeneous paths. In *ACM CoNEXT*, 2017.

[20] Yihua Ethan Guo, Ashkan Nikravesh, Z. Morley Mao, Feng Qian, and Subhabrata Sen. Accelerating multipath transport through balanced subflow completion. In *ACM MobiCom*, 2017.

[21] Bo Han, Feng Qian, Lusheng Ji, and Vijay Gopalakrishnan. Mp-dash: Adaptive video streaming over preference-aware multipath. In *ACM CoNEXT*, 2016.

[22] Cheng-Lin Tsao and Raghupathy Sivakumar. On effectively exploiting multiple wireless interfaces in mobile hosts. In *ACM CoNEXT*, 2009.

[23] Onjeinika Brooks. Best cell phone plans of 2022. https://www.reviews.org/internet-service/best-internet-service-providers/, 2022.

[24] Zhilong Zheng, Yunfei Ma, Yanmei Liu, Furong Yang, Zhenyu Li, Yuanbo Zhang, Jiuhai Zhang, Wei Shi, Wentao Chen, Ding Li, Qing An, Hai Hong, Hongqiang Harry Liu, and Ming Zhang. Xlink: Qoe-driven multi-path quic transport in large-scale video services. In *ACM SIGCOMM*, 2021.

[25] Start cloud gaming. https://start.qq.com/, 2023.

[26] Thomas Wiegand, Gary J Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the h. 264/avc video coding standard. *IEEE Transactions on circuits and systems for video technology*, 2003.

[27] Taehyun Kim, Niranjan Avadhanam, and Sridharan Subramanian. Dimensioning receiver buffer requirement for unidirectional vbr video streaming over tcp. In *IEEE International Conference on Image Processing*, 2006.

[28] Michael Rudow, Francis Y. Yan, Abhishek Kumar, Ganesh Ananthanarayanan, Martin Ellis, and K.V. Rashmi. Tambur: Efficient loss recovery for videoconferencing via streaming codes. In *USENIX NSDI*, 2023.

[29] Serhat Arslan, Yuliang Li, Gautam Kumar, and Nandita Dukkipati. Bolt: Sub-RTT congestion control for Ultra-Low latency. In *USENIX NSDI*, 2023.

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

[31] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. In *ACM SIGCOMM*, 2020.

[32] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. *ACM SIGCOMM Computer Communication Review*, 2002.

[33] Alexander Frommgen, Tobias Erbshäußer, Alejandro Buchmann, Torsten Zimmermann, and Klaus Wehrle. Remp tcp: Low latency multipath tcp. In *IEEE ICC*, 2016.

[34] Markus Amend, Veselin Rakocevic, and Joachim Habermann. Cost optimized multipath scheduling in 5g for video-on-demand traffic. In *IEEE WCNC*, 2021.

[35] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 1951.

[36] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 1951.

[37] Xinlei Yang, Hao Lin, Zhenhua Li, Feng Qian, Xingyao Li, Zhiming He, Xudong Wu, Xianlong Wang, Yunhao Liu, Zhi Liao, Daqiang Hu, and Tianyin Xu. Mobile access bandwidth in practice: Measurement, analysis, and implications. In *ACM SIGCOMM*, 2022.

[38] Haiqing Jiang, Zeyu Liu, Yaogong Wang, Kyunghan Lee, and Injong Rhee. Understanding bufferbloat in cellular networks. In *ACM SIGCOMM CellNet*, 2012.

[39] Jing Wang, Yufan Zheng, Yunzhe Ni, Chenren Xu, Feng Qian, Wangyang Li, Wantong Jiang, Yihua Cheng, Zhuo Cheng, Yuanjie Li, Xiufeng Xie, Yi Sun, and Zhongfeng Wang. An active-passive measurement study of tcp performance over lte on high-speed rails. In *ACM MobiCom*, 2019.

[40] Junxian Huang, Feng Qian, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *ACM MobiSys*, 2012.

[41] Arvind Narayanan, Xumiao Zhang, Ruiyang Zhu, Ahmad Hassan, Shuowei Jin, Xiao Zhu, Xiaoxuan Zhang, Denis Rybkin, Zhengxuan Yang, Zhuoqing Morley Mao, Feng Qian, and Zhi-Li Zhang. A variegated look at 5g in the wild: Performance, power, and qoe implications. In *ACM SIGCOMM*, 2021.

[42] R. Netravali A. Sivaraman and K. J. Winstein. Mpshell. https://github.com/ravinet/mahimahi/releases/tag/old%2Fmpshell_scripted, 2020.

[43] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for http. In *USENIX ATC*, 2015.

[44] Cailian Deng, Xuming Fang, Xiao Han, Xianbin Wang, Li Yan, Rong He, Yan Long, and Yuchen Guo. Ieee 802.11be wi-fi 7: New challenges and opportunities. *IEEE Communications Surveys & Tutorials*, 2020.

[45] Xiao Zhu, Jiachen Sun, Xumiao Zhang, Y. Ethan Guo, Feng Qian, and Z. Morley Mao. Mpbond: Efficient network-level collaboration among personal mobile devices. In *ACM MobiCom*, 2020.

[46] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *ACM SIGCOMM*, 2017.

[47] Zahaib Akhtar, Yun Seong Nam, Ramesh Govindan, Sanjay Rao, Jessica Chen, Ethan Katz-Bassett, Bruno Ribeiro, Jibin Zhan, and Hui Zhang. Oboe: Auto-tuning video abr algorithms to network conditions. In *ACM SIGCOMM*, 2018.

[48] Kuntai Du, Ahsan Pervaiz, Xin Yuan, Aakanksha Chowdhery, Qizheng Zhang, Henry Hoffmann, and Junchen Jiang. Server-driven video streaming for deep learning inference. In *ACM SIGCOMM*, 2020.

[49] Xianshang Lin, Yunfei Ma, Junshao Zhang, Yao Cui, Jing Li, Shi Bai, Ziyue Zhang, Dennis Cai, Hongqiang Harry Liu, and Ming Zhang. Gso-simulcast: Global stream orchestration in simulcast video conferencing systems. In *ACM SIGCOMM*, 2022.

[50] C. Perkins, O. Hodson, and V. Hardman. A survey of packet loss recovery techniques for streaming audio. *IEEE Network*, 1998.

[51] Justin Uberti. Webrtc forward error correction requirements. *draft-ietf-rtcweb-fec-01 (work in progress)*, 2015.

[52] Yu Guan, Chengyuan Zheng, Xinggong Zhang, Zongming Guo, and Junchen Jiang. Pano: Optimizing 360° video streaming with a better understanding of quality perception. In *ACM SIGCOMM*, 2019.

[53] Devdeep Ray, Jack Kosaian, K. V. Rashmi, and Srinivasan Seshan. Vantage: Optimizing video upload for time-shifted viewing of social live streams. In *ACM SIGCOMM*, 2019.

[54] Mallesham Dasari, Kumara Kahatapitiya, Samir R. Das, Aruna Balasubramanian, and Dimitris Samaras. Swift: Adaptive video streaming with layered neural codecs. In *USENIX NSDI*, 2022.

[55] Jinyang Li, Zhenyu Li, Ri Lu, Kai Xiao, Songlin Li, Jufeng Chen, Jingyu Yang, Chunli Zong, Aiyun Chen, Qinghua Wu, Chen Sun, Gareth Tyson, and Hongqiang Harry Liu. Livenet: A low-latency video transport network for large-scale live streaming. In *ACM SIGCOMM*, 2022.

[56] Tan Zhang, Aakanksha Chowdhery, Paramvir (Victor) Bahl, Kyle Jamieson, and Suman Banerjee. The design and implementation of a wireless video surveillance system. In *ACM MobiCom*, 2015.

[57] Le Xu, Shivaram Venkataraman, Indranil Gupta, Luo Mai, and Rahul Potharaju. Move fast and meet deadlines: Fine-grained real-time stream processing with cameo. In *USENIX NSDI*, 2021.

[58] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. Achieving consistent low latency for wireless real-time communications with the shortest control loop. In *ACM SIGCOMM*, 2022.

[59] Tingfeng Wang, Zili Meng, Mingwei Xu, Rui Han, and Honghao Liu. Enabling high frame-rate uhd real-time communication with frame-skipping. In *ACM HotEdgeVideo*, 2021.

[60] Alan Ford, Costin Raiciu, Mark Handley, and Olivier Bonaventure. Tcp extensions for multipath operation with multiple addresses. Technical report, 2013.

[61] Daniel Lukaszewski and Geoffrey Xie. Multipath transport for virtual private networks. In *10th USENIX Workshop on Cyber Security Experimentation and Test*, 2017.

[62] Ashkan Nikravesh, Yihua Guo, Feng Qian, Z. Morley Mao, and Subhabrata Sen. An in-depth understanding of multipath tcp on mobile devices: Measurement and system design. In *ACM MobiCom*, 2016.

[63] Quentin De Coninck and Olivier Bonaventure. Multipath quic: Design and evaluation. In *ACM CoNEXT*, 2017.

[64] Quentin De Coninck, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, and Olivier Bonaventure. Pluginizing quic. In *ACM SIGCOMM*, 2019.

[65] Yunzhe Ni, Feng Qian, Taide Liu, Yihua Cheng, Zhiyao Ma, Jing Wang, Zhongfeng Wang, Gang Huang, Xuanzhe Liu, and Chenren Xu. POLYCORN: Data-driven cross-layer multipath networking for high-speed railway through composable schedulerlets. In *USENIX NSDI*, 2023.

[66] Ashkan Nikravesh, Yihua Guo, Xiao Zhu, Feng Qian, and Z. Morley Mao. Mp-h2: A client-only multipath solution for http/2. In *ACM MobiCom*, 2019.

[67] Yung-Chih Chen, Don Towsley, and Ramin Khalili. Msplayer: Multi-source and multi-path video streaming. *IEEE Journal on Selected Areas in Communications*, 2016.

[68] Juhoon Kim, Yung-Chih Chen, Ramin Khalili, Don Towsley, and Anja Feldmann. Multi-source multipath http (mhttp): A proposal. In *ACM SIGMETRICS*, 2014.

[69] Zhenhua Li, Weiwei Wang, Tianyin Xu, Xin Zhong, Xiang-Yang Li, Yunhao Liu, Christo Wilson, and Ben Y. Zhao. Exploring Cross-Application cellular traffic optimization with baidu TrafficGuard. In *USENIX NSDI*, 2016.

[70] Yanyuan Qin, Shuai Hao, Krishna R. Pattipati, Feng Qian, Subhabrata Sen, Bing Wang, and Chaoqun Yue. Quality-aware strategies for optimizing abr video streaming qoe and reducing data usage. In *ACM MMSys*, 2019.

Zombie: Middleboxes that Don't Snoop

Collin Zhang,[†] Zachary DeStefano,[*] Arasu Arun,[*] Joseph Bonneau,[*] Paul Grubbs,[‡] Michael Walfish[*]

[*]NYU [†]Cornell (work done mostly at NYU) [‡]University of Michigan

Abstract. Zero-knowledge middleboxes (ZKMBs) are a recent paradigm in which clients get privacy while middleboxes enforce policy: clients prove in zero knowledge that the plaintext underlying their encrypted traffic complies with network policies, such as DNS filtering. However, prior work had impractically poor performance and was limited in functionality.

This work presents Zombie, the first system built using the ZKMB paradigm. Zombie introduces techniques that push ZKMBs to the verge of practicality: preprocessing (to move the bulk of proof generation to idle times between requests), asynchrony (to remove proving and verifying costs from the critical path), and batching (to amortize some of the verification work). Zombie's choices, together with these techniques, reduce client and middlebox overhead by $\approx 3.5\times$, lowering the critical path overhead for a DNS filtering application on commodity hardware to less than 300ms or, in the asynchronous configuration, to 0.

As an additional contribution that is likely of independent interest, Zombie introduces a portfolio of techniques to encode regular expressions in probabilistic (and zero-knowledge) proofs. These techniques significantly improve performance over a standard baseline, asymptotically and concretely. Zombie builds on this portfolio to support policies based on regular expressions, such as data loss prevention.

1 Introduction

A fundamental conflict frequently arises in network security: administrators' policy enforcement vs. users' privacy. Organizations want, or in some cases need (by legal obligation), to enforce network usage policies. Users want end-to-end encrypted protocols like TLS to provide privacy against network observers, including administrators. Traditionally, policy enforcement requires a middlebox to scan traffic and block policy-violating use. End-to-end encryption is in direct conflict with middleboxes, which can't see plaintext and therefore can't assess policy compliance. This conflict has led some administrators to take draconian steps, like inserting themselves as an all-seeing middleperson (MITM) proxying TLS connections ("split TLS"), or even blocking the use of TLS.

Resolving this conflict has been a goal of the network security research community for some time. Existing approaches have fallen into two categories. First are protocols that use novel cryptography to enable policy checks on encrypted data, but require server support and/or changes to standard protocols like TLS [63, 79, 98] (§7). Changing TLS is a huge task though: it took ten years of extensive design effort to go from TLS 1.2 [30] to TLS 1.3 [90]. Deploying server-side changes is also slow: five years after the standardization of TLS 1.3, only 60% of HTTPS servers on the web support it [60]. Furthermore, implementing TLS securely is notoriously complex and subtle [24], meaning that protocol changes are risky. Second, by contrast, are middleboxes designed to work with standard TLS-encrypted traffic but rely on trusted hardware enclaves (TEEs) to enforce policy [31, 44]. Our perspective is that, although it promises good performance, trusted hardware has a wide attack surface (§7), as demonstrated by many attacks [47, 77, 92, 102, 109–112].

Our goal is to support policy enforcement on standard TLS 1.3 traffic, inheriting its existing security guarantees and avoiding any changes to existing TLS code bases. We eschew any trusted hardware assumptions. We do, however, accept modifications to clients, observing that modern browser vendors can update the vast majority of users within months [114].

Zero-knowledge middleboxes. We build on top of the recently-proposed ZKMB paradigm [49]. With ZKMBs, clients prove in zero knowledge [43] to the middlebox that the plaintext underlying their encrypted traffic is policy-compliant. The middlebox verifies these proofs, allowing only policy-compliant traffic to pass. Because the proof is zero knowledge, the middlebox learns nothing about the underlying plaintext, except that it is policy-compliant. ZKMBs require no changes to existing encryption protocols and no trusted hardware; they promise an elegant solution to the policy vs. privacy conflict. However, the initial ZKMB prototype offered implausible performance for most network applications, adding several seconds of latency to traffic even under optimistic assumptions and with a relatively simple policy.

The key question remains: *Can ZKMBs perform well enough, and express a wide-enough range of policies, for real-world use?* This paper gives a cautious affirmative answer, with the design, implementation, and experimental evaluation of a system called *Zombie*.

Contributions and results. Zombie applies three techniques to reduce end-to-end delay (both client proving costs and middlebox verifying costs on the critical path). First, Zombie precomputes and pre-proves part of the encryption step in TLS, moving it off the critical path to periods when the client is idle (§3.1). This part includes legacy cryptographic primitives like ChaCha20 encryption, which, for reasons we explain later (§2, §3.1), are expensive to represent in proof frameworks. Such a split is perhaps surprising: how could a client

precompute an encryption before the plaintext is known?

Zombie's next two performance enhancements are conditional on assumptions about client and middlebox behavior. For these, our contribution is primarily analyzing and evaluating the techniques in this context, rather than the mechanics of the techniques. One of these is *optimistic approval* (§3.2), via asynchronous verification. We make the simple but consequential observation that, in many applications, administrators may be willing to allow client traffic to proceed as normal, on the condition that clients supply valid proofs in short order. A similar approach, *near real-time verification*, is already taken by some real-world middleboxes [18, 23, 38]. The other is *batch proof verification* by the middlebox, reducing the overall verification burden by amortizing it across proofs for multiple packets (§3.3).

Another set of contributions enables Zombie to handle policies based on regular expression matching, a crucial building block in various middleboxes, including intrusion detection systems (IDS), network traffic classification, and data loss prevention (DLP). The core challenge derives from probabilistic proofs themselves: using these frameworks requires representing the target computation in arithmetic circuits or constraints. Meanwhile, as with ChaCha20 mentioned earlier, circuits and constraints are inefficient and inhospitable for many computations (§2), including (at first glance) regular expressions. Zombie tackles this challenge with a collection of techniques (§4), including a new encoding of substring matching in arithmetic constraints, a new encoding of Boolean algebra in arithmetic constraints, and a new finite automaton formalism. Some of these techniques are likely to be of independent interest for other applications of probabilistic proofs, even beyond regular-expression matching.

We implement Zombie for TLS 1.3 with the ChaCha20 cipher (§5). The result of all this work is near-practicality for some ZKMB uses (§6). In the precomputation regime, Zombie adds less than 300 ms of delay to DNS queries, which may be tolerable (§8). In the asynchronous regime, this number drops to 0. Proofs are large (30KB) but never leave the local network. Memory requirements for prover and verifier can be substantial, but small packet sizes mitigate this issue. The sticking point is middlebox resources: although throughput improves almost 5× from batching, even this improvement (380 255-byte packets/second in our experiments) is too low to imagine proof verification on every packet. Similarly, Zombie's regular expression techniques reduce the overhead of encoding real-world DLP policies in zero knowledge by over an order of magnitude; however, the resulting overhead, 1–2 ms processing delay per byte, is uncomfortably large for networking applications.

Thus, although Zombie is designed to be extensible to any read-based public policy, it is most practical for multi-packet flows, with small packets, that represent a fraction of traffic, for example enforcing a domain blocklist on a long-running connection with a DNS server or enforcing a keyword or regexp blocklist on search engine queries. Our work has other limitations (§8). Most notably, our implementation requires that policies be public, per-packet, read-only, and stateless, though these restrictions are not fundamental.

2 Background

Zero-knowledge proofs. At a high level, a ZKP is a cryptographic protocol between two parties: a *prover* and a *verifier*. The protocol pertains to a computation $\mathbf{S}$ (we also call this the "statement"), which we formulate as having two inputs $\mathbf{X}$ and $\mathbf{W}$, each a vector of variables, and producing an output $\mathbf{Y}$. We call $\mathbf{X}$ the *public input* and $\mathbf{Y}$ the *output*, respectively.

In this paper, we consider non-interactive ZKPs, which work as follows. Both the verifier and the prover agree on a computation $\mathbf{S}$. To convince the verifier that a particular $(\mathbf{X}, \mathbf{Y})$ pair known to both parties is *valid*, the prover sends the verifier a *proof* π. Validity here is defined as the existence of a *witness* $\mathbf{W}$ such that $\mathbf{S}(\mathbf{X}; \mathbf{W}) = \mathbf{Y}$ for a particular $(\mathbf{X}, \mathbf{Y})$. Following convention, we use a semicolon to separate public and private variables in statements. The proof also convinces the verifier that the prover *knows* this witness—this guarantee is called *knowledge soundness*. Moreover, it hides the witness from the verifier—this is the *zero-knowledge* guarantee. The notions of soundness and zero-knowledge have precise cryptographic definitions that we elide here; Zombie inherits these properties directly from the underlying cryptographic tools.

In general, there is a deep cryptographic literature on ZKPs; for a survey, we refer the reader to Thaler [103].

A concrete example. Consider using ZKPs to prove that an encrypted packet does not contain a DNS query for a blocked domain [49, §7]. The output $\mathbf{Y}$ is true/false, the input $\mathbf{X}$ is the encrypted packet, and the witness $\mathbf{W}$ includes the decryption key. The computation $\mathbf{S}$ asserts that the packet, after decrypting to plaintext using the decryption key and extracting the domain name, does not contain a domain in the blocklist. Under standard cryptographic assumptions, knowing $\mathbf{X}$ and $\mathbf{Y}$, without knowing $\mathbf{W}$, is insufficient to verify the correctness of $(\mathbf{X}, \mathbf{Y})$ with respect to $\mathbf{S}$. A ZKP, by contrast, convinces a verifier, who has no access to $\mathbf{W}$, of the correctness of $(\mathbf{X}, \mathbf{Y})$.

Zero-knowledge proof pipelines. Most generic ZKP schemes decompose into a *front-end* and a *back-end*. The front-end takes $\mathbf{S}$, a high-level specification of a program, for example in C code or a domain-specific language (DSL). The front-end compiles this program into an *intermediate representation*, often called a *circuit* (see below). This circuit acts as a blueprint for provers to show that a program produces specific outputs, given specific inputs.

The back-end then enables the prover to take the circuit representation of the program, along with $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{W}$, and output a proof π. The verifier also has access to a circuit representation of the program and uses the back-end, $\mathbf{X}$, and $\mathbf{Y}$ to verify a proof π, outputting a true/false value.

R1CS instances. Most modern ZKP front-ends compile programs to a generalization of arithmetic circuits called *rank-one constraint systems* (R1CS). An R1CS instance is a collection of algebraic constraints. The instance is parameterized by a finite field $\mathbb{F}$, a number of constraints m, a number of variables n, and three $m \times n$ matrices A, B, C. An input-output pair $(\mathbf{X}, \mathbf{Y})$ satisfies the R1CS instance if there exists a $\mathbf{W}$ such that for the vector $z = (\mathbf{X}, \mathbf{Y}, 1, \mathbf{W})$, $Az \circ Bz = Cz$, where the operation $\circ$ is entry-wise multiplication. Notice that an R1CS instance consists of m constraints in n variables, where each constraint $i \in \{1, \ldots, m\}$ restricts any satisfying $z = (z_1, \ldots, z_n)$ as follows:

$$(A_{i,1}z_1 + \ldots + A_{i,n}z_n) \cdot (B_{i,1}z_1 + \cdots + B_{i,n}z_n) = (C_{i,1}z_1 + \cdots + C_{i,n}z_n).$$

Following convention, we sometimes refer to an R1CS representation as a set of *constraints* or loosely as a *circuit*.

Efficiently expressing a computation as a circuit is challenging. First, the primary efficiency metric of a circuit representation is the *number of constraints*, as the back-end's costs—specifically, the prover's costs—scale linearly or superlinearly in this quantity. Second, circuits are frequently verbose, as they are algebraic constructs, not hardware circuits or a general-purpose processor.

Among other limitations, circuits do not support looping, conditionality, order comparisons, bitwise operations, or random-access memory. Compiling a high-level computation to a circuit requires the front-end to unroll all loops to their maximum iteration count, inline all function calls, represent all branches of conditionals explicitly, and then *arithmetize* each statement (translating it into constraints), often introducing additional variables [16, 17, 85, 94, 96, 115, 124].

As a simple example, consider this line of `C` code:

```
y = (x == 0);
```

where the mathematical variable x (representing the program variable `x`) is in $\mathbf{X}$ and y (likewise representing `y`) is in $\mathbf{Y}$. To compile this to constraints, one introduces a variable W in $\mathbf{W}$ and writes the following, called `EQUALS-ZERO` [96, Appx D]:

$$\left\{ \begin{array}{c} y \cdot x = 0 \\ W \cdot x = 1 - y \end{array} \right\}$$

The constraints can be satisfied if and only if y is 1 when x is 0 and y is 0 otherwise, thus enforcing the desired computation. These constraints can be expressed in the form of an R1CS instance as the following A, B, and C matrices:

$$\begin{array}{cccc} x & y & 1 & W \\ \end{array} \quad \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A$$

$$\begin{array}{cccc} x & y & 1 & W \\ \end{array} \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad B$$

$$\begin{array}{cccc} x & y & 1 & W \\ \end{array} \quad \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 \end{bmatrix} \quad C$$

Spartan ZKP. As its back-end, Zombie uses Spartan [93], specifically the SpartanNIZK variant (which we refer to as just Spartan for simplicity). By contrast, prior work [49] used Groth16 [48]. Spartan is a non-interactive ZKP protocol that strikes an attractive balance among prover time (lower than in Groth16), verifier time (higher than Groth16 but sufficient for our purposes, §6), and proof size (again, higher than Groth16 but sufficient). Like many ZKP protocols, Spartan has a *setup phase* to generate parameters that are used in the proof protocol; in Spartan, (unlike Groth16), this phase does not require trusting any party, only a source of public randomness. Consequently, provers (clients) can use the generated parameters across different verifiers (networks).

3 Zombie's protocol

We start with the existing ZKMB paradigm [49], though our notation differs from the original. A middlebox begins with encryption protocol E (such as TLS 1.3), content type F, and policy P, with the goal of enforcing P on traffic of type F that is sent via E. The middlebox—or a third-party—defines the following subcomputations, which are composed into statements (§2) that the client proves and the middlebox verifies:

(1) A *channel-opening* subcomputation $\mathbf{S}_E$ takes as input a packet and the information required to re-derive a session key. This subcomputation outputs the decrypted packet, in the sense of delivering that decrypted packet to the next composed subcomputation; for clarity, we note that the decrypted packet itself is never available to the middlebox, which has no access to the values of the circuit wires used in the client's proof. We follow the amortized ZKMB model, which, for TLS 1.3, reuses the expensive work of this phase over multiple per-packet proofs. Specifically, $\mathbf{S}_E$ is split into $\mathbf{S}_{E.1}$ (*derive-and-commit*) and $\mathbf{S}_{E.2}$ (*decrypt*).

(2) A *parse-and-extract* subcomputation $\mathbf{S}_F$ takes as input the decrypted packet and outputs (in the sense above) a snippet of policy-relevant data from the packet.

(3) A *policy-check* subcomputation $\mathbf{S}_P$ takes as input the snippet of policy-relevant data and outputs whether or not the policy is satisfied (for example if a domain being queried is part of a blocklist or not).

Figure 1 depicts the high-level protocol. The middlebox sends $\mathbf{S}_P$ to each client when it joins the network [49]. (We discuss other deployment possibilities in Section 8.) When a client wants to communicate with a particular server, it first negotiates the shared key K using a *handshake* protocol, the transcript of which is public but the generation of which involves secrets shared between the client and the server. $\mathbf{S}_{E.1}$ re-derives this session key K by taking the handshake transcript as public input and the client's secrets as witness, and then hashes the session key to produce h_K. The client sends to the middlebox h_K and the proof π_K of $\mathbf{S}_{E.1}$. This proof

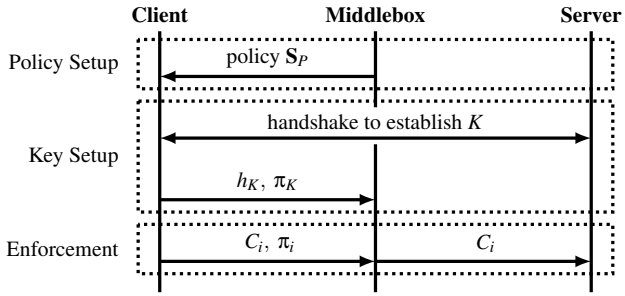


Figure 1: The ZKMB paradigm with amortized key setup [49] divided into 3 phases. The *policy setup* step occurs once when the client connects to the middlebox; the middlebox sends the policy $\mathbf{S}_P$ to the client. The *key setup* step occurs once per session; it involves a handshake between the client and the server, a commitment h_K to a session key K, and a proof of this commitment π_K. Finally, in per-packet *enforcement*, the client sends the middlebox ciphertext C_i and a proof π_i of the policy-compliance of the plaintext corresponding to C_i and to the key commitment; π_i is with reference to the composition of the $\mathbf{S}_{E.2}$, $\mathbf{S}_F$, and $\mathbf{S}_P$ subcomputations.

convinces the middlebox that h_K is indeed the commitment to some key that is consistent with the handshake.

Then, for each packet, $\mathbf{S}_{E.2}$ takes the ciphertext C and the key commitment h_K as public inputs, and the session key K as witness. After ensuring that K hashes to h_K, $\mathbf{S}_{E.2}$ outputs (again, in the sense above) the decrypted packet. Finally, the client needs to convince the middlebox that C is valid with respect to h_K and the composition of $\mathbf{S}_{E.2}$, $\mathbf{S}_F$, and $\mathbf{S}_P$. It does so with a proof π_i. When the middlebox receives (C, h_K, π_i), it verifies π_i, and only then forwards C.

Zombie's enhancements. Zombie introduces three changes to the ZKMB paradigm. *Precomputation* (§3.1) allows Zombie to generate and verify the most expensive part of the proof during idle times, before the ciphertext is known to the client, reducing proving times in the critical latency path. *Asynchronous verification* (§3.2) relaxes the requirement that proofs about traffic are verified before each packet leaves the network. This moves the main ZKP-related costs out of the critical path entirely, greatly reducing delay but changing Zombie's security model. *Batching* (§3.3) lets Zombie middleboxes reuse the results of expensive computations when verifying a batch of proofs created by the client. Figure 2 comprehensively classifies when these techniques should be used and combined.

3.1 Precomputation

Precomputation in Zombie changes both the statement being proved and the protocol flow (adding an extra message). At a high level, precomputation splits the per-packet computation $\mathbf{S}_{E.2}$ (*decrypt*) into two subcomputations $\mathbf{S}_{E.2a}$ (*pad-commit*) and $\mathbf{S}_{E.2b}$ (*decrypt-from-pad*), the first of which can be computed before the plaintext is known. As noted in the introduction, it may be surprising that it is possible to prove $\mathbf{S}_{E.2a}$

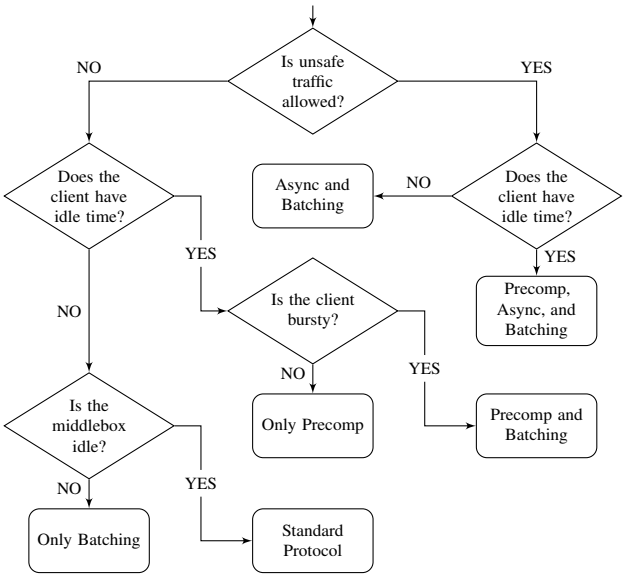


Figure 2: Appropriate combinations of protocol enhancements. The goal is to minimize client latency while respecting security. Unsafe traffic means that some non-compliant traffic is allowed to exit the network. To summarize the logic: (1) If unsafe traffic is allowed, then asynchronous mode is appropriate; (2) If clients have idle time, precomputation is appropriate; (3) If {(a) unsafe traffic is allowed, or (b) neither the client nor the middlebox has idle time, or (c) the client workload is bursty and the client has idle time} then batching is appropriate.

before any plaintext is known, but when using stream ciphers the keystream can be generated (and proved) independently of the plaintext. We explain how this works for TLS 1.3 below. The technique may be more broadly relevant; it is orthogonal (and complementary) to the split between $\mathbf{S}_{E.1}$ and $\mathbf{S}_{E.2}$.

We will simplify by omitting some operations. Let K be the session key output by the handshake. TLS 1.3 encrypts session data using a stream cipher, which can be thought of as a pseudorandom one-time pad. This pad is derived via a function PadGen that takes K, a packet number SN, and a length L_{pad}, and outputs an L_{pad}-byte pseudorandom $\mathsf{pad}_{\mathsf{SN}}$. For an L_{pad}-byte message M, its ciphertext is $\mathsf{pad}_{\mathsf{SN}} \oplus M$.

We make two key observations. First, the inputs to PadGen are independent of the message, so $\mathsf{pad}_{\mathsf{SN}}$ can be computed by the client before M is known. Second, computing PadGen is the most expensive part of the channel-opening subcomputation $\mathbf{S}_E$. This is because PadGen involves the legacy stream cipher ChaCha20, which is difficult to represent in the constraint formalism used for ZKPs (§2). In our evaluation of a DNS filtering application [49] (§6), we find that PadGen accounts for nearly two-thirds of the total constraint size (§6.3).

Zombie uses these observations to move PadGen into the subcomputation $\mathbf{S}_{E.2a}$ (*pad-commit* in Figure 3). This involves running PadGen and then hashing its output to produce $h_{\mathsf{pad}_{\mathsf{SN}}}$. Also, K is hashed to ensure that it corresponds to the hash h_K provided as a public input. The client computes the

pad-commit$(h_K, \mathsf{SN}, L_{pad}; K)$:

$\mathsf{pad}_{\mathsf{SN}} \leftarrow \mathsf{PadGen}(K, \mathsf{SN}, L_{pad})$

$h_{\mathsf{pad}_{\mathsf{SN}}} \leftarrow H(\mathsf{pad}_{\mathsf{SN}})$

Return $(h_K = H(K))\ ?\ h_{\mathsf{pad}_{\mathsf{SN}}} : \perp$

decrypt-from-pad$(C, h_{\mathsf{pad}_{\mathsf{SN}}}, \mathsf{SN}; \mathsf{pad})$:

$M \leftarrow \mathsf{pad} \oplus C$

Return $(h_{\mathsf{pad}_{\mathsf{SN}}} = H(\mathsf{pad}))\ ?\ M : \perp$

Figure 3: Pseudocode for the statements $\mathbf{S}_{E.2a}$ (*pad-commit*) and $\mathbf{S}_{E.2b}$ (*decrypt-from-pad*) used in Zombie's precomputation.

proof $\pi_{E.2a}$ for this subcomputation and sends it, along with the pad hash $h_{\mathsf{pad}_{\mathsf{SN}}}$, to the middlebox. The middlebox verifies this proof and stores the hash with the corresponding sequence number.

In the second part of the protocol, run once the Zombie client receives the plaintext M, the client encrypts M with $\mathsf{pad}_{\mathsf{SN}}$ for sequence number SN to get the ciphertext C. Then, it generates the proof for the statement $\mathbf{S}_{E.2b}$ (see *decrypt-from-pad* in Figure 3). This statement takes a pad pad as the witness, which is purportedly $\mathsf{pad}_{\mathsf{SN}}$; ensures that this witness hashes to the stored hash $h_{\mathsf{pad}_{\mathsf{SN}}}$; and passes the decrypted $M = \mathsf{pad} \oplus C$ to the $\mathbf{S}_F$ subcomputation.

Roughly, the security of Zombie's precomputation follows from the zero-knowledge and soundness properties of the proof protocol, and the hiding and binding properties of the hash function H. Specifically, neither the hash $h_{\mathsf{pad}_{\mathsf{SN}}}$ nor $\pi_{E.2a}$ reveal $\mathsf{pad}_{\mathsf{SN}}$—thus, the middlebox cannot learn anything about the client's traffic. The collision-resistance of H and the soundness of the proof system prevent the client from lying about $\mathsf{pad}_{\mathsf{SN}}$, or the key used to derive it—thus, the client cannot equivocate about the sent message M.

3.2 Optimistic approval via asynchronous verification

While precomputation can greatly reduce the per-packet delay incurred by proof generation (down to under 300 ms, §6.1), some applications (for example, web browsing) require even less delay. This section describes how Zombie can perform the ZKP-related parts of its protocol asynchronously, that is, without blocking the flow of normal traffic. Client traffic passes optimistically while the middlebox detects policy violations retroactively.

In more detail, after the client encrypts its packet, it immediately sends the ciphertext C. The middlebox forwards this packet to the server and sets a timer. Then the client generates π and sends it to the middlebox. If the timer has not expired and π is valid, the middlebox does nothing. If the proof is invalid, or the timer fires, the middlebox takes some action, for example, blocking the client (§8).

Asynchronous verification requires the middlebox to keep track of unverified ciphertexts until the client sends the corresponding proofs. Done naively, this could require high memory usage at the middlebox. This memory usage can be reduced by storing a hash of the ciphertexts instead of the ciphertexts themselves. The client would re-send the ciphertext along with the proof, and the middlebox would use the hash to validate the ciphertext. This increases bandwidth usage, since a single ciphertext must traverse the client-middlebox link twice. Since bandwidth usage was not the bottleneck in any of our experiments, we did not implement this optimization.

Security. The middlebox cannot prevent non-compliant packets from leaving the network, nor can it prevent clients from receiving responses to such packets. Zombie will only eventually learn if this has occurred. We claim, though, that this relaxed security is sufficient for many applications. For DNS filtering, the policy goal is to prevent users from browsing blocked sites. Even if the user learns the IP address of a blocked site by sending a non-compliant DNS query, as long as the middlebox can detect this reasonably quickly, further browsing can be blocked. As another example, if Zombie is used to stop users from uploading sensitive data to external sites, it may be sufficient to detect and shut down uploads in time to prevent too much sensitive data from being uploaded, even if (say) the beginning of a file is successfully uploaded. Other context-specific policies may be appropriate, for example a middlebox might optimistically send packets to servers but hold the response packets pending proof verification.

3.3 Batching in Zombie

The final protocol improvement Zombie makes is batch proof generation and verification. Concretely, given ciphertexts $C_1, \ldots, C_b$, Zombie can generate one proof π that verifies only if all b underlying plaintexts are policy-compliant. This single proof is much more efficient for the middlebox to verify than b separate ones. The batch size does not need to be fixed by the middlebox; it can be dynamically chosen by clients based on their current workloads. Figure 2 depicts the conditions under which batching is useful. The combination of batching and asynchronous verification is potent: clients can gather larger batches because they can wait longer to send proofs. Batching is also complementary to precomputation: the client can batch together multiple $\pi_{E.2a}$ proofs.

At a high level, batching works by modifying Zombie's underlying ZKP protocol, Spartan [93], to allow for parallel runs of proof generation to share randomness. Sharing randomness to batch proofs is a known technique, and has been used in other modern proof systems [95, 113]; however, its application to Spartan in this work is novel. We use "SpartanBatch" to refer to our variant of Spartan that supports batching. Appendix A gives details and security analysis.

4 Regular expressions in Zombie

This section describes how Zombie supports middlebox functionality based on regular expressions, which we sometimes call *regexps*. Regexps feature in real-world policies for data

loss prevention (DLP) [71], intrusion detection (IDS) [22, 39], and traffic classification [119, 120]. For example, a DLP system might use a regexp to specify that all outgoing packets containing a social security number should be blocked.

The high-level picture is as follows. Zombie begins with a policy P that uses regexps. This policy (§3) is a restriction on the plaintext payloads (which this section calls simply *payloads*) allowed to pass through the middlebox, and is expressed as a computation $\mathbf{S}_P$ that takes the payload as input and returns 1 or 0 depending on whether the payload adheres to the policy. The policy can be as simple as whether any substring of the payload matches the given regular expression. Or it could include more sophisticated combinations, for example, whether two regexps match within close proximity, or whether there are more than four matches to a given regexp.

Zombie produces both a constraint representation of the computation $\mathbf{S}_P$ and a prover recipe for executing this computation and satisfying those constraints. The constraints $\mathcal{C}_P$ are constructed to be satisfiable if and only if the prover correctly reports whether the payload adheres to P.

As we have described (§1–§2), constraints are an inefficient way to represent general-purpose computations. The same holds for regular expressions: one cannot simply take $\mathbf{S}_P$ to be a regexp library parameterized by a specific regexp, because that would involve compiling, say, C code that uses program constructs that are prohibitive when expressed in constraints.

For this reason, we depart from prior work on regular expressions [119, 120], which aims to make *matching* fast. In our context, matching is the step where the prover executes its recipe to identify a satisfying assignment, and this step is swamped by the costs of proving. Instead, we focus on the driver of those costs: number of constraints (§2). Specifically, our metric of interest is constraints per character in the payload, which we want to be small.

The rest of this section describes how Zombie lowers this metric versus a naive approach. Zombie introduces a series of techniques that achieve substantial improvements in both constants and asymptotics (§6).

4.1 Setup and framework

A given policy P comprises one or more regexps, Boolean combinations of them, and proximity checks. So $\mathbf{S}_P$ has one or more subcomputations, which we denote $\mathbf{S}_R$, referring to a specified regular expression R.

The input to one such $\mathbf{S}_R$ is the payload T (of length L_T); typically, L_T is in the thousands (the number of bytes in a plaintext network packet). The output of a given $\mathbf{S}_R$ is an *array* of L_T Boolean variables; slot ℓ is True if there is a match to R ending at position ℓ and False otherwise; notice that $\mathbf{S}_R$ thus captures not only whether the given R matches any substring(s) of T but also the (ending) position of the match(es).

$\mathbf{S}_P$ processes the output array produced by $\mathbf{S}_R$, or multiple such arrays if there are multiple regexps. Section 4.7 describes that process in detail; until then, we focus on a given $\mathbf{S}_R$.

Zombie encodes $\mathbf{S}_R$ in constraints via several translation phases: $R \to FA \to IR \to \mathcal{C}_R$, where $\mathcal{C}_R$ is the constraint representation of $\mathbf{S}_R$, FA is a finite automaton, and IR is an intermediate representation that has Boolean logic (`AND`, `OR`, `NOT`), augmented with equality and inequality tests (`==`, `!=`, `<=`, etc.).

Sections 4.2–4.6 describe the main ideas in this translation: a new string matching primitive (§4.2), Zombie's translation from NFAs to constraints (§4.3), a new arithmetization of Boolean logic to substantially lower the cost of encoding Boolean `OR` (at the expense of Boolean `NOT`) (§4.4), techniques for rewriting the regular expression to admit a more efficient translation (§4.5), a new FA formalism that memoizes the results of character class matching (§4.6), and finally exploiting structure in character classes (§4.6).

4.2 Efficient string matching in constraints

Suppose R represents a fixed string, say `a{k}` (`a` repeated k times), so $\mathbf{S}_R$ must determine for each $\ell \in \{0, \ldots, L_T - 1\}$ whether the pattern appears in the payload, ending at position ℓ. If so, a Boolean $b^{(\ell)}$ is 1 and otherwise 0. For illustration, we skip FA, so the translations are $R \to IR \to \mathcal{C}_R$. The IR is:

$$b^{(\ell)} := (T[\ell]\texttt{==a}) \wedge (T[\ell-1]\texttt{==a}) \wedge \cdots \wedge (T[\ell-k+1]\texttt{==a}).$$

To encode this in R1CS constraints (§2), one expresses $\wedge$ using field multiplication and `==` using `EQUALS-ZERO` (§2, see also [96, Appx D]):

$$\begin{aligned}
b^{(\ell)}_{k-1} &:= \texttt{EQUALS-ZERO}(T[\ell-k+1]-\texttt{a}) \\
b^{(\ell)}_{k-2} &:= b^{(\ell)}_{k-1} \cdot \texttt{EQUALS-ZERO}(T[\ell-k+2]-\texttt{a}) \\
&\cdots \\
b^{(\ell)}_{1} &:= b^{(\ell)}_{2} \cdot \texttt{EQUALS-ZERO}(T[\ell-1]-\texttt{a}) \\
b^{(\ell)} &:= b^{(\ell)}_{1} \cdot \texttt{EQUALS-ZERO}(T[\ell]-\texttt{a}) \qquad (1)
\end{aligned}$$

Notice that $b^{(\ell)}$ equals 1 iff there is a match, and 0 otherwise.

Of course, expression (1) is not literal constraints. To produce those, one expands lines of the form $b^{(\ell)}_i = b^{(\ell)}_{i+1} \cdot \texttt{EQUALS-ZERO}(T[\ell-i]-\texttt{a})$, as follows:

$$\left\{ \begin{array}{l} b^{(\ell)}_i = b^{(\ell)}_{i+1} \cdot M_i, \\ M_i \cdot (T[\ell-i]-\texttt{a}) = 0, \\ Z_i \cdot (T[\ell-i]-\texttt{a}) = 1 - M_i \end{array} \right\}$$

The variable M_i represents the outcome of $\texttt{EQUALS-ZERO}(T[\ell-i]-\texttt{a})$, and Z_i is non-deterministically supplied. Altogether, $\mathbf{S}_R$ for this pattern requires roughly $3 \cdot k$ constraints per character position, so $3 \cdot k \cdot L_T$ in all.

As a more efficient alternative, Zombie introduces a primitive: `STRING-MATCH`. `STRING-MATCH` exploits the observation that, in constraints, the indivisible unit (akin to a bit on a

CPU) is a finite field element, which holds many bits, and thus conceptually "has room" for packing the information about whether many characters matched. Letting Λ be the alphabet, $|\Lambda|$ be its size (256 for ASCII), and S_1, S_2 be two strings:

$$\texttt{STRING-MATCH}(S_1[0]\ldots S_1[k-1], S_2[0]\ldots S_2[k-1]) \triangleq \texttt{EQUALS-ZERO}\left(\sum_{i=0}^{k-1} |\Lambda|^i \cdot (S_1[i] - S_2[i])\right).$$

Zombie replaces expression (1) with $\texttt{STRING-MATCH}(T[\ell-k+1]\ldots T[\ell], \texttt{a....a})$, which (assuming loose limits on k; see below) is 2 constraints per input character, down from $3 \cdot k$. To see why, note that the argument to EQUALS-ZERO is a weighted sum of the variables $T[i]$ plus a constant term, with the weights and constant term known at compile time. Plugging that argument into EQUALS-ZERO keeps the constraints in R1CS format (§2).

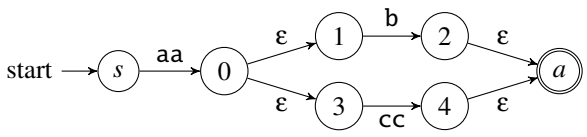
The loose limits are determined by the size of the alphabet and the size of the field that the constraints are expressed over. Assuming a field of size q, the maximum length of a pattern that can be compiled into a single STRING-MATCH is $\lfloor \log_{|\Lambda|}(q) \rfloor$. For our application, we consider the alphabet of ASCII characters ($|\Lambda| = 2^8$) and a 255-bit prime field (the base field of curve25519 [11]); thus, patterns of at most 31 characters can be compiled into a single STRING-MATCH.

If these loose limits do not hold, the pattern compiles into several STRING-MATCHs, connected by AND ($\wedge$).

4.3 From regular expressions to constraints

Real-world systems [25, 58, 67] translate regular expressions to executable code in two steps. First, they produce a nondeterministic finite automaton (NFA), via Thompson's algorithm [104]. Second, they determinize the NFA to get a DFA [101, Ch. 1]. This step represents the DFA's state transition function as a table: an entry for every state and every character. This representation makes execution very fast. However, in our context, the entire exponentially-sized table would turn into constraints, exploding proving costs.

Thus, Zombie stops after the Thompson step. Because of its packing technique, Zombie produces FAs that have *string* transitions instead of the usual character transitions. As an example, consider the regular expression: aa(b|cc). Here is the NFA (ε refers to the empty string; s and a are the start and accepting states):



Zombie's *IR* representation of this FA uses functions, one for the final state and each intermediate state that has non-epsilon incoming transitions. Each function encodes, for each character position ℓ, whether the FA could be in the given state at character position ℓ.

$$\begin{aligned} f_0(\ell) &:= \texttt{STRING-MATCH}(T[\ell-1]T[\ell], \texttt{aa}) \\ f_2(\ell) &:= f_0(\ell-1) \wedge (T[\ell]\texttt{==b}) \\ f_4(\ell) &:= f_0(\ell-2) \wedge \texttt{STRING-MATCH}(T[\ell-1]T[\ell], \texttt{cc}) \\ f_a(\ell) &:= f_2(\ell) \vee f_4(\ell) \end{aligned} \quad (2)$$

Translating a function $f(\cdot)$ to constraints means that each evaluation $f(0), \ldots, f(L_T - 1)$ is separately translated and possibly assigned to a constraint variable. For example, $f_2(\ell) \vee f_4(\ell)$ translates to $f_2[\ell] + f_4[\ell] - f_2[\ell] \cdot f_4[\ell]$, where $f_2[\ell]$ is a constraint variable that represents $f_2(\ell)$. Notice that the translation of $\vee$ requires a constraint, because of the multiplication. Also, each AND ($\wedge$) translates to a constraint that multiplies ($\cdot$) its terms. So, expression (2) is 9 constraints for each position ℓ (2 for ==, 2 for each of two STRING-MATCH, and 1 for each of the three multiplications). Notice from the definition of STRING-MATCH earlier that the cost is relatively insensitive to the length of the substrings. For example, if the pattern were a{k}(b{k}|c{k}) (a k-length run of a followed by a k-length run of b or c), then the number of constraints is unchanged (assuming the aforementioned loose limits on k).

4.4 A new arithmetization of Boolean logic

Traditionally, when *arithmetized*—that is, translated to constraints—Boolean logic maps True to 1 and False to 0. Letting p, q, r be Boolean variables [8, 85, 94–97]:

$r := p \wedge q$	customarily translates to:	$r = p \cdot q$
$r := p \vee q$	customarily translates to:	$r = p + q - p \cdot q$
$q := \neg p$	customarily translates to:	$q = 1 - p$

Above, multiplication ($\cdot$) and addition ($+, -$) are over the underlying finite field $\mathbb{F}$ (§2).

Zombie introduces an alternate arithmetization: False still maps to 0 but any non-zero value in the underlying finite field functions as True:

$r := p \wedge q$	now translates to:	$r = p \cdot q$, as above
$r := p \vee q$	now translates to:	$r = p + q$ (assuming no overflow; see below)
$q := \neg p$	now translates to:	$q = \texttt{EQUALS-ZERO}(p)$

For example, in (2), $f_a(\ell)$ translates to $f_2[\ell] + f_4[\ell]$, shedding the term $f_2[\ell] \cdot f_4[\ell]$. This concretely goes from 9 to 8 constraints.The source of the savings is that $f_a(\ell)$ no longer needs a constraint itself: any other constraint that uses $f_a(\ell)$ can substitute in the sum $f_2[\ell] + f_4[\ell]$. Notice that any such substitution retains R1CS format (§2), whether the substitution happens in the "A"-part of the constraint, the "B"-part, the "C"-part, or combinations thereof. That is, $f_2[\ell]$ and $f_4[\ell]$

are components of the z vector from Section 2, and their inclusion in a constraint simply adds 1 to the corresponding coefficients. More generally, arithmetizations that are linear combinations (that is, no degree-2 terms, meaning no multiplications of two or more variables) cost no constraints. We will use this fact over and over again.

Consequently, `OR` has become mostly free: addition of degree-1 terms, being a linear combination, doesn't require constraints. We say *mostly* because, for this to work, $p+q$ must not overflow, that is, wrap around the finite field modulus and become 0 when at least one of the summands is non-zero. Our implementation of Zombie (§5) handles this issue at compile time. The compiler tracks the maximum possible value of variables and, if overflow is possible, inserts constraints to reduce a summand to a 0-1 term before it has the chance to overflow. The specific constraints are `NOT-EQUALS-ZERO` [96, Appx D], which maps 0 to 0 and non-zero values to 1.

By contrast, `NOT` ($\neg$) has gone from free (because it was a linear combination) to requiring two constraints, for `EQUALS-ZERO` (see §2). Finally, `AND` ($\wedge$) costs one constraint in both arithmetizations. The overall trade, then, is to make `NOT`s more expensive in exchange for free `OR`s.

This trade not only is a dramatic improvement but also carries broader significance. In our context, the 9-to-8 savings in the earlier example is a restricted case; in fact, this arithmetization has a quadratic-to-linear improvement. To see why, consider a state that has $s-1$ inbound paths, one for each of the other states in an s-state FA. For example: $f_a(\ell) = f_1(\ell) \vee f_2(\ell) \vee \cdots \vee f_{s-1}(\ell)$. In the traditional arithmetization, each disjunct requires a constraint with a field multiplication, each of which costs one constraint; the total for $f_a(\ell)$ in this example is s constraints. In the worst case, then, $O(s)$ states can each require $O(s)$ constraints, for a total of $O(s^2)$ constraints for each $\ell \in \{1, \ldots L_T\}$. In Zombie, by contrast, f_a would be translated into $f_1[\ell] + f_2[\ell] + \cdots + f_{s-1}[\ell]$. This costs 0 constraints because it is a linear combination.

Qualitatively, this arithmetization means that Zombie gains enormously from devising *IR* representations that use mainly `OR`, with `AND` entering only when necessary. Beyond Zombie, this point applies to the constraint translation of any problem naturally expressed with many conjunctions and disjunctions, such as 3-SAT.

4.5 Preprocessing regular expressions

Another technique in Zombie is rewriting regular expressions at compile time to favor longer substring matches. Doing so exploits packing (§4.2) to reduce the number of `AND`s and the number of states in the IR. For example, Zombie rewrites `aa(b|cc)` as `(aab|aacc)`, yielding the following *IR*, which should be compared to (2):

$$f_0(\ell) := \texttt{STRING-MATCH}(T[\ell-2]T[\ell-1]T[\ell], \texttt{aab})$$
$$f_1(\ell) := \texttt{STRING-MATCH}(T[\ell-3]T[\ell-2]T[\ell-1]T[\ell], \texttt{aacc})$$
$$f_a(\ell) := f_0(\ell) \vee f_1(\ell) \quad (3)$$

Whereas we saw in the previous section that the formulation in (2) costs 8 constraints for each character position ℓ, the one in (3) costs 4 constraints (two for each `STRING-MATCH`).

4.6 Character classes and a new FA formalism

A common and convenient feature of regular expressions is *character classes*, for example, `[0-9]` or `[A-Za-z]`, which respectively match any digit and any ASCII alphabet character. Naively treating a character class as a union (using the `|` operator) would be expensive. Although real-world regexp frameworks have special optimizations for character classes, these would not contribute to efficient constraint representations, for the reasons discussed at the beginning of this section. Instead, Zombie applies several of its own optimizations.

First, Zombie deduplicates so that the costs associated with matching to a class are paid once, even if there are multiple instances of the class in the regular expression. To do so, Zombie constructs a new kind of FA, one that uses "sub-FAs" to *write* to separate tapes (FAs are not typically modeled as writing to a tape) and then reads the tapes in the "main" FA. The sub-FAs are each supposed to produce an array of Booleans. As an example, consider the regexp `[0-9]a[0-9]`. Zombie produces the following *IR*:

$$t_0[\ell] := \texttt{MATCH-CLASS}(T[\ell], \texttt{[0-9]})$$
$$f_a(\ell) := \texttt{STRING-MATCH}(t_0[\ell-2]T[\ell-1]t_0[\ell], 1_{\mathbb{F}}\texttt{a}1_{\mathbb{F}})$$

$1_{\mathbb{F}}$ is 1 in the finite field and is used to encode the Boolean result of `MATCH-CLASS`. Think of t_0 as memoizing the sites of matches found by a sub-FA; notice how the values in t_0 are reused in $f_a(\cdot)$.

Outside of the present context, the requirement for an additional tape would seemingly require more memory for the prover. In our context (constraints), each extra tape *saves* memory, by reducing the number of variables necessary to represent a match to the character class.

Besides deduplication, another benefit of Zombie's FA formalism is that it enables longer substring matches. The idea is similar to the example in Section 4.5. Here, the packing technique (§4.2), this time applied to the results of other tapes, lets f_a consist of a single `STRING-MATCH`. Conversely, can we use deduplication on that earlier example? No, because the union components were different lengths.

As another optimization, Zombie exploits structure in the character class. For example, Zombie encodes `[A-Za-z]` with only 25 constraints (fewer than the 52 characters in the class!):

$$\texttt{MATCH-CLASS}(T[\ell], \texttt{[A-Za-z]}) = (T[\ell] \ \texttt{>=}\ \texttt{A}) \wedge (T[\ell] \ \texttt{<=}\ \texttt{z}) \wedge \cdots$$

The elided terms check that $T[\ell]$ is not one of the few ASCII characters between `Z` and `a`. This approach relies on the IR primitives <= and >=, which translate to $\log_2|\Lambda|$ + 1 constraints [96], which is 9 if Λ is the 8-bit ASCII characters.

Zombie's compiler tries to optimize the encoding of a class; for example, treating the class `[0-9]` as a range with <= and >= operators is not worthwhile. Larger classes see greater benefit from being treated as a range.

4.7 Applying regexp-based policies in ZK

In this section, we move from considering a single $\mathbf{S}_R$ to a higher-level policy P, expressed as $\mathbf{S}_P$.

One-shot expressions. Consider a basic case: P is simply whether there is a match to some R somewhere in the payload. Recall that $\mathbf{S}_R$ is already encoded as constraints for $\{f_a(\ell)\}_{\ell=0,\ldots,L_T-1}$. $\mathbf{S}_P$, then, is `NOT-EQUALS-ZERO`$(f_a(0) \vee f_a(1) \vee \cdots \vee f_a(L_T-1))$. Assuming no overflow (so the `OR`s are free; §4.4), the overhead of $\mathbf{S}_P$ beyond $\mathbf{S}_R$ is two constraints, stemming from `NOT-EQUALS-ZERO` (§4.4). Zombie's compiler handles possible overflow as described earlier (§4.4).

Proximity. In network security, simple regexp searches can have too many false positives. Thus, the policy P is sometimes concerned with context: individually two patterns are not sensitive, but close together they are. For example, a DLP policy might disallow a pattern matching a driver's license number within 100 characters of strings like "driving license," "driver's license," "DL," etc. (§6.3).

Perhaps surprisingly, Zombie can handle such policies with very little overhead beyond the cost of matching the individual regexps. Consider a computation $\mathbf{S}_P$ that returns 1 if there are respective matches to two regular expressions R_1 and R_2 within d characters of each other. Notice that, for correctness, all possible combinations of occurrences of the two patterns have to result in $\mathbf{S}_P$ returning 1. To capture these possibilities, $\mathbf{S}_P$ performs two steps. First, it takes the f_a array of R_1, call it f_{r_1}, and produces a new array $f^d_{r_1}$, which for each position ℓ holds a Boolean indicating whether there is a match within d characters of ℓ. Concretely,

$$f^d_{r_1}[\ell] = \sum_{k=1-d}^{d-1} f_{r_1}[\ell+k].$$

Because each $f^d_{r_1}[\ell]$ is a linear combination of existing variables, there is no cost in constraints to produce it (§4.4). Second, $\mathbf{S}_P$ checks whether the entrywise product of $f^d_{r_1}$ and the f_a array of R_2, call it f_{r_2}, has any non-zero entries. This check requires L_T+2 constraints: one for each product, and two for a `NOT-EQUALS-ZERO` applied to the sum of these products.

Thus, in total, the requirement for proximity costs an amortized 1 constraint per character in the payload. In contrast, naively encoding proximity as a single regexp, $(R_1(.\{0,d\})R_2)\,|\,(R_2(.\{0,d\})R_1)$, would introduce an extra $O(L_T \cdot \log d)$ constraints.

5 Implementation

Our implementation of Zombie has two main components: a client and a middlebox. We currently support two classes of applications. The first is DNS filtering [49] (see also §2), as applied to the DNS-over-TLS and DNS-over-HTTPS protocols. The second is arbitrary policies involving regular expressions, for example DLP policies for files sent by clients via HTTPS, applied to text files (as opposed to formats such as PDF).

5.1 ZKP implementation

Circuits. The circuits used for Zombie's ZKPs are specified in the ZoKrates domain-specific language (DSL) [34] and compiled to R1CS using CirC [84], a ZKP compiler framework. The circuits comprise 1832 handwritten lines of ZoKrates code and 630 lines automatically generated by our own regexp compiler (which is a standalone component that could be integrated with other projects). The handwritten code was optimized and features a large improvement in the encoding of $\mathbf{S}_F$ (§3).

The regexp compiler takes as input (a) a list of regexps and (b) a list of proximity restrictions between pairs of regexps. Using built-in knowledge of the constraint-level costs of ZoKrates' semantics, the compiler applies the techniques in Section 4. The compiler is 5425 lines of C++, 463 lines of `yacc`, and 50 lines of `lex` code on top of the BNFC library [1].

ZKP improvements. In implementing Zombie, we made several improvements to CirC and the existing Spartan implementation. First, we created an adapter that integrates CirC with Spartan. We have also configured Spartan to use curve25519 [11] as its underlying cryptographic group, a standard choice believed to offer ≈128 bits of security.

Our CirC improvements make witness generation more efficient; in early experiments, witness generation was slower than proof generation. We modified internal CirC data structures to prevent unnecessary memory copying, which greatly improves performance.

We improved the Spartan prover and verifier to take full advantage of parallelism, resulting in better performance for generating multiple proofs even in the non-batch setting.

5.2 Client implementation

The client implementation comprises 1976 lines of Python. When performing DNS filtering, the Zombie client acts as a local DNS proxy. It accepts UDP DNS requests then sends them to a recursive DNS resolver (we use Google's 8.8.8.8 resolver [46]) over TLS (DoT [53]) or HTTPS (DoH [52]). The web browser is configured to point to the local proxy for DNS resolution. The client performs the channel-opening (§3) with the Zombie middlebox on startup to set up a session. It uses this session for as long as the recursive resolver will allow (up to five minutes in our testing). It generates and sends proofs, and forwards traffic to the middlebox; we ensure this via routing tables.

Precomputation. In our implementation, the client has a child process for precomputation (§3.1) that has lower priority than the main proxy process, constantly generating *pad-*

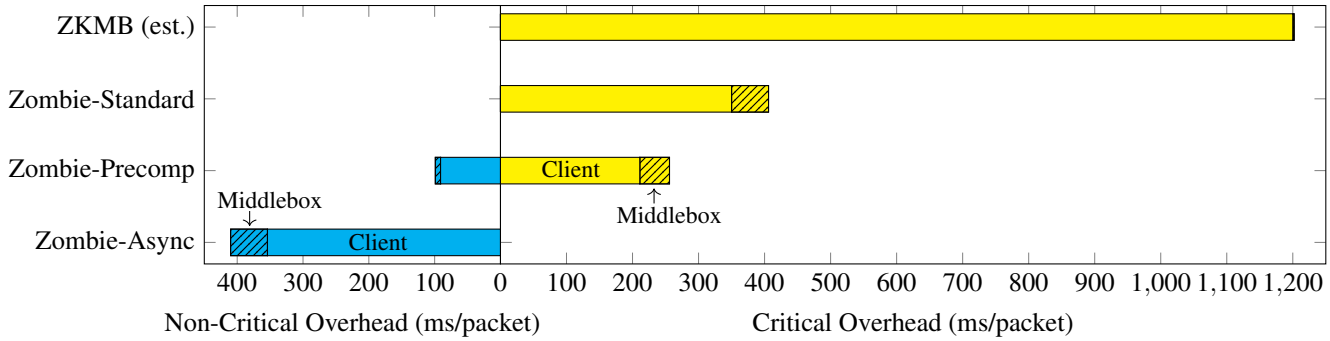


Figure 4: Per-Packet critical and non-critical overheads (relative to a no-policy baseline) for enforcing a blocklist on DNS requests over TLS in zero-knowledge with ZKMB [49] and with Zombie-Standard, Zombie-Precomputation, and Zombie-Asynchronous configurations.

commit proofs when the proxy is idle. When generating these proofs ($\pi_{E.2a}$) (§3.1), the client has two parameters to balance. First is the length of the pad, L_{pad}, which, for each proof, must match or exceed the size of each packet to be encrypted. Choosing a larger L_{pad} will result in more proving work, but will allow the client to send larger packets. The second parameter, m, is the number of *pad-commit* proofs to batch to amortize prover and verifier work. A higher m is less likely to be exhausted by a burst of traffic, but it risks performing excess precomputation that might not be used. In our implementation, for DNS-over-TLS we set $L_{pad} = 255$ (the size of DNS request payloads) and $m = 16$.

5.3 Middlebox implementation

The middlebox is implemented in 1595 lines of Rust. The middlebox configures IP packet filter rules using `iptables`. When packets arrive at the middlebox, they are put on a queue implemented via `libnetfilter_queue` in Linux. The middlebox dequeues packets and performs the following steps. First, it determines whether they are policy-relevant. If so, the middlebox increments the TLS sequence number; it needs to have an accurate count of the sequence number to verify proofs. Then, it buffers the received packet for verification. When the middlebox receives the proof from the client, it links the proof to the packet by sequence number, verifies the proof, and forwards it.

6 Evaluation

We evaluate Zombie with these questions:

(1) What are the overheads added by different configurations of Zombie?

(2) How does batching improve middlebox throughput?

(3) What are the costs of different components of Zombie?

(4) How effective are Zombie's regexp techniques?

Method, applications, and baselines. Our experiments measure client, server, and overall end-to-end delay introduced by Zombie, and we compare these overheads against those introduced by the original ZKMB work [49]. We evaluate Zombie for DNS filtering and DLP policies applied to traffic over TLS 1.3. The DNS filtering benchmarks use a representative adult-content domain blocklist from prior work [49, §8.2] [69] (with 2 million domains). The DLP benchmarks use policies from Microsoft DLP Purview [71].

Our experiments that require networking run on CloudLab [33] while those that do not are run on Amazon Web Services (AWS). On CloudLab, we use `c6525-25g` instances. Each has a 16-core 3GHz AMD 7302P CPU, with 128GB RAM, SSDs, and two Mellanox 25Gb/s NICs. On AWS, we use an instance with a 16-core 2.90GHz Intel 8375C CPU with 128GB RAM and SSDs. This instance has similar computation resources to that of CloudLab `c6525-25g`.

6.1 Computational overhead and delay, no batching

We measure the overhead introduced by Zombie with respect to a no-policy setup, in which the middlebox forwards the traffic without enforcing any policy. We also compare this overhead to ZKMB [49], via conservative estimates of ZKMB's performance extrapolated from microbenchmarks on the same hardware; this is labeled "ZKMB (est.)".

We run three configurations of Zombie, with the following settings of precomputation and synchrony:

(1) *Zombie-Standard*: No precomputation, no asynchrony.

(2) *Zombie-Precomputation*: Precomputation, no asynchrony.

(3) *Zombie-Asynchronous*: No precomputation, asynchrony.

We do not evaluate precomputation combined with asynchrony here; asynchrony moves all overheads to the non-critical path, so precomputation does not affect latency in this case. Batching can be applied to all of the above; we evaluate it separately (§6.2). We run this experiment on CloudLab and report the average of 20 255-byte DNS requests.

Figure 4 depicts the results. Zombie-Standard incurs approximately 3× lower latency than ZKMB. A major difference in client and middlebox work comes from the transition

from the Groth16 [48] proof system to Spartan [93] (§2). We see additional gains from the enhancements detailed in Section 5. The average additional latency is about 400 ms for Zombie-Standard: approximately 350 ms for proof generation and 50 ms for verification.

With precomputation, the average latency is lower, about 250 ms. While much worse than the average latency for a DNS request, which is about 20 ms [86], it may still be tolerable (§8). With optimistic approval via asynchronous verification, Zombie introduces no additional latency. Of course, asynchronous mode requires (in addition to assumptions about policy enforcement; §3.2) additional storage requirements for the middlebox, for buffering packets. We return to these storage requirements when evaluating batching, as the effects on storage are more pronounced there.

Observe that Zombie-Standard and Zombie-Asynchronous introduce the same total per-packet overhead. This is expected: they involve proving and verifying the exact same statements. Surprisingly, Zombie-Precomputation requires slightly less overall per-packet overhead despite the need to produce and verify more proofs than Zombie-Standard. This decreased overhead is because the work to generate multiple *pad-commit* proofs is parallelized in our implementation.

Turning to the communication overhead of sending proofs: in the synchronous setting with precomputation, each online proof is approximately 30 KB. As we will argue (§8), the overall increase in required bandwidth is expected to be small.

6.2 Effect of batching

We investigate the effect of batching (§3.3) on the throughput (number of (proof, packet) pairs processed per second) and the storage requirements of the middlebox. We do not separately evaluate batch proof size; a batch of proofs is the same size as the sum of the non-batched proofs.

We run the DNS benchmark in a new Zombie-Async-Batch configuration. We model each client as a Poisson process, whose parameter is scaled by the batch size. For example, if the batch size is 8, then we set the average interarrival time to be 8× longer than when the batch size is 1, and when an arrival event happens, the client sends 8 packets. For each batch size, as offered load increases, throughput does not collapse, but instead it approaches a maximum. We interpret this maximum throughput as the middlebox's empirical capacity for that particular batch size.

We measure this quantity under four batch sizes; for each size, we average three experiments. We also create a model to predict maximum throughput: we measure the time to verify a single Zombie DNS proof on one thread on CloudLab, and individually measure the execution time of the fixed cost code block and the marginal cost code block, obtaining 121 ms per batch and 38 ms per proof, respectively.

Figure 5 compares the empirical measurements and the model; the divergence is around 5%. This discrepancy owes to lower-order middlebox costs related to packet forwarding,

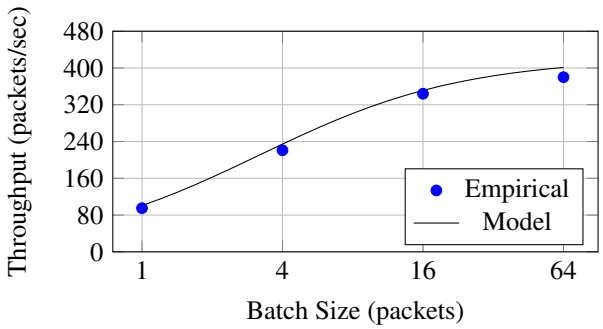


Figure 5: Middlebox throughput vs batch size.

listening, and proof parsing. The maximum throughput we observe is 380 packets per second, at a batch size of 64. Although larger batches would further increase throughput, the possible improvement levels off (even in theory); besides, larger batches are impractical.

Batching brings additional storage requirements (to accumulate ciphertexts). However, storage capacity is not a limiting factor here, owing to the small size of DNS requests and the throughput of 380 packets per second. Even if we assume that the middlebox is exactly at capacity, never falls behind, always has a proof to check (even if it might be waiting on other proofs), and has a generous window of 60 seconds for proofs to arrive after the first corresponding ciphertext in the batch, then the middlebox would still need to store less than 6 MB of ciphertexts at any given time. This is well within the capacity of even the smallest of middleboxes [83].

6.3 Circuit benchmarks

To pinpoint the costs of individual components, we run the prover and verifier in isolation on AWS, taking the average of 5 executions, for various circuits. Figure 6 shows the results.

DNS-related circuits. The first five rows are related to the DNS blocklist policy. The first and second rows serve to compare Zombie and ZKMB, using the same circuit as the one for the Zombie-Standard and Zombie-Asynchronous benchmarks in Figure 4. We see a significant reduction in the number of constraints in Zombie, which owes to the optimizations mentioned in Section 5; thus, the performance gain in Figure 4 for Zombie results not only from the Spartan back-end but also our optimizations. The total number of constraints for the third row is less than one-third of the number of constraints in the second row, indicating that the dominant cost in our implementation of Zombie-Standard is $\mathbf{S}_{E.2}$. The sum of the constraints in the fourth and fifth rows (measuring offline and online cost with precomputation) is slightly larger than the number of constraints in the second row; this is expected because there is extra work to commit to $\mathsf{pad}_{\mathsf{SN}}$ (Figure 3).

Regexp circuits. The last four rows of Figure 6 include costs for a DLP benchmark. This benchmark combines five Microsoft Purview policies for detecting sensitive information in the US locale: bank account number [72], driver's license

Benchmark: Circuit description and size								
$\mathbf{S}_E$	$\mathbf{S}_F$	$\mathbf{S}_P$	**Payload Size**	**Constraints**	**Prover Time**	**Verifier Time**	**Memory**	**Proof Size**
E.2	DNS	Blocklist	255 B	*176000*	*1200 ms*	*2 ms*	-	*128 B*
E.2	DNS	Blocklist	255 B	128702	365.4 ms	39.8 ms	531.8 MB	30.2 KB
	DNS	Blocklist	255 B	40295	191.2 ms	29.8 ms	212.3 MB	21.3 KB
E.2a			255 B	86568	278.0 ms	36.4 ms	377.2 MB	30.2 KB
E.2b	DNS	Blocklist	255 B	48562	221.4 ms	31.2 ms	258.5 MB	21.3 KB
E.2	HTTP	Microsoft DLP	100 B	64438	207.0 ms	29.0 ms	225.7 MB	21.3 KB
E.2	HTTP	Microsoft DLP	2000 B	1186241	6363.8 ms	319.6 ms	10114.6 MB	49.1 KB
	HTTP	Microsoft DLP	100 B	20080	112.2 ms	23.2 ms	89.2 MB	20.5 KB
	HTTP	Microsoft DLP	2000 B	490966	4909.0 ms	245.4 ms	8402.9 MB	48.0 KB

Figure 6: Costs of various circuits in Zombie. The first row shows estimated overheads for the prior work, ZKMB [49], using the Groth16 [48] back-end (§2). Memory is a single column because the prover and verifier have the same memory requirements with the Spartan back-end. Blank cells indicate that a subcomputation of that type is not included in that row's benchmarked circuit.

number [73], taxpayer number (ITIN) [74], social security number [75], and passport number [76]. These policies use substring matches, regular expressions, and proximity checks. A message must pass all five to pass; for brevity, we refer to the overall policy simply as "Microsoft DLP". We encode this policy using Zombie's regexp pipeline (§4–§5) and benchmark it on HTTP POST messages of varying sizes.

Per byte, the cost of this policy is 25–50% more expensive than the DNS blocklist benchmark. This can be seen by comparing the DNS blocklist (no $\mathbf{S}_{E.2}$) and HTTP Microsoft DLP (no $\mathbf{S}_{E.2}$) rows, and dividing the metrics by the payload size. The result is approximately 200–245 constraints per byte for DLP and 160 constraints per byte for the blocklist.

Packet size brings a complication. However, this is not because of latency, which scales linearly with packet size, so overall latency is driven by total bytes (in a policy-relevant flow), rather than the distribution of bytes over packets. The main issue with packet size is in memory consumption, which scales linearly with circuit size, and reaches into the GB in our experiments (Figure 6). The mitigant is that the memory cost is paid once per circuit, and circuits are reusable, creating benefit from small packets. Concretely, using a circuit for a 100-byte packet 20 times is much better for middlebox memory than using a circuit for a 2000-byte packet once.

Regexp circuit techniques. Figure 7 depicts the results of each of our regexp techniques in more detail. We show the decrease in per-byte overheads from incrementally applying the optimizations discussed in Section 4. We use the Microsoft DLP policy for these benchmarks and exclusively consider the $\mathbf{S}_P$ subcomputation.

For this policy, our optimizations reduce the per-byte overhead by almost an order of magnitude, from over 20 ms per byte to under 2 ms per byte. The most substantive improvements come from `STRING-MATCH` (§4.2) and from creating multiple tapes (§4.6).

To contextualize these costs, we compare them to the per-byte overhead of decryption, approximated by fixing a payload size, subtracting the costs for a given row without $\mathbf{S}_{E.2}$ from the corresponding cost with $\mathbf{S}_{E.2}$ (or by looking at the $\mathbf{S}_{E.2a}$ row), and dividing by the payload size. The result is 348 constraints per byte, which should be compared to the 242 constraints per byte at the bottom of Figure 7 (down from 1566, at the top of the figure). That is, after the techniques in Section 4, the dominant cost is no longer regular expression handling but rather representing decryption in the circuit.

Techniques	**# Constraints**	**Prover time**	**Verifier time**
Baseline	1566 / B	*19000 ms*	*1400 ms*
+ `STRING-MATCH`	996 / B	*11000 ms*	*860 ms*
+ Alt Arithmetization	901 / B	*10000 ms*	*760 ms*
+ Regexp Preprocessing	873 / B	*9800 ms*	*740 ms*
+ Additional Tapes	288 / B	*2300 ms*	*160 ms*
+ Optimized Classes	242 / B	1705 ms	38 ms

Italicized entries indicate estimates.

Figure 7: Effect of techniques, in the order they are introduced in Section 4, on the Microsoft DLP policy (1 KB payload). The final line is the result of running CirC on the zok file produced from all optimizations. The times in the other lines are estimates from the compiler with all prior optimizations enabled and all subsequent optimizations still disabled.

7 Related work

Systems built using probabilistic proofs. Probabilistic proofs are a foundational concept in complexity theory with a deep and rich literature [6,7,9,42,43]; for a survey, we recommend Goldreich [41]. The last decade has seen rapidly growing interest from the applied cryptography community, with a particular emphasis on zero-knowledge proofs. For a survey, we recommend Walfish and Blumberg [117] or Thaler [103].

Zombie is part of a growing line of work applying proba-

bilistic proofs to solve practical problems, such as privacy-preserving payments [28, 91] private smart contracts [15, 19, 57], proofs of solvency [4, 27], verifiable delay functions [14, 55], proofs of software vulnerability [26] and cryptographic transparency logs [21, 107, 108]. Of particular relevance to our work are DECO [123] and Reclaim [100], which employ probabilistic proofs about TLS plaintext. However, both systems aim to prove statements about a TLS session (e.g. "My bank account balance is greater than $\$X$") to an out-of-band third party, rather than an in-band middlebox. This makes the proof more challenging, as the verifier needs to be convinced that the claimed ciphertext really came from a session with the claimed server. To solve this, DECO relies on multiparty computation between the client and a third-party notary. However, these applications do not face tight latency constraints, as ZKMBs do, enabling much different performance tradeoffs.

Regular expressions in zero knowledge. Zombie introduces a portfolio of techniques for encoding regular expressions in probabilistic proofs. Previous to Zombie, the approach taken [3] was direct translation of a DFA, with exponential costs. We are aware of only two other works, both concurrent with Zombie, that improve on the exponential baseline; like Zombie, both target network security applications.

Exciting work by Luo et al. [68] transforms a regular expression to a Thompson NFA [104], like Zombie does (§4.3). Unlike Zombie, Luo et al. transform the NFA to a Boolean circuit and then use MPC-in-the-head [29, 54]. In addition to the setting where the client knows the policy, Luo et al. also consider the setting where the middlebox wants to keep the policy private but still apply it to the client's traffic. This part of their application thus has a significantly different performance profile than ours (an extra logarithmic term is introduced in the size of the regexp, and extra overheads are incurred to preserve the privacy of the policy itself.) Additionally, because most of our optimizations rely on constraints over large finite fields while theirs are tailored Boolean circuits, the respective techniques do not seem to be applicable to each other. While a detailed comparison has yet to be done, Zombie appears to have an order of magnitude lower communication cost (proof size) and computation (prover time) in the public policy case.

The other concurrent work, `zkreg` [89], compiles a large collection of regular expressions (mostly string matches) into an Aho-Corasick automaton [2], encodes this automaton as an arithmetic circuit, and then uses a custom Commit-and-Prove scheme [20] to prove membership and non-membership in zero knowledge on extremely large dictionaries of strings. For example, they consider proofs involving an automaton with 19 million states and over 300 million transitions. To handle an automaton this large, they represent it as a multiset of transitions and handle transition checking partially using set membership. This incurs a significantly higher computational overhead than our transition checking, but it scales far better for large automata (for which it is explicitly designed). Future work is to investigate ways of combining relevant techniques in `zkreg` with Zombie to efficiently support larger policies.

Middlebox architectures. Many proposed middlebox architectures aim to enforce policies on encrypted traffic. For helpful surveys, we refer the reader to Sherry [99] and Naylor et al. [78]. Work prior to ZKMB [49] largely falls into two broad categories:

Trusted hardware. ETTM [31] first proposed shifting policy enforcement logic from middleboxes to network users (end hosts) themselves. This requires trusted hardware to assert that a virtual machine run by the end host is faithfully checking that the plaintext is policy-compliant. Endbox [44] refined this vision using the then-emerging *trusted execution environment* (TEE) abstraction, specifically using Intel's SGX implementation. An obvious limitation is that all users must have a TEE to take advantage of this approach.

mbTLS [78] proposes relying on a TEE at the middlebox itself, acting as a middleperson (MITM) between a TLS session established with the client machine and one with the server (which can also be extended to multiple hops). This undermines the typical end-to-end nature of TLS, but if the TEE remains secure users can trust that their plaintext will only be used by the TEE for policy checks. Another approach is to shift policy enforcement from a local middlebox to a TEE run on a cloud server [51, 87, 105]. Other works, too, rely on trusted hardware [32, 45, 50, 59, 118].

We wish to avoid trusted hardware, given the cavalcade of exploits demonstrated against real-world TEE implementations [36, 47, 77, 80, 92, 102, 109–112], using information like power consumption or electromagnetic emanations to extract secrets from an enclave non-invasively. One might note that zero-knowledge prover implementations can also be subject to side-channel attacks [40, 106]. However, there is a subtle but essential difference: TEE-based middleboxes inherently require running code with access to secrets in an enclave *that is placed under the direct control of an adversary*, for example a policy-enforcing enclave hosted by a client machine (where a TEE break undermines integrity) or an enclave with access to decryption keys hosted by the middlebox (where a TEE break undermines privacy). Either setup makes side-channel attacks considerably easier to mount, versus the ZKMB paradigm, in which the relevant adversary (for example, the network administrator) is *remote from the party executing the relevant algorithm* (for example, the ZK prover).

TLS modifications. Several proposals to reconcile widespread TLS adoption with network policy enforcement envision modifying TLS to make it "middlebox-aware," with middleboxes gaining the ability to read and/or modify some (but not necessarily all) of the plaintext data sent in a TLS connection and users typically getting some visibility into the process [10, 12, 64, 65, 78, 79, 121]. An example is "multi-context TLS" or mcTLS [79], with different middleboxes on

the network path receiving context-specific keys based on the permissions the client and server are willing to grant. In the case of DNS filtering, a middlebox might require read-only access to the request body of a DNS query. While this approach enables finer-grained tradeoffs than disabling encryption completely, it is still a blunt instrument that sacrifices user privacy considerably; in the DNS example users fully give up privacy of their query history. It also requires server-side changes.

Blindbox [98] proposed modifying TLS to support policy enforcement by middleboxes. Specifically, Blindbox supplements the standard, semantically-secure symmetric encryption used in TLS with *searchable encryption*. This second ciphertext, along with techniques from circuit garbling and oblivious transfer, allows middleboxes to obliviously execute policy checks on ciphertext, specifically tailored to searching for keywords in text. A rich line of follow-up work extends this basic model [35, 56, 61–64, 66, 81, 82, 88, 122].

All of these works use some variant of *functional encryption*, which allows middleboxes to compute a limited function of the underlying plaintext, with different proposals tailored to different functionality. These works all face the challenging requirement of changing TLS servers, as well as relying on servers to check consistency of the TLS plaintext and that of the supplemental functional encryption (without this check, clients might send policy-violating traffic over TLS but append a functional encryption of benign traffic to satisfy the middlebox). A key goal in our work is not to require changes to, or participation of, existing TLS servers (§1).

8 Discussion

Zombie is 3× cheaper than its progenitor ZKMB [49], with an overhead of 406 ms for enforcing a blocklist on DNS requests over TLS (§6.1). Assuming client idle time, this drops to 256 ms, via precomputation. This number may be tolerable for DNS filtering; by comparison, traditionally satellite Internet connections have added at least 600 ms of latency [13] (though, to be fair, modern low-earth-orbit satellite Internet service offer significantly lower latency, as low as 25 ms [70]). Regardless, under optimistic approval, there is zero online overhead (§6.1). Zombie is thus plausibly practical for clients. Also, our regular expression techniques allow for new policies in the ZKMB paradigm, reducing the cost of enforcing complex policies to the point where the dominant cost is decryption (§6.3).

Although the communication overhead of 30 KB per proof (§6.1) is 120× larger than 255-byte DNS requests, the proof size is small when compared to the average website size of 2–3 MB [5]. Furthermore, these proofs are transmitted only from client to middlebox, which are typically on the same local network. Moreover, Zombie is best geared to settings where most packets do not need proofs (see below), so the overall increase in bandwidth is expected to be small. Memory requirements at prover and verifier are an issue but are mitigated by small packets (§6.3).

The real snag is the middlebox's resource requirements, independent of configuration. Despite batching, which can experimentally increase throughput by almost 5×, the middlebox still requires at least 38 ms per packet on a single thread (§6.2), which is too high for most applications.

Consequently, Zombie is not truly practical outside of a few specific use cases: policy-relevant traffic must be a small fraction of all traffic, with small packets, and ideally occurs over multi-packet flows, to amortize channel opening (§3). Two promising examples are enforcing policy over DNS requests and over search engine queries.

Apart from performance, a number of concerns remain for real-world deployment. Zombie only supports public, stateless, read-only policies. Although confidential policies with zero-knowledge middleboxes are possible in principle [49], they require extra round-trips, and composing them with batching and asynchrony is an open challenge. Supporting stateful and write-based policies in the ZKMB framework is also conceptually possible, but we leave this to future work.

To deploy and update Zombie, clients need to learn the setup, such as the proving algorithm and the subcomputations (§3), when they first connect to the network. This material can be supplied or stored in browsers or as a required download to use the network.

To avoid barring honest clients from the network, the middlebox needs to handle dropped packets and proofs gracefully. One option is for the middlebox to send an acknowledgment when it receives a proof. If the client does not receive a timely acknowledgment, it will know to resend the proof. Another concern is soundness under load: the middlebox cannot simply skip verification, say if it runs out of storage (§6.2). Instead it must drop proofs and packets, and expect clients to resend. In the asynchronous setting, the middlebox can go so far as to request batch proofs from specific clients when it has freed up space for them.

Finally, the middlebox needs a way to identify, and deal with, clients who violate policy. This concern is not specific to the ZKMB paradigm, and is left to network administrators.

Ultimately, despite the sometimes equivocal performance results, Zombie has taken a substantial step forward in demonstrating the possible practicality of the ZKMB paradigm.

Acknowledgments

We thank the anonymous reviewers for constructive comments that improved this work. This research was supported by DARPA under Agreement No. HR00112020022. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Government or DARPA.

The code for Zombie is available at
https://github.com/PepperSieve/Zombie

A Details of SpartanBatch

A.1 Spartan protocol details

This section provides a sketch of the Spartan algorithm, focusing on the parts relevant to our extension of it to support batching.

For an R1CS instance $\mathcal{C}$ with matrices (A,B,C) and public input $\mathbf{X}$, public output $\mathbf{Y}$, and witness $\mathbf{W}$, Spartan works by transforming the validity check for $(\mathbf{X},\mathbf{Y})$ into a polynomial that equals zero at every point if (and only if) $(\mathbf{X},\mathbf{Y})$ satisfies the constraints $\mathcal{C}$. This polynomial is large, and some of its coefficients are elements of $\mathbf{W}$, so the verifier cannot do this check itself; instead, the prover and verifier engage in a subprotocol that lets the verifier check whether the polynomial is zero efficiently, by evaluating it at a random point. The details of this process are unimportant for us, save for one: in the last step of the subprotocol, the verifier must evaluate special polynomial encodings (a *multilinear extension*) $\tilde{A},\tilde{B},\tilde{C}$ of each R1CS matrices A,B,C at a random point. These evaluations are the most expensive part of the protocol for the verifier; in fact, they are asymptotically as expensive as rerunning the entire computation. The random point that the polynomial encodings are to be evaluated on is of the form (r_x,r_y) where $r_x,r_y \in \mathbb{F}^{\log n}$ and each element of the r_x and r_y is a random value provided by the verifier in some step of the protocol.

In the non-interactive version of Spartan, these are chosen by hashing prefixes of the proof as the prover generates it (that is, via the Fiat-Shamir transform [37]). We observe that, since these expensive evaluations: $\tilde{A}(r_x,r_y)$, $\tilde{B}(r_x,r_y)$, and $\tilde{C}(r_x,r_y)$, depend only on the R1CS statement and not the input (in our setting, the ciphertext), they can be done just once for a batch of proofs as long as each of their respective subprotocols "coordinate", that is, use the same r_x,r_y values. We ensure this by having the prover hash the prefixes of each proof in the batch together, instead of separately. (Some hashing steps in Spartan generate randomness that is not part of r_x or r_y; we do not batch generate randomness for these steps.) We call the resulting protocol SpartanBatch.

Below, we show SpartanBatch retains the security guarantees of Spartan; in particular, we show that a malicious client has about the same (very low) probability of proving a false statement with SpartanBatch as it does with Spartan. Our analysis is based on analogous results for AND-composition in Σ-protocols and related results [95, 113].

A.2 SpartanBatch and its security proof

We can define (interactive) SpartanBatch from (interactive) Spartan below. Applying the standard Fiat-Shamir heuristic results in the non-interactive version described above. (Interactive) SpartanBatch is b parallel instances of (interactive) Spartan with the verifier following two different methods depending on the step involved:

- **Coordinated** steps are those where the verifier provides a random value that is an element of r_x or r_y (which are each in $\mathbb{F}^{\log n}$), and thus part of the evaluation point (r_x,r_y) for the polynomials $\widetilde{A},\widetilde{B},\widetilde{C}$ in the final step of Spartan's verification algorithm. In these steps, the SpartanBatch Verifier provides a *single* random value that is taken to be the response to all b parallel instances.

- All other steps are **uncoordinated**. Here, the verifier provides a b-tuple of independent responses, one for each parallel proof.

The analysis below is based on that of similar techniques applied in other proof systems [95, 113].

Theorem 1 *SpartanBatch is a succinct non-interactive argument of knowledge for the language $\mathcal{L}^b$, where b is the batch size.*

Proof:

We analyze the *interactive* version of SpartanBatch, noting that all the properties proven below are retained when using the standard Fiat-Shamir heuristic to obtain non-interactivity.

Completeness: The verifier for SpartanBatch can be seen as performing the checks of b separate Spartan verifiers. Completeness is thus immediate from the completeness of Spartan.

Soundness: Using an argument similar to standard AND-composition analysis in Σ-protocols, we show that the soundness error of SpartanBatch is *at most* the soundness error (ε) of Spartan. The proof proceeds by contradiction. Assume that there exists a false instance $x^* \notin \mathcal{L}$ and a SpartanBatch prover P_B that produces a convincing proof of a batch of statements $X^* = \{x^*, x_2, \ldots, x_b\}$ with probability $\geq 1-\varepsilon$ (we place the false instance in the first position without loss of generality). We use P_B to construct a Spartan prover P that convinces a Spartan verifier V of the same false statement x^* with the same probability.

In this reduction, P doubles as the SpartanBatch verifier when interacting with P_B: that is, $\langle P,V\rangle$ run an instance of Spartan on input x^* while $\langle P_B,P\rangle$ run an instance of SpartanBatch on input X^*. The reduction proceeds as follows:

- When P_B provides a tuple of values, P forwards the value corresponding to the false instance (here, the first one) to V.

- When V sends randomness r, P forwards the following to P_B based on the step:

  - In coordinated steps: P forwards r.
  - In uncoordinated steps: P sample randomness $r_2,\ldots,r_b$ and forwards $(r,r_2,\ldots,r_b)$.

Thus, if P_B passes the SpartanBatch verification checks, P must pass the Spartan verification checks. This is a contradiction as P_B was assumed to pass with probability $\geq 1-\varepsilon$.

Zero-knowledge: Like Spartan, SpartanBatch being a public-coin interactive protocol allows us to leverage existing compilers to satisfy zero-knowledge [116].

Knowledge soundness: We prove the stronger notion of witness-extended emulation. As this property is satisfied by Spartan, we have an emulator E that interacts with any Spartan prover P as an oracle and is allowed to rewind P to any step and resume with new verifier randomness. Using E, we construct $E_B^{P_B}$ that runs on input $X = \{x_1, \ldots, x_b\}$ interacting with a SpartanBatch prover P_B as follows:

For $i \in 0 \ldots b$:

- E_B runs emulator E on input x_i.
- When E sends randomness r to its oracle, E_B sends the following values to its oracle P_B based on the step:
  - In coordinated steps, E_B forwards value r.
  - In uncoordinated steps, E_B forwards a b-tuple with value r in position i and freshly sampled randomness in all other positions.
- When P_B responds with a tuple of values, E_B forwards the value at position i to E as a response to E's oracle query.
- When E rewinds its prover P to a step, E_B rewinds P_B (and thus rewinding all parallel instances in the batch) to that step, as well.

This way, E_B accurately simulates the required oracle for E and thus has it extract witness w_i for all inputs x_i in the batch. As E_B sequentially runs E on b inputs, E_B also runs in expected polynomial time when b is a constant.

References

[1] BNF converter. http://bnfc.digitalgrammars.com/.

[2] Efficient string matching: an aid to bibliographic search. *Communications of The ACM*, 18(6):333–340, 1975.

[3] zk-regex. https://github.com/zkemail/zk-regex, 2023.

[4] Shashank Agrawal, Chaya Ganesh, and Payman Mohassel. Non-Interactive Zero-Knowledge Proofs for Composite Statements. In *CRYPTO*, 2018.

[5] HTTP Archive. Web almanac HTTP archive's annual state of the web report. https://almanac.httparchive.org/en/2022/page-weight#request-bytes.

[6] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof Verification and the Hardness of Approximation Problems. *Journal of the ACM*, 45(3), May 1998.

[7] Sanjeev Arora and Shmuel Safra. Probabilistic Checking of Proofs: A New Characterization of NP. *Journal of the ACM*, 45(1), January 1998.

[8] László Babai and Lance Fortnow. Arithmetization: A new method in structural complexity theory. *Computational Complexity*, 1:41–66, March 1991.

[9] László Babai, Lance Fortnow, Leonid A Levin, and Mario Szegedy. Checking Computations in Polylogarithmic Time. In *ACM STOC*, 1991.

[10] Joonsang Baek, Jongkil Kim, and Willy Susilo. Inspecting TLS anytime anywhere: a new approach to TLS interception. In *Asia CCS*, 2020.

[11] Daniel J. Bernstein. Curve25519: new diffie-hellman speed records. https://cr.yp.to/ecdh/curve25519-20060209.pdf, 2006.

[12] Karthikeyan Bhargavan, Ioana Boureanu, Antoine Delignat-Lavaud, Pierre-Alain Fouque, and Cristina Onete. A formal treatment of accountable proxying over TLS. In *IEEE Symposium on Security and Privacy*, 2018.

[13] Anas A Bisu, Alan Purvis, Katharine Brigham, and Hongjian Sun. A framework for end-to-end latency measurements in a satellite network environment. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2018.

[14] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable Delay Functions. In *CRYPTO*, 2018.

[15] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *IEEE Symposium on Security and Privacy*, pages 947–964. IEEE, 2020.

[16] Benjamin Braun. Compiling computations to constraints for verified computation. UT Austin Honors thesis HR-12-10, December 2012.

[17] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *ACM SOSP*, 2013.

[18] Broadcom Near Real-Time Scan. https://techdocs.broadcom.com/us/en/symantec-security-software/endpoint-security-and-management/cloud-workload-protection-for-storage/1-0/Scan_Configuration_7/about-near-real-time-scan-v123769597-d4995e65807.html, 2023.

[19] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In *Financial Crypto*, 2020.

[20] Matteo Campanelli, Dario Fiore, and Anaïs Querol. LegoSNARK: Modular design and composition of succinct zero-knowledge proofs. In *ACM CCS*, page 2075–2092, 2019.

[21] Weikeng Chen, Alessandro Chiesa, Emma Dauterman, and Nicholas P. Ward. Reducing Participation Costs via Incremental Verification for Ledger Systems. Cryptology ePrint Archive, Paper 2020/1522, 2020.

[22] Cisco. Snort intrusion detection system. https://www.snort.org/.

[23] Cisco Umbrella. https://umbrella.cisco.com/, 2023.

[24] Jeremy Clark and Paul C Van Oorschot. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *IEEE Symposium on Security and Privacy*, 2013.

[25] Russ Cox. Regular expression matching can be simple and fast. https://swtch.com/ rsc/regexp/regexp1.html, 2007.

[26] Santiago Cuéllar, Bill Harris, James Parker, Stuart Pernsteiner, and Eran Tromer. Cheesecloth: Zero-Knowledge Proofs of Real-World Vulnerabilities. *arXiv preprint arXiv:2301.01321*, 2023.

[27] Gaby G Dagher, Benedikt Bünz, Joseph Bonneau, Jeremy Clark, and Dan Boneh. Provisions: Privacy-preserving Proofs of Solvency for Bitcoin Exchanges. In *ACM CCS*.

[28] George Danezis, Cedric Fournet, Markulf Kohlweiss, and Bryan Parno. Pinocchio Coin: building Zerocoin from a succinct pairing-based proof system. In *ACM workshop on Language Support for Privacy-Enhancing Technologies*, 2013.

[29] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, and Titouan Tanguy. Limbo: Efficient zero-knowledge MPCitH-based arguments. In *ACM CCS*, CCS '21, page 3022–3036, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Tim Dierks and Eric Rescorla. RFC 5246: The transport layer security (TLS) protocol version 1.2. RFC 5246, 2008.

[31] Colin Dixon, Hardeep Uppal, Vjekoslav Brajkovic, Dane Brandon, Thomas Anderson, and Arvind Krishnamurthy. ETTM: A scalable fault tolerant network manager. In *USENIX NSDI*, 2011.

[32] Huayi Duan, Xingliang Yuan, and Cong Wang. Lightbox: SGX-assisted secure network functions at near-native speed. *arXiv preprint arXiv:1706.06261*, 2017.

[33] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[34] Jacob Eberhardt and Stefan Tai. ZoKrates - Scalable Privacy-Preserving Off-Chain Computations. In *IEEE Conference on Internet of Things (iThings)*, 2018.

[35] Jingyuan Fan, Chaowen Guan, Kui Ren, Yong Cui, and Chunming Qiao. Spabox: Safeguarding privacy during deep packet inspection at a middlebox. *IEEE/ACM Transactions on Networking*, 25(6), 2017.

[36] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. Security vulnerabilities of SGX and countermeasures: A survey. *ACM Computing Surveys*, 54(6), 2021.

[37] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, 1986.

[38] Fortra Digital Guardian. https://www.digitalguardian.com/, 2023.

[39] Open Information Security Foundation. Suricata intrusion detection system. https://suricata.io/.

[40] Sanjam Garg, Abhishek Jain, and Amit Sahai. Leakage-Resilient Zero Knowledge. In *CRYPTO*, 2011.

[41] Oded Goldreich. Probabilistic proof systems – a primer. *Foundations and Trends in Theoretical Computer Science*, 3(1), 2008.

[42] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Delegating computation: interactive proofs for muggles. *Journal of the ACM*, 62(4), 2015.

[43] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1), 1989.

[44] David Goltzsche, Signe Rüsch, Manuel Nieke, Sébastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Cosa, Christof Fetzer, Pascal Felber, et al. Endbox: Scalable middlebox functions using client-side trusted execution. In *IEEE/IFIP DSN*, 2018.

[45] Deli Gong, Muoi Tran, Shweta Shinde, Hao Jin, Vyas Sekar, Prateek Saxena, and Min Suk Kang. Practical verifiable in-network filtering for DDoS defense. In *IEEE ICDCS*, 2019.

[46] Google. Google public DNS. https://developers.google.com/speed/public-dns.

[47] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[48] Jens Groth. On the size of pairing-based non-interactive arguments. In *IACR Eurocrypt*, 2016.

[49] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. Zero-Knowledge Middleboxes. In *USENIX Security*, 2022.

[50] Juhyeng Han, Seongmin Kim, Daeyang Cho, Byungkwon Choi, Jaehyeong Ha, and Dongsu Han. A Secure Middlebox Framework for Enabling Visibility Over Multiple Encryption Protocols. *IEEE/ACM Transactions on Networking*, 28(6), 2020.

[51] Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. SGX-Box: Enabling visibility on encrypted traffic using a secure middlebox module. In *Asia-Pacific Workshop on Networking*, 2017.

[52] Paul E. Hoffman and Patrick McManus. DNS Queries over HTTPS (DoH). RFC 8484, 2018.

[53] Zi Hu, Liang Zhu, John Heidemann, Allison Mankin, Duane Wessels, and Paul E. Hoffman. Specification for DNS over Transport Layer Security (TLS). RFC 7858, 2016.

[54] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *ACM STOC*, STOC '07, page 21–30, New York, NY, USA, 2007. Association for Computing Machinery.

[55] Dmitry Khovratovich, Mary Maller, and Pratyush Ranjan Tiwari. MinRoot: Candidate Sequential Function for Ethereum VDF. Cryptology ePrint Archive, Paper 2022/1626, 2022.

[56] Jongkil Kim, Seyit Camtepe, Joonsang Baek, Willy Susilo, Josef Pieprzyk, and Surya Nepal. P2DPI: Practical and Privacy-Preserving Deep Packet Inspection. *AsiaCCS*, 2021.

[57] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. In *IEEE Symposium on Security and Privacy*, 2016.

[58] Feodor Kulishov. DFA-based and SIMD NFA-based regular expression matching on cell BE for fast network traffic filtering. In *2nd Intl. Conference on Security of Information and Networks (SIN)*. ACM Press, 2009.

[59] Dmitrii Kuvaiskii, Somnath Chakrabarti, and Mona Vij. Snort intrusion detection system with Intel software guard extension (Intel SGX). *arXiv preprint arXiv:1802.00508*, 2018.

[60] SSL Labs. SSL Pulse. https://www.ssllabs.com/ssl-

pulse/.

[61] Shangqi Lai, Xingliang Yuan, Joseph K Liu, Xun Yi, Qi Li, Dongxi Liu, and Surya Nepal. OblivSketch: Oblivious Network Measurement as a Cloud Service. In *Network and Distributed System Security Symposium*, 2021.

[62] Shangqi Lai, Xingliang Yuan, Shifeng Sun, Joseph K. Liu, Ron Steinfeld, Amin Sakzad, and Dongxi Liu. Practical Encrypted Network Traffic Pattern Matching for Secure Middleboxes. *IEEE TDSC*, 2021.

[63] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely outsourcing middleboxes to the cloud. In *USENIX NSDI*, 2016.

[64] Hyunwoo Lee, Zach Smith, Junghwan Lim, Gyeongjae Choi, Selin Chun, Taejoong Chung, and Ted Taekyoung Kwon. maTLS: How to make TLS middlebox-aware? In *Network and Distributed System Security Symposium*, 2019.

[65] Jie Li, Rongmao Chen, Jinshu Su, Xinyi Huang, and Xiaofeng Wang. ME-TLS: Middlebox-enhanced TLS for Internet-of-Things devices. *IEEE Internet of Things Journal*, 7(2), November 2019.

[66] Cong Liu, Yong Cui, Kun Tan, Quan Fan, Kui Ren, and Jianping Wu. Building generic scalable middlebox services over encrypted protocols. In *IEEE INFOCOM*, 2018.

[67] Yanbing Liu, Li Guo, Ping Liu, and Jianlong Tan. Compressing regular expressions' dfa table by matrix decomposition. In Michael Domaratzki and Kai Salomaa, editors, *Implementation and Application of Automata*, pages 282–289, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[68] Ning Luo, Chenkai Weng, Jaspal Singh, Gefei Tan, Ruzica Piskac, and Mariana Raykova. Privacy-preserving regular expression matching using nondeterministic finite automata. Cryptology ePrint Archive, Paper 2023/643, 2023. https://eprint.iacr.org/2023/643.

[69] Chad Mayfield. my-pihole-blocklists/pi_blocklist_porn_all.list. https://github.com/chadmayfield/my-pihole-blocklists, 2021.

[70] François Michel, Martino Trevisan, Danilo Giordano, and Olivier Bonaventure. A First Look at Starlink Performance. In *Proceedings of the Internet Measurement Conference*, 2022.

[71] Microsoft. Data loss prevention. learn.microsoft.com/en-us/microsoft-365/compliance/dlp-learn-about-dlp.

[72] Microsoft. U.S. bank account number. https://learn.microsoft.com/en-us/microsoft-365/compliance/sit-defn-us-bank-account-number.

[73] Microsoft. U.S. drivers license number. https://learn.microsoft.com/en-us/microsoft-365/compliance/sit-defn-us-drivers-license-number.

[74] Microsoft. U.S. individual taxpayer identification number. https://learn.microsoft.com/en-us/microsoft-365/compliance/sit-defn-us-individual-taxpayer-identification-number.

[75] Microsoft. U.S. social security number. https://learn.microsoft.com/en-us/microsoft-365/compliance/sit-defn-us-social-security-number.

[76] Microsoft. U.S./U.K. passport number. https://learn.microsoft.com/en-us/microsoft-365/compliance/sit-defn-us-uk-passport-number.

[77] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *IEEE Symposium on Security and Privacy*, 2020.

[78] David Naylor, Richard Li, Christos Gkantsidis, Thomas Karagiannis, and Peter Steenkiste. And Then There Were More: Secure Communication for More Than Two Parties. In *ACM CoNEXT*, 2017.

[79] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-context TLS (McTLS): Enabling secure in-network functionality in TLS. In *ACM SIGCOMM*, 2015.

[80] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on Intel SGX. *arXiv preprint arXiv:2006.13598*, 2020.

[81] Jianting Ning, Xinyi Huang, Geong Sen Poh, Shengmin Xu, Jia-Chng Loh, Jian Weng, and Robert H Deng. Pine: Enabling privacy-preserving deep packet inspection on TLS with rule-hiding and fast connection establishment. In *ESORICS*, 2020.

[82] Jianting Ning, Geong Sen Poh, Jia-Ch'ng Loh, Jason Chia, and Ee-Chien Chang. PrivDPI: privacy-preserving encrypted traffic inspection with reusable obfuscated rules. In *ACM CCS*, 2019.

[83] OpenWrt. OpenWrt table of hardware. https://openwrt.org/toh/views/toh_extended_all.

[84] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. CirC: compiler infrastructure for proof systems, software verification, and more. In *IEEE Symposium on Security and Privacy*, 2022.

[85] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, 2013.

[86] PerfOps. Dns performance analytics and comparison. https://www.dnsperf.com/#!dns-resolvers.

[87] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Safebricks: Shielding network functions in the cloud. In *USENIX NSDI*, 2018.

[88] Geong Sen Poh, Dinil Mon Divakaran, Hoon Wei Lim, Jianting Ning, and Achintya Desai. A Survey of Privacy-Preserving Techniques for Encrypted Traffic Inspection over Network Middleboxes. *arXiv preprint arXiv:2101.04338*, 2021.

[89] Michael Raymond, Gillian Evers, Jan Ponti, Diya Krishnan, and Xiang Fu. Efficient zero knowledge for regular language. Cryptology ePrint Archive, Paper 2023/907, 2023. https://eprint.iacr.org/2023/907.

[90] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, 2018.

[91] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE Symposium on Security and Privacy*, 2014.

[92] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *DIMVA*, 2017.

[93] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *IACR CRYPTO*, 2020.

[94] Srinath Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. Resolving the conflict between generality and plausibility in verified computation. In *Eurosys*, 2013.

[95] Srinath Setty, Richard McPherson, Andrew Blumberg, and Michael Walfish. Making argument systems for outsourced computation practical (sometimes). 2012.

[96] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security*, 2012.

[97] Adi Shamir. IP = PSPACE. *J. ACM*, 39(4):869–877, oct 1992.

[98] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. BlindBox: Deep packet inspection over encrypted traffic. In *ACM SIGCOMM*, 2015.

[99] Justine M. Sherry. *Middleboxes as a Cloud Service*. PhD thesis, University of California, Berkeley, 2016.

[100] Adhiraj Singh, Madhavan Malolan, and Abhilash Inumella. Reclaim Protocol: Privacy preserving consensus to export reputation from webservers. https://www.reclaimprotocol.org/, 2022.

[101] Michael Sipser. *Introduction to the Theory of Computation*. Boston, MA, third edition, 2013.

[102] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W Fletcher. Microscope: Enabling microarchitectural replay attacks. In *ISCA*, 2019.

[103] Justin Thaler. Proofs, Arguments, and Zero-Knowledge. http://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html, 2020.

[104] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of The ACM*, 11(6):419–422, June 1968.

[105] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Shieldbox: Secure middleboxes using shielded execution. In *Symposium on SDN Research*, 2018.

[106] Florian Tramèr, Dan Boneh, and Kenny Paterson. Remote side-channel attacks on anonymous transactions. In *USENIX Security*, 2020.

[107] Nirvan Tyagi, Ben Fisch, Andrew Zitek, Joseph Bonneau, and Stefano Tessaro. VeRSA: Verifiable Registries with Efficient Client Audits from RSA Authenticated Dictionaries. In *ACM CCS*, 2022.

[108] Ioanna Tzialla, Abhiram Kothapalli, Bryan Parno, and Srinath Setty. Transparency Dictionaries with Succinct Proofs of Correct Operation. In *Network and Distributed System Security Symposium*, 2022.

[109] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[110] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *IEEE Symposium on Security and Privacy*, 2020.

[111] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice. https://sgaxeattack.com/, 2020.

[112] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. In *IEEE Symposium on Security and Privacy*, 2021.

[113] Victor Vu, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE Symposium on Security and Privacy*, 2013.

[114] W3Schools. Chrome statistics. https://www.w3schools.com/browsers/browsers_chrome.asp.

[115] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Network and Distributed System Security Symposium*, 2015.

[116] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *IEEE Symposium on Security and Privacy*, 2018.

[117] Michael Walfish and Andrew J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near practicality. *Communications of the ACM*, 58(2), February 2015.

[118] Juan Wang, Shirong Hao, Yi Li, Zhi Hong, Fei Yan, Bo Zhao, Jing Ma, and Huanguo Zhang. TVIDS: Trusted virtual IDS with SGX. *China Communications*, 16(10), 2019.

[119] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *USENIX NSDI*, 2019.

[120] Yu Wang, Yang Xiang, Wanlei Zhou, and Shunzheng Yu. Generating regular expression signatures for network traffic classification in trusted network management. *Journal of Network and Computer Applications*, 35(3):992–1000, 2012. Special Issue on Trusted Computing and Communications.

[121] Florian Wilkens, Steffen Haas, Johanna Amann, and Mathias Fischer. Passive, transparent, and selective TLS decryption for network security monitoring. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, 2022.

[122] Xingliang Yuan, Huayi Duan, and Cong Wang. Assuring string pattern matching in outsourced middleboxes. *IEEE/ACM Transactions on Networking*, 26(3), 2018.

[123] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. DECO: Liberating web data using decentralized oracles for TLS. In *ACM CCS*, 2020.

[124] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *IEEE Symposium on Security and Privacy*, 2017.

Solving Max-Min Fair Resource Allocations Quickly on Large Graphs

Pooria Namyar[†‡], Behnaz Arzani[†], Srikanth Kandula[†], Santiago Segarra[†⋆],
Daniel Crankshaw[†*], Umesh Krishnaswamy[†], Ramesh Govindan[‡], Himanshu Raj[†]
[†]*Microsoft,* [‡]*University of Southern California,* [⋆]*Rice University*

Abstract– We consider the max-min fair resource allocation problem. The best-known solutions use either a sequence of optimizations or waterfilling, which only applies to a narrow set of cases. These solutions have become a practical bottleneck in WAN traffic engineering and cluster scheduling, especially at larger problem sizes. We improve both approaches: (1) we show how to convert the optimization sequence into a single fast optimization, and (2) we generalize waterfilling to the multi-path case. We empirically show our new algorithms Pareto-dominate prior techniques: they produce faster, fairer, and more efficient allocations. Some of our allocators also have theoretical guarantees: they trade off a bounded amount of unfairness for faster allocation. We have deployed our allocators in Azure's WAN traffic engineering pipeline, where we preserve solution quality and achieve a roughly 3× speedup.

1 Introduction

Multi-resource fair allocators have become essential for cloud operators as multi-tenancy, availability, and efficiency grow in importance. These allocators divide the resources fairly among different requests (applications, users, or network flows). Operators use them to meet customer expectations, especially during congestion and for network neutrality.

Recent works present fair allocators in settings such as WAN traffic engineering [17, 30, 34, 38] and GPU scheduling [14, 42, 56]. We show these allocators achieve fairness at the cost of *speed* (crucial for maintaining high utilization as loads change [4]) and *efficiency*[1] (essential for profit).

We aim to achieve a better balance between fairness, efficiency, and speed, and our novel algorithms offer operators greater flexibility to control the trade-off between them. We focus on max-min fairness — where we cannot allocate more to one request without reducing the allocation of another with *an equal or smaller value* — because it is simple, commonly used in practice [30, 34, 39, 56], and can promote efficiency.[2]

The definition of max-min fairness naturally leads to iterative solutions that prioritize smaller requests over larger ones and assign rates in order from smallest to largest. In multi-resource settings, these solutions solve either mixed-integer or linear optimizations [52, 59] at *each* step. Their scalability depends on the size of each individual optimization and the number of iterations – typically a function of the number of resources and requests.

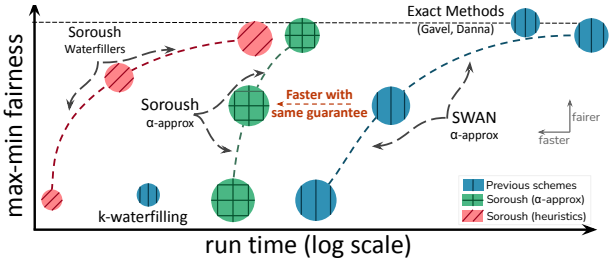


FIGURE 1: Comparing the new allocators with state-of-the-art. Soroush offers parameterizable max-min fair allocators. The axes are fairness and speed; the marker size corresponds to efficiency (larger is more efficient). Our new allocators empirically Pareto-dominate other schemes, and some of them have theoretical guarantees on fairness (§4).

Operators need to invoke resource allocators when workloads change or failures occur. However, today's exact [17,56] or approximate solutions (*e.g.,* that trade-off fairness for speed [30]) are too slow in reacting to these events at the production scale. They take *tens of minutes to hours* (§4) on WANs with 100s of routers that serve millions of flows or clusters with 1000s of jobs.

We ask: are *iterative* optimizations necessary for max-min fair resource allocation? Max-min fair algorithms must maximize smaller allocations before assigning more capacity to larger ones. Current solutions iterate because they do not know the sorted order of these rate allocations apriori. One of our ideas is to (1) use sorting networks [7] to discover the sorted order of max-min fair allocations *dynamically within the optimization* and (2) use a linear weighted objective that explicitly incentivizes the optimization to allocate more rates to requests with smaller indices in the sorted order. The result is a single-shot optimization for max-min fair allocation.

The above single-shot optimization is not always practical as modeling sorting networks within an optimization can significantly increase its size, and the linear weighted objective can cause double-precision issues when there are many requests. To develop a practical solution, we combine this idea with an approximate max-min fair allocator from SWAN [30]. This combination results in a new allocator, GeometricBinner (or GB), which is fundamentally faster than SWAN, does not need sorting networks, has no double precision issues, and provides the same fairness guarantees as SWAN.

Waterfilling-based algorithms [8] can be faster than black-

*The author contributed to this work while at Microsoft.
[1]In this paper, we use efficiency and utilization interchangeably.
[2]We defer extending to other notions of fairness to future work.

Allocator	*Properties*	*Parameters*
Geometric Binner	(**T**) α-approx fairness guarantee (**E**) Faster than other α-approx methods	α ϵ
Adaptive Waterfiller	(**T**) Solution in a small set containing optimal (**E**) Fastest	#iterations
Equi-depth Binner	(**T**) Better than Adaptive Waterfiller (**E**) Fairest and fast	#iterations #bins, ϵ

TABLE 1: The Soroush allocators, their properties (**T**heoretical and **E**mpirical), and their parameters. These allocators provide different trade-offs among fairness, efficiency, and speed.

box optimizations, but they are specialized for cases where each request seeks rates on a single path. In a broad class of problems [14, 17, 30, 34, 38, 42, 56], requests ask for allocation from multiple paths. In these cases, the global fair share is not locally fair along each path, and waterfilling does not apply. Our solution, AdaptiveWaterfiller (or AW), extends waterfilling to multi-path settings. AW is faster than GB but does not have a worst-case fairness guarantee. We prove AW can converge to a small set of allocations that contain the optimal.

Our third algorithm, which is empirically the fairest, combines the above approaches. We apply GB with one change: use the allocations from AW to spread requests more uniformly among bins (instead of the fixed geometrically increasing bin sizes in GB). This allocator, EquidepthBinner (or EB), is slower than both AW and GB (executes each once) but intuitively improves fairness for the same reason that equi-depth binning reduces histogram approximation error [32].

Soroush[3] is the collection of these algorithms, each providing a different trade-off among fairness, efficiency, and speed. Operators can use our simple decision process to choose the allocator (and its hyper-parameters) that achieves their desired trade-off. Table 1 lists our allocators, their theoretical and empirical properties, and their parameters.

To show Soroush is general, we introduce a graph model for multi-resource, max-min fair resource allocation problems where edges model resources and paths capture groups of resources the allocator must assign together. Requests (demands) can then ask for resources on any choice of multiple paths. This compact and general model subsumes problems from at least two domains: traffic engineering (TE) and cluster scheduling (CS). Soroush can solve any future max-min fair allocation problem if the user can specify it in this model.

Our extensive evaluation in both TE and CS (which we summarize in Fig. 1) show the new allocators Pareto-dominate the state-of-the-art in fairness, speed, and efficiency.

We deployed GeometricBinner in the production TE pipeline at Azure where it provides a $2.4\times$ average speedup (up to $5.4\times$ in some cases) without any impact on fairness and efficiency compared to the previous allocator.

2 Motivation and Overall Approach

Faster workload dynamics [4] and higher availability requirements [45] have made fast resource allocation a necessity.

[3] Our code is available at https://github.com/microsoft/Soroush

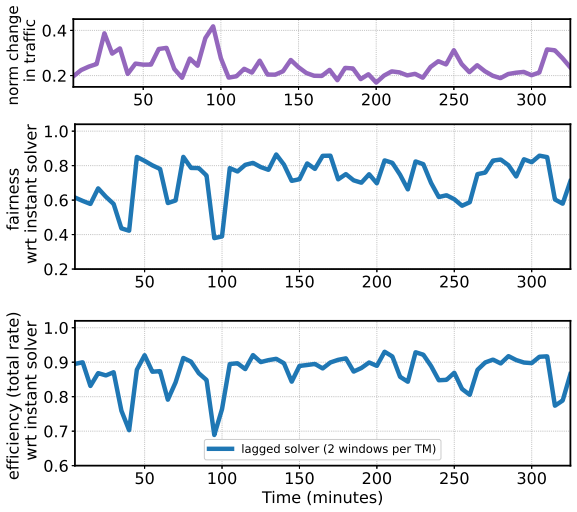


FIGURE 2: Slow max-min fair resource allocators cause under-utilization and unfairness. We compare two instances of the SWAN solver – one computes the allocations instantly while the other needs two windows – on a 5-hour trace from Azure's WAN. The results indicate a large gap between the two solutions in fairness and efficiency.

Operators of multi-tenant clouds further require solutions that ensure fairness and maintain high efficiency [30, 34]. Prior work ([4, 45, 55, 66] in TE or [56] in CS) fails to meet one or more of these requirements.

Efficient and fair solvers [17, 30, 56] cannot adapt quickly to conditions that frequently change. Some production environments [4] use the most recent previous allocation when the solver cannot allocate resources within a fixed time window. This is sub-optimal: nodes that increase their demands in the new window do not get enough resources, and others who request less may receive more than they need.

We quantify the impact of this strategy in the TE setting using a 5-hour trace from Azure's production WAN (Fig. 2), which uses a 5-minute window. We observe a solver that needs two windows (10 minutes) to allocate resources reduces fairness by 20% – 60% and efficiency by 10% – 30% relative to a solver that completes within one window.

How often do solvers miss their deadline? We use traces from [4] to show the distribution of the number of windows an exact solver (Danna *et al.* [17]) and Microsoft's approximate solver (SWAN [30]) need to compute max-min fair allocations. For nearly half of the traffic trace, these solvers exceed the 5-minute window and often need 2 to 3 windows to finish (Fig. 3, left). This is because these approaches invoke expensive optimizations multiple times (Fig. 3, right).

Soroush invokes at most one optimization and always completes within a single window. Whether a one-shot optimization is faster than an iterative approach that solves multiple optimizations depends on two factors: (a) the number of optimizations in the iterative approach, and (b) the size of the optimization in the one-shot approach compared to those in

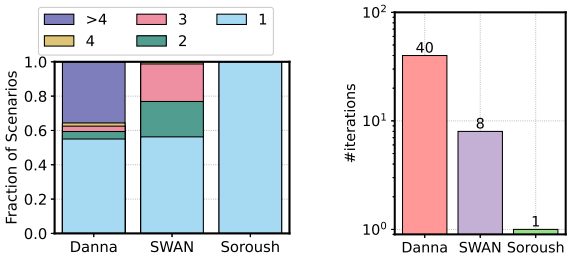


FIGURE 3: State-of-the-art methods can not keep up with frequently changing demands. We capture the number of windows (left) and the number of iterations (right) each approach needs. To keep up with demands, they must finish within a single 5-minute window. However, SWAN [30] and Danna *et al.* [17] often need more than one window and miss their deadline. The results are on a topology with ∼200 nodes and ∼500 edges. Left captures 160 different scenarios. Right is a highly loaded scenario [4] – the results hold across all the algorithms in Soroush.

the iterative solution[4]. Our one-shot optimizations are faster than previous solutions [17, 30, 56] because we only add a small number of variables to convert the problem into one that can be solved in one shot, and we avoid the overhead in solving multiple optimizations. See §F for a detailed analysis.

While examples here are from TE, CS resource allocators are similar [56]. We omit the details for brevity.

2.1 Our model

We model the max-min fair resource allocation problem as a capacitated graph[5]. Each *edge* represents a different resource, and *edge capacities* show the amount of available resources. *Paths* in the graph encode a collection of resources we must allocate together (*e.g.,* GPU and memory), and *demands* can request resources on a subset of these paths. Our model also supports other affine constraints over graph variables (*e.g.,* text in maroon below). Soroush solves any max-min fair resource allocation problem we can specify in this model.

Our model takes as input:

- A set of resources $\mathcal{E}$, each with capacities $c_e, e \in \mathcal{E}$.
- A set of paths $\mathcal{P}$ where each path is a group of dependent resources that we must allocate together.
- A set of demands $\mathcal{D}$ where each demand $k \in \mathcal{D}$:
  - Requests some rate d_k.
  - Has weight w_k (for weighted max-min fairness).
  - Can be routed over a set of multiple paths $P_k \in \mathcal{P}$.
  - Consumes r_k^e of the capacity on edge e for each unit rate we assign.
  - Has utility q_k^p on path p for each unit rate.

A max-min fair allocator assigns rates to demands such that the weighted ratios $\{\frac{f_k}{w_k}\}$ are max-min fair: to increase the allocation of any demand, we have to reduce the allocation of another demand with a smaller ratio.

[4]LP solver latency is polynomial in the problem size [13, 15].

[5]We present the formulation of this model in §A

Term	Interpretation
$\mathcal{E}, \mathcal{D}, \mathcal{P}$	sets of resources, demands and paths
c_e	capacity of resource $e \in \mathcal{E}$
$\mathbf{f}, f_k^p$	rate allocation vector and rate for demand k on path p
d_k, w_k	requested rate and weight for demand $k \in \mathcal{D}$
r_k^e, q_k^p	scaling resource consumption and rate utility for demand k

TABLE 2: Notation for our max-min fair resource allocation model. (more details in Table A.1)

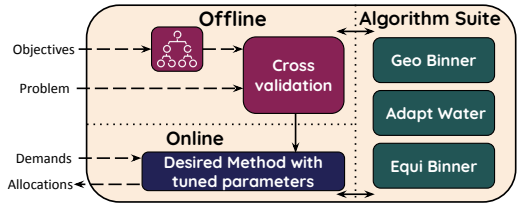


FIGURE 4: Choosing allocators (and their parameters).

We can use this model to specify max-min fair allocation problems in traffic engineering (TE) [30, 34] and cluster scheduling (CS) [22, 25, 56].

TE. The actual links in the network are the resources, and demands are services that require a specific rate between nodes in the network. The TE scheme picks the paths for each demand. Weights describe how the operator wants to divide the rates (*e.g.,* split rate between search and ads services).

CS. Each path corresponds to a server and contains multiple edges. Each edge models a different type of resource on each server (*e.g.,* CPU, memory, or GPU). Demands are jobs that require a number of workers. We model heterogeneity (workers may progress at different rates on different servers) with the utility term q_k^p and scale how much of each resource the worker uses with r_k^e.[6] We also support extensions, such as jobs with varying resource requirements [22, 25].

We are unaware of any model as general as ours for max-min fair allocation. However, solving this general model is hard: when we must allocate resources along multiple paths (*groups* of *resources*), local fairness does not imply global fairness, so single-path solutions [36] are ineffective. In Soroush, we focus on this model and leave the extension to problems that we cannot model as graphs [21] for future work.

2.2 Soroush Overview

Soroush offers a suite of *allocators* that produce approximate solutions for graph-based max-min fair allocation problems. An allocator is either an algorithm or optimization (or a combination of both) that assigns max-min fair rates that meet the demand and capacity constraints.

Table 1 lists our allocators, their key properties, and salient parameters that let the user trade-off between fairness, speed,

[6]We can also use these terms to model similar aspects in TE.

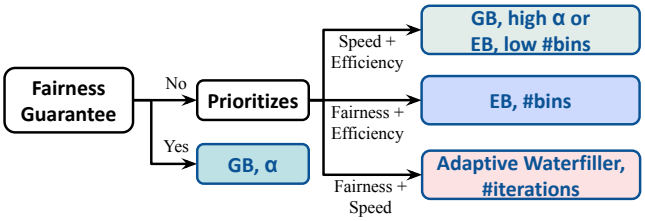


FIGURE 5: An example of how to pick the right allocator.

and efficiency (see §3 for more details). We have many choices here and naïvely running multiple allocators in parallel can waste computational resources. We suggest a simple decision process along with a hyperparameter search to help practitioners choose a suitable allocator (Fig. 4 and 5). A sensitivity analysis in §4 indicates that this decision process is robust. We do not claim any credit for it but use it as evidence that we have effective mechanisms to make the choice.

The GeometricBinner (§3.1) is the only allocator in Soroush with worst-case fairness guarantees: for a given $\alpha > 1$, it provably assigns rates to each demand within $[\alpha^{-1}, \alpha]$ times its optimal max-min fair rate. EquidepthBinner (§3.3) is the best choice for users who prefer fairness and efficiency but do not need formal worst-case guarantees. AdaptiveWaterfiller (§3.2) is suitable for users who prefer speed over efficiency.

3 Max-Min Fair Resource Allocators

We present two novel kinds of multi-resource max-min fair resource allocation algorithms.

3.1 One-shot Optimizations

Overview. We can think of max-min fair resource allocation as an optimization with a prioritized list of objectives: first, we maximize the smallest allocation, then the second smallest, and so on. This intuition naturally leads to a sequence of linear optimization problems (LPs) [17, 30].

Prior exact methods are slow since they solve nearly as many LPs as the number of unique resources [17] (number of edges in a network or machines in a cluster).

We will show how to linearize a prioritized list of objectives such that we can solve one optimization instead of a (long) sequence. The optimization we arrive at is analytically interesting but can encounter double-precision errors, requires a sorting network to sort allocations, and is consequently slower in practice — we instead linearize an approximate version.

SWAN [30] uses an approximate solution that needs to solve fewer LPs. It gradually and geometrically increases the maximum possible rate for each demand and guarantees the final allocations are within $\alpha\times$ their optimal fair rates. Users pick an α based on their requirements for fairness and speed (e.g., $\alpha = 2$ in SWAN). A larger α requires fewer LPs but results in less fair allocations.[7] Microsoft has been using SWAN in production for many years [39].

[7] The number of LPs is $\log_\alpha Z$ where Z is the ratio between the largest and the smallest request.

Term	Meaning
$\mathbf{t}, t_i$	sorted rate vector and the ith smallest rate
$N_\beta, \mathcal{D}_b$	number of bins and set of demands in bin b
ℓ_b, s_b	boundary and slackness of bin b
$\mathbf{f}_b, f_{kb}$	bin allocation vector and rate of demand k in bin b

TABLE 3: Additional notation for Soroush.

We develop an approximate one-shot optimization by linearizing SWAN's approximate geometric method. Our idea is to define "bins" that capture the geometric rate increase at each iteration and introduce new variables to model each flow's allocation from each bin instead of the cumulative total. This combination of techniques is novel and has the same worst-case fairness guarantees as SWAN. By linearizing at the granularity of bins, we no longer encounter double precision issues, do not need a sorting network, and achieve an empirically faster solution.

We flesh out the details next (Tables 2 and 3 show our notations). We discuss why this one-shot optimization is fundamentally smaller and faster than SWAN's sequence of optimizations. We present results from our production deployment in §4.2.

Max-min fair allocation as a sequence of LPs. If we have n demands, we can use n LPs to compute max-min fair allocations — the ith LP in the sequence maximizes the ith smallest rate. Let t_i be the ith smallest rate, then:

$$\begin{aligned} \mathsf{MaxMin}_i(\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq \quad & \arg\max \ t_i \\ \text{s.t.} \quad & (t_1,\ldots,t_{i-1}) \in \mathsf{MaxMin}_{i-1}(\mathcal{E},\mathcal{D},\mathcal{P}), \\ & f_k \geq t_i, \qquad \forall k \ \text{ whose rate is not yet frozen} \\ & \mathbf{f} \in \mathsf{FeasibleAlloc}(\mathcal{E},\mathcal{D},\mathcal{P}). \end{aligned} \tag{1}$$

Note that the algorithm freezes demands that can not receive more than t_i in each iteration. These demands will not receive any allocations in later iterations.

Our one-shot optimal max-min fair solution. We change Eqn. 1 (changes are in color) to a single optimization by (1) using a sorting network [7, 35, 45] to sort the rates as part of the optimization (Fig. A.1), and (2) using a linear weighted objective where the weight of a demand decreases based on its rank in the sorted order — these weights incentivize the optimization to increase the smaller rates.

Eqn. 1 does not need sorting because each LP maximizes the next smallest rate. The one-shot optimization, however, must explicitly sort the allocations in order to weight them appropriately in the objective. Let $\epsilon < 1$, then:

$$\begin{aligned} \mathsf{OneShotOpt}(\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq \quad & \arg\max_{\mathbf{f}} \sum_{i=1}^{n} \epsilon^{i-1} t_i \\ \text{s.t.} \quad & (t_1,\ldots,t_n) = \mathsf{sorted\ rates}(\mathbf{f}), \\ & \mathbf{f} \in \mathsf{FeasibleAlloc}(\mathcal{E},\mathcal{D},\mathcal{P}). \end{aligned} \tag{2}$$

We prove OneShotOpt leads to max-min fair rates:

Theorem 1. *There exist values of ϵ for which the optimization in Eqn. 2 yields the same max-min fair rate allocations as the sequence of optimizations shown in Eqn. 1.*

Proof Sketch. Let $\mathbf{t}^\dagger$ be the rate vector solution from Eqn. 2. Notice that the optimal max-min fair rate vector, say $\mathbf{t}^*$, is a feasible solution to Eqn. 2. Thus, $\sum_i \epsilon^{i-1} t_i^\dagger \geq \sum_i \epsilon^{i-1} t_i^*$ (otherwise $\mathbf{t}^\dagger$ is not the optimal solution to Eqn. 2). We can re-arrange and get $t_1^* - t_1^\dagger \leq \epsilon \left(\sum_{i>1} \epsilon^{i-1} t_i^\dagger - \sum_{i>1} \epsilon^{i-1} t_i^* \right)$.

By definition of max-min fairness, we have $t_1^* \geq t_1^\dagger$ since t_1^* is the smallest allocation in the optimal max-min fair solution. If we can find a feasible assignment where the smallest rate $t_1^\dagger$ is higher, then t^* cannot be max-min fair — we can increase the smallest rate without hurting any other demand with an smaller allocation (because no such demand exists). These statements together imply the smallest rates must match as $\epsilon \to 0$. The rest follows by induction. □

OneShotOpt is not practical. We need a small ϵ to find an optimal solution (see proof), but we will encounter double precision errors if the smallest weight (ϵ^{n-1}) is too small. In this case, the formulation may have to sacrifice optimality and use a large ϵ to solve the one-shot optimization (especially when there are many demands). Even with a large ϵ, we find that solving LPs with a full sorting network is slow [35, 45] since sorting networks add $O(nlog^2(n))$ additional constraints.

Our one-shot GeometricBinner (or GB) linearizes the following approximate max-min fair technique. Compared to Eqn. 1, Eqn. 3 shows a shorter sequence of LPs inspired by SWAN [30] (changes in color) but also differs from SWAN [8] in one crucial way (it introduces new variables to track the *increase* in the allocation of each demand in each iteration):

$$\mathsf{ApproxMaxMin}_b(\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq \arg\max \sum_{k\in\mathcal{D}} f_k \tag{3}$$

$$\begin{aligned}
\text{s.t.} \quad & f_k = \sum_{\text{bins } j\leq b} f_{kj}, && \forall k \in \mathcal{D} \\
& f_{k1} \leq U, && \forall k \in \mathcal{D} \\
& f_{kb} \leq U(\alpha^{b-1} - \alpha^{b-2}), && \forall b > 1, \forall k \in \mathcal{D} \\
& f_{kb} = 0 \quad \text{if} \sum_{j<b} f_{kj} < U\alpha^{b-2}, && \forall b > 1, \forall k \in \mathcal{D} \\
& (\mathbf{f}_1, \ldots, \mathbf{f}_{b-1}) \in \mathsf{ApproxMaxMin}_{b-1}, \\
& \mathbf{f} \in \mathsf{FeasibleAlloc}(\mathcal{E},\mathcal{D},\mathcal{P}).
\end{aligned}$$

The changes cause the bth LP, where index b begins at 1, to allocate only *up to* $U(\alpha^{b-1} - \alpha^{b-2})$ for $b > 1$ and up to U for $b = 1$. The algorithm also freezes any demand that does not receive the full rate from the previous iteration. α and U are input parameters that control the fairness guarantee and the minimum rate, respectively. Observe that each LP in the sequence allocates rates unfairly, but the unfairness is bounded as each LP allocates rates only within a small range.

[8] Eqn. 9 in §C shows SWAN's formulation for comparison.

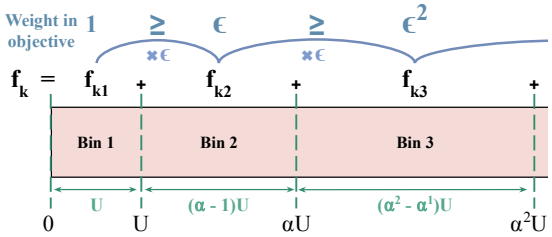


FIGURE 6: Geometric Binning: Approximate one-shot max-min fair allocations. We can model the problem in one shot because we add a new variable to track the allocation to each demand from each bin. With this idea, we can then change the objective to incentivize the optimization to make sure its allocation saturates smaller bins before allocating from subsequent bins.

Fig. 6 shows the key idea behind our one-shot geometric binner. If we consider each allocation as the sum of contributions from different, geometrically-sized bins, we can use ϵ-weighting per bin to incentivize the optimization to allocate more from the smaller bins. The resulting formulation is:

$$\mathsf{GeoBinning}(\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq \arg\max_{\mathbf{f}} \sum_{k\in\mathcal{D}} \sum_{\text{bins } b} \epsilon^{b-1} f_{kb} \tag{4}$$

$$\begin{aligned}
\text{s.t.} \quad & f_k = \sum_{\text{bins } b} f_{kb}, && \forall k \in \mathcal{D} \\
& f_{k1} \leq U, && \forall k \in \mathcal{D} \\
& f_{kb} \leq U(\alpha^{b-1} - \alpha^{b-2}), && \forall b > 1, \forall k \in \mathcal{D} \\
& \mathbf{f} \in \mathsf{FeasibleAlloc}(\mathcal{E},\mathcal{D},\mathcal{P}).
\end{aligned}$$

The geometric binner (Eqn. 4):

- Applies to various bin choices beyond the geometric ones we used here. We use a similar intuition from equi-depth binning in databases [32] to show in §3.3 that we can choose bin boundaries to improve fairness.
- Offers the same fairness guarantee as SWAN [30] when using the same (geometric) bins. GeoBinning allocates rates within an α ratio of the optimal max-min fair rates for any demand.

  Theorem 2. *Eqn. 4 assigns resources to a demand k in bin b only if it has assigned demand k the full rate from all of the larger-weighted bins.*

  Proof Sketch. Assume otherwise: Eqn. 4 has assigned a non-zero rate to some demand k in some bin b without assigning the full rate from some other bin $j < b$. Then, we can move some δ rate from bin b to j and not violate any constraints yet improve the objective value.[9] □
  We can combine Theorem 2 with the proof technique of Theorem 1 to prove Eqn. 4 will allocate the same rates as Eqn. 3, so the approximation ratio proof from [30] applies directly.
- Lets users adjust α to balance the trade-off between fairness approximation guarantee and the solver time.

[9] Smaller indexed bins have larger weights because $\epsilon < 1$.

Algorithm 1: Waterfilling algorithm to compute single-path weighted max-min fair rates.

Input: $\boldsymbol{\Gamma}$ where $\boldsymbol{\Gamma}[e,k]$ is the weight of single-path demand k on link e.
Input: $\mathbf{c}$: link capacity vector.
Output: $\mathbf{f}$: max-min rate allocation vector.

$\mathcal{D}_a \leftarrow [0,\ldots,K-1]$ — initial list of active demands
$\mathbf{f} \leftarrow \mathbf{0}$ — initial rate vector
while $|\mathcal{D}_a| > 0$ **do**
  $\mathbf{n} \leftarrow \boldsymbol{\Gamma}\mathbf{1}$ — total weight per link
  $\boldsymbol{\zeta} \leftarrow \frac{\mathbf{c}}{\mathbf{n}}$ — vector division, fairshare per link
  $e \leftarrow \arg\min\ \boldsymbol{\zeta}$ — link with minimum fair share
  $\mathcal{D}_e \leftarrow \{k : \boldsymbol{\Gamma}[e,k] > 0\}$ — active demands on link e
  foreach $k \in \mathcal{D}_e$ **do**
    $\mathbf{f}[\mathcal{D}_a[k]] \leftarrow \boldsymbol{\zeta}[e]\boldsymbol{\Gamma}[e,k]$ — fix to weighted fair share
    foreach $l : \boldsymbol{\Gamma}[l,k] > 0$ **do**
      $\mathbf{c}[l] \leftarrow \mathbf{c}[l] - \mathbf{f}[\mathcal{D}_a[k]]$ — deduct allocations
    end
  end
  $\mathbf{c} \leftarrow \mathbf{c}[\setminus\{e\}]$ — remove link e
  $\boldsymbol{\Gamma} \leftarrow \boldsymbol{\Gamma}[\setminus\{e\};\setminus\mathcal{D}_e]$ — remove link e and its demands
  $\mathcal{D}_a \leftarrow \mathcal{D}_a \setminus \mathcal{D}_e$ — update set of active demands
end
return $\mathbf{f}$

- Is less likely to run into precision issues compared to Eqn. 2 since there are fewer bins than demands.
- Requires no sorting constraints unlike Eqn. 2.
- Is only slightly larger in size compared to *each of the optimizations* in Eqn. 3 (see §F for more details). The key difference is that we can now run one LP instead of many. The one-shot optimization is empirically faster, likely due to redundant computation between the LPs in the sequence of optimizations in Eqn. 3.

3.2 Multi-path Waterfilling

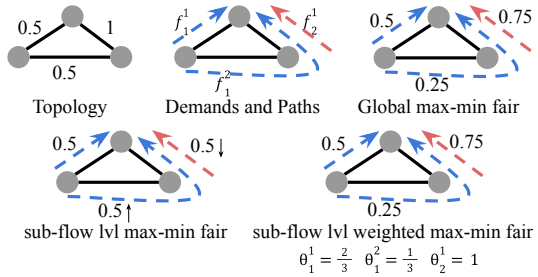
We also generalize the classical waterfilling algorithm for max-min fair allocation over multiple resources. We present parallelizable combinatorial algorithms (and not optimizations) with better empirical performance compared to §3.1 but weaker fairness guarantees.

Waterfilling is a well-known method that applies to scenarios where all the demands are *unconstrained* and require resources on *exactly one path* [36]. Under these conditions, we can achieve max-min fairness by visiting resources in the ascending order of their fair share and splitting their capacities *fairly* among the demands [67] (see Alg. 1[10])

We extend this approach to constrained demands by adding a virtual edge with a capacity equal to the requested rate for each demand. This augmented topology ensures that demands receive at most what they asked for. For small requests, the virtual edge becomes the bottleneck and limits the allocation.

It is harder to generalize waterfilling to multi-path settings because the local max-min fair allocation at individual resources is not globally fair. Fig. 7 shows a simple example where the blue demand — which has access to more paths — must receive a locally *unfair* share on the common link in order to produce a globally max-min fair solution. We next modify waterfilling to produce approximate, globally max-min fair rates in the general multi-resource setting.

[10]Notice weighted max-min fair allocation is roughly the same with one minor change: we relatively weigh the rate we allocate to each flow.



(a) Global and Local (per-link) max-min fairness are different.

	# iteration $t \longrightarrow$							
θ_1^1	$\frac{1}{2}$	$\frac{3}{5}$	$\frac{7}{11}$	$\frac{15}{23}$	$\frac{31}{47}$	$\frac{63}{95}$	...	$\frac{2}{3}$
θ_1^2	$\frac{1}{2}$	$\frac{2}{5}$	$\frac{4}{11}$	$\frac{8}{23}$	$\frac{16}{47}$	$\frac{32}{95}$	...	$\frac{1}{3}$
θ_2^1	1	1	1	1	1	1	...	1
f_1^1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	...	$\frac{1}{2}$
f_1^2	$\frac{1}{3}$	$\frac{2}{7}$	$\frac{4}{15}$	$\frac{8}{31}$	$\frac{16}{63}$	$\frac{32}{127}$	...	$\frac{1}{4}$
f_2^1	$\frac{2}{3}$	$\frac{5}{7}$	$\frac{11}{15}$	$\frac{23}{31}$	$\frac{47}{63}$	$\frac{95}{127}$	...	$\frac{3}{4}$

(b) AW's weight multipliers and allocations converge to global max-min fair allocation.

FIGURE 7: An example to illustrate the difficulty in extending waterfilling to multi-path settings and how our AdaptiveWaterfiller effectively tackles the issue. Waterfilling (single-path) computes local fair shares and is ineffective in multi-path settings as it ignore the dependencies between different paths of a single demand.

ApproxWaterfiller (or aW): For each demand, aW creates several "subdemands", each going through one of the demand's paths. Subdemands of each demand pass through a shared virtual edge to ensure the algorithm does not allocate more than the requested rate. We then use waterfilling to assign rates to these subdemands. This algorithm simply ignores the coupling between multiple paths and does not reach global max-min fair rates, but we use it as the starting point to generalize waterfilling. As a solution, it is fast, and we also use a variant of Alg. 1 to speed it up further (Alg. 2).

The new algorithm simplifies Alg. 1 by retaining the initial order of the links in subsequent iterations. In each iteration, it only recomputes the fair share for the link under consideration and fixes the rates for the demands bottlenecked by that link. It is approximate (even in the one-path case [5, 54]) but is faster and more parallelizable.

Global max-min fairness assigns lower rates to subdemands that are on congested paths but their corresponding demand can get enough allocation from its other paths. For example, the blue demand in Fig. 7(a) should receive a lower allocation on the path through the congested link. Intuitively, the allo-

Algorithm 2: Our Approx. Waterfilling algorithm to compute single-path weighted max-min fair rates.

Input: $\boldsymbol{\Gamma}$ where $\boldsymbol{\Gamma}[e,k]$ is the weight of single-path demand k on link e.
Input: $\mathbf{c}$: link capacity vector.
Output: $\mathbf{f}$: max-min rate allocation vector.

$\mathbf{f} \leftarrow \boldsymbol{\infty}$ — initial max-min rate set to ∞
$\mathbf{n} \leftarrow \boldsymbol{\Gamma}\mathbf{1}$ — total weight per link
$\mathcal{L} \leftarrow \arg\text{sort } \frac{\mathbf{c}}{\mathbf{n}}$ — vector division, sort links in ascending order
foreach $e \in \mathcal{L}$ **do**
  $\mathcal{D}_e \leftarrow \{k : \boldsymbol{\Gamma}[e,k] > 0\}$ — demands on link e
  while $\mathcal{D}_e \neq \emptyset$ **do**
    $\zeta \leftarrow \frac{\mathbf{c}[e]}{\sum_{k \in \mathcal{D}_e} \boldsymbol{\Gamma}[e,k]}$ — fair share of link e
    $\mathcal{B} \leftarrow \{k \in \mathcal{D}_e : \mathbf{f}[k] < \zeta \boldsymbol{\Gamma}[e,k]\}$
    if $\mathcal{B} = \emptyset$ **then** — if no flows bottlenecked elsewhere,
      $\mathbf{f}[\mathcal{D}_e] \leftarrow \zeta \boldsymbol{\Gamma}[e,\mathcal{D}_e]$ — fix rate to weighted share.
      break
    else — otherwise, remove those bottlenecked elsewhere.
      $\mathbf{c}[e] \leftarrow \mathbf{c}[e] - \sum_{k \in \mathcal{B}} \mathbf{f}[k]$
      $\mathcal{D}_e \leftarrow \mathcal{D}_e \setminus \mathcal{B}$
    end
  end
end
return $\mathbf{f}$

cator can get closer to global max-min fair assignments by moving each demand's allocation from more congested paths to less congested ones. We can achieve this by iteratively seeking more rates from subdemands that have received higher rates (*i.e.,* on less-contended paths) in previous iterations.

AdaptiveWaterfiller (or AW): Motivated by this intuition, AW uses a weighted version of aW (using Alg. 1 or Alg. 2) and adjusts the input weights ($\boldsymbol{\Gamma}$) to seek more rate from subdemands on less congested paths.

Let θ_k^p be the weight multiplier for the subdemand of demand k on path p. AW initializes these multipliers as $\theta_k^p = \frac{1}{\|\{p \in \mathcal{P}_k\}\|}$. In each iteration, AW first computes $\boldsymbol{\Gamma}$ directly from θ. $\boldsymbol{\Gamma}[e,k_p]$ is the weight of the subdemand k_p (demand k on path p) on link e[11]. Following the definition, $\boldsymbol{\Gamma}[e,k_p] = \theta_k^p \; \mathbb{1}[e \in p]$. AW then invokes one of the waterfilling algorithms[12] with these weights $\boldsymbol{\Gamma}$. For iteration $t+1$, AW sets $\theta_k^p(t+1) = \frac{f_k^p(t)}{\sum_p f_k^p(t)}$ where $f_k^p(t)$ is the rate demand k obtains from path p in iteration t. We show how multipliers and rates evolve in our example in Fig. 7(b).

AW converges when $\theta_k^p(t+1) = \theta_k^p(t)$. We can prove that adapting weight multipliers gets close to global max-min fairness: we say a rate assignment in the multi-path setting is *bandwidth-bottlenecked* if for all demands k, (i) each of its subdemands f_k^p is bottlenecked on some link l, and (ii) $f_k \geq f_j$, for all demands j that have any subdemand on any such link l. We prove in §D.1 that:

Theorem 3. If *the adaptive waterfiller converges, it converges to a bandwidth bottlenecked assignment.*

We prove the global max-min fair allocation is bandwidth-bottlenecked (see §D.2). The converse is not true — not all bandwidth-bottlenecked allocations are max-min fair. However, the set of bandwidth-bottlenecked allocations is *significantly* smaller than the set of all feasible allocations. We also prove that AW converges when its assignment is bandwidth-bottlenecked (*i.e.,* it stops iterating). Empirically, AW's allocations stabilize within 5 – 10 iterations on average (§4.4).

Adaptive waterfiller produces allocations that belong to a constrained set containing the optimal max-min fair rates. It is slower than approximate waterfiller because it iterates and updates weight multipliers. It is faster than the Geometric Binner as it does not solve an LP. Users can tune the maximum number of iterations to trade-off between fairness and speed.

3.3 Combinations and Extensions

Empirically, we find that the geometric binner (§3.1) is fairer than what its worst-case guarantee suggests (recall, we prove the rates will be within $[\alpha^{-1}, \alpha]$ times the optimal max-min fair rate). We can attribute most of the unfairness to bins that happen to contain many demands (Fig. A.5): can we set the bin boundaries differently to improve fairness?

We use the generalized waterfillers (§3.2) — which are fast but lack worst-case guarantees — to set bin boundaries in a way that spreads demands more uniformly across bins:

Equi-depth Binner (or EB) applies GeoBinning (Eqn. 4) with a few changes: it uses the rate allocation from AdaptiveWaterfiller to approximate the order across demands; distributes demands more uniformly over bins; and finds the bin boundaries as part of the optimization. Specifically, EB divides demands $\mathcal{D}$ into N_β equi-sized sets ($\mathcal{D}_1 \ldots \mathcal{D}_n$) based on their increasing order of rates from AW. In EB, the demands in a set D_b only receive rates from one bin with index b. EB dynamically chooses bin boundaries: $\forall k \in D_b, \ell_{b-1} \leq f_k < \ell_b + s(b)$ where $s(b)$ is a small constant that helps reducing the impact of inaccuracies from AW. We provide a more formal definition of EB in §E.

EB is slower than GB and AW because it executes both but we expect it to be fairer than GB — it spreads demands more uniformly across bins. We empirically confirm this hypothesis. It is hard to formally analyze EB but we suspect it also offers tighter worst-case guarantees. This is subject for future work.

Extensions: We did not explicitly account for weighted max-min fairness (e.g., w_k in §2.1) when describing our one-shot optimization. This extension is straightforward. For example, we can replace the first constraint in the geometric binner (GB) in Eqn. 4 with $f_k/w_k = \sum_{\text{bins } b} f_{kb}$. Algorithm 2, which we use in our generalized waterfillers in §3.2, already supports

[11] Note that waterfilling requires each demand to be on a single path. We use the notation k_p to show the single-path subdemands.

[12] We use Alg. 2 for our experiments since it is an order of magnitude faster with only a slight decrease in fairness (Fig. 8).

weights. We compute the per-edge per-subdemand weighted routing matrix as $\boldsymbol{\Gamma}[e, k_p] = w_k \theta_k^p \mathbb{1}[e \in p]$.

We have also omitted the heterogeneous utilities, different resource consumption scales, and other affine functions in our model (§2.1) when describing the solutions. These extensions are also straightforward and we show them in §A. The key is to appropriately manipulate the constraints that determine when an allocation is feasible (FeasibleAlloc, Eqn. 5).

4 Evaluation

Implementation. We implemented Soroush in Python and C# using Gurobi 9.1.1 [27] as the solver.

Summary of results. We apply Soroush to traffic engineering (TE) and cluster scheduling (CS). We show Soroush captures the trade-off between speed, fairness, and efficiency. We also show the results from integrating Soroush with Azure's production TE system where it reduces the run-times by up to $5.4\times$ without any impact on efficiency and fairness.

In TE, all the allocators in Soroush are faster than both the optimal algorithm by Danna *et al.* [17] (referred to as Danna) and the more practical α-approximate SWAN [30]. Soroush contains algorithms that match or exceed the efficiency or fairness of these methods while running orders of magnitude faster. Soroush can also trade-off (a little) fairness and efficiency for up to 3 orders of magnitude speed up.

Our solution scales to one of the largest WAN topologies (over 1000 nodes and 1000 edges), which is significantly larger than those in [10, 17, 30, 34, 75] and matches the size of topologies in [4]. We also analyze the sensitivity of Soroush to demand variations and other relevant inputs.

In CS, we show Soroush outperforms two variants of Gavel [56]. Our Equi-depth binner (EB) has the same fairness and efficiency as the optimal variant of Gavel (the one with waterfilling), but is 2 orders of magnitude faster.

4.1 Benchmarks and Metrics

Benchmarks. We use state-of-the-art solutions in both WAN-TE and CS as benchmarks to evaluate Soroush:

WAN-TEs. We use Danna [17], SWAN [30], and a modified version of the k-waterfilling algorithm [36] as benchmarks. We also provide limited comparisons with B4 [34] for completeness (see §4.2). The k-waterfilling algorithm only applies to single-path, infinite-demand scenarios — we extend it to multi-path, demand-constrained cases. We tune each benchmark for maximum speed (see §G.1). Following [30], we set $\alpha = 2$ for SWAN and GB unless mentioned otherwise.

CS. We compare with two variants of Gavel [56], the state-of-the-art max-min fair allocator in CS (with and without waterfilling). We use Gavel's public implementation.

Metrics. We use the following metrics for comparisons:

Fairness. We report fairness of a particular allocation ($\mathbf{f}$) as its distance from the optimal max-min fair allocation ($\mathbf{f}^*$)[13]. For fairness distance, we use the q_ϑ metric [46, 47]. This metric is resilient to numerical instability and is computed as $\min\left(\frac{\max(f_k,\vartheta)}{\max(f_k^*,\vartheta)}, \frac{\max(f_k^*,\vartheta)}{\max(f_k,\vartheta)}\right)$ for a given demand k. We report the geometric mean of q_ϑ across all the demands as the overall fairness measure (the geometric mean is less sensitive than the arithmetic mean to outliers). For our evaluations, we use $\vartheta = 0.01\%$ of the resource (link or GPU) capacities.

Topology	# Nodes	# Edges
WANLarge	∼1000s	∼1000s
WANSmall	∼100s	∼1000s
Cogentco	197	486
UsCarrier	158	378
GtsCe	149	386
TataNld	145	372

TABLE 4: Topologies used for the evaluation of Soroush.

Efficiency. We measure efficiency in TE as the total rate allocated to flows relative to Danna (*i.e.,* $\frac{e}{e_{danna}}$). For CS, we measure the effective throughput which is the progress rate of a job given an allocation. We report CS efficiency relative to Gavel (*i.e.,* $\frac{e}{e_{gavel}}$).

Runtime. In most cases, we report speed up (*i.e., relative* runtime compared to a baseline $\frac{s_{baseline}}{s}$). Our runtimes consist of the time each algorithm needs to compute the allocations. We measure runtimes on an AMD Operaton 2.4GHz CPU (6234) with 24 cores and 64GB of memory.

4.2 WAN Traffic Engineering

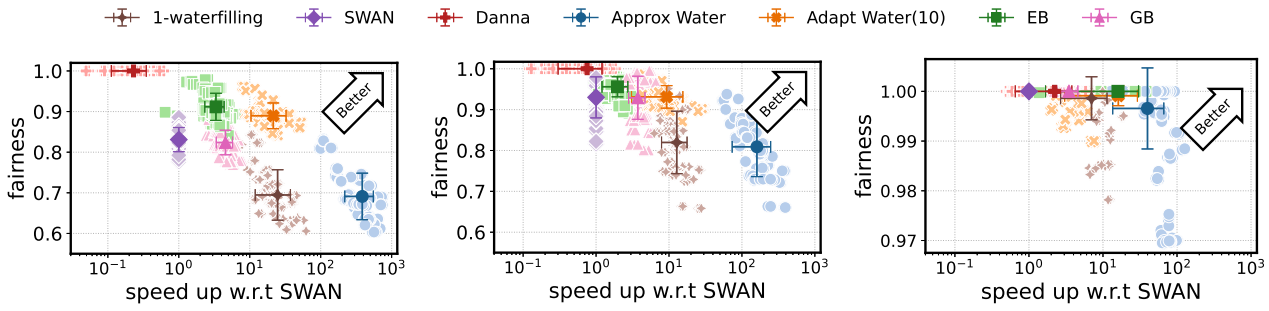
Experiment Setup. Table 4 summarizes the topologies in our evaluation. We show the results for both Azure's production WAN topology and the topologies from the Topology Zoo [1]. We use K-shortest paths [73] to find the paths between node pairs (K=16 unless mentioned otherwise).

For topologies from Topology Zoo, we generate traffic using Poisson [6], Uniform, Bimodal, and Gravity [6, 62] distributions. We follow [4] and generate traffic at different *scale factors*. Our traffic spans a range of loads: light (scale factors {1, 2, 4, 8}), medium ({16, 32}), and high ({64, 128}). At higher loads, more flows compete for traffic than at medium or light loads. We report results of over 640 experiments, which capture different traffic and topology combinations.

Comparison to benchmarks (Fig. 8 and 9). All algorithms in Soroush are faster than SWAN and Danna (Fig. 8). Each approach is in a different color in this figure, and each point corresponds to a single traffic demand on a single topology. The plot also shows the mean and standard deviations along the fairness and speedup axes.

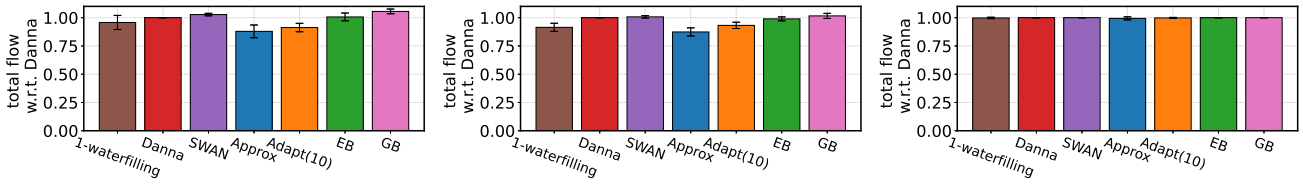
We see the trade-off across these different max-min fair resource allocators: (a) Danna is optimal but also by far the

[13] Danna and Gavel (w waterfilling) compute the optimal max-min fair allocations in TE and CS respectively. They are too slow for practice but we can run them to completion outside of a production environment.



(a) High Load (scale factor ∈ {64, 128}) (b) Medium Load (scale factor ∈ {16, 32}) (c) Light Load (scale factor ∈ {1, 2, 4, 8})

FIGURE 8: The fairness vs speed trade-off across different approaches. As in [4], we use the scale-factor to denote the level of load. We observe even the slowest algorithm in Soroush is faster than SWAN and Danna. While 1-waterfilling is faster than most of the algorithms in Soroush, it has to sacrifice much more in terms of fairness (it is 30% less fair than Danna in the high load case).



(a) High Load (scale factor ∈ {64, 128}) (b) Medium Load (scale factor ∈ {16, 32}) (c) Light Load (scale factor ∈ {1, 2, 4, 8})

FIGURE 9: The efficiency of Soroush's algorithms and our benchmarks. We report numbers relative to Danna. Empirically, Soroush Pareto-dominates SWAN, 1-waterfilling, and Danna on the efficiency, agility, and fairness. In (c), the error bar is small because of light load — most solutions can satisfy all the demands (fairness is also close to one for most algorithms in these cases.)

slowest (on average taking 4.3× longer than the second slowest algorithm, SWAN, under high-load); (b) 1-waterfilling is the fastest of the baselines but does not consider flow-level fairness (30% less fair than Danna on average but 4 orders of magnitude faster); (c) SWAN sits somewhere in between. It is faster than Danna (solves fewer optimizations), but slower than 1-waterfilling (1-waterfilling does not solve any optimization). It is fairer than 1-waterfilling but unlike Danna does not achieve optimal max-min fairness; (d) Soroush empirically Pareto-dominates these baselines as each of its algorithms provide a different point on the trade-off space.

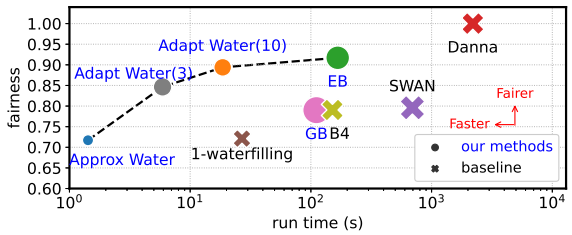
Our algorithms are most effective under *high loads* (arguably, speed *and* fairness matter most). Soroush's Geometric Binner (GB) is faster than SWAN by 4.5× on average (6× in the 90th percentile) because it only solves a single optimization. GB also has worst-case fairness guarantees. The Equi-depth Binner (EB) is faster than SWAN, slightly slower than GB, and fairer than both. Soroush's Approximate Waterfiller is even faster than 1-waterfilling (by an order of magnitude) with the same flow-level fairness. Soroush's Adaptive Waterfiller improves fairness (19% higher on average) at a slight speed reduction (still 21.4× faster than SWAN on average).

Fig. 9 compares the efficiency of different methods. Under low loads, all schemes are comparable. The differences become evident at higher loads, where EB is approximately as efficient as Danna. GB and SWAN are more efficient, likely because they sacrifice fairness.

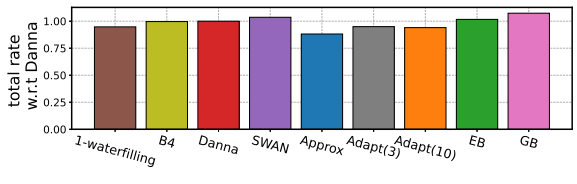
We can see these differences more clearly when we focus on a single topology and workload in Fig. 10. Soroush's allocators Pareto-dominate other approaches. The Approximate Waterfiller, Adaptive Waterfiller (number of iterations = 3 and 10), and EB are faster than SWAN and Danna. Adaptive Waterfiller and EB are also fairer than SWAN while having comparable efficiency. Operators can use GB to get strong worst-case guarantees (at the cost of reduced fairness). B4 [34]'s TE algorithm is just as fast and fair as GB (albeit slightly less efficient) but does not have fairness guarantees. Note that we can control the fairness and runtime of GB by tuning α, whereas we can not control either in B4.

In summary, in settings where Danna is impractical, Soroush outperforms other TE algorithms (SWAN, 1-waterfilling, B4). Depending on the requirements, users can opt for Adaptive or Approximate Waterfillers, or EB (or GB if fairness guarantees are important). They can also customize the parameters in each allocator to further tune the balance.

Production deployment (Fig. 11). We have successfully deployed Soroush in the production TE pipeline of Azure. Microsoft opted for GB as it has the same fairness guarantees as their existing TE solver. Fig. 11(a) shows cumulative density

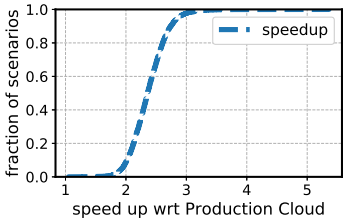


(a) Fairness vs Run-time

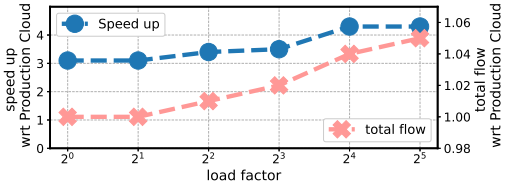


(b) Efficiency wrt Danna

FIGURE 10: The empirical Pareto-dominance of Soroush over all of our baselines on an example topology (Cogentco) and an example workload with $64\times$ scale factor. The size of the markers in (a) are in proportion to the efficiency of each algorithm — we report exact comparisons in (b).



(a) Speedup



(b) Impact of load

FIGURE 11: Results from deploying Soroush in production. (a) Month-long measurements show substantial speedup with no impact on efficiency or fairness compared to the provider's previous max-min fair allocator. (b) Using production traces, we show the benefit of Soroush improves as loads [4] increase.

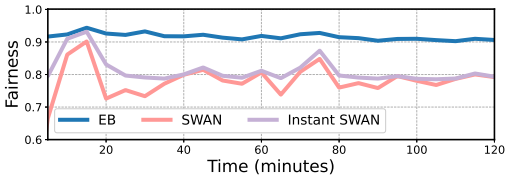


FIGURE 12: Impact of solver runtimes on fairness when demands change. SWAN can not react to the new demands quickly and faces another 10% reduction in fairness whereas EB can keep track of the changes. These results are on Cogentco following NCFlow's change distribution [4] on medium load traffics.

function (CDF) of the relative speed up of Soroush compared to the provider's previous allocator. These measurements are over a month-long deployment in a WAN with thousands of nodes. Soroush reduces the run-time on average by $2.4\times$ (up to $5.4\times$) without impacting fairness or efficiency.

We compare Soroush with the previous allocator on production demands at different loads (Fig. 11(b)). Soroush's speedup increases with the load because the previous iterative solver invokes more optimizations at higher loads. Soroush's efficiency also increases because its ϵ-trick can exploit minor fairness violations to improve efficiency. In all cases, Soroush is within 1% of the previous solver's fairness.

Tracking Changing Demands (Fig. 12). We evaluated each method on a sequence of traffic, arriving every five minutes (a window), starting from a medium load traffic demand. Our methodology is the same as NCFlow [4]. In this scenario, SWAN needs two windows to compute each allocation – it only computes allocations for half of the demands. This results in up to 10% reduction in fairness compared to an instant SWAN (a hypothetical scheme that computes the allocation instantly). However, EB[14] reacts to changes quickly and meets all the deadlines. In general, SWAN's inability to keep track of demands leads to even higher unfairness than EB (relative to what we reported in Fig. 8). Also, as we move from medium to high load, we expect the difference to be more as SWAN is even slower and needs to solve more optimizations.

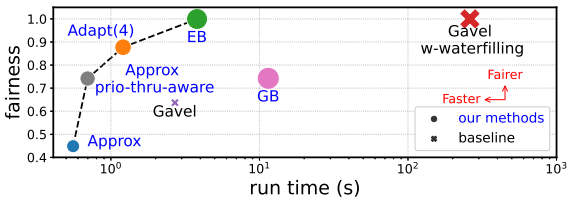
4.3 Cluster Scheduling

Experiment Setup. We generate job requests from Gavel's job generator: we consider 3 types of GPUs (V100, P100, K80) and uniformly sample jobs from the 26 different job types available in Gavel (see §G.2). Jobs are heterogeneous: they require a different number of workers (which we derive from the Microsoft public trace [3]) and have different priorities (which we sample uniformly from the set $\{1, 2, 4, 8\}$).

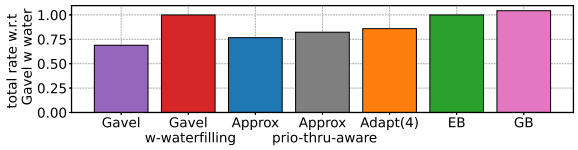
Comparison to benchmarks. We report results on over 40 different scenarios, which capture different number of available GPUs and competing jobs (see §G.2 for more details). Our results match our observations from WAN-TE; Soroush Pareto-dominates both Gavel and Gavel with waterfilling. We present these results in Fig. A.2 in §G.2 for space.

We provide further insight into Soroush's performance through an example scenario where 8192 jobs compete for resources (Fig. 13). Adaptive Waterfiller outperforms standard Gavel in fairness, efficiency, and speed. For CS, GB is slower than Gavel but fairer (more than 10%) and more efficient (more than 30%). We can augment Gavel with waterfilling [56] to improve it, but with a substantial slowdown. In contrast, EB provides comparable fairness and efficiency as Gavel with waterfilling and is ~ 2 orders of magnitude faster.

[14] GB is faster than EB. If the latter can keep up, so can GB. We have omitted an evaluation based on GB for this reason.



(a) Fairness vs Run-time



(b) Efficiency wrt Gavel w-waterfilling

FIGURE 13: Trade-off between efficiency, fairness, and speed in CS on an example scenario (with 8192 jobs). (a) shows the fairness vs run-time behavior of the different approaches; (b) shows the efficiency relative to the Gavel w waterfilling. Empirically, Soroush Pareto-dominates both variants of Gavel.

4.4 Convergence and Sensitivity Analysis

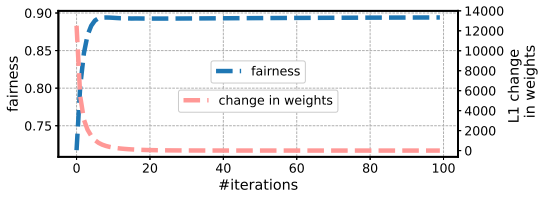
Convergence. We empirically evaluate the convergence of the Adaptive Waterfiller. In §D, we proved the algorithm in §3.2 only converges to and stops if it finds a bandwidth-bottlenecked allocation but may not converge if it does not find one. We empirically find that Adaptive Waterfiller always converges. Fig. 14(a) shows how its weights and fairness properties change with the number of iterations: the weights stabilize after 5 iterations.

Impact of number of bins. Fig. 14(b) and 14(c) show fairness and efficiency of binners (GB/EB) for different number of bins. Soroush uses this parameter to tune the trade-off between efficiency, fairness, and run-time. Using more bins increases fairness because the number of demands within each bin decreases but at the cost of higher run-time (more variables in the optimization). EB is fairer than GB for up to 16 bins because GB suffers from bin-imbalance. However, GB does not incur bin-imbalance for ≥ 32 bins and both methods have roughly the same fairness. The slightly lower fairness of EB is due to Adaptive Waterfiller making small mistakes when estimating the order of rates and influencing EB's binning.

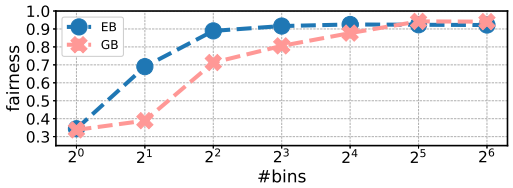
4.5 Other Experiments

Impact of number of paths. We explore how sensitive our solutions are to the number of paths by varying this parameter and comparing our fairest methods (*i.e.,* Adaptive Waterfilling and EB) to SWAN (Fig. 15). Increasing the number of paths improves the benefit of Soroush in both speedup and fairness. With more paths, each optimization of SWAN becomes more expensive, while Adaptive Waterfiller as well as EB can exploit path diversity better to achieve higher fairness.

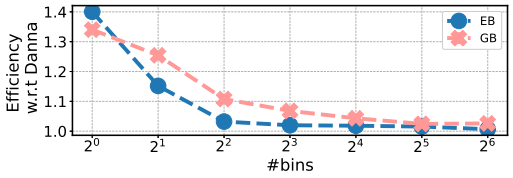
Impact of topology size. The benefit of Soroush's allocators increases with the topology size (Fig. 16): SWAN needs to



(a) Convergence of the Adaptive Waterfiller



(b) Impact of the number of bins on the fairness



(c) Impact of the number of bins on the efficiency

FIGURE 14: Convergence and sensitivity analysis. (a) Adaptive Waterfiller empirically converges within 5 – 10 iterations. (b, c) The number of bins controls the trade-off between fairness and efficiency in EB and GB (fewer bins lead to higher efficiency and lower fairness). Results are on the Cogentco topology and Gravity traffic distribution (scale factor = 64). (see Fig. A.3 for Poisson)

solve more optimizations for larger topologies while Soroush solves a fixed number of optimizations (=1 for EB/GB and =0 for adaptive waterfilling).

Comparison to NCFlow and POP. NCFlow [4] and POP [55] decompose the resource allocation problem to scale but do not directly address max-min fairness [65]. NCFlow only maximizes the total flow, and the authors mention in the paper that it is hard to extend it to max-min fairness objective [4]. Similarly, POP maximizes total flow and maximum concurrent flow (*i.e.,* the smallest fractional allocation) but does not provide any results on max-min fairness. To understand how POP compares to Soroush, we adapt both SWAN and Soroush to use it. We randomly divide demands (with client splitting as needed per POP's guidelines) among different partitions and run SWAN or Soroush in parallel on each partition (Fig. 17, extended evaluation in §G.3).

We use GB to ensure a fair comparison to SWAN: it has the same theoretical guarantees, is more than $10\times$ faster, and maintains the same level of fairness. When we apply POP to SWAN, we lose the worst-case guarantee [53] and have to sacrifice over 10% in fairness to achieve the same speed as Soroush. We also observe that applying POP to Soroush results in the same fairness as SWAN for the same number of partitions (but is also substantially faster).

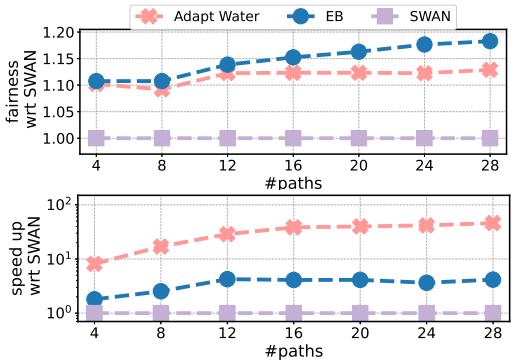


FIGURE 15: Increasing the number of paths improves the fairness and speedup of Soroush compared to SWAN. Results are on the Cogentco topology and Gravity traffic distribution (scale factor = 64). (see Fig. A.4 for Poisson)

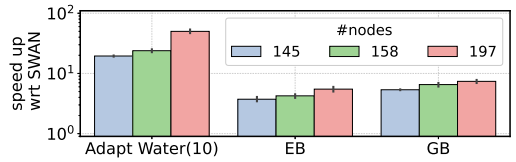


FIGURE 16: Impact of topology size. Soroush's speed up relative to SWAN improves with the size of the topology.

5 Discussion

Soroush allows operators to adjust the trade-off between fairness, speed, and efficiency. We focus on multi-path allocations but our solutions apply to single-path settings too [5,54]. Under this setting, our experiments show the Approximate Waterfiller is an order of magnitude faster than the fastest single-path allocator with only a slight decrease in efficiency. We defer the following to future work:

Other fairness metrics. Soroush does not apply to other, less commonly used, fairness metrics [11,12,33].

Other problem domainss. Soroush applies to any graph-based resource allocation problem which seeks to achieve max-min fairness. We demonstrate significant benefit of Soroush using examples from CS and WAN-TE. To use Soroush in other domains [5,24,36,43,50,54,64,71], users need to model the additional constraints in our graph model. We aim to provide tools to simplify this in future work.

Distributed extension. Soroush applies to centralized resource allocation problems. Our future work aims to extend it to distributed settings [36,61,70,74].

6 Related Work

TE and CS resource allocation. Prior approaches to both TE and CS aim to produce fast and efficient allocations [14, 16, 17, 23, 30, 34, 36, 37, 42, 56, 59]. In §4, we show Soroush outperforms the sate-of-the-art in multi-resource max-min fair allocation (SWAN, Danna, B4, waterfilling, and Gavel).

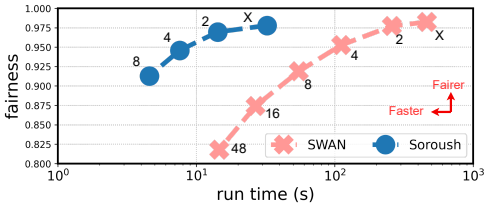


FIGURE 17: Impact of POP [55]. The results are on 3 randomly generated traffic, following Poisson distribution with a scale factor 64, on the Cogentco topology. Consistent with POP, we use client splitting (ratio= 0.75) for this traffic distribution. ["X" indicates that POP is not used, and "Numbers" = number of POP partitions.]

Prior work employs ML in TE [58, 68, 72] to optimize objectives that are already solved using a single LP (*e.g.,* max flow). These objectives are either convex or quasi-convex [58]. However, the exact from of max-min fairness is sequential and we are unaware of any work that considers end-to-end training on a sequence of LPs. In fact, it may be more tractable for ML methods to learn our GB which is a single LP. Applying ML to further speed up Soroush is an interesting future direction.

Recent work [4,55] uses decomposition techniques to scale resource allocation problems. However, they focus on simpler objectives such as max flow or max concurrent flow that require single LPs and do not explicitly support max-min fairness. Extending NCFlow to max-min fairness is non-trivial as the authors mentioned [4]. We have extended POP to support max-min fairness and empirically compare it with Soroush. Our results show POP's performance depends on the traffic distribution, whereas Soroush works consistently well. POP also does not have worst-case fairness guarantees [53], whereas Soroush has allocators that do (GB).

Algorithms for computing max-min fair rates. Prior work has expanded our understanding of max-min fair resource allocation [57, 60]. These are largely theoretical and do not provide a practical and fast solution. Bandit-based solutions [9] lack worst-case guarantees and do not allow users to control the trade-off between fairness, efficiency and speed.

7 Conclusion

Soroush enables fast max-min fair resource allocation for graph-based problems such as traffic engineering and cluster scheduling. It provides a suite of allocators that Pareto-dominate state-of-the-art in both of these domains. Some of the allocators in Soroush have theoretical guarantees, and all of them have parameters for users to control the trade-offs. We have deployed Soroush in Azure's WAN traffic engineering pipeline. Future work can explore other applications and other notions of fairness.

Acknowledgements: We thank Ymir Vigfusson and the anonymous reviewers for their feedback on this paper. We also thank Luis Irun-Briz for his support of this work. This material is based upon work supported by Microsoft and the U.S. National Science Foundation under grant No. CNS-1901523.

References

[1] Internet topology zoo. http://www.topology-zoo.org/.

[2] Imagenet training in pytorch. https://github.com/pytorch/examples/tree/main/imagenet, 2020.

[3] Microsoft philly trace. https://github.com/msr-fiddle/philly-traces, 2022.

[4] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. Contracting wide-area network topologies to solve flow problems quickly. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 175–200. USENIX Association, April 2021.

[5] Omid Alipourfard, Jiaqi Gao, Jeremie Koenig, Chris Harshaw, Amin Vahdat, and Minlan Yu. Risk based planning of network changes in evolving data centers. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 414–429, New York, NY, USA, 2019. Association for Computing Machinery.

[6] David Applegate and Edith Cohen. Making intra-domain routing robust to changing and uncertain traffic demands: Understanding fundamental tradeoffs. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, page 313–324, New York, NY, USA, 2003. Association for Computing Machinery.

[7] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 307–314, New York, NY, USA, 1968. Association for Computing Machinery.

[8] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice-Hall, Inc., USA, 1987.

[9] Ilai Bistritz, Tavor Baharav, Amir Leshem, and Nicholas Bambos. My fair bandit: Distributed learning of max-min fairness with multi-player bandits. In *International Conference on Machine Learning*, pages 930–940. PMLR, 2020.

[10] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. Teavar: striking the right utilization-availability balance in wan traffic engineering. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 29–43. Association for Computing Machinery, 2019.

[11] Thomas Bonald, Laurent Massoulié, Alexandre Proutiere, and Jorma Virtamo. A queueing analysis of max-min fairness, proportional fairness and balanced fairness. *Queueing systems*, 53(1):65–84, 2006.

[12] Thomas Bonald and Alexandre Proutiere. On performance bounds for balanced fairness. *Performance Evaluation*, 55(1-2):25–50, 2004.

[13] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[14] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Michael B. Cohen, Yin Tat Lee, and Zhao Song. Solving linear programs in the current matrix multiplication time. *Journal of the ACM*, 2021.

[16] Emilie Danna, Avinatan Hassidim, Haim Kaplan, Alok Kumar, Yishay Mansour, Danny Raz, and Michal Segalov. Upward max-min fairness. *J. ACM*, 64(1), mar 2017.

[17] Emilie Danna, Subhasree Mandal, and Arjun Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *2012 Proceedings IEEE INFOCOM*, pages 846–854, 2012.

[18] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[19] Steven Diamond and Stephen Boyd. Cvxpy: A python-embedded modeling language for convex optimization. *J. Mach. Learn. Res.*, 17(1):2909–2913, jan 2016.

[20] Desmond Elliott, Stella Frank, Khalil Sima'an, and Lucia Specia. Multi30K: Multilingual English-German image descriptions. In *Proceedings of the 5th Workshop on Vision and Language*, pages 70–74, Berlin, Germany, August 2016. Association for Computational Linguistics.

[21] Alexander Gersht and Robert Weihmayer. Joint optimization of data network design and facility selection. *IEEE Journal on Selected Areas in Communications*, 8(9):1667–1681, 1990.

[22] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant

resource fairness: Fair allocation of multiple resource types. In *NSDI*. USENIX, 2011.

[23] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 365–378, 2013.

[24] Ada Gogu, Dritan Nace, Supriyo Chatterjea, and Arta Dilo. Max-min fair link quality in wsn based on sinr. *Journal of applied mathematics*, 2014, 2014.

[25] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[26] David Griffis. Rl a3c pytorch. https://github.com/dgriff777/rl_a3c_pytorch, 2020.

[27] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022.

[28] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4), dec 2015.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[30] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 15–26, New York, NY, USA, 2013. Association for Computing Machinery.

[31] Yu-Hsiang Huang. Attention is all you need: A pytorch implementation. https://github.com/jadore801120/attention-is-all-you-need-pytorch, 2020.

[32] Hosagrahar Visvesvaraya Jagadish, Nick Koudas, S Muthukrishnan, Viswanath Poosala, Kenneth C Sevcik, and Torsten Suel. Optimal histograms with quality guarantees. In *VLDB*, 1998.

[33] Raj Jain, Arjan Durresi, and Gojko Babic. Throughput fairness index: An explanation. In *ATM Forum contribution*, volume 99, 1999.

[34] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 3–14, New York, NY, USA, 2013. Association for Computing Machinery.

[35] Virajith Jalaparti, Ivan Bliznets, Srikanth Kandula, Brendan Lucier, and Ishai Menache. Dynamic pricing and traffic engineering for timely inter-datacenter transfers. SIGCOMM '16, page 73–86, New York, NY, USA, 2016. Association for Computing Machinery.

[36] Lavanya Jose, Stephen Ibanez, Mohammad Alizadeh, and Nick McKeown. A distributed algorithm to calculate max-min fair rates without per-flow state. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(2), jun 2019.

[37] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. Walking the tightrope: Responsive yet stable traffic engineering. *SIGCOMM Comput. Commun. Rev.*, 35(4):253–264, aug 2005.

[38] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. Decentralized cloud wide-area network traffic engineering with BLASTSHIELD. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 325–338, Renton, WA, April 2022. USENIX Association.

[39] Umesh Krishnaswamy, Rachee Singh, Paul Mattes, Paul-Andre C Bissonnette, Nikolaj Bjørner, Zahira Nasrin, Sonal Kothari, Prabhakar Reddy, John Abeln, Srikanth Kandula, Himanshu Raj, Luis Irun-Briz, Jamie Gaudette, and Erica Lan. OneWAN is better than two: Unifying a split WAN architecture. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 515–529, Boston, MA, April 2023. USENIX Association.

[40] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The cifar-10 dataset. http://www.cs.toronto.edu/kriz/cifar.html, 2014.

[41] kuang liu. Train cifar10 with pytorch. https://github.com/kuangliu/pytorch-cifar, 2020.

[42] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. Allox: Compute allocation in hybrid clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[43] Rui Li and Paul Patras. Max-min fair resource allocation in millimetre-wave backhauls. *IEEE Transactions on Mobile Computing*, 19(8):1879–1895, 2019.

[44] Erik Linder-Norén. Pytorch-gan. https://github.com/eriklindernoren/PyTorch-GAN#cyclegan, 2020.

[45] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 527–538, New York, NY, USA, 2014. Association for Computing Machinery.

[46] Yao Lu, Srikanth Kandula, Arnd Christian König, and Surajit Chaudhuri. Pre-training summarization models of structured datasets for cardinality estimation. *Proceedings of the VLDB Endowment*, 15(3):414–426, 2021.

[47] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711*, 2019.

[48] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. 2016.

[49] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.

[50] Sourav Mondal and Marco Ruffini. A min-max fair resource allocation framework for optical x-haul and du/cu in multi-tenant o-rans. In *ICC 2022-IEEE International Conference on Communications*, pages 3016–3021. IEEE, 2022.

[51] Abdallah Moussawi. Towards large scale training of autoencoders for collaborative filtering. *ArXiv*, abs/1809.00999, 2018.

[52] Dritan Nace, Linh Nhat Doan, Olivier Klopfenstein, and Alfred Bashllari. Max-min fairness in multi-commodity flows. *Comput. Oper. Res.*, 35(2):557–573, feb 2008.

[53] Pooria Namyar, Behnaz Arzani, Ryan Beckett, Santiago Segarra, Himanshu Raj, and Srikanth Kandula. Minding the gap between fast heuristics and their optimal counterparts. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, HotNets '22, page 138–144, 2022.

[54] Pooria Namyar, Behnaz Arzani, Daniel Crankshaw, Daniel S. Berger, Kevin Hsieh, Srikanth Kandula, and Ramesh Govindan. Mitigating the performance impact of network failures in public clouds, 2023.

[55] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving large-scale granular resource allocation problems efficiently with pop. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 521–537, New York, NY, USA, 2021. Association for Computing Machinery.

[56] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498. USENIX Association, November 2020.

[57] Nhan-Tam Nguyen, Trung Thanh Nguyen, and Jörg Rothe. Approximate solutions to max-min fair and proportionally fair allocations of indivisible goods. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 262–271, 2017.

[58] Yarin Perry, Felipe Vieira Frujeri, Chaim Hoch, Srikanth Kandula, Ishai Menache, Michael Schapira, and Aviv Tamar. DOTE: Rethinking (predictive) WAN traffic engineering. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1557–1581. USENIX Association, 2023.

[59] Michal Pióro, Gábor Fodor, Pål Nilsson, and Eligijus Kubilinskas. On efficient max-min fair routing algorithms. In *Proceedings of the Eighth IEEE Symposium on Computers and Communications. ISCC 2003*, ISCC'03, page 365, USA, 2003. IEEE Computer Society.

[60] Bozidar Radunovic and Jean-Yves Le Boudec. A unified framework for max-min and min-max fairness with applications. *IEEE/ACM Transactions on networking*, 15(5):1073–1083, 2007.

[61] Jordi Ros-Giralt and Wei Kang Tsai. A theory of convergence order of maxmin rate allocation and an optimal protocol. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 2, pages 717–726. IEEE, 2001.

[62] Matthew Roughan, Albert Greenberg, Charles Kalmanek, Michael Rumsewicz, Jennifer Yates, and Yin Zhang. Experience in measuring backbone traffic variability: Models, metrics, measurements and meaning. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment*, IMW '02, page 91–92, New York, NY, USA, 2002. Association for Computing Machinery.

[63] Mark Saroufim. Word-level language modeling using rnn and transformer. https://github.com/pytorch/examples/tree/main/word_language_model, 2020.

[64] Adrian Schad and Marius Pesavento. Max-min fair transmit beamforming for multi-group multicasting. In *2012 International ITG Workshop on Smart Antennas (WSA)*, pages 115–118. IEEE, 2012.

[65] Rachee Singh, Nikolaj Bjørner, and Umesh Krishnaswamy. Traffic engineering: From isp to cloud wide area networks. In *Proceedings of the Symposium on SDN Research*, SOSR '22, page 50–58, 2022.

[66] Rachee Singh, Manya Ghobadi, Klaus-Tycho Foerster, Mark Filer, and Phillipa Gill. Radwan: Rate adaptive wide area network. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 547–560, New York, NY, USA, 2018. Association for Computing Machinery.

[67] Ion Stoica, Scott Shenker, and Hui Zhang. Core-Stateless Fair Queueing: A Scalable Architecture to Approximate Fair Bandwidth Allocations in High Speed Networks. In *SIGCOMM*, 1998.

[68] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. Learning to route. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, page 185–191, New York, NY, USA, 2017. Association for Computing Machinery.

[69] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[70] Weitao Wang, Masoud Moshref, Yuliang Li, Gautam Kumar, T. S. Eugene Ng, Neal Cardwell, and Nandita Dukkipati. Poseidon: Efficient, robust, and practical datacenter CC via deployable INT. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 255–274, Boston, MA, April 2023. USENIX Association.

[71] Yiting Xia, Ying Zhang, Zhizhen Zhong, Guanqing Yan, Chiun Lin Lim, Satyajeet Singh Ahuja, Soshant Bali, Alexander Nikolaidis, Kimia Ghobadi, and Manya Ghobadi. A social network under social distancing: Risk-Driven backbone management during COVID-19 and beyond. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 217–231. USENIX Association, April 2021.

[72] Zhiying Xu, Francis Y. Yan, Rachee Singh, Justin T. Chiu, Alexander M. Rush, and Minlan Yu. Teal: Learning-accelerated optimization of wan traffic engineering, 2023.

[73] Jin Y. Yen. Finding the K Shortest Loopless Paths in a Network. *Management Science*, 17(11):712–716, 1971.

[74] Liangcheng Yu, John Sonchack, and Vincent Liu. Cebinae: Scalable in-network fairness augmentation. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 219–232, New York, NY, USA, 2022. Association for Computing Machinery.

[75] Zhizhen Zhong, Manya Ghobadi, Alaa Khaddaj, Jonathan Leach, Yiting Xia, and Ying Zhang. Arrow: restoration-aware traffic engineering. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 560–579, 2021.

[76] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2242–2251, 2017.

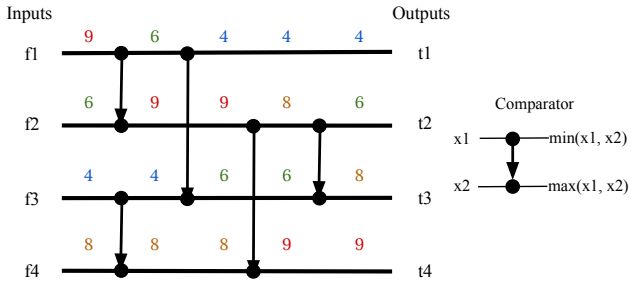


FIGURE A.1: Sorting Network Example.

A Max-min fair allocation optimization

Soroush offers a range of general algorithms that can solve any max-min fair resource allocation problem expressed using the model described in §2.1. In this section, we present the formulation behind this model. Table A.1 describes all the notations, their meanings, and their mappings to WAN Traffic Engineering (WAN-TE) and Cluster Scheduling (CS).

Feasible Allocation. Given a set of demands and a set of paths over a group of resources, an allocation is feasible if it satisfies demand and capacity constraints. We define the feasible allocation as:

$$\begin{aligned}
&\mathsf{FeasibleAlloc}(\mathcal{E},\mathcal{D},\mathcal{P},\mathcal{Q},\mathcal{R}) \triangleq \big\{\mathbf{f} \mid && (5)\\
&f_k = \sum_{p\in\mathcal{P}_k} q_k^p f_k^p, && \forall k \in \mathcal{D}\text{(allocation for demand } k)\\
&\sum_{p\in\mathcal{P}_k} f_k^p \le d_k, && \forall k \in \mathcal{D}\text{(allocation below volume)}\\
&\sum_{k,p\mid p\in\mathcal{P}_k, e\in p} r_k^e f_k^p \le C_e && \forall e \in \mathcal{E}\text{(allocation below capacity)}\\
&f_k^p \ge 0 && \forall p \in \mathcal{P}_k, k \in \mathcal{D}\text{(non-negative allocation)}\big\}
\end{aligned}$$

Max-Min Fairness. Among all the feasible allocations, the optimal max-min fair solution seeks:

$$\begin{aligned}
\mathsf{OptFair}(\mathcal{E},\mathcal{D},\mathcal{P},\mathcal{Q},\mathcal{R}) \quad &\triangleq \quad \arg\max_{\mathbf{f}} \quad \text{fair}(\mathbf{f}/\mathbf{w}) && (6)\\
&\text{s.t.} \quad \mathbf{f} \in \mathsf{FeasibleAlloc}(\mathcal{E},\mathcal{D},\mathcal{P},\mathcal{Q},\mathcal{R})
\end{aligned}$$

where the function $\text{fair}(\mathbf{x})$ encodes the max-min fairness objective. To our knowledge, prior works do not present a closed form of this function. In §B, we introduce two potential candidates (one exact and one that converges in the limit).

B Closed-form max-min fair objective

We present two closed form representations of the max-min fair objective: one exact, and one that converges to the max-min fair objective in the limit. The exact form is the following:

$$\text{fair}(\mathbf{f}) = \arg\max_{\mathbf{f}} \bigcup_{\{\mathcal{F}_A \mid \mathcal{F}_A \subseteq \mathbf{f}\}} \min(\{f_k \mid f_k \in \mathcal{F}_A\}) \quad (7)$$

Intuitively, this is a collection of maximization problems, where each maximizes the smallest flow in a given subset of $\mathbf{f}$ (a total of $2^{|\mathbf{f}|}$ maximizations). We next prove that this objective, in the instance that $\mathbf{f}$ are bounded, results in max-min fair allocations.

Proof. Without loss of generality, we assume if $i < j$ then $f_i \le f_j$ for all $f_i, f_j \in \mathbf{f}$.

Suppose the theorem is not true: there exists an allocation $\mathbf{f}^*$ which is optimum as measured by the objective in 7 but is not max-min fair. Three scenarios might have caused this;

Case 1. A flow i exists with unbounded f_i^*, which can not be true as we assume all the flows are bounded.

Case 2. A flow i exists that we can improve its rate without hurting other flows with $\le$ rate. One of the constraints in Eqn. 7 is to maximize f_i as a result such i can not exist.

Case 3. Two flows i and j exist $(i < j)$ with optimal max-min fair rates of $\hat{f}_i$ and $\hat{f}_j$ such that $\hat{f}_j < f_j^*$ and $f_i^* < \hat{f}_i$. This means that in the solution from Eqn. 7, flow j is receiving more than its share and is hurting flow i. This also can not happen since it violates one of the constraints in Eqn. 7 that maximizes the minimum of i and j. (Note that this holds even if $\hat{f}_i = \hat{f}_j$ since maximizing the minimum of these two ensures they get equal rates.)

As a result, each flow is guaranteed to be bounded, achieve its maximum possible rate, and can not hurt any other flow with less than or equal rate. This is the definition of max-min fairness ($\mathbf{f}^*$ is max-min fair). □

An alternate closed form representation of max-min fair is the following:

$$\text{fair}(\mathbf{f}) = \arg\max_{\mathbf{f}} \sum_i \epsilon^{\sum_{j\neq i} \mathbb{I}(f_i \le f_j)} f_i \quad (8)$$

We can prove this converges to the max-min fair rate allocations as $\epsilon \to 0$ similar to the proof of Theorem 1.

C SWAN as a sequence of LPs

The original formulation of the bth iteration of SWAN [30] is the following:

$$\begin{aligned}
\mathsf{SWANMaxMin}_b(\mathcal{E},\mathcal{D},\mathcal{P}) \quad &\triangleq \quad \arg\max \sum_{k\in\mathcal{D}} f_{kb} && (9)\\
\text{s.t.} \quad & f_{kb} \le U\alpha^{b-1}, && \forall k \in \mathcal{D}\\
& f_{kb} \begin{cases} = f_{k(b-1)} & \text{if } f_{k(b-1)} < U\alpha^{b-2} \\ \ge f_{k(b-1)} & \text{otherwise} \end{cases}, && \forall b > 1\\
& (\mathbf{f}_1,\ldots,\mathbf{f}_{b-1}) \in \mathsf{SWANMaxMin}_{b-1},\\
& \mathbf{f} \in \mathsf{FeasibleAlloc}(\mathcal{E},\mathcal{D},\mathcal{P}).
\end{aligned}$$

where f_{kb} is the total allocated rate to demand k up to iteration b.

Term	Meaning	CS	WAN-TE
$\mathbf{w}$	inverse of coefficient vector in the objective of fair(.) where w_k indicates the weight for the k-th demand and encodes the desired proportional max-min fair allocations.	priority of job k (In [56] = user specified priority $\times$ effective average throughput / number of workers)	priority of different services (*e.g.,* search and ads)
q_k^p	the utility obtained by demand k when assigned 1 unit on path p.	progress rate of the k-th job when assigned 1 unit on server p	$= 1$
r_k^e	capacity consumed on resource e (*i.e.,* link or GPU) when allocating 1 unit to demand k	capacity consumed by job k from resource e (CPU, GPU or memory)	$= 1$
c_e	capacity of resource $e \in \mathcal{E}$	capacity of CPU, GPU or any other resources on a server	capacity of link e
d_k	the resource requested by the k-th demand	job k's requested duration of time ($= 1$ in [56])	flow k's requested rate
f_k, f_k^p	f_k : demand k's total utility f_k^p : demand k's obtained allocation from path p	f_k : job k's total progress rate f_k^p: fraction of time server p is assigned to job k	f_k: flow k's total rate f_k^p: flow k's rate on path p

TABLE A.1: Additional notation for the general multi-resource max-min fair formulation in §A.

D Proofs of results for AdaptiveWaterfiller

We present the proofs of the various results mentioned in §3.2 for Adaptive Waterfiller.

D.1 Proof of Theorem 3

If we denote by $\mathbf{f}(\theta)$, the solution of solving the weighted waterfilling sub-flow problem with weights $\theta = \{\theta_k^p\}$, then convergence implies that

$$\theta_k^p = \frac{f_k^p(\theta)}{f_k(\theta)}, \tag{10}$$

so that $\theta_k^p(t+1) = \theta_k^p(t)$ for all p, k. From the definition of single-path weighted waterfilling, it must be that if f_k^p is bottlenecked at link l, then $\frac{f_k^p}{\theta_k^p} \geq \frac{f_j^{\hat{p}}}{\theta_j^{\hat{p}}}$ for all non-zero $f_j^{\hat{p}}$ going through that link. Using Eqn. 10 to replace the weights in this inequality, it immediately follows that $f_k \geq f_j$. Since this must hold for every j such that there exists a non-zero subflow $f_j^{\hat{p}}$ going through link l, it must be that $\mathbf{f}$ is bandwidth-bottlenecked (see definition before Theorem 3).

D.2 Other results

In the discussion after Theorem 3, two results are stated without proof: the max-min fair rate allocation is bandwidth-bottlenecked and the adaptive waterfiller converges when it finds a bandwidth-bottlenecked rate allocation. Here, we provide their proofs in the form of the two following lemmas:

Lemma 1. *If $\mathbf{f}$ is a max-min fair rate allocation then it must be bandwidth-bottlenecked.*

Proof. Suppose that this is not true and a max-min rate allocation is *not* bandwidth-bottlenecked. This must mean that for some subflow f_k^p bottlenecked on link l, there is another non-zero subflow $f_j^{\hat{p}}$ going through that link and $f_j > f_k$. This implies that we can increase the subflow f_k^p at the expense of $f_j^{\hat{p}}$. Ultimately, this increases the allocation of f_k without reducing the allocation of any other equal or smaller allocation (only reducing the allocation of f_j, which was larger to start with). We arrived at a contradiction since this violates the definition of max-min fair allocation. □

Lemma 2. *Every bandwidth-bottlenecked rate allocation $\mathbf{f}$ is a fixed point of the adaptive waterfiller algorithm.*

Proof. Assume that $\mathbf{f}$ is bandwidth-bottlenecked and we use these flows (and subflows) to construct weights $\theta_k^p = f_k^p / f_k$. Let us denote by $\tilde{\mathbf{f}}$ the solution of solving the weighted waterfilling with those weights. We want to show that $\mathbf{f} = \tilde{\mathbf{f}}$. Notice that the following must hold for a subflow f_k^p bottlenecked at link l:

$$\frac{f_k^p}{\theta_k^p} = \frac{f_k^p}{f_k^p} f_k = f_k \geq f_j = f_j \frac{f_j^{\hat{p}}}{f_j^{\hat{p}}} = \frac{f_j^{\hat{p}}}{\theta_j^{\hat{p}}}, \tag{11}$$

where the inequality follows from the definition of bandwidth bottleneck (prior to Theorem 3) and the equality after that one assumes that $f_j^{\hat{p}}$ is a non-zero subflow also going through link l. Hence, we have established that for every f_k^p bottlenecked at link l, it must hold that $\frac{f_k^p}{\theta_k^p} \geq \frac{f_j^{\hat{p}}}{\theta_j^{\hat{p}}}$ for all non-zero flows $f_j^{\hat{p}}$ going through that link. This implies that $\mathbf{f}$ is a solution to the weighted waterfilling problem. However, we denoted by $\tilde{\mathbf{f}}$ the solution to this problem. From uniqueness of the weighted waterfilling solution, it must be that $\mathbf{f} = \tilde{\mathbf{f}}$. □

E Equi-depth binner formulation

In this section, we present two variants of Equi-depth binner — one where boundaries are elastic but the demands get allocation from exactly one bin, and the other where the

bin boundaries are fixed but the demands are allowed to get allocation from multiple bins.

Equi-depth binner with elastic bin boundaries. In this variant, we use the output of AdaptiveWaterfiller to sort demands by their estimated max-min fair rates. We then divide the demands from smallest to largest into equal-sized bins (D_b), each assigned to one specific bin. The order of bins is maintained using bin boundaries ℓ_b, which are determined by the optimization. During the allocation process, we prioritize bins with smaller demands, following a similar linearization technique described in §3.1:

$$\begin{aligned}
&\mathsf{ElasticBoundaryEquiBinning}(\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq \qquad (12)\\
&\quad \arg\max_{\mathbf{f},\ell} \sum_{\text{bins } b} \sum_{k\in\mathcal{D}_b} \epsilon^{b-1} f_k \\
&\text{s.t.} \quad f_k < \ell_b + s_b, && \forall b < N_\beta, \forall k \in \mathcal{D}_b \\
&\quad f_k \geq \ell_{b-1}, && \forall b > 1, \forall k \in \mathcal{D}_b \\
&\quad \ell_b \geq 0, && \forall b \\
&\quad \mathbf{f} \in \mathsf{FeasibleAlloc}(\mathcal{E},\mathcal{D},\mathcal{P}).
\end{aligned}$$

where N_β is the number of bins, D_b denotes the set of demands in bin b, ℓ_b shows the boundary of bin b (determined by the optimization), and s_b is the slack in quantization boundary of bin b (input to the optimization).

Equi-depth binner with multi-bin allocations. In this variant, we use the output of AdaptiveWaterfiller to compute the bin boundaries ℓ_b that result in roughly the same number of demands per bin. Then, we reuse the Geometric Binner's formulation from Eqn. 4 but with the estimated bin boundaries instead of geometrically increasing sizes:

$$\begin{aligned}
&\mathsf{MultiBinEquiBinning}(\mathcal{E},\mathcal{D},\mathcal{P},\{\ell_b\}) \triangleq \qquad (13)\\
&\quad \arg\max_{\mathbf{f}} \sum_{k\in\mathcal{D}} \sum_{\text{bins } b} \epsilon^{b-1} f_{kb} \\
&\text{s.t.} \quad f_k = \sum_{\text{bins } b} f_{kb}, && \forall k \in \mathcal{D} \\
&\quad f_{k1} \leq \ell_1, && \forall k \in \mathcal{D} \\
&\quad f_{kb} \leq \ell_b - \ell_{b-1}, && \forall b > 1, \forall k \in \mathcal{D} \\
&\quad \mathbf{f} \in \mathsf{FeasibleAlloc}(\mathcal{E},\mathcal{D},\mathcal{P}).
\end{aligned}$$

F Expected Run-time Benefit of GB and EB

Solving a linear program (with #constraints = Ω(#variables) – holds for resource allocation problems such as TE and CS) has worst-case time complexity of $O(\nu^a)$ where $a \approx 2.373$ [15] and ν is the number of variables in the optimization.

One can argue that simply solving a single optimization (as in the case of EB and GB) does not guarantee lower run-times compared to solving multiple optimizations (*e.g.,* SWAN). Solving multiple optimizations adds a multiplicative term to the time complexity. However, a naive single-shot optimization may use too many additional variables and ends up being slower. In this part, we theoretically analyze the expected run-time benefit of GB and EB. We show that the speed up of Soroush's optimization-based methods is due to their carefully designed approaches that only require a small number of additional variables compared to each optimization in the multi-shot variant.

SWAN uses 1 variable per demand per path to demonstrate the allocation from an specific path ($\nu = PK$ where P is the number of paths and K is the number of demands). Therefore, if SWAN needs N_β^S iterations, its worst-case complexity is $O(N_\beta^S P^a K^a)$.

GB needs 1 extra variable per demand per bin to measure the allocation from each bin ($\nu = (N_\beta^G + P)K$ – note that the number of bins in GB (N_β^G) is the same as the number of iterations in SWAN (N_β^S)). This leads to a worst-case complexity of $O((N_\beta^G + P)^a K^a)$. Compared to SWAN, the run-time saving of GB is proportional to $N_\beta \left[1 + \frac{N_\beta}{P}\right]^{-a}$. For $P = 16$ paths and $N_\beta = 8$ bins, we expect GB to be $\sim 3.06\times$ faster.

Our analysis of GB is only valid when the number of bins is small. When there are many bins, GB's allocation may have too many zero variables. For example, if we have 128 bins, and a demand only needs allocation from the first bin, the remaining 127 bin variables will be zero. Existing solvers such as Gurobi [27] exploit this sparsity to improve their runtime.

EB[15] only uses 1 extra variable per bin to show the bin boundaries ($\nu = N_\beta^E + PK$). Therefore, its worst-case complexity is $O((N_\beta^E + PK)^a)$. Compared to SWAN, EB has a run-time saving proportional to $N_\beta^S \left[1 + \frac{N_\beta^E}{PK}\right]^{-a}$. Since the number of demands is usually substantially larger than the number of bins N_β^E, we can approximate the run-time saving by N_β^S. For $N_\beta^S = 8$, we expect EB to be $\sim 8\times$ faster.

Although theoretical analysis can help us understand the speedup of Soroush, solvers can typically perform better than their worst-case and can take advantage of the structure of the optimization (*e.g.,* sparsity). In §4, we empirically show that the speedup of GB compared to SWAN is larger than what our theoretical analysis predicted. In fact, it is even faster than EB. Similarly, we found that the speed up of EB compared to SWAN is only slightly less than the theoretical analysis.

G Extended Evaluation

In this section, we provide both additional experiment details as well as an extended evaluation of Soroush.

[15] We analyze the variant with elastic bin boundaries. The other multi-bin variant has the same complexity as GB since it uses the same optimization.

Model	Task	App/Dataset	#Batch sizes
ResNet-18 [29,41]	Image Classification	CIFAR-10 [40]	16, 32, 64, 128, 256
ResNet-50 [2,29]	Image Classification	ImageNet [18]	16, 32, 64, 128
CycleGAN [44,76]	Image-to-Image Translation	monet2photo [76]	1
LSTM [63]	Language Modeling	Wikitext2 [48]	5, 10, 20, 40, 80
Transformer [31,69]	Language Translation	Multi30k [20] (de-en)	16, 32, 64, 128, 256
A3C [26,49]	Deep RL	Pong	4
Autoencoder [51]	Recommendation	ML-20M [28]	512, 1024, 2048, 4096, 8192

TABLE A.2: Type of jobs used for the evaluation of Soroush. We use Gavel's job generator [56].

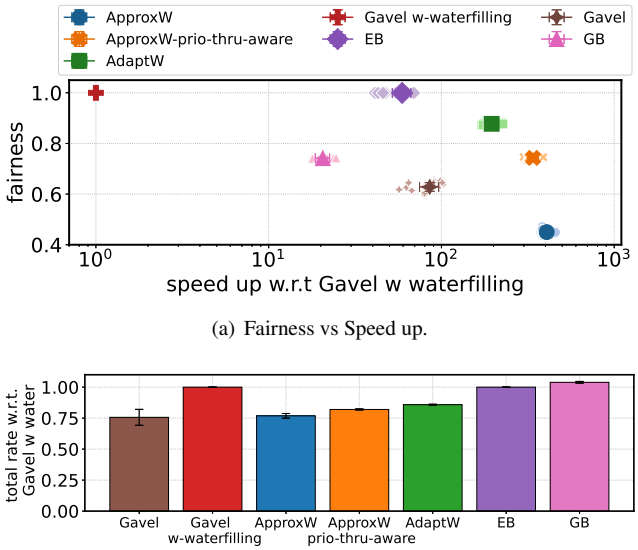
G.1 Tuning benchmarks for performance

- We warm start SWAN's and Danna's optimizations for iterations > 1 to reduce the run-times. We further tune Gurobi's solver parameters using 5% of the traffic demands to achieve the best run-time.
- Our Danna's implementation is that of Figure 2 in [17] (*i.e.,* binary and linear search): we found this algorithm is faster than the other proposed by the same work (*i.e.,* binary then linear search in Figure 4) as it can find and eliminate more demand-constrained flows.
- Our modified K-waterfilling algorithm uses K=1 which is the fastest and most parallelizable version of the K-waterfilling [36].
- For cluster scheduling (CS), we changed Soroush's implementation to use CVXPY [19] to match Gavel's implementation and ensure fair run-time comparisons.

G.2 Evaluation on CS

Experiment details. We consider 3 different types of GPUs (V100, P100, K80) and base the number of GPUs on the number of jobs. We randomly select jobs from a set of 26 different job types available in Gavel (see Table A.2). Each job has a specific priority and requires a certain number of workers. We determine the number of workers by randomly sampling from the distribution obtained from the Microsoft public trace [3] – 70% of jobs need a single worker, 25% need between 2 and 4, and the remaining 5% need 8. We also sample job priorities uniformly from $\{1, 2, 4, 8\}$. To compute the weights in the weighted max-min fair objective, we use the job throughput estimations from Gavel.

Results. In Fig. A.2, we compare Soroush to two variants of Gavel (with and without waterfilling) in 40 different scenarios. The number of competing jobs in each scenario is selected from the set $\{1024, 2048, 4096, 8192\}$ – 10 scenarios from each. Following Gavel, we set the number of each type of GPU to one-fourth of the total number of jobs and generate each job using the methodology explained above.



(a) Fairness vs Speed up.

(b) Efficiency (total effective rate)

FIGURE A.2: Soroush **empirically Pareto-dominates Gavel**. These results are on 40 different scenarios with varying number of jobs and GPUs.

The results are in line with our observations from WAN-TE; (a) Gavel with waterfilling is an optimal max-min fair algorithm, but it is more than $20\times$ slower than other methods, (b) Gavel is $\sim 100\times$ faster than the Gavel w waterfilling but at the cost of $\sim 40\%$ drop in fairness and $\sim 15\%$ drop in efficiency, and (c) Soroush's algorithms empirically Pareto-dominate both Gavel and Gavel w waterfilling.

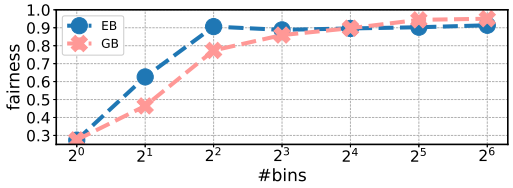
We also observe that GB is slower than the rest of the methods except (Gavel with waterfilling), but it provides worst-case per-demand fairness guarantees.

G.3 Evaluation of POP

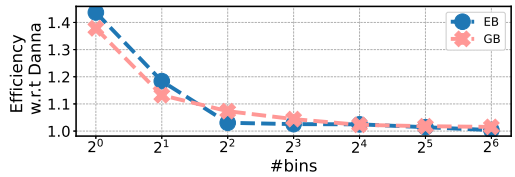
POP [55] is a decomposition technique used to scale granular resource allocations. It involves dividing demands uniformly at random into partitions, assigning an equal share of each resource to each partition, and then, solving the resource allocation for each partition in parallel. This procedure is called *resource splitting*.

For large demands, POP incorporates an additional method called *client splitting*, where demands are divided among multiple partitions. [55] recommends using client splitting for Poission traffic distribution, as this can improve resource utilization. However, it is unnecessary for the other distributions such as Gravity.

POP focuses on objectives such as maximizing utilization or maximizing the minimum allocation, a different objective than max-min fairness. To assess the impact of POP on max-min fairness, we adapt POP to work with both Soroush and SWAN; We use the same procedure for partitioning the problem (resource splitting and client splitting as needed). We then allocate resources in each partition using a max-min fair



(a) Impact of the number of bins on the fairness



(b) Impact of the number of bins on the efficiency

FIGURE A.3: Impact of number of bins on fairness and efficiency of GB and EB. These results are on the Cogentco topology and Poisson traffic distribution (scale factor = 64).

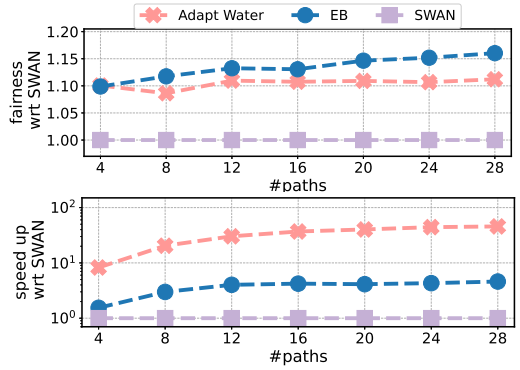


FIGURE A.4: Impact of number of paths in TE. These results are on the Cogentco topology and Poisson Traffic distribution (scale factor = 64).

solver such as SWAN or Soroush.

Theoretical Guarantee. POP results in losing all the theoretical guarantees (α-approximation for Soroush and SWAN). In fact, [53] shows a substantial worst-case optimality gap for POP. However, Soroush is faster than SWAN because of its single optimization reformulation, while maintaining the same theoretical guarantees.

Empirical Evaluation. In Fig. A.6, we evaluate the performance of POP when applied to Soroush (specifically, GB) and SWAN for two different topologies, two load factors and two traffic distributions.

We find that the performance of POP depends on the traffic distribution whereas Soroush maintains the same level of fairness in all cases while being up to 15× faster than SWAN. For distributions with granular demands such as Gravity, POP speeds up both SWAN and Soroush with only a minor drop in fairness. However, using POP causes significant fairness degradation – more than 10% to match the run-time of Soroush– for traffic distributions such as Poisson that are not granular and require client splitting to avoid resource underutilization. This unfairness is a result of per-partition max-min fairness in POP, which differs from global max-min fairness in Soroush or SWAN.

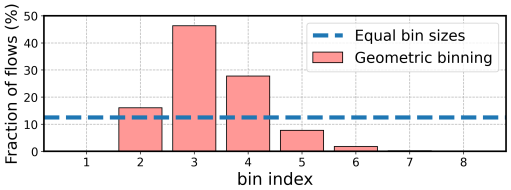
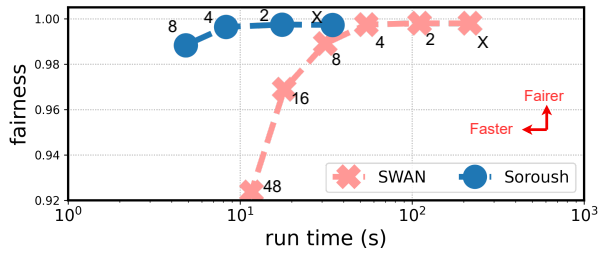
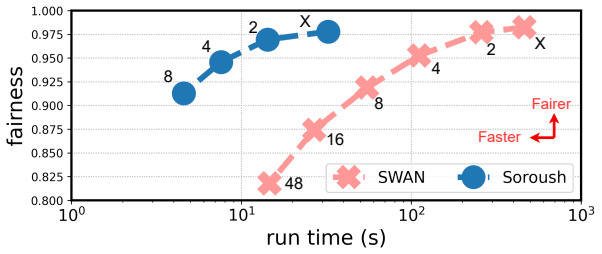


FIGURE A.5: Example of imbalanced bins in GB for the TE usecase.

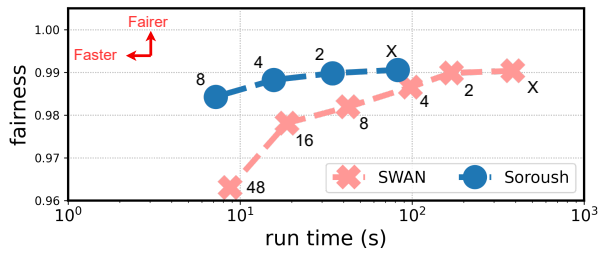
We also observe that applying POP to Soroush results in the same fairness level as SWAN for the same number of partitions. In each partition, Soroush is guaranteed to produce an allocation similar to SWAN (see §3.1), and therefore, the aggregated allocation is guaranteed to be the same. Since Soroush is faster in each partition, the overall run-time of Soroush + POP is substantially lower.



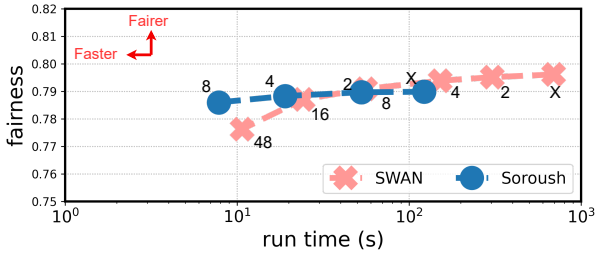
(a) Topo: Cogentco, Traffic: Poisson, Scale factor=16, w client splitting



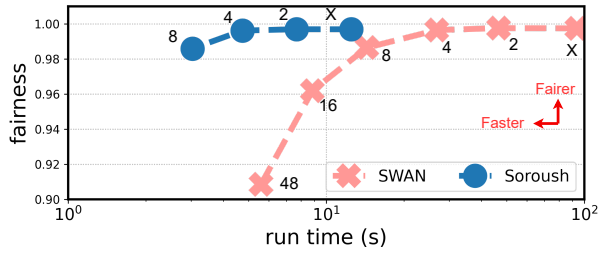
(b) Topo: Cogentco, Traffic: Poisson, Scale factor=64, w client splitting



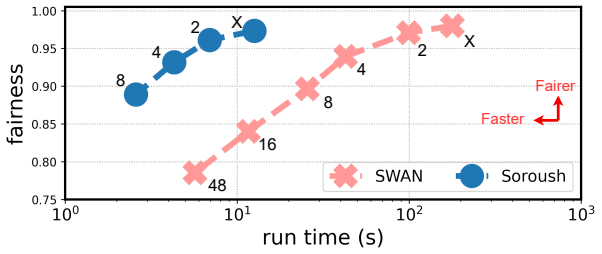
(c) Topo: Cogentco, Traffic: Gravity, Scale factor=16, no client splitting



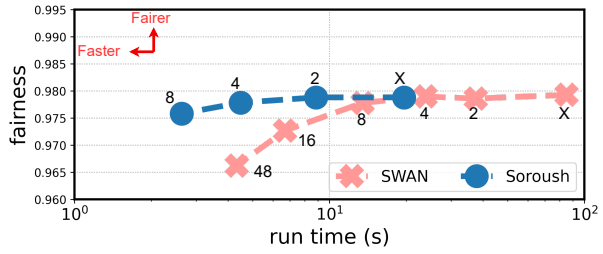
(d) Topo: Cogentco, Traffic: Gravity, Scale factor=64, no client splitting



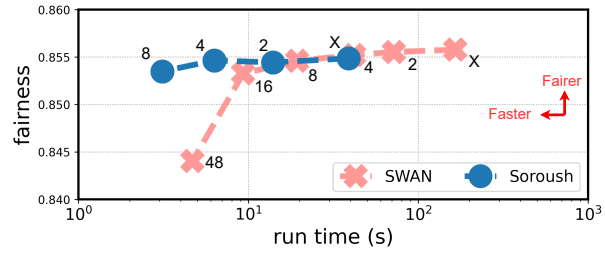
(e) Topo: GtsCe, Traffic: Poisson, Scale factor=16, with client splitting



(f) Topo: GtsCe, Traffic: Poisson, Scale factor=64, with client splitting



(g) Topo: GtsCe, Traffic: Gravity, Scale factor=16, no client splitting



(h) Topo: GtsCe, Traffic: Gravity, Scale factor=64, no client splitting

FIGURE A.6: Impact of POP [55]. POP is not designed for max-min fair allocation and can cause drop in fairness depending on the traffic distribution (both on Soroush and SWAN). In contrast, Soroush achieves lower runtime compared to SWAN while maintaining the same level of fairness and theoretical guarantees. The figure reports the average over 3 randomly generated demands. "X" indicates that POP is not used, and the "Numbers" represent the number of POP partitions.

Cloud-LoRa: Enabling Cloud Radio Access LoRa Networks Using Reinforcement Learning Based Bandwidth-Adaptive Compression

Muhammad Osama Shahid[*1] Daniel Koch[*1] Jayaram Raghuram[1]
Bhuvana Krishnaswamy[1] Krishna Chintalapudi[2] Suman Banerjee[1]
[1]*University of Wisconsin-Madison* [2]*Microsoft Research* [*]Co-Primary authors

Abstract

The Cloud Radio Access Network (CRAN) architecture has been proposed as a way of addressing the network throughput and scalability challenges of large-scale LoRa networks. CRANs can improve network throughput by coherently combining signals, and scale to multiple channels by implementing the receivers in the cloud. However, in remote LoRa deployments, a CRAN's demand for high-backhaul bandwidths can be challenging to meet. Therefore, bandwidth-aware compression of LoRa samples is needed to reap the benefits of CRANs. We introduce *Cloud-LoRa*, the first practical CRAN for LoRa, that can detect sub-noise LoRa signals and perform bandwidth-adaptive compression. To the best of our knowledge, this is the first demonstration of CRAN for LoRa operating in real-time. We deploy Cloud-LoRa in an agricultural field over multiple days with USRP as the gateway. A cellular backhaul hotspot is then used to stream the compressed samples to a Microsoft Azure server. We demonstrate SNR gains of over 6 dB using joint multi-gateway decoding and over 2x throughput improvement using state-of-the-art receivers, enabled by CRAN in real-world deployments.

1 Introduction

LoRa [1] is one of the most popular long-range, low-power wide area network (LPWAN) technology for IoT applications such as smart city [2, 3], smart agriculture [4, 5]. Operating in the license-free ISM band, anyone can independently deploy a LoRa LPWAN, where IoT devices make use of off-the-shelf LoRa radios to transmit messages to a gateway. Like most common wireless systems, a LoRa gateway performs all physical layer processing such as receiving and decoding transmissions. Often, gateways relay these decoded packets to the cloud to enable cloud-based IoT applications.

The last decade has seen the emergence of a new wireless architecture – a Cloud Radio Access Network (CRAN), where the gateway continuously streams the raw received digitized radio signals (I/Q samples) to a virtual gateway in the cloud over a back-haul link for physical-layer processing (Fig. 1). CRANs, applied to LoRa, offer three disruptive advantages.

- *Joint Multi-Gateway Packet Decoding:* Economic viability of a LoRa deployment is often dictated by its range – a long range necessitates fewer base-stations thereby reducing capital and operating costs. Weak signals from multiple gateways, when combined constructively, boost the signal-to-noise ratio (SNR) [6, 7], and in turn, extend the range. These approaches require centralization to jointly process the raw radio signals from multiple base-stations.
- *Rapid Physical-Layer Innovation to Boost Capacity:* A major challenge in dense areas is capacity scaling, as mushrooming uncoordinated LoRa deployments lead to increased collisions [8]. Recently, several promising PHY-layer demodulation techniques [9–16] have shown an order of magnitude improvement in capacity. CRAN enables their rapid deployment and A/B testing in the field, as virtual software receivers can be deployed in the cloud.
- *Elastic Scaling to Multiple Channels:* As capacity needs increase, traditional LoRa gateways need to be physically upgraded to high-end gateways with parallel receiver chains baked into their ASIC. CRAN gateways capture a wide spectrum and allow for the potential to dynamically scale the number of channels in the cloud depending on demand.

While researchers have demonstrated the potential of CRANs to enable the deployment of new physical layer techniques [6, 17–19], to the best of our knowledge, there is no end-to-end implementation of a LoRa CRAN till date where gateways continuously stream radio samples to virtual receivers in the cloud to be decoded in real-time. *The key contribution of this paper is* **Cloud-LoRa***, the first end-to-end practically-deployable LoRa CRAN for urban and rural deployments.* We demonstrate that Cloud-LoRa allows for the practical deployment of real-time joint multi-gateway packet decoding in the cloud with techniques such as Charm [6] and achieves **an SNR gain of over 6 dB**. This gain could translate to a doubling of the range. We also show that rapid deployment of novel decoding techniques such as [9, 10, 12] offer a **2X improvement in throughput** as well as scaling to multiple

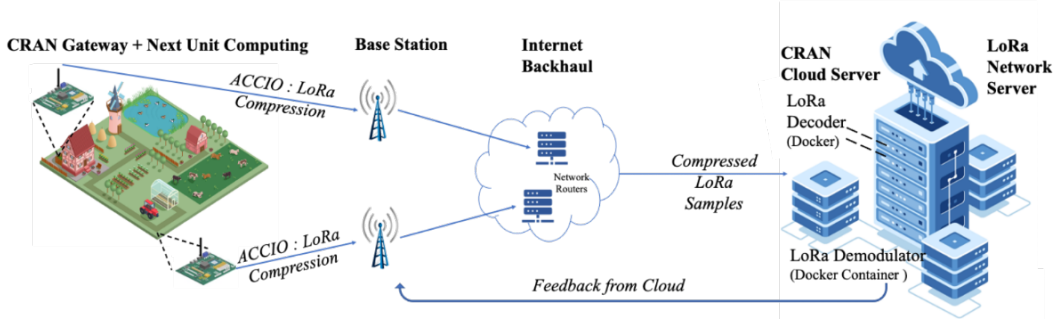


Fig. 1: An Illustration of CRAN and Cloud-LoRa

channels using per-channel virtual receivers in the cloud.

Cloud-LoRa comprises three key components: i) a *CRAN gateway* that can be deployed on software-defined radios (e.g., USRP and a NUC), ii) *ACCIO*, an online reinforcement learning-based adaptive compression algorithm, and iii) a cloud gateway with user-defined receivers. *Cloud-LoRa allows researchers to deploy their own physical layer demodulators* as containers at the cloud gateway (Figure 1).

Extreme Back-Haul Bandwidth Gap in a LoRa CRAN A common challenge to every CRAN is the need to stream a high volume of raw signals (I/Q samples) to the cloud. Each 1 MHz of radio spectrum generates a continuous data stream at 64 Mbps[1] to the cloud. However, LoRa LPWANs are intended for low-cost deployments and often, the only backhaul available in several rural and remote deployments is a cellular link offering as low as 500 kbps [20–22]. Therefore, a viable LoRa CRAN must be capable of streaming samples even on such low-bandwidth links to the cloud.

Detecting Sub-Noise LoRa signals buried in noise. At such low backhaul bandwidths, channel activity detection to avoid streaming noise signals to a cloud server is an integral component of a CRAN gateway. A majority of the existing activity detection methods rely on observing signal strength above a fixed threshold [23]. Due to its spread-spectrum technology, LoRa signals are received at sub-zero dB SNR, *i.e.*, they are buried in noise and hence signal strength-based thresholding fails. Even recent works, such as SparSDR [24] that uses time-frequency analysis for activity detection cannot distinguish sub-noise LoRa from noise in real-time. SparSDR can detect sub-noise signals at the cost of significantly higher false positives, a tradeoff that defeats the purpose of activity detection. In this work, we develop an activity detection approach that detects sub-noise LoRa signals in real-time across multiple channels with relatively low false positives.

Adaptive Lossy Compression and Streaming. A key component for any CRAN is compressing real-time streaming of radio samples to the cloud. CRANs typically employ lossless compression techniques since lossy compression degrades the quality of the signal and adversely affects its decodability. Further, it is hard to predict in advance at the gateway how much "lossiness" will allow a specific part of the signal to be decoded, since decodability of the signal depends on several dynamic factors such as SNR of the received wireless signal, the specific demodulation being used in the cloud etc.

As we evaluate in Section 5, state-of-the-art lossless compression techniques provide up to 70% compression for LoRa signals, depending on the SNR. At this compression, a single 500 kHz wide LoRa channel with 10% channel activity will generate a 960 kbps stream – about twice greater than the capacity of a rural cellular backhaul link of 500 kbps.

To address this challenge, we propose *ACCIO*, an online reinforcement learning (RL)-based wavelet compression that is lossy and adaptive, to compress sub-noise LoRa transmissions. *ACCIO* receives rewards (feedback) from a cloud receiver for decoding successfully, and learns to employ the right level of compression based on the available backhaul bandwidth and SNR at the gateway. *ACCIO* uses TCP BBR [25] to reliably deliver the lossy-compressed radio signals to the cloud. The bandwidth estimates from BBR, signal SNR estimates, and application buffer levels are then used by ACCIO's RL agent to dynamically set the appropriate level of compression. The motivation behind using RL for adaptive compression is discussed in detail in Section 3.2.

Open Source, Deployable Implementation. Our implementation of the Cloud-LoRa gateway performs channelization, activity detection, and runs *ACCIO* in real-time. Its cloud gateway runs on Azure and *implements several recent LoRa demodulators* including CIC [12] and Charm [6]. To demonstrate its practical viability, we deploy and test Cloud-LoRa in an 8-channel CRAN in two scenarios: **1)** a rural outdoor setting with cellular backhaul, and **2)** an urban outdoor setting. We have open-sourced our framework[2] with well-defined APIs to plug-in new physical-layer demodulators, and allow scalability with the number of channels. We hope that Cloud-LoRa will encourage and enable future researchers to deploy and compare novel physical-layer demodulators. The major contributions of this paper are:

- We present Cloud-LoRa, the first practically-deployable end-to-end LoRa CRAN solution. Our current implementation streams and processes signals from up to 8 LoRa channels in real-time to the cloud. We hope that researchers will be

[1] Two 32-bits for each complex-valued sample at 1 Msamples/s

[2] https://github.com/UW-CONNECT/cloud-lora.git

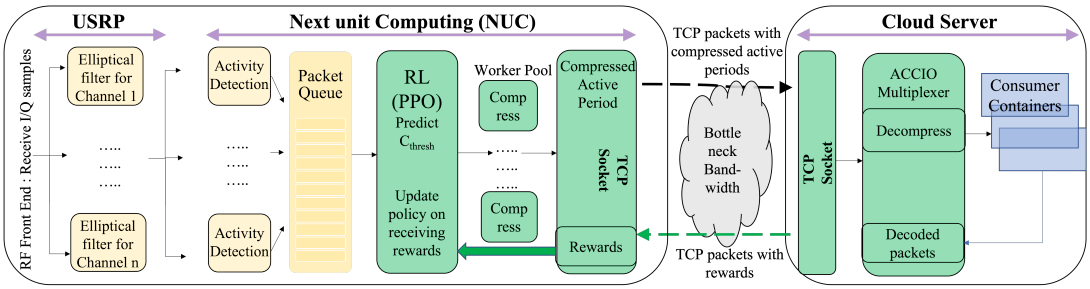


Fig. 2: Components of Cloud-LoRa : CRAN gateway (USRP) performs channelization, followed by activity detection at the NUC. ACCIO then compresses active periods using the RL agent and streams to the cloud server. The cloud server decodes the received packets and provides reward feedback to the ACCIO RL agent.

able to use Cloud-LoRa to deploy, test, and compare new physical layer demodulators in the field.

- We propose a novel LoRa activity detection approach that detects even sub-noise LoRa signals across multiple channels in real-time, without demodulating them.
- We propose ACCIO, a reinforcement learning-based adaptive compression technique that aims to maximize packet decodability in the cloud CRAN receiver. ACCIO also compresses the active LoRa signals to meet the available backhaul bandwidth and desired latency requirements.
- We provide an open-sourced implementation of Cloud-LoRa including CRAN gateway (on USRP), ACCIO, and several LoRa receivers as Dockerized containers.
- We demonstrate Cloud-LoRa through rural field deployments and by testing other recently published techniques [6, 12]. We show an SNR gain of over 6dB when signals from multiple gateways were jointly decoded in the cloud. We see a 2X boost in network throughput using novel physical-layer innovations deployed in the cloud. We process up to 8 LoRa channels in real-time, further improving throughput.

We attest that this work complies with the applicable ethical standards of our home institution.

2 Background and Motivation

LoRa Modulation-DeModulation: LoRa uses Chirp Spread Spectrum (CSS) modulation, a spread-spectrum technology that enables LoRa to operate at sub-zero dB SNR due to its resilience to noise and interference. The spreading factor (SF), which defines the number of bits per symbol, along with the RF bandwidth of a LoRa channel (BW) determine the symbol duration, datarate, energy consumption, and range of communication. As the SF increases, the range will also increase, but at the cost of reduced datarate. As SF and BW are predetermined for a transmitter, a LoRa receiver searches for preambles of a single SF and BW to detect the start of a packet. LoRa demodulates by dechirping the received signal, which enables it to receive sub-noise signals [26].

Variable Backhaul Bandwidths: While access to broadband connectivity has been expanding, available bandwidths still vary vastly across the country [22, 27]. FCC 2020 reports on broadband access determined that potentially over 50% of rural Americans lack broadband access [20, 28] of 25 Mbps download/3 Mbps upload speeds. Broadband speeds lower than 1 Mbps have been identified as a bottleneck for the adoption of precision agriculture [21, 29, 30]. Additionally, significant variability in data-rates can be expected over wireless links even in urban areas due to changes in load, environment, service providers, among other factors [31, 32].

Recent works on CRAN-based LoRa: The benefits of CRAN-based LoRa have been identified in recent works such as Charm [6], Nephalai [19], and OPR [7]. Charm and OPR demonstrate that the range of communication can be improved by coherently combining signals using CRANs. Nephalai proposes a static compression technique using compressed sensing [33] to stream multiple LoRa channels to the cloud to improve the network throughput. More generally, physical-layer-agnostic CRAN for IoT has been proposed in works such as SparSDR [24] and CharIoT [34]. While these existing works have identified and demonstrated the benefits of CRAN, dynamic compression that adapts to the signal characteristics and meets the variable backhaul bandwidths of LoRa gateways remains an open challenge. We propose an RL-based adaptive compression to address this challenge.

3 Proposed System - Cloud LoRa

Towards a practical, real-time LoRa CRAN, our Cloud-LoRa framework consists of three components, illustrated in Fig. 2:

1. CRAN gateway : a software defined radio (SDR) gateway that continuously streams samples from a wideband spectrum. The gateway performs channelization to filter LoRa channels and detects activity in each individual channel. The activity detection module at the gateway is designed to detect even sub-noise LoRa signals, and stream only those signals corresponding to active LoRa transmissions.
2. ACCIO (green blocks in Figure 2) : the active LoRa transmissions need further compression. We propose ACCIO, an online RL-based compression algorithm that adaptively

predicts the compression threshold for each active period. ACCIO's goal is to maximize the total packets decoded in the cloud gateway, while meeting the backhaul-bandwidth and latency constraints.

3. Cloud Server : we implement standard LoRa as well as user-defined LoRa receivers in a Microsoft Azure cloud server as Docker containers. The cloud server reconstructs the compressed samples, which are then demodulated and decoded. The number of packets decoded per active period is sent as reward feedback to ACCIO's RL agent.

3.1 CRAN Gateway

LoRa transmitters typically have a low duty-cycle to conserve their battery. As a result, a majority of the samples captured at a CRAN gateway are noise. Since it is wasteful to transport noise to the cloud, activity detection is critical in CRAN.

Multi-Channel Filter. The gateway performs channelization to filter an individual channel from the wideband spectrum before detecting activity. We first convert each channel to baseband and then apply a 4th-order IIR Elliptical filter (Figure. 2) [35]. This light-weight filter both suppresses other channels by 100 dB and offers a small transition band, ensuring minimal cross-channel leakage and real-time operation.

Sub-Noise LoRa Activity Detection. Activity detection is typically performed using energy-based approaches such as carrier sensing, which fail to distinguish low-SNR LoRa signals from noise [24]. At received SNRs below $0\,dB$, the energy of LoRa samples becomes comparable to that of noise.

A standard LoRa receiver performs dechirping followed by Fast Fourier Transform (FFT) to accumulate energy in a single frequency, in-turn distinguishing noise from LoRa samples [26, 36]. However, dechirping is specific to a spreading factor (SF). Current multi-channel LoRa gateways have a dedicated RF front-end for each SF. A naive sub-noise LoRa activity detection is to dechirp the received samples with each possible SF (7 through 12), and then perform energy-based detection. This is computationally intensive and requires 6× more multiplications than a single demodulator. Therefore, an SF-agnostic activity detection is desirable for LoRa CRAN.

We propose an SF-agnostic LoRa activity detection algorithm to detect sub-noise LoRa signals at the CRAN gateway. Our activity detection leverages two properties of LoRa: **1)**. Two LoRa signals of different SFs are Pseudo-orthogonal (more in Appendix.A.1). *e.g.*, dechirping an SF7 signal with SF8 downchirp would result in pseudo-random noise. **2)** For a given bandwidth, the downchirp of one SF is a time-scaled function of the downchirp of another SF. Based on these properties, we design ***superDC****, a custom downchirp, that can dechirp and hence detect the activity of more than one SF by superimposing downchirps of multiple SFs.*

The CRAN gateway continuously dechirps an array of samples with the *superDC*, followed by an FFT. *An active LoRa transmission results in a sharp peak above the noise floor in the FFT, triggering activity detection at the gateway.* For instance, a superDC that superimposes SF7, SF8, and SF9 downchirps detects an activity *only if* the active signals are in SF7 through SF9. Since an SF9 downchirp is 4× as long as that of SF7, and 2× as that of SF8, we construct the superDC by superposing one SF9 downchirp with two consecutive SF8 and four consecutive SF7 downchirps (more in Appendix.A).

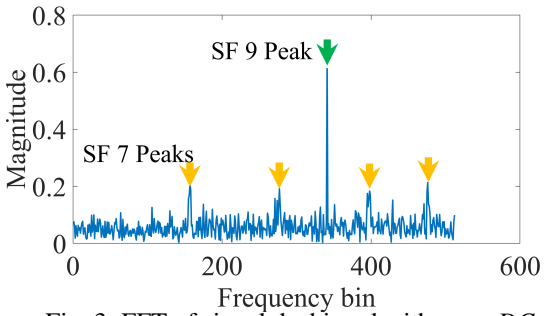


Fig. 3: FFT of signal dechirped with *superDC*

Figure 3 shows the FFT of a signal containing SF7 and SF9 chirps, each with 10 dB SNR, dechirped with this *superDC*. We observe four SF7 peaks and one SF9 peak since the superDC includes four SF7 and one SF9 downchirps.

The active LoRa signals that can be detected by the *superDC* are determined by the superposed downchirps, which in turn determine the length of the superDC. A superDC to detect all SFs (7 to 12) must accommodate at least one SF12, two SF11, four SF10, eight SF9, sixteen SF8, and thirty-two SF7 downchirps. In this case, the FFT peak-gain (ratio of the maximum peak in an FFT window to its noise floor [3]) of an SF7 symbol is 32 times lesser than that of an SF12 symbol. Hence, low-SNR SF7 symbols could go undetected. On the other hand, using narrower windows would lead to missing higher SF symbols. To combat this challenge, we design two superDCs: $superDC_{low}$ to detect symbols with SF 7 through 9 and $superDC_{high}$ to detect symbols with SF 10 through 12. The former can be defined in time domain as

$$superDC_{low}(t) = \sum_{m=0}^{3} C(t - mT_{SF7}, 7) + \sum_{m=0}^{1} C(t - mT_{SF8}, 8) + C(t, 9), \quad 0 \le t \le T_{SF9}, \quad (1)$$

where $T_{SF} = \frac{2^{SF}}{BW}$ and $C(t,i)$ is the downchirp of SFi. We can similarly define $superDC_{high}$. By choosing two groups of SF, we minimize the impact of excessive window sizes, while still maintaining SF-agnostic detection. The received samples are dechirped using both $superDC_{low}$ and $superDC_{high}$ to detect activity. As we demonstrate in our evaluation, the two groups of superDC signals can detect all LoRa activity in real-time.

As the SNR of the received samples decreases, the peak-gain of the received signal dechirped with a superDC signal

[3] We maintain a running estimate of the noise floor to confer resilience against temporal variations.

also decreases. This could result in spurious samples triggering activity detection. To reduce such false positives, the gateway signals activity in the channel whenever a minimum of 3 consecutive peak-gains, which correspond to 12 symbols of the smallest SF (*i.e.*, SF7 or SF10), are observed to be higher than a threshold (average peak gain for noise signal). We push such an active period's I/Q samples to the *Packet Queue* for compression and transport to the cloud server.

In summary, dechirping received samples using our custom-designed superDCs ($\text{superDC}_{\text{low}}$ and $\text{superDC}_{\text{high}}$) provides the processing gain needed to detect LoRa activity even when the received signals are much below zero dB SNR. The proposed activity detection is agnostic to the SF of the transmitter, making it a general-purpose front-end, with only $2\times$ multiplications of a single LoRa demodulator, as opposed to the state-of-the-art gateways that incur $6\times$ multiplications.

3.2 ACCIO : Reinforcement Learning-based Adaptive Compression

While activity detection reduces the volume of noise samples streamed, when the network traffic increases, even active period samples can be too high for some backhaul bandwidths to support. Even with perfect activity detection, the required bandwidths for 64 channels with $\approx 10\%$ channel occupancy is over 200 Mbps. Nephalai [19], a recent work on LoRa CRAN, utilizes downsampling and compressive sensing for compression. But, novel demodulators leverage oversampling to resolve packet collisions [9, 11–13], and hence compression without downsampling is necessary. Moreover, LoRa's chirp spread-spectrum (CSS) modulation renders dictionary-based lossless compression methods ineffective. We propose ACCIO , a light-weight RL-based adaptive compression algorithm that works on top of a Discrete Wavelet Transform (DWT)-based lossy compression scheme in order to maximize the number of packets decoded at the cloud server, without exceeding the backhaul bandwidth and latency constraints.

Lossy Active-Period Compression. We propose to utilize the Discrete Wavelet Transform (DWT) [37] as our lossy compression scheme for oversampled active LoRa signals. DWT, being a multi-resolution time-frequency analysis, is a suitable compression tool for CSS modulation which uses both time and frequency for modulation. Each DWT coefficient represents the energy of the received signal corresponding to a particular frequency (level) and time (shift). Also, DWT's linear complexity ($O(N)$, where N is the length of the signal) makes it light-weight, allowing it to compress in real-time. More background on DWT is presented in Appendix B.

We compress active LoRa signals by first applying DWT to the signals, and then retaining only those DWT coefficients with magnitude greater than a threshold C_{thresh}. The compressed signals can be reconstructed if sufficient energy is retained in the DWT coefficients. As we increase C_{thresh}, we retain fewer coefficients and compress more; but the energy of the signal (coefficients) retained will decrease, leading to lossy compression. Determining the optimum threshold that ensures reliable reconstruction of signals at the receiver (cloud server), while maintaining a compression to match the network bandwidth is a challenging problem.

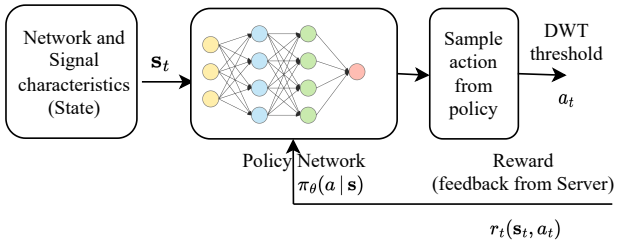
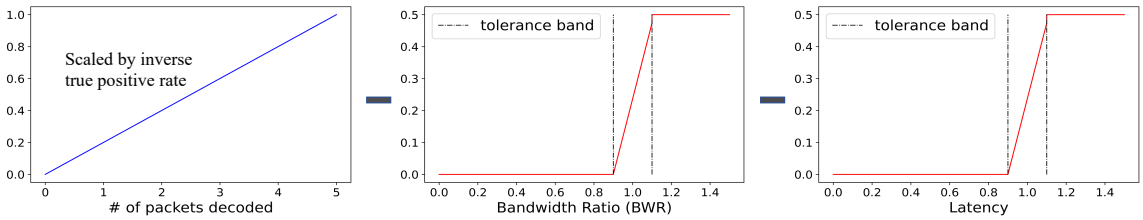


Fig. 4: Overview of the RL algorithm of ACCIO

Bandwidth-Adaptive Compression. The optimal compression threshold of DWT coefficients has a non-linear dependence on three factors: i) the SNR of samples at the gateway, ii) the backhaul network conditions, and iii) the LoRa demodulator at the cloud. Current compression approaches are static [19, 24] and do not adapt to these factors. Therefore, an adaptive compression that can learn this dependence is required. While DWT provides an effective way to compress based on the time-frequency characteristics of the signals, we still need a method to determine the appropriate amount of compression based on the backhaul network conditions (which are usually non-stationary). To address this, we propose an RL-based (DWT) threshold prediction algorithm that adaptively selects the level of compression for any given active period, with the goal of maximizing the cumulative number of packets decoded at the cloud server.

Choice of RL.

While supervised learning methods (particularly DNNs) are effective at modeling non-linear dependencies between the input and the target, they are designed for an offline scenario where the data distribution is not changing over time [38]. This makes them less effective when the network traffic and conditions (e.g., duty cycle, SNR, and the number of channels) are non-stationary, and also when there is lack of visibility into demodulator design used at the cloud receiver. Reinforcement learning is particularly well suited for this scenario since by design it learns an agent or policy in an unknown, dynamic environment such that the agent can perform a sequence of actions with the goal of maximizing its cumulative reward feedback [39]. In our setting, the agent performs the task of adaptively selecting the DWT threshold for each active period (based on various signal and network characteristics), with the goal of maximizing the total number of packets decoded at the cloud receiver over a transmission interval. Moreover, our choice of an online policy gradient-based RL algorithm does *not require pretraining* on a large collection of offline data from the target (or a similar) environment. It can start learning the compression policy from scratch based on data



(a) Reward for packet decoding (b) Penalty for bandwidth ratio (c) Penalty for latency

Fig. 5: Components of the reward function : (a) $u_{dec}(\mathbf{s},a)$, (b) $u_{band}(\mathbf{s},a)$, (c) $u_{lat}(\mathbf{s},a)$.

from the target environment, and still learn a good stable policy (see § 5.6). Hence, it can be applied to a wide variety of applications and deployments.

An RL agent at the gateway learns to take sequential actions based on the current state of the environment such that its cumulative-discounted rewards received from the environment over multiple time-episodes is maximized (more background in Appen. D). Crucial to the success of an RL agent are the design of the state variables and the reward function. Based on a careful study and evaluation, *we propose suitable state variables and a reward function that enable the agent to adaptively predict the (DWT) threshold in order to achieve high decodability under varying signal and backhaul conditions*. A unique challenge to the design of the reward function in this setting is the lack of perfect ground truth for providing the reward signal. In a typical RL system, the environment would have ground truth for providing the reward. This occurs because some of the transmitted active periods could be false positives (without actual packets), and it is unknown to the server whether a received active period is a false positive or not. We address this in the design of our reward function.

We focus on the Policy Gradient class of RL methods [40, 41], whose goal is to directly learn an optimal policy function that is parameterized by a neural network. Specifically, we use the Proximal Policy Optimization (PPO) method with clipped objective [42] for online training of the RL agent. PPO is widely adopted as a state-of-the-art online policy-gradient method due to its better computational and sample efficiency, and stable policy function updates. We next discuss the components of our RL algorithm.

Action. The action of the RL agent a corresponds to selecting the DWT threshold C_{thresh}. In principle, the threshold is continuous-valued. However, we simplify the design by choosing a discrete set of eight threshold levels. Specifically, if $a \in \{0,1,\cdots,7\}$ is the action taken, then $C_{thresh} = 5\,d_{avg}\,a$, where d_{avg} is the average DWT coefficient value over the current active period. (The factor 5 is chosen to cover a wider range of thresholds.). Our action space is discrete and the policy function $\pi_{\boldsymbol{\theta}}(a\,|\,\mathbf{s})$ will be a conditional probability mass function that sums to 1 over all the actions.

States. We provide the RL agent with a state vector $\mathbf{s}$ that broadly consists of the network (pipeline) characteristics and the signal characteristics from the recent active periods. The state variables based on the network are functions of the estimated bandwidth and the current fraction of the packet buffer that is filled. The state variables based on the signal characteristics include the normalized packet length, the ratio of signal-to-noise magnitude in the time domain, and a histogram of the peak-gain values of the active period. The full list of states with a description is given in Table 1.

State variable	Description
norm_pkt_len	(AP size in samples) / (sampling rate); AP - Active period
mag_time	(AP magnitude in the time domain) / (noise magnitude)
dcmp_avg	Average value of the first-level decomposition DWT coefficients
PG_hist	A 4-bin histogram of the peak gain values
BW_obs	log(BW_btl / 50,000,000); BW_btl - estimated bottleneck bandwidth
BW_ratio	(#bits sent to cloud in the last 10s) / $\int BW_btl$ over the last 10s
BW_ratio_5	(#bits sent to cloud over last 5 AP) / $\int BW_btl$ over last 5 AP
BW_ratio_10	(#bits sent to cloud over last 10 AP) / $\int BW_btl$ over last 5 AP
buffer_size	Fraction of the packet queue filled with AP

Table 1: State variables used by ACCIO . The first four states capture LoRa characteristics and the rest capture the network. $\int BW_btl$ is computed as a Riemann sum over time period.

Reward Function. A well-designed reward function is a crucial part of the RL design. As discussed earlier, the goal of ACCIO is to compress the LoRa signals in the active periods such that it maximizes the number of packets correctly decoded, while also meeting the bandwidth and latency constraints. We design the reward function as a sum of four terms: **i)** a positive reward term $u_{dec}(\mathbf{s},a)$ that is a weighted count of the number of packets decoded correctly (Fig. 5 a); **ii)** a negative penalty term $u_{band}(\mathbf{s},a)$ that strongly discourages the bandwidth utilized from getting very close to the available bandwidth (Fig. 5 b); **iii)** a negative penalty term $u_{lat}(\mathbf{s},a)$ that strongly discourages the overall latency (from client-side processing and network delays) from getting very close to a preset limit (*e.g.*, 2 seconds) (Fig. 5 c); and **iv)** a strong negative penalty $u_{over}(\mathbf{s},a)$ (equal to -10) that prevents the packet queue from filling up close to its limit (dictated by the hardware). The last penalty term $u_{over}(\mathbf{s},a)$ is applied preemptively at the client side whenever an action of the RL agent could potentially lead to a transmission that will cause buffer overflow and/or exceed the acceptable transmission time. The overall reward function is given by,

$$r(\mathbf{s},a) \;=\; u_{dec}(\mathbf{s},a) - u_{band}(\mathbf{s},a) - u_{lat}(\mathbf{s},a) - u_{over}(\mathbf{s},a),$$

The reward terms are discussed formally in Appen. C. Details of our PPO implementation is given in Sec. 4, and more background on the PPO algorithm is given in Appen. D.

False Positives & Reward Feedback. The cloud server does

not know the ground truth about LoRa packets, *i.e.*, a transmitted active period could just be noise (false positive). Moreover, in a low-duty-cycle network, the cloud receiver may decode only a small number of packets relative to the total number of active periods. Therefore, the RL agent could learn to compress more since there is a higher chance of incurring penalties from overshooting BW and/or latency limits, while the positive rewards for decoding the occasional packets are small. This could drastically increase the overall learning time necessary for the RL to reach an optimal policy. To address this, in the reward term $u_{dec}(\mathbf{s}, a)$, we weight the number of packets decoded by the *inverse of the true positive rate*, which is estimated as the fraction of LoRa packets decoded correctly over the last 100 detected active periods.

3.3 CRAN Cloud Server

The active LoRa signals compressed using ACCIO are streamed to the cloud server using a reliable TCP connection. Each compressed active period is packetized with metadata such as gateway ID, time-stamp at the gateway, length of the active period, sampling rate, channel number, number of DWT levels, among others. The cloud server in our architecture receives the packet, reads the metadata, and performs inverse DWT to reconstruct the signal. It is then input to user-defined LoRa receivers, implemented as Docker containers in the cloud. We separate the LoRa demodulator and decoder so that a custom LoRa demodulator can be deployed by simply updating the demodulator, while retaining the rest of the cloud implementation. The number of decoded packets is sent back to the CRAN gateway as a reward (component) to ACCIO , which then uses the reward to update its RL policy.

Two key objectives of our cloud-server design are i) scalability and ii) ease of deployment of user-defined LoRa receivers. To address scalability, we deploy parallel Docker containers per consumer. As the network scales, the cloud server increases the number of consumers to keep up. To facilitate user-defined LoRa receivers, we include a multiplexer that receives compressed signals and routes them to consumer containers based on their metadata. The link between the multiplexer and the consumers is simply a set of sockets, where each consumer listens on a unique port. The consumer is unaware of the compression and reconstruction. A configuration file maintains the global mappings of the (base-station, channel) pairs to ports. Note that users can map multiple base stations to the same ports to easily apply coherent combining such as Charm [6] atop our implementation. Each Cloud-LoRa packet contains time-stamps for coarse time synchronization between the base stations.

4 Implementation

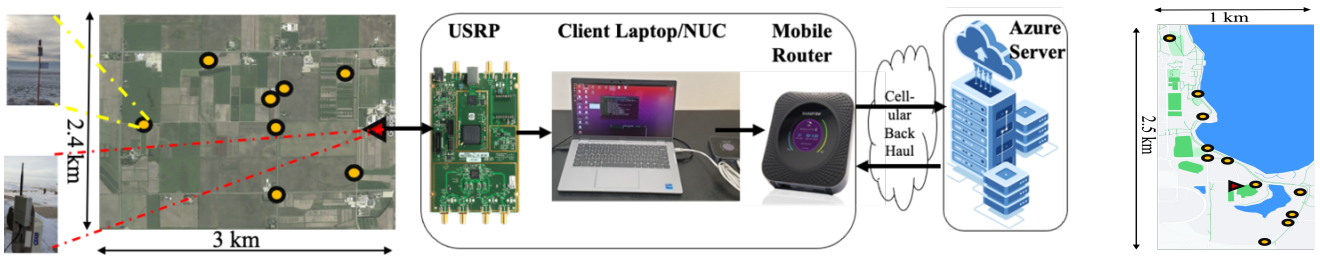
We describe in detail our end-to-end implementation of the three components of Cloud-LoRa : 1) SDR as CRAN gateway 2) ACCIO , the RL-based compression, 3) the cloud server.

CRAN Gateway - Activity Detection. We use a USRP B200 [43] as the CRAN gateway to capture a 2 MHz spectrum that includes 8 LoRa channels (125 kHz bandwidth and 75 kHz guard band, as per LoRaWAN specs). Channelization is performed using eight parallel 4th-order elliptical low-pass filters. Each filtered channel is input to the activity detection module implemented using Python. We use $superDC_{low}$ and $superDC_{high}$ to detect active periods of SF7 to SF9 and SF10 to SF12 respectively. We advance the superDC windows every 1/3-rd of the lengths of respective window samples, to ensure alignment with higher SFs.

CRAN Gateway - ACCIO The adaptive compression of ACCIO is implemented on a client laptop. On detecting active periods in each channel, the corresponding I/Q samples are pushed to the *Application Packet Queue*. The RL agent pops the oldest active period and extracts the state variables from the current network and the active period's coefficients. Table 1 lists the state variables used by the RL agent.

The RL agent was trained using the PPO algorithm [42], whose implementation is provided in the Keras and TensorFlow libraries [44]. Both the policy function and the value (or advantage) function in our PPO-based agent are realized using a fully-connected neural network with two hidden layers of sizes 48 and 32 respectively. This small network enables lightweight training and action determination, while maintaining sufficient complexity for complex approximations. The output layer of the policy network uses the Softmax activation to return probabilities over the set of actions. We set the discount factor of the cumulative rewards to $\gamma = 0.9$. We use the variant of PPO with a clipped objective, and set the clip ratio ε to 0.2. Optimization is based on the stochastic gradient descent method Adam [45], whose learning rate for the policy network and value-estimation network are set respectively to 0.00025 and 0.009. The episode length was defined as 50 active periods, *i.e.*, the agent performs online training, and after every 50 active periods, rewards are returned from the server. To maintain strict reward ordering, each active period (once popped from the application packet queue) is given an ID counter. The active period statistics are then cached and re-ordered after the decoding information is returned. As our implementation is run online, this re-ordering and training process runs in a background daemon.

The RL agent chooses an action that determines the compression threshold C_{thresh}, and DWT coefficients with magnitude $< C_{thresh}$ are set to zero. We further compress the DWT coefficients using Lz4 (preferred over Gzip due to its faster compression). The compressed coefficients are packetized with metadata and sent through TCP to the cloud server. Packet metadata includes the SDR gateway's ID, the active period's ID, the channel it was received on, sampling rate, time the active period was received at the client, as well as other useful information such as the data-section size and its



(a) (b)

Fig. 6: (a) RURAL : Outdoor rural deployment where LoRa Tx (yellow circles) transmit to a USRP B200 (red triangle), which is then connected to a client running ACCIO that streams to Azure server through cellular hotspot in real-time. (b) URBAN : LoRa Tx transmit to a USRP which stores the received samples in a local file.

DWT level sizes (needed for Inverse DWT).

We utilize BBR as the TCP variant; it provides the estimated network bandwidth to the client. We use the socket statistics tool to obtain the bottleneck bandwidth, delivery rate estimated by BBR, and the link's average round-trip time. The bottleneck bandwidth is a key state used by the RL agent to determine a compression threshold.

LoRa receiver at the Cloud Server. The cloud server is implemented in Microsoft Azure as Docker Containers [46], demodulating packets and sending rewards back in real-time(Cloud-LoRa is amenable to deployment on other cloud providers as well.) Our server utilized 8 Docker containers, each reading on unique ports corresponding to each LoRa channel. The Docker containers were booted using Docker Compose [47] and each container was running on an Azure VM. The first module of the cloud server is a multiplexer that decompresses the received samples: it first performs the inverse of Lz4, reads the metadata, and then decompresses using Inverse DWT. Using the metadata, the multiplexer routes the decompressed DWT coefficients to the corresponding user-defined consumer (demodulator). In other words, the multiplexer is responsible for reconstructing the active periods and placing them in the queue of the appropriate consumer based on the metadata of the received TCP packets. The consumers are Docker containers that take the reconstructed active period samples as input, and run the user-defined LoRa demodulator algorithm that outputs symbols, followed by the LoRa decoder that outputs bits. The number of packets decoded per active period, weighted by the inverse-true-positive-rate is used by the RL agent as a reward component in the feedback channel back to the corresponding CRAN gateway.

5 Evaluation

We have deployed the first LoRa CRAN operating in real-time, in two practical outdoor deployments/scenarios. In RURAL(Fig 6(a)), we deployed eight LoRa transmitters in an agricultural farm. Here, we use a cellular backhaul, whose bandwidth varies with time; the backhaul bandwidth is the bottleneck in the network. We show the real-time operation of Cloud-LoRa in this scenario averaged across multiple days. URBAN(Fig 6(b)) shows an urban deployment where the backhaul does not pose a limitation in bandwidth. We leverage this scenario to perform controlled experiments, evaluate the micro-benchmarks and perform an ablation study. Towards evaluating Cloud-LoRa , we answer the following questions.

1. How well does Cloud-LoRa perform in rural settings with impoverished cellular backhaul?
2. Can Cloud-LoRa enable real-time joint decoding of LoRa packets from multiple gateways in the cloud to improve coverage or capacity?
3. Can Cloud-LoRa enable rapid deployment of recently developed state-of-the-art LoRa demodulators?
4. Can Cloud-LoRa scale elastically to provision for network capacity by increasing the number of channels?
5. How does ACCIO adapt in real-time to changing backhaul bandwidth, network latency, and channel quality?
6. How does ACCIO's adaptive compression respond to varying backhaul network latency, and how well does it adapt to bandwidth variations?

5.1 Real-world Deployment Settings

We describe our rural and urban deployments in detail below.

RURAL : Rural deployment scenario. As shown in Fig. 6(a), our deployment includes 8 LoRa transmitters (yellow circles) that broadcast data from humidity sensors to a CRAN gateway (red triangle). Each transmitter operates in a dedicated 125 kHz BW LoRa channel and chooses a random SF and packet length to emulate rate adaptation in LoRaWAN networks while actively transmits 10% of the time. Our CRAN gateway receives over 902.2 MHz to 904.2 MHz and receives samples over 8 different LoRa channels one for each transmitter. It uses a Netgear cellular mobile hotspot as backhaul to a CRAN cloud server in Microsoft Azure (Fig. 6(a)). The backhaul bandwidths achieved by the cellular hotspot varied over a wide range: 1 Mbps to 15 Mbps at different locations and times. As shown in Fig. 6(a), Cloud-LoRa streams samples to the cloud server in real-time, using ACCIO to learn and adapt to the varying available bandwidth. The transmitters were left in the field over 2 days with a total of ≈470000 packets transmitted. The CRAN gateway did not

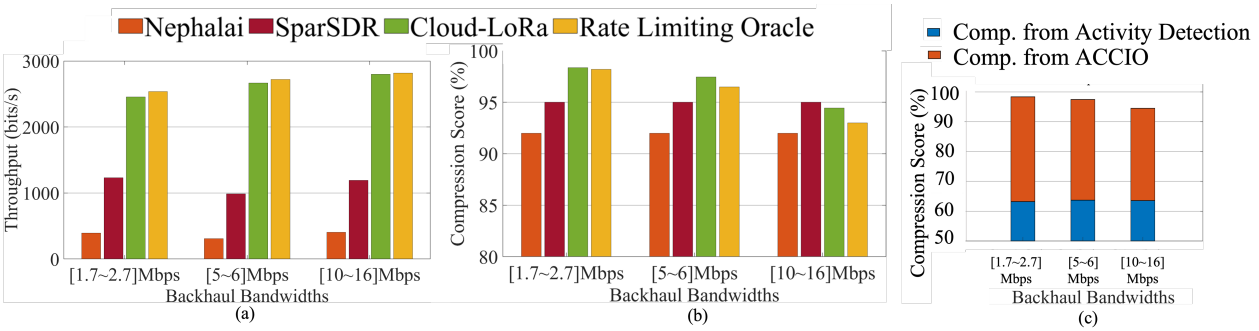


Fig. 7: (a) : Cloud-LoRa throughput performance across 8 parallel LoRa channels, averaged over multiple days in RURAL scenario. (b) Corresponding compression performance. (c) Ablation study on Compression.

have any pre-trained model for ACCIO to use; instead, the RL agent learned from scratch and adapted in real-time.

URBAN: Urban deployment used for Ablation Study. We deploy 9 off-the-shelf LoRa transmitters, each operating in a different channel in an urban, outdoor setting (Figure 6(b)). The transmitters were deployed over an area of 2.5 km x 1 km. The CRAN gateway receives the samples from all the transmitter over a wide bandwidth and stores them locally with time stamps to enable replay. The stored samples are then replayed in real-time to the cloud server to emulate real-time streaming. We connect the USRP to the cloud server via a router. This activity is to ensure consistency across multiple microbenchmark experiments that run with different parameters. This setup allows us to simulate different backhaul bandwidths and latencies to the cloud by using Linux Traffic Control (TC) [48] at the router, a tool for shaping traffic. We perform controlled, comparative, and ablation studies using this deployment by varying various factors such as backhaul bandwidths(Sec. 5.6), LoRa channel quality(Sec. 5.7), network load(Sec. 5.5), backhaul latency(Sec. 5.7), and others.

Backhaul Compression Baselines compared. We implement and compare the compression and throughput performance of Cloud-LoRa against five baselines: 1) **Standard LoRa** – a LoRa gateway that demodulates each packet at the gateway *i.e.*, without CRAN; 2) CRAN with **No compression**; 3) **Nephalai** [19], which proposes a compressed-sensing-based static compression; 4) **SparSDR** [24] – a sparsity-aware compression which is agnostic to the PHY-layer technology; and 5) **Rate-limiting Oracle**. This oracle provides a theoretical upper bound on the throughput and compression performance. We assume that the oracle has a global view of the incoming traffic and bottleneck bandwidth, is not limited by computational resources, performs perfect activity detection with zero false positives and, is able to compress the active periods exactly to meet the available bandwidth.

5.2 Performance in a Rural, Bandwidth-Constrained Deployment

In RURAL we repeated the deployment over three separate 8hr sessions. In each session, the cellular hotspot was placed at roughly the same location (within a 2m radius). Despite using roughly the same location for the hotspot, we found significant variation in the backhaul bandwidth ranges in these three sessions. Arranging the sessions in increasing average backhaul bandwidth, the ranges were 1.7-2.7Mbps, 5-6Mbps and 10-16Mbps. The distribution of the received SNR collected over all 24hrs from all the transmitters (≈470000 packets) are shown in Fig. 8. As seen from Fig. 8, there is a wide variation in received SNRs at the gateway from -15dB to 30dB. ACCIO continuously learns and adapts its compression to meet the available cellular backhaul bandwidths (Fig. 6(a)) and simultaneously streams 8 channels. We plot the average LoRa throughput over all 8 channels (bits/second) achieved by Cloud-LoRa in Fig. 7(a) for the three different sessions. We also compare the achieved average network throughput of Cloud-LoRa with Nephalai, SparSDR, and Rate-limiting Oracle. The maximum achievable throughput *with compression* is upper bounded by the rate-limiting oracle. We observe that the throughput of Cloud-LoRa approaches that of the oracle at higher backhaul bandwidths of 10-16 Mbps, while achieving ≈ 96% of that of the Oracle even at 2Mbps backhaul badwidths. Further, Cloud-LoRa is able to use its adaptive compression effectively and significantly outperforms other LoRa compression solutions such as Nephalai by 7x and 6.2x, and SparSDR by 2.3x and 1.9x (on average).

In Fig. 7(b), we plot the compression score, defined as (# samples streamed to the cloud) / (# samples captured by the CRAN gateway), for the same backhaul bandwidths. We observe that Cloud-LoRa has a higher compression score of 98.4% at 1.7 to 2.7 Mbps, while a lower compression of 94% at 10 to 16 Mbps, *i.e.*, it compresses more at low backhaul bandwidths. On the other hand, Nephalai and SparSDR adopt static compression and hence face packet losses when the compression does not meet the backhaul bandwidth. The most appropriate compression necessary using CRAN is that of the Oracle, as it has a global view; Cloud-LoRa is within 98.5% of the oracle's comrpession score on average.

Two key reasons for the improved throughput performance of Cloud-LoRa over existing approaches are i) sub-noise activity detection and ii) adaptive compression. The contribution of each of these modules to the overall compression is shown in Fig. 7(c). On average, the activity detection achieves a

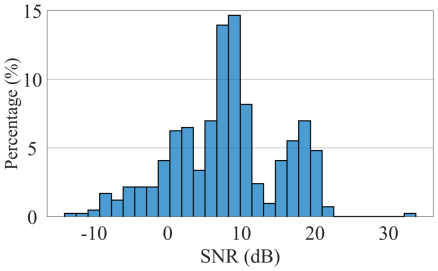


Fig. 8: Distribution of the received SNR at the Gateway in RURAL

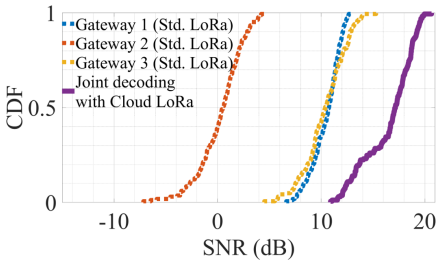


Fig. 9: SNR Gains from Joint Multi-Gateway Packet Decoding

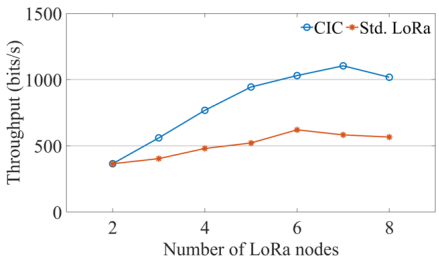


Fig. 10: Throughput Improvements due to Rapid Deployment of state-of-the-art

compression score of 63% in our setting by distinguishing active LoRa transmission from noise samples. The remaining compression is achieved by ACCIO , which varies the compression threshold of the active period to meet the backhaul bandwidth. While the compression score achieved by the activity detection block does not change with backhaul or LoRa signal characteristics, that of ACCIO changes with backhaul bandwidth, as evident in Fig. 7(c).

5.3 Joint Multi-Gateway Packet Decoding

Cloud-LoRa offers centralization, which is key to jointly process raw radio signals across multiple gateways such that the SNR of weak LoRa links could be enhanced through coherent combining with relatively stronger links. In this section, we deploy Charm [6] using Cloud-LoRa using 3 Cloud-LoRa gateways deployed on the floor of a large building spanning over 100m $\times$ 50m along with a LoRa transmitter transmitting packets. On detecting activity, the gateways would attach a time stamp, gateway ID and relay the samples to the cloud. In the cloud, we deploy Charm which coherently combines samples from the 3 gateways based on time stamps and packet/gateway IDs and decodes 100% packets. In Fig. 9, we plot the CDF of packet SNR received across 3 USRP gateways streaming samples. Jointly decoded packets achieve a mean SNR of 16dB, a 6dB improvement from even the stronger links, i.e. Gateways 1 and 3. This SNR boost almost doubles the coverage area. At each gateway, ACCIO achieves 93% compression. We also plot the network throughput achieved by joint decoding when each gateway faces varying backhaul bandwidths. These results are in Appendix E.

5.4 Rapid Deployment of state-of-the-art

Another key advantage of LoRa CRAN is the rapid deployment of novel PHY techniques to practice. In this section, we evaluate a state-of-the-art LoRa receiver in the cloud server, i.e. Concurrent Interference Cancellation (CIC) [12] published as recently as 2021 using Cloud-LoRa . CIC improves LoRa network throughput by decoding multi-packet collisions. In Fig. 10, we plot the network throughput with CIC as the demodulator in the cloud and compare it against Std. LoRa in the cloud. We increase the number of concurrent nodes (same SF and BW) in a single channel in the x-axis and stream the samples to the cloud, where CIC is used as the demodulator. Cloud-LoRa was capable of transporting the samples in real-time for CIC to demodulate. Hence, the network throughput shows an improvement of 1.9x over standard LoRa when 7 nodes are colliding. The throughput begins to drop beyond 7, the maximum collisions CIC can resolve.

5.5 Elastic Scaling to Multiple Channels

Cloud-LoRa allows for scaling to provision for higher capacity by increasing the number of channels from single to multiple channels without any hardware change. However, as the number of channels increases, the volume of samples increases. With limited backhaul bandwidths, such an increase in samples demands higher compression. In this section, we answer the question of *how well ACCIO adapts to an increasing number of channels for a given bottleneck bandwidth*. We plot the number of packets decoded in the cloud at 10 Mbps bottleneck bandwidth for increasing number of channels in Fig. 11. When only one channel has active LoRa signals, a 10Mbps backhaul can support low compression. As the number of active channels increases, the load and hence the number of samples increases. At 10 Mbps bandwidth, a static compression of 50% for Nephalai is best suited for up to 2 channels since packet losses due to compression would be the bottleneck in this case. With 4 or more channels, Nephalai-75 is better suited since packet losses due to network congestion would dominate. ACCIO on the other hand adapts its compression to meet the bandwidth, despite the increase in traffic load. Cloud-LoRa 's network throughput increases linearly by an order of magnitude as the number of channels increases from 1 to 8. Cloud-LoRa decodes about 2X and 4X more packets than Nephalai-75 and Nephalai-50 respectively for 4 channels, and 12X and 20X for 8 channels.

5.6 Varying Backhaul Conditions

Bandwidth-Aware ACCIO In the practical deployment, we have witnessed the adaptation of Cloud-LoRa 's compression performance to different backhaul conditions. However, due

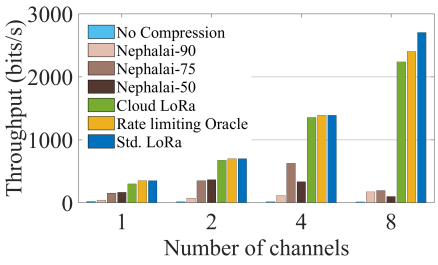


Fig. 11: Elastic Scaling to Multiple Channels - Throughput Vs # of channels

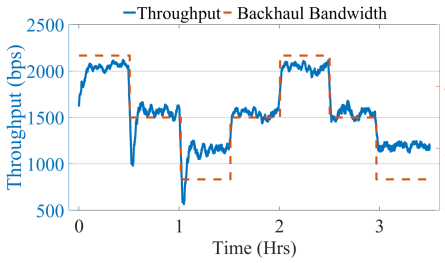


Fig. 12: ACCIO Adaption to Varying Backhaul Bandwidths

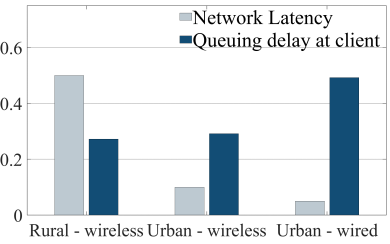


Fig. 13: ACCIO Adaption to Varying Backhaul Latency

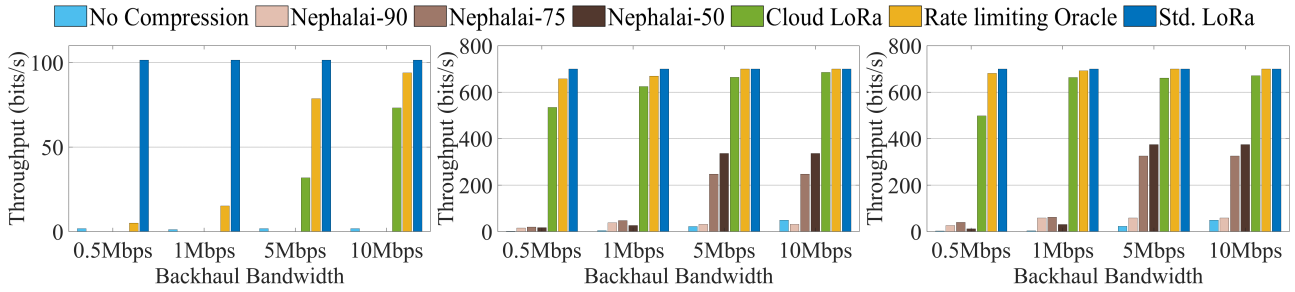


Fig. 14: ACCIO adaptation to varying LoRa Channel Quality
(a) Low SNR (-20 to -5 dB) (b) Medium SNR (-5 to 10 dB) (c) High SNR (10 to 25 dB) LoRa signals.

to the uncontrolled cellular backhaul, it is challenging to observe its time to learn and adapt. In this section, we study the adaptation of Cloud-LoRa to varying backhaul bandwidths in a controlled setting. Using the I/Q samples collected and stored at the CRAN gateway in the outdoor RURAL, we replay the real-time streaming of samples to a cloud server *i.e.*, the gateway streams samples stored in a file to the cloud server, where the samples are decoded, which are then used as rewards by ACCIO at the gateway. During this replay, we control the backhaul bandwidths using Linux-TC. We vary the backhaul bandwidths every 0.5 hours, as shown by the dotted orange line in Fig. 12, from 5 Mbps down to 1 Mbps, and back to 5 Mbps, and then 1 Mbps in steps of 2 Mbps every 0.5 hours. During the first ramp down, LoRa throughput drops and then ramps up every time backhaul bandwidth changes. This is because, in the first 1.5 hours, the ACCIO module at the gateway is untrained *i.e.*, it has not faced the new backhaul conditions before, and hence takes time to learn a new policy at the gateway. Once it learns the policy, the throughput flattens. This is evident from the dip in throughput (blue curve) at 0, 0.5, and 1 hour mark in Fig. 12. The RL agent takes approximately 10 minutes to learn a new policy when it faces a new backhaul bandwidth. After the 1.5 hour mark, when the backhaul bandwidth ramps up to 5 Mbps, LoRa throughput approaches the steady state quickly at 1.5, 2, 2.3, and 3 hour marks as the RL agent has learned a policy for these backhaul bandwidths in the first 1.5 hours. The adaptation time of Cloud-LoRa to varying backhaul conditions therefore depends on the historical data.

Latency-Aware ACCIO Figure 13 plots the queueing delay at the SDR gateway for three networks with different network latencies. In each of these networks, the application latency requirement is designed to be 2 seconds. Therefore, the RL agent learns to drop packets at the client and/or compress more when the network latency is high such that the overall latency is below 2 seconds. In the case of networks with low latency, the RL agent tolerates more queueing delay at the client, allowing it to send more packets.

5.7 Varying LoRa Channel Quality

Compression depends on the SNR of the LoRa samples at CRAN gateway. We evaluate the throughput and compression performance of ACCIO as a function of SNR at different backhaul bandwidths in Figure 14. The corresponding compression performance are presented in Appen. E. Figure 14 (a), (b), (c) correspond to SNR : low (-20 to -5 dB), medium (-5 to 10 dB), and high (10 to 25 dB) respectively. At SNR $<$ 0 dB, Nephalai decodes less than 5% of the packets even at 50% compression (Nephalai-50). This is due to its inability to differentiate between noise and active LoRa signal. Cloud-LoRa improves network throughput by over 20X compared to Nephalai at low backhaul bandwidths. Cloud-LoRa's throughput performance is limited by the false positive rate of the activity detection, which results in a higher volume of active LoRa samples to be transported. At a backhaul bandwidth of 1 Mbps, the RL agent chooses compression scores of over 99% to meet the backhaul constraints (see Fig. 19 in Appendix),

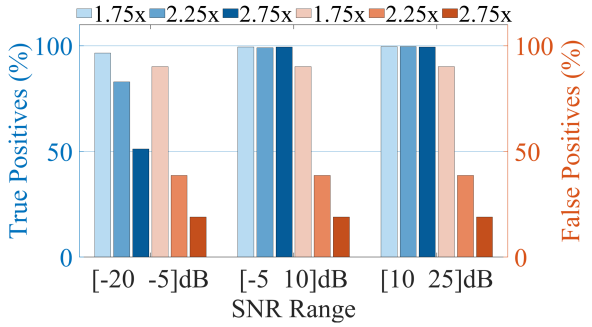


Fig. 15: Activity detection performance

resulting in poor reconstruction in the cloud. As the backhaul bandwidth increases, the achievable throughput improves to over 70% of that of a standard gateway. The network throughput of Cloud-LoRa for Medium SNR signals (green bars in Fig. 14(b)) is about 90% at 1 Mbps backhaul with a compression score of approximately 91%. Nephalai fails to decode more than 10% at 1 Mbps; this can be attributed to the lack of activity detection in Nephalai which leads to redundant samples taking up available bandwidth.

Lossless compression such as Lz4-0 and Gzip9 offers better decodability than ACCIO at bandwidths >= 5Mbps. As the bandwidth decreases, Lz4-0 faces over 50% packet loss due to network losses, similar to Nephalai-50. While Gzip9 achieves 75% compression and has lower packet loss even at low bandwidths, it is too slow to use for real time compression. Even at a backhaul bandwidth of over 5 Mbps, Nephalai-50% compression decodes only about 50% of the packets, while Cloud-LoRa decodes more than 95% of the packets with an impressive compression of over 91%. For high SNRs ACCIO transports over 95% of packets (Fig. 14(c)) to the cloud with an average compression score of 98%.

5.8 Activity Detection of Cloud-LoRa

We evaluate the tradeoff between the sensitivity to low SNR and false positive rate in detecting active LoRa packets of our proposed activity detection algorithm. The proposed multi-channel, sub-noise LoRa activity detection is agnostic to the transmitters' SF. Therefore, the sensitivity of the module determines the minimum SNR that can be detected. We plot the percentage of true active periods and false positives as a function of SNR in Figure 15. The two y-axes presented are true positives, the percentage of active periods correctly detected, normalized to that of a standard LoRa gateway (blue bars), and the percentage of active periods that were detected but did not contain a packet (false positives), normalized to the total samples received (red bars). Each bar represents a different sensitivity used by the activity detection module. A higher sensitivity results in detecting more packets even at lower SNRs at the cost of higher false positives. As SNR increases, the true positive does not vary with the threshold, as the signal energy is high. However, false positives increase with increased sensitivity, even at high SNR. Therefore, the right choice of the threshold is particularly critical in detecting the most active period at low SNRs, without trading off too many false positives. Cloud-LoRa settings that detect over 80% of the packets (Cloud-LoRa -2.25x) only result in 40% false positives, which is further compressed by RL. We observe that, combined with the RL compression, the volume of non-active samples transported to the cloud by Cloud-LoRa is minimal. In contrast, SparSDR incurs more than 90% overheads to accommodate near-zero dB SNR.

6 Discussions and Future Work

To the best of our knowledge, Cloud-LoRa is the first end-to-end practically deployable LoRa CRAN. There are plenty of promising future research directions emerging from this framework. In Cloud-LoRa , we spawn a new process for each channel and stream upto 9 channels in real-time. We can scale to the maximum 64 channels using a NUC with more cores, and a more efficient software implementation. Beyond standard LoRa, recent research such as CurvingLoRa [49] and Falcon [16] that changes the transmitter requires further study to understand the impact of compression on packet decodability. Further research is needed to incorporate performance guarantees such as throughput fairness across channels.

7 Conclusions

We proposed, designed, and implemented Cloud-LoRa , the first practical CRAN for LoRa networks. Cloud-LoRa streams LoRa signals to a cloud server that performs baseband signal processing, in turn providing opportunities for dynamic network scaling and rapid deployment of novel LoRa receivers in the cloud. Towards realizing this end-to-end framework, we developed an *activity detection* algorithm that can detect sub-noise active LoRa signals and reduce signals being streamed to the cloud. We also developed ACCIO, an *RL-based adaptive compression* technique, whose compression threshold adapts to variations in backhaul bandwidth, latency requirements, and input-signal characteristics at the gateway in real-time. We implement and deploy Cloud-LoRa as a Docker container in an Azure server, and experimentally show the feasibility of CRAN for real-world LoRa deployments.

8 Acknowledgements

We would like to thank our shepherd Kate Lin and the anonymous reviewers for the valuable comments and for helping us improve the paper. The authors are partially supported through the following NSF grants : CCSS-2034415, CNS-2142978, 2213688, 1838733, 2112562, 1719336, 1647152, 1629833, 2107060, 2212688, and 2003129 and the US Department of Commerce award 70NANB21H043.

References

[1] LoRa. https://www.semtech.com/lora.

[2] E. Asimakopoulou and N. Bessis. Buildings and crowds: Forming smart cities for more effective disaster management. In *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 229–234, 2011.

[3] María V Moreno, Miguel A Zamora, and Antonio F Skarmeta. User-centric smart buildings for energy sustainable smart cities. *Transactions on emerging telecommunications technologies*, 25(1):41–55, 2014.

[4] Achim Walter, Robert Finger, Robert Huber, and Nina Buchmann. Opinion: Smart farming is key to developing sustainable agriculture. *Proceedings of the National Academy of Sciences*, 114(24):6148–6150, 2017.

[5] Climate Smart Agriculture. https://www.worldbank.org/en/topic/climate-smart-agriculture.

[6] Adwait Dongare, Revathy Narayanan, Akshay Gadre, Anh Luong, Artur Balanuta, Swarun Kumar, Bob Iannucci, and Anthony Rowe. Charm: exploiting geographical diversity through coherent combining in low-power wide-area networks. In *2018 17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 60–71. IEEE, 2018.

[7] Artur Balanuta, Nuno Pereira, Swarun Kumar, and Anthony Rowe. A cloud-optimized link layer for low-power wide-area networks. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 247–259, 2020.

[8] Branden Ghena, Joshua Adkins, Longfei Shangguan, Kyle Jamieson, Philip Levis, and Prabal Dutta. Challenge: Unlicensed lpwans are not yet the path to ubiquitous connectivity. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–12, 2019.

[9] Xianjin Xia, Yuanqing Zheng, and Tao Gu. Ftrack: Parallel decoding for lora transmissions. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, pages 192–204, 2019.

[10] Shuai Tong, Jiliang Wang, and Yunhao Liu. Combating packet collisions using non-stationary signal scaling in lpwans. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 234–246, 2020.

[11] Rashad Eletreby, Diana Zhang, Swarun Kumar, and Osman Yağan. Empowering low-power wide area networks in urban settings. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 309–321, 2017.

[12] Muhammad Osama Shahid, Millan Philipose, Krishna Chintalapudi, Suman Banerjee, and Bhuvana Krishnaswamy. Concurrent interference cancellation: decoding multi-packet collisions in lora. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 503–515, 2021.

[13] Chenning Li, Hanqing Guo, Shuai Tong, Xiao Zeng, Zhichao Cao, Mi Zhang, Qiben Yan, Li Xiao, Jiliang Wang, and Yunhao Liu. Nelora: Towards ultra-low snr lora communication with neural-enhanced demodulation. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, pages 56–68, 2021.

[14] Qian Chen and Jiliang Wang. AlignTrack: Push the limit of lora collision decoding. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2021.

[15] Zhenqiang Xu, Pengjin Xie, and Jiliang Wang. Pyramid: Real-time lora collision decoding with peak tracking. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2021.

[16] Shuai Tong, Zilin Shen, Yunhao Liu, and Jiliang Wang. Combating link dynamics for reliable lora connection in urban settings. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 642–655, 2021.

[17] Christophe Delacourt, Patrick Savelli, and Vincent Savaux. A cloud ran architecture for lora. In *Radio Science Letters*, 2020.

[18] Eryk Schiller, Silas Weber, and Burkhard Stiller. Design and evaluation of an sdr-based lora cloud radio access network. In *2020 16th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 1–7. IEEE, 2020.

[19] Jun Liu, Weitao Xu, Sanjay Jha, and Wen Hu. Nephalai: towards lpwan c-ran with physical layer compression. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–12, 2020.

[20] FCC Report on Broadband Aceess. https://www.fcc.gov/reports-research/reports/broadband-progress-reports/2020-broadband-deployment-report.

[21] Tyler B Mark, Terry W Griffin, and Brian E Whitacre. The role of wireless broadband connectivity on 'big data'and the agricultural industry in the united states and australia. *International Food and Agribusiness Management Review*, 19(1030-2016-83150):43–56, 2016.

[22] FCC Working Paper on Digital Divide . https://www.fcc.gov/reports-research/working-papers/digital-divide-us-mobile-technology-and-speeds.

[23] Amalinda Gamage, Jansen Christian Liando, Chaojie Gu, Rui Tan, and Mo Li. Lmac: Efficient carrier-sense multiple access for lora. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–13, 2020.

[24] Moein Khazraee, Yeswanth Guddeti, Sam Crow, Alex C Snoeren, Kirill Levchenko, Dinesh Bharadia, and Aaron Schulman. Sparsdr: Sparsity-proportional backhaul and compute for sdrs. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 391–403, 2019.

[25] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based congestion control. *ACM Queue*, 14, September-October:20 – 53, 2016.

[26] LoRa and LoRaWAN. https://lora-developers.semtech.com/documentation/tech-papers-and-guides/lora-and-lorawan//.

[27] James E Prieger. The broadband digital divide and the economic benefits of mobile broadband for rural areas. *Telecommunications Policy*, 37(6-7):483–502, 2013.

[28] Christopher Ali. The politics of good enough: Rural broadband and policy failure in the united states. *International Journal of Communication*, 14:23, 2020.

[29] John Lai and Nicole O Widmar. Revisiting the digital divide in the covid-19 era. *Applied economic perspectives and policy*, 43(1):458–464, 2021.

[30] FCC Task Force. www.fcc.gov/task-force-reviewing-connectivity-and-technology-needs-precision-agriculture-united-states.

[31] Muhammad Iqbal Rochman, Vanlin Sathya, Norlen Nunez, Damian Fernandez, Monisha Ghosh, Ahmed S Ibrahim, and William Payne. A comparison study of cellular deployments in chicago and miami using apps on smartphones. In *Proceedings of the 15th ACM Workshop on Wireless Network Testbeds, Experimental evaluation & CHaracterization*, pages 61–68, 2022.

[32] Yuanjie Li, Chunyi Peng, Zhehui Zhang, Zhaowei Tan, Haotian Deng, Jinghao Zhao, Qianru Li, Yunqi Guo, Kai Ling, Boyan Ding, et al. Experience: a five-year retrospective of mobileinsight. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 28–41, 2021.

[33] David L Donoho. Compressed sensing. *IEEE Transactions on Information Theory*, 52(4):1289–1306, 2006.

[34] Revathy Narayanan, Swarun Kumar, and Siva Ram Murthy. Cross technology distributed mimo for low power iot. *IEEE Transactions on Mobile Computing*, 2020.

[35] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall Press, USA, 3rd edition, 2009.

[36] LoRa Modulation Basics. https://www.frugalprototype.com/wp-content/uploads/2016/08/an1200.22.pdf.

[37] Stéphane Mallat. *A wavelet tour of signal processing*. Elsevier, 1999.

[38] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[39] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[40] Richard S. Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, pages 1057–1063. The MIT Press, 1999.

[41] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897. PMLR, 2015.

[42] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[43] USRP B200. https://www.ettus.com/all-products/ub200-kit/.

[44] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[45] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[46] Docker containers on Azure. https://docs.docker.com/cloud/aci-integration/.

[47] Docker Compose. https://docs.docker.com/compose/.

[48] Linux Traffic Control. https://man7.org/linux/man-pages/man8/tc.8.html.

[49] Chenning Li, Xiuzhen Guo, Longfei Shangguan, Zhichao Cao, and Kyle Jamieson. {CurvingLoRa} to boost {LoRa} network throughput via concurrent transmission. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 879–895, 2022.

[50] Richard S. Sutton and Andrew G. Barto. Reinforcement learning: An introduction. *IEEE Transactions on Neural Networks*, 9(5):1054–1054, 1998.

[51] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

Appendix

A LoRa Modulation and Demodulation

LoRa uses Chirp Spread Spectrum (CSS) as its PHY layer modulation. In CSS, the instantaneous frequency of the signal increases linearly with time within a predefined Bandwidth BW over a symbol duration of T_s as shown by chirp equation 2. The start frequency f_{sym} of the data chirp $S(t, f_{sym})$ encodes the data to be transmitted. The slope of the datachirp in frequency-time plot as shown in Fig.16 denotes the Spreading Factor SF which in turn determines the symbol duration $T_s = \frac{2^{SF}}{BW}$, Data Rate and range of operation. Higher SF offers longer range at the cost of reduced data rate. SF can take values $\in \{7,8,9,10,11,12\}$ and for increasing SF, symbol duration doubles. That means an $SF8$ chirp is twice the length of an $SF7$ chirp and an $SF12$ chirp is 32 times the length of an $SF7$ chirp.

$$C(t) = e^{j2\pi(\frac{BW^2}{2\times 2^{SF}}t - \frac{BW}{2})t}, 0 \leq t \leq T_s \quad (2)$$

$$S(t, f_{sym}) = C(t) \cdot e^{j2\pi f_{sym} t} \quad (3)$$

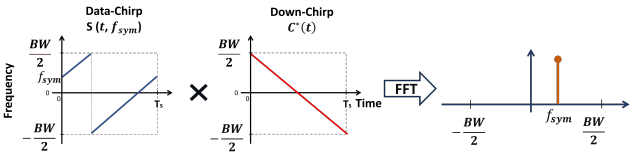


Fig. 16: LoRa Demodulation

At the receiver, LoRa demodulation starts by correlating the received buffer with a preamble (8 base upchirps $C(t)$) to determine the start of LoRa packet. The receiver correlates the buffer with preambles of all possible SFs to reveal the spreading factor SF_x of received packet. It then demodulates the data by multiplying the data symbols with downchirp of same SF_x as shown in Fig.16. Downchirp is a conjugate of base upchirp whose frequency decreases linearly with time. This multiplication is called dechirping and it concentrates the signal energy into a single frequency which is equal to the start frequency of the data symbol. Index of the peak in the FFT of the dechirped signal gives us the demodulated data.

A.1 Chirps of different SF are Pseudo-orthogonal.

As discussed above, to detect the presence and thus start of a LoRa packet, receiver correlates the incoming I/Q samples with base upchirps of all SFs. Once a packet of specific $SF = x$ is detected, datachirps are then demodulated using a downchirp of $SF = y : x = y$. Two chirps (a datachirp and a downchirp) for which $x \neq y$ cannot concentrate energy of the datachirp through dechirping since chirps of unequal SFs

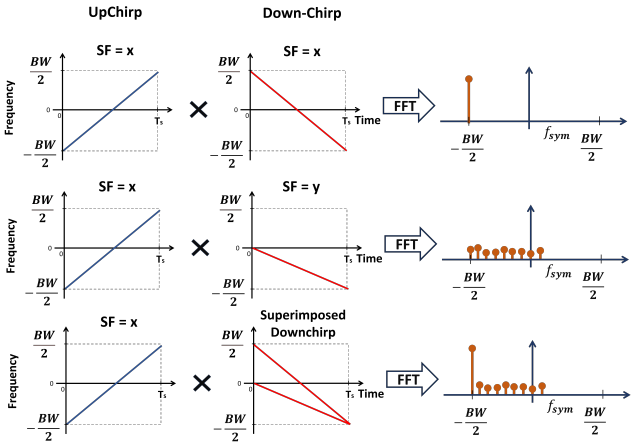


Fig. 17: Pseudo-Orthogonal LoRa chirps

are pseudo-orthogonal. For such cases, dechirping only removes the linear increase in frequency-time trend of the chirps when the magnitude of frequency-time slope of both chirps is equal as shown in top plot of Fig.17. It therefore results in clear peak in the FFT of the dechirped signal. In contrast, if the two chirps have unequal SF, and therefore different magnitude of slopes as shown in the middle plot of Fig.17, dechirping followed by FFT will not concentrate signal energy into a single peak. Instead, the energy is spread over multiple frequencies (hence the term *pseudo*) depending upon the difference in x and y. Therefore, pseuso-orthogonality of SF means that chirps of similar SF can only dechirp signals to a single frequency whereas chirps of unequal SF do not show this behaviour.

Based on this observation, if we design a downchirp which is superposition of two downcirps of $SF = x$ & y respectively and use it to dechirp a datachirp of $SF = x$, we will obtain energy concentration due to downchirp of $SF = x$ and will obtain an increased noise floor due to downchirp of $SF = y$ as shown in bottom plot of Fig.17.

B Primer on the Discrete Wavelet Transform

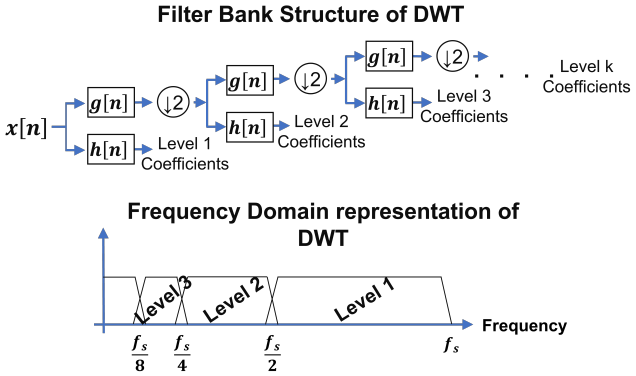


Fig. 18: Filter Bank Implementation of DWT

DWT is a multi-resolution time-frequency analysis tool that

is widely used in image signal processing. It decomposes the input signal into a set of mutually orthogonal wavelet basis functions, that are shifted and scaled versions of a *mother wavelet*. With the appropriate choice of the mother wavelet, DWT provides high time and frequency resolution. Hence, it is an efficient tool to de-noise and compress signals with varying frequency content, such as chirp spread spectrum used by LoRa.

Figure 18(top) shows a k-stage DWT that uses a series of high-pass (h[n]) and low-pass (g[n]) filter banks to decompose the signal into coefficients at multiple levels. This filter-bank implementation of DWT of an input signal of length N has a complexity of $O(N)$, making it computationally a minimally intensive operation, and hence enabling compression in real-time. The DWT coefficients from higher levels of decomposition provide detailed time information. The coefficients from lower levels of the DWT provide the more detailed frequency information. Each DWT coefficient represents the energy of the received signal corresponding to the frequency (level) and time (shift).

C Reward Function Shaping

We discuss the specific choice of the reward function used in our RL algorithm. Recall that the reward function is designed as follows:

$$r(\mathbf{s},a) \;=\; u_{dec}(\mathbf{s},a) - u_{band}(\mathbf{s},a) - u_{lat}(\mathbf{s},a) - u_{over}(\mathbf{s},a). \tag{4}$$

The first reward term encourages the RL algorithm to maximize the total number of LoRa packets decoded correctly over an episode of compression, consisting of (potentially) multiple APs. As discussed in Section 3.2, we would like to weight the number of packets decoded by the estimated true positive rate (TPR) in order to account for the false positives included in the count. Since, the TPR cannot be estimated exactly due to absence of ground truth about when an actual LoRa transmission occurs, we estimate the TPR as the fraction of LoRa packets decoded correctly over the last 100 detected active periods. Suppose N_{dec} is the number of packets decoded correctly and $\hat{P}_{tpr}$ is the estimated TPR, then

$$u_{dec}(\mathbf{s},a) \;=\; \frac{N_{dec}}{\hat{P}_{tpr}}. \tag{5}$$

The second term in the reward function $u_{band}(\mathbf{s},a)$ penalizes the bandwidth utilized B from getting very close to the available bandwidth $B_{\max}$ and is defined as follows:

$$u_{band}(\mathbf{s},a) \;=\; \begin{cases} 0, & \text{if } \frac{B}{B_{\max}} \in [0,\alpha] \\ \frac{0.5\alpha}{1-\alpha^2}\left(\frac{B}{B_{\max}} - \alpha\right), & \text{if } \frac{B}{B_{\max}} \in [\alpha, \frac{1}{\alpha}] \\ 0.5, & \text{if } \frac{B}{B_{\max}} > \frac{1}{\alpha}. \end{cases} \tag{6}$$

Here $\alpha \in (0,1)$ is a constant that determines how close to the maximum bandwidth we want to start penalizing the RL algorithm. We set $\alpha = 0.9$ in our experiments. As shown in Fig. 5, the graph of $u_{band}(\mathbf{s},a)$ as a function of the bandwidth ratio (BWR) $B/B_{\max}$ would be a constant 0 for BWR values less than α; a straight line ranging from 0 to 0.5 for BWR values in the interval $[\alpha, 1/\alpha]$; and a constant value of 0.5 for BWR values larger than $1/\alpha$. The idea is that we start giving as negative reward (penalty) as the BWR approaches 1 and the penalty increases until the BWR exceeds a value slightly larger than 1.

The third term in the reward function $u_{lat}(\mathbf{s},a)$ is very similar to the second term. We simply replace the BWR with the ratio of the overall latency T_{lat} to the maximum latency $T_{\max}$, and it is given below for completeness

$$u_{lat}(\mathbf{s},a) \;=\; \begin{cases} 0, & \text{if } T_{lat} \in [0,\alpha] \\ \frac{0.5\alpha}{1-\alpha^2}\left(\frac{T_{lat}}{T_{\max}} - \alpha\right), & \text{if } \frac{T_{lat}}{T_{\max}} \in [\alpha, \frac{1}{\alpha}] \\ 0.5, & \text{if } \frac{T_{lat}}{T_{\max}} > \frac{1}{\alpha}. \end{cases} \tag{7}$$

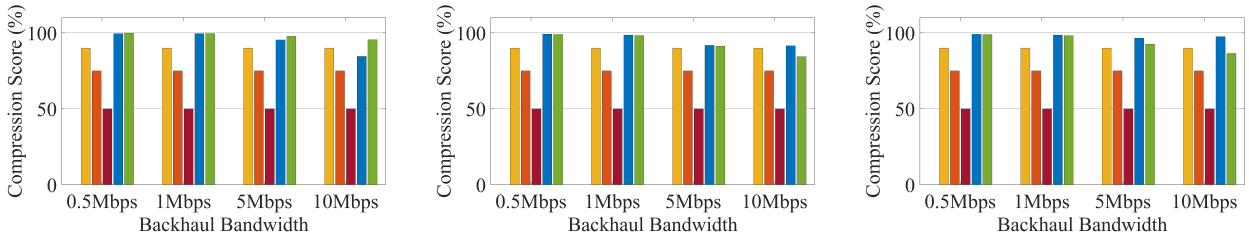
We set $T_{\max} = 0.2$ in our experiments. The factor 0.5 is included in both the penalty terms in order to balance out the rewards due to correct packet decoding and the two penalty terms. We want to avoid giving a higher overall weight to the penalty terms in order to encourage the RL to focus on its main goal of accurate packet decoding.

As discussed in Section 3.2, the final penalty term $u_{over}(\mathbf{s},a)$ is included to preemptively avoid packet buffer overflow, which could result in dropped packets. It is set to a constant -10 whenever an action of the RL agent could result in a packet buffer overflow.

D Background on Reinforcement Learning

Reinforcement learning (RL) is a machine learning paradigm of sequential decision making where an agent acting in an uncertain (stochastic) environment learns to perform actions by interacting with the environment using trial and error (rather than direct supervision) in a way that it maximizes its cumulative reward [39, 50]. This is a popular setting for learning when it is hard to obtain supervised data (*e.g.*, labeled inputs) and the data distribution is non-stationary.

The main component of RL is an agent (*e.g.*, a self-driving car), which is the decision-making center that acts in an uncertain environment (*e.g.*, a street). At any time step t, the agent senses the environment and obtains a signal or reading known as the state $\mathbf{s}_t \in \mathcal{S} \subseteq \mathbb{R}^{d_s}$ (*e.g.*, processed sensor inputs from cameras, Radar, LIDAR etc), which is typically a vector denoted in bold. Given the state $\mathbf{s}_t$, the agent performs an action $\mathbf{a}_t \in \mathcal{A}$ (*e.g.*, the steering angle, brake position *etc.*) that has an effect on the environment. Here $\mathcal{S}$ and $\mathcal{A}$ are the state space and action space respectively. The environment transitions to a new state $\mathbf{s}_{t+1}$ according to its (usually unknown) *state-transition function* $\mathbb{P}(\mathbf{s}_{t+1} \,|\, \mathbf{s}_t, \mathbf{a}_t)$ that governs its dynamics. This is a conditional probability distribution of the state at time $t+1$ given the state and action at time t. The environment



(a) (b) (c)

Fig. 19: (a-c): Compression performance of Cloud-LoRa in a single channel (a) Low SNR (-20 to -5 dB) (b) Medium SNR (-5 to 10 dB) and (c) High SNR (10 to 25 dB) LoRa signals.

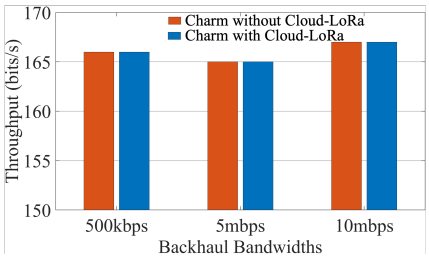


Fig. 20: Throughput performance of Charm with and w/o Cloud-LoRa

provides feedback to the agent in the form of a *reward signal* $r_t := r(\mathbf{s}_t, \mathbf{a}_t) \in \mathbb{R}$ that informs the agent of how good or bad its action $\mathbf{a}_t$ was in the state $\mathbf{s}_t$. Starting from an initial state $\mathbf{s}_0$, this sequence of state observation, action, reward, and state transition is repeated for a number of time steps, known as *an epsiode or trajectory* $\tau := (\mathbf{s}_0, \mathbf{a}_0, r_0, \mathbf{s}_1, \mathbf{a}_1, r_1, \cdots, \mathbf{s}_T, \mathbf{a}_T, r_T)$.

The agent learns a strategy or policy to perform actions in different states by repeatedly interacting with the environment over several episodes, with the goal of maximizing its total rewards. Formally, the policy $\pi : \mathcal{S} \times \mathcal{A} \mapsto [0,1]$ of the agent is a conditional probability distribution over the set of actions given the state, *i.e.*, $\pi(\mathbf{a} \,|\, \mathbf{s}) := \mathbb{P}(\mathbf{A}_t = \mathbf{a} \,|\, \mathbf{S}_t = s)$. We define the *discounted return* U_t at time t as the *discounted cumulative future reward*, with a discount factor $\gamma \in [0,1]$, given by $U_t = R_t + \gamma R_{t+1} + \cdots + \gamma^{T-t} R_T$ [4]. The discount factor γ determines the relative importance of the current reward over future rewards. For a given policy π, the *state-value function* $V_\pi(\mathbf{s}_t)$ and *action-value function* $Q_\pi(\mathbf{s}_t, \mathbf{a}_t)$ are important quantities that define the value of a given state or a state-action pair in terms of the (discounted) cumulative future rewards. The action-value function captures how good an action $\mathbf{a}_t$ is while being in state $\mathbf{s}_t$, and is defined as

$$Q_\pi(\mathbf{s}_t, \mathbf{a}_t) \;=\; \mathbb{E}[U_t \,|\, \mathbf{S}_t = \mathbf{s}_t, \mathbf{A}_t = \mathbf{a}_t],$$

where the expectation is over all the future states and actions, and defined by the policy and the state transition function. The state-value function (or simply value function) captures how good a given state is under the policy π, and defined as

$$V_\pi(\mathbf{s}_t) \;=\; \mathbb{E}[U_t \,|\, \mathbf{S}_t = \mathbf{s}_t] \;=\; \mathbb{E}_{\mathbf{A} \sim \pi(\cdot \,|\, \mathbf{s}_t)}[Q_\pi(\mathbf{s}_t, \mathbf{A})].$$

The goal of an RL agent is to find an optimal policy that maximizes the expected value function $\mathbb{E}_{\mathbf{S}}[V_\pi(\mathbf{S})]$. It is common to use the *advantage function* $A_\pi(\mathbf{s}_t, \mathbf{a}_t) = Q_\pi(\mathbf{s}_t, \mathbf{a}_t) - V_\pi(\mathbf{s}_t)$, which captures the excess return (in a state $\mathbf{s}_t$) obtained by performing action $\mathbf{a}_t$, compared to the expected return obtained over all possible actions.

There are many classes of RL methods, and recently deep learning methods have been adopted to solve the traditionally challenging setting of large and continuous state/action spaces [51]. We focus on a particular type of on-policy RL known as Policy Gradient methods [40, 41], whose goal is to directly learn an optimal policy function that is parameterized by a neural network. Specifically, we use the Proximal Policy Optimization (PPO) method with clipped objective [42] for online training of the RL agent. PPO is widely adopted as a state-of-the-art online policy-gradient method due to its better computational and sample efficiency, and stable policy function updates.

E Supplementary Results

The amount of compression by ACCIO depends on the SNR of the LoRa samples at CRAN gateway. The compression performance of ACCIO for varying LoRa channel quality are presented in Figure 19. Figure 19 (a), (b), (c) correspond to low SNR (-20 to -5 dB), medium SNR (-5 to 10 dB), and high SNR (10 to 25 dB) respectively.

In Fig. 20, we plot the network throughput of Charm with 3 USRP gateways streaming samples to the cloud. Charm coherently combines samples from the 3 gateways and decodes the packets. We compare Charm with Cloud-LoRa , against Charm on a local machine without any compression (*i.e.*, Charm without a bottleneck backhaul). It can be noted that the throughput is unaffected by Cloud-LoRa even at bandwidths as low as 500 kbps, indicating that the compression did not affect the quality of the LoRa signals, and still allows Charm to coherently combine signals in the cloud.

[4] The state, action, and reward random variables are denoted in uppercase, taking on specific values denoted in lowercase.

Cloudy with a Chance of Cyberattacks: Dangling Resources Abuse on Cloud Platforms

Jens Frieß[§‡], Tobias Gattermayer[§*], Nethanel Gelernter[⋆], Haya Schulmann[§†], and Michael Waidner[§‡*]

[§]National Research Center for Applied Cybersecurity ATHENE
[*]Fraunhofer Institute for Secure Information Technology SIT
[‡]Technische Universität Darmstadt [⋆]IONIX [†]Goethe-Universität Frankfurt

Abstract

Recent works showed that it is feasible to hijack resources on cloud platforms. In such hijacks, attackers can take over released resources that belong to legitimate organizations. It was proposed that adversaries could abuse these resources to carry out attacks against customers of the hijacked services, e.g., through malware distribution. However, to date, no research has confirmed the existence of these attacks.

We identify, for the first time, real-life hijacks of cloud resources. This yields a number of surprising and important insights. First, contrary to previous assumption that attackers primarily target IP addresses, our findings reveal that the type of resource is not the main consideration in a hijack. Attackers focus on hijacking records that allow them to determine the resource by entering freetext. The costs and overhead of hijacking such records are much lower than those of hijacking IP addresses, which are randomly selected from a large pool.

Second, identifying hijacks poses a substantial challenge. Monitoring resource changes, e.g., changes in content, is insufficient, since such changes could also be legitimate. Retrospective analysis of digital assets to identify hijacks is also arduous due to the immense volume of data involved and the absence of indicators to search for. To address this challenge, we develop a novel approach that involves analyzing data from diverse sources to effectively differentiate between malicious and legitimate modifications. Our analysis has revealed 20,904 instances of hijacked resources on popular cloud platforms. While some hijacks are short-lived (up to 15 days), $\frac{1}{3}$ persist for more than 65 days.

We study how attackers abuse the hijacked resources and find that, in contrast to the threats considered in previous work, the majority of the abuse (75%) is blackhat search engine optimization. We also find fraudulent certificates and stolen cookies. We cluster the abuse resources and abuse content to identify about 1,800 individual attacking infrastructures.

1 Introduction

Digital resources form the fabric of modern societies. They provide the fundamental platform for digital services and assets, e.g., for financial services, critical infrastructure, government services. Due to their importance, digital resources pose a lucrative target for attackers. Therefore securing these resources and correctly managing them is crucial for the security of the Internet. Managing resources requires not only creating and configuring them, but also releasing them correctly after they are no longer required. Previous work [12] showed that when organizations release resources of services that are no longer needed, they often do not purge the infrastructure that was set up for them, creating *dangling resources*.

Dangling records. Previous work [1, 3, 12, 16, 18] studied the threat introduced by *dangling records*, i.e., Domain Name System (DNS) records that point to resources that were released. The concept of dangling records is related to dangling pointers in programming, which occur when a variable's memory is deallocated. Similarly, DNS records become dangling when domain owners forget to purge the records. For example: a domain owner does not remove a mapping `foo.com A 1.2.3.4` of service `foo.com` to a cloud IP address `1.2.3.4` from the authoritative DNS server after the resource at `1.2.3.4` is discontinued and released. Adversaries, which succeed in taking over the released resources that are pointed to by the existing DNS record, can launch attacks against clients that attempt to access the domain. In our example, if an adversary can take over `1.2.3.4` it can obtain control over all the records that point to that IP address, since all requests to `foo.com` are sent to the adversary. The attack does not require any extensive capabilities. All that it requires is some way of collecting domain names (e.g., via passiveDNS or Certificate Transparency), checking if the resource is hosted in the cloud and is reachable, and if not, registering the resource through an account with the cloud provider.

Research finds multiple dangling records. In 2016 [12] analyzed dangling records on cloud and other platforms and the threat that they create for hijacking domains. [12] found 467 dangling records in top 10K Alexa domains and 52 `.edu` domains. A follow-up study [3] extended the methodology of [12] for identifying dangling records on cloud platforms and identified over 700,000 dangling DNS records. [16] im-

proved the subdomain enumeration of [12] and discovered exploitable vulnerabilities in 887 domains. [1] studied the risk of stale NS records finding 628 hijackable domains. Recently [18] developed a hostingChecker, eventually finding 10K vulnerable subdomains. Although dangling records and their threat have been extensively studied in previous work, no research has provided evidence that dangling records are abused for attacks and demonstrated real-life abuses.

We study real-life abuse of dangling records. The most remarkable result of our work is the first evidence and analysis of actual, real-life attacks that abuse dangling DNS records. Detecting real-life abuses is hard. The fundamental challenge is detecting malicious vs. legitimate changes in resources. We find that the hijacked resources often do not stand out and even have valid (yet fraudulent) certificates. Approaches that look for changes in the infrastructure or in the content do not work, since changes are often legitimate and happen not only in abused resources. In addition, the huge data volumes involved and lack of known indicators make finding abuses equivalent to looking for a needle in a haystack. We show that the key to finding real-life abuses is a combination of longitudinal data analysis from multiple sources with clustering of changes according to similarities and manual keyword derivation. Applying this approach we derive indicators which enable detection of real-life hijacks. Our longitudinal study of abuses in 12 cloud platforms identified 20904 hijacks that hosted malicious content. We detect hijacked domains in 219 Top Level Domains (TLDs) and abuses on popular clouds.

Selection of hijacked resources is financially motivated. Previous work measuring dangling records on cloud platforms looked for released IP addresses that were still pointed to by DNS records. In our study we surprisingly find no IP takeovers among real-life abuses of dangling records. The analysis of the abuse cases in our longitudinal dataset shows that the selection of resources by attackers is financially motivated: attackers target dangling resources which can be easily and cost-effectively taken over. These requirements do not apply to IP addresses on popular cloud platforms. Therefore, although the threat of IP address take-over considered in previous work is real, we find that attackers target different resources than previously assumed. We characterize the resources abused by real-life adversaries and explain which factors make them lucrative targets.

We also show that, surprisingly, the most popular abuse of the hijacked resources (75%) is blackhat Search Engine Optimization (SEO), rather than, e.g., malware distribution, as suggested previously [16].

Definition: hijacks & abuse. We use the term "hijack" to refer to the appropriation of a (sub)domain name through the re-registration of a released cloud resource pointed to by a dangling DNS record. We use the term “abuse" to refer to the subsequent use of such a resource for malicious purposes, such as blackhat SEO, clickjacking, phishing, etc. Our definition therefore falls into the type 2 and 3 categories of DNS abuse, defined by [4], and subsumes their definitions of “malicious conduct", “abusive activity", “DNS abuse" and “DNS misuse". We refer to “blackhat SEO” as “SEO", unless being specifically discussed in the context of regular SEO.

Ethics and notifications. We initiated a notification campaign and already notified more than 300 organizations of the abuse we found in their resources, which already confirmed the hijacks. Large-scale vulnerability studies pose risks and therefore such research, e.g., [2, 5–8, 10, 13, 17], explicitly takes ethics of scans and collected data into account. Due to the sensitive nature of our findings we also take extensive measures to ensure the security of the organizations studied in this work. In our study and data collection we follow the ethical guidelines for network measurements [7, 14], which were also approved by the ethics committee (IRB) in our organization. By following these guidelines we ensure that the equipment of target organizations and cloud platforms is not affected or overloaded, and that the organizations' private data is not compromised. In addition, we conducted a privacy impact assessment with our legal department, which allowed us to conduct the study. For each organization in our dataset we send at most two HTTP requests per Fully Qualified Domain Name (FQDN) to check an abuse: the first request is for the page itself, and if we cannot establish an abuse with confidence, we send another request for the sitemap. We repeat this data collection on a weekly basis.

Contributions. We develop a methodology and use it to find and analyze real-life abuse of dangling records.

▷ *Longitudinal comprehensive dataset.* We collect a longitudinal dataset of (sub)domains pointing to deallocated cloud assets, which started with 1,508,273 records and after three years grew to 3,101,992 records.

▷ *Methodology to detect abuses.* We develop the first methodology that identifies abuse of dangling records. Key to finding abuses are longitudinal data collection from multiple sources and a novel analysis methodology. In our dataset we find 20,904 cases of abused dangling records that belong to organizations in multiple sectors.

▷ *Attackers prefer cheap and easy hijacks.* [3] showed it was possible to take over cloud IP addresses pointed to by dangling DNS records. We find that adversaries avoid IP addresses, which are typically randomly allocated from a large pool, and instead target cheap and easy-to-take-over resources.

▷ *Attacker capabilities.* We show that the abuse that the attacker can launch against the clients of the victim domains is a function of the dangling resource that the attacker takes over. We develop a model of the attacker capabilities as a function of the dangling resource, extending [16] which only focused on the configuration in the legitimate service.

▷ *Characterization of abuses.* We find that the main abuse (75%) is SEO. The attackers mostly target domains with established reputation to increase the ranking of their malicious content by search engines. We also identify cookie stealing attacks, fraudulent certificates and malware distribution - and

we analyze these attacks. Overall, we find that the hacking groups successfully attacked 31% of the Fortune 500 companies and 25.4% of the Global 500 companies, some over long periods of time. Many of the victim organizations were abused more than once, with one even suffering abuse across more than 100 different subdomains.

▷ *Methodology to detect vulnerability to cookie theft.* We develop the first method which enables IT staff to automatically and easily check their own (sub)domain setup for vulnerability to cookie theft, should one of them be hijacked. Using this method we perform a large-scale study of top 10K Alexa domains, finding 65% of domains to be vulnerable to cookie theft and 23% to theft of authentication cookies.

▷ *Characterization of attackers.* We develop a methodology for clustering attackers into groups, using content and metadata on the abuse sites and the infrastructure of the attackers. We identify 1,800 individual attackers infrastructures.

Organization. We review related work in Section 2. In Section 3.1 we develop a methodology for collecting hijacks. In Section 4 we characterize hijacked resources and in Section 5 analyze the abuse deployed on hijacked resources. In Section 6 we develop methods to cluster the abuse by attacker infrastructure and conclude in Section 7.

2 Related Work

The threat of hijacking domains by taking over released resources, pointed to by dangling (stale) DNS records, was considered in previous work [3, 12, 16, 18]. The idea is that the attacker attempts to get assigned a recently released resource, therefore taking over the domain which points to that resource. [12] showed that it was practical for an attacker to obtain the desired IP address from the cloud pool by repeatedly allocating and releasing IP addresses. The authors scanned cloud IP addresses to find dangling DNS records from their dataset of domains using Zmap [7] and found hundreds of dangling records on cloud platforms and on top 10K Alexa domains. [3] extended the dataset of [12] and collected 130M domains that point to IP addresses in cloud platforms. [3] found that over 700,000 domains point to cloud IP addresses that were free and hence vulnerable to domain takeover attacks. [3] also estimated that it would be economically practical for attackers to obtain a target IP address from the cloud pool. [16] further improved the subdomain enumeration and their analysis of deprovisioned cloud instances yielded 13,532 potentially vulnerable domains with dangling records. [1] analyzed dangling NS records and found 628 hijackable domains. Recently, [18] developed an automated framework for detecting dangling records by reconstructing DNS resolution chains and found 10K subdomains among the top 1M Tranco domains with dangling records. Our research augments the previous work in the following aspects:

Targeted dangling records. While [16] suggested that the likelihood of a domain being vulnerable is directly related to the number of subdomains it has, we show that the ease of taking over the dangling DNS record and the reputation of the target domain define the likelihood of an attack.

Analysis of the attack surface. To measure the prevalence of the dangling records previous work sent liveness probes to IP addresses in cloud IP ranges to determine if the IP addresses were allocated and in use. [12] checked TCP ports 80 and 443 and TCP/UDP port 53, [3] sent ICMP pings and TCP/UDP requests to 36 common TCP/UDP ports, and [16] scanned 148 TCP/UDP ports. Records that pointed to IP addresses that did not respond on any ports were classified as dangling.

Due to virtual hosting, TCP/UDP and ICMP pings do not accurately reflect the availability of a (web)service. Reaching a virtually hosted service requires connecting on the application layer, rather than the transport layer, in order to traverse the forwarding logic in the webserver. To illustrate this we compare ICMP, TCP and HTTP requests using our dataset of cloud-hosted, hijacked domains. Using ICMP pings we receive responses from 72% of the domains in the cloud. On TCP ports 80/443 we receive responses from 93% of the domains. An HTTP request to the respective FQDN results in 89% responsive domains. The results indicate that ICMP pings tend to overestimate unresponsiveness, and therefore vulnerability of services by around 20%, whereas TCP pings tend to underestimate by around 4%, compared to HTTP requests to the actual FQDNs. Therefore, to accurately check domain liveness we download HTML files via HTTP/S from the *services* rather than simply probing the ports on the target *servers*, therefore capturing the availability of each individual FQDN, regardless of virtual hosting.

Abuse of dangling records. In contrast to previous work that measured the prevalence of dangling records, in our work we identify and analyze real-life abuse of the dangling records. We analyze the resources that the attackers take over, characterize the target domains and the abuses of the dangling resources for attacks. Previous work proposed that dangling records could be exploited for stealing cookies, issuing fraudulent certificates, loading malware or authentication bypass. In our work we show that the most common abuse of dangling records is for SEO, which was not considered in prior work.

3 Collection of Abused Resources

In January 2020 we started monitoring 1.5 million cloud assets in use by large organizations. After a couple of months we noticed that, after a time period of being inactive, domains of large organizations became active again. However, all those pages that became active after a period of inactivity, shared something in common: they had similar error pages, in different languages, reporting that the website was under maintenance; an interested reader is referred to Figure 23 in the Appendix for one such example in English on a Fortune 500 company domain. Further analysis of these domains revealed that behind the web pages with the error messages were thousands of other pages. In all those cases the hackers gained control over the domains by taking over abandoned re-

Algorithm 1 Collection of Cloud-pointing FQDNs.

```
function COLLECT_FQDNS(fqdns, cloud_suffixes, cloud_IPs)
    fqdns_to_analyze ← { }
    for fqdn in fqdns do
        A_results, CNAME_results ← DNS_A_query(fqdn)
        for CNAME in CNAME_results do
            if CNAME.ends_with_any(cloud_suffixes) then
                fqdns_to_analyze.add(fqdn)
            end if
        end for
        for IP in A_results do
            if IP in cloud_IPs then
                fqdns_to_analyze.add(fqdn)
            end if
        end for
    end for
    return fqdns_to_analyze
end function
```

sources on cloud platforms. We identified such abuse patterns in hundreds of domains, belonging to governments, universities and enterprises worldwide. Periodically adding more organizations to our list, we were identifying more hijacked domains. In June 2020 we were already tracking more than two million domains, over multiple cloud assets, most of them in Azure and AWS (see Table 2 for full list of asset types). In this section we describe our three-year data collection methodology and the resulting dataset of assets. An overview of this collection process is shown in the top left (blue) of Figure 25. This data forms the basis for the analyses, shown in the top right (green), which are covered in Sections 4, 5 and 6.

3.1 Dataset

Our initial search space contains domains across a number of sectors: a list of 2M government domains[1], Fortune 1000[2] and Global 500 enterprise domains[3], and 1M-top Alexa domains[4]. We also use a list of 9,933 university domains[5]. These domains serve as candidates for finding potential hijacks. We do not sanitize these further, as inaccessible domains are automatically removed as part of our search methodology. Using the FarSight passive DNS service, with global sources across all continents, we also discover all subdomains observed for these domains. From this initial list of known high-profile (sub)domains we determine the subset that points to cloud assets, resulting in a list of 1,508,273 (sub)domains, which constitutes the dataset we study in this work.

The pseudocode of our methodology is described in Algorithm 1. We select domains and subdomains that have a CNAME DNS record referencing an FQDN with a known cloud suffix (e.g., `*.azurewebsites.net`, `*.amazonaws.com`); we provide a list of the known cloud suffixes relevant for our research in Appendix A.1. For domains without CNAMEs we check if one of its IPs falls within a subnet used for cloud hosting. This is often the case with domains

[1] `.gov` filtered from `https://domainsproject.org/`
[2] `https://fortune.com/analytics/fortune-1000`
[3] `https://fortune.com/ranking/global500/`
[4] `https://www.kaggle.com/datasets/cheedcheed/top1m`
[5] `https://github.com/Hipo/university-domains-list`

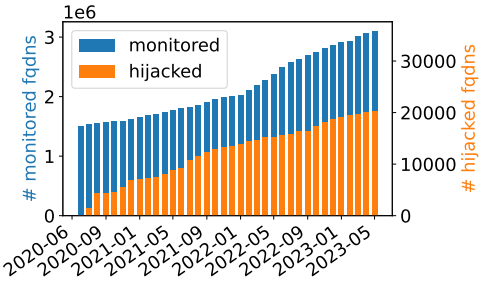


Figure 1: Monitored vs. hijacked cloud-hosted domains over time.

hosted on AWS S3 buckets or dedicated VM servers, which refer to these servers using A records rather than CNAME. Subnet information is published by the cloud providers (see Appendix A.1). By downloading these regulary we ensure that we are using up-to-date information.

Over the 3 year period, we kept updating the list by consuming a commercial feed of FQDNs for the enterprises in our dataset (in Section 3.1). Within that time, we filtered through more than 87,000,000 non-NXDOMAIN (i.e., with at least one DNS record) domains and subdomains, resulting in a doubling of our initial list to 3,101,992 monitored FQDNs. Figure 1 shows this monthly increase of monitored cloud-hosted FQDNs, overlayed with the cumulative number of abuses seen until that time. We collected the data and monitored the changes in DNS records and in HTML files for over 3 years.

3.2 Detection of Abused Domains

Finding abuses is hard, hence it is not surprising that so far no analysis of dangling records has found actual abuses. The main problem is that changes in DNS records and websites are often legitimate, and without knowing what malicious content to look for, finding abuse is virtually impossible. To determine whether assets were abused, we track changes of site content. To accomplish this we take regular samples of each site, downloading the index HTML as well as the sitemap stored in a database. By comparing these snapshots, including changes to DNS, HTTP response, sitemap (e.g., size changes of 100KB), language changes, and keywords, differences can be detected. The components in our analysis are illustrated in Appendix, Figure 25).

Signatures. Having identified content changes in groups of assets within a short time frame, we manually inspect the new content to ensure no false positives when notifying affected organizations. We then create signatures by automatically extracting keywords from the index page and other pages, as well as sitemap features, JavaScript and other loaded objects. We validate that these are shared across multiple abuse pages and finally test these against a large dataset of benign assets to ensure they do not yield false positives; those that do are discarded. The benign assets are also assembled from Alexa websites, Fortune 500 sites and university sites, ensuring cross-sector representation, and verified to not contain malicious content. Once created, these signatures can be used

to detect similar malicious changes on other domains we monitor. Examples of the signatures we create include:

```
(1) index page includes "Comming soon ..."
//"Comming" is written with a typo instead of "Coming".
(2) index page refers to 3 other pages in a specific
structure; the referred pages are written to create a
new window on each click.
(3) loading a particular "popunder.js" script
(4) sitemap with several thousand pages (> 5 MB);
each page is structured similarly; consistent random name
generation
(5) New sitemap or 100KB increase in sitemap size.
(6) Language change.
(7) Keywords related to content uploaded by the attackers
```

We validated the changes manually on the abused assets. Figure 2 shows what percentage of domains in our dataset match different types of indicators or combinations thereof. For example, some domains (30.2%) can be identified with just keywords (such as those shown in Table 5), whereas others (10.1%) require using keywords as well as attacker infrastructure-related indicators, such as hyperlinks or scripts and images loaded from other domains, as differentiating features. If the required features are present on the site, the signature matches and the domain is classified as abused.

In our analysis we find that the page contents, i.e., keywords, are the most telling indicators of abused assets, whereas infrastructure-related indicators are only useful in combination with keywords or sitemap features. Looking at the sitemap and keywords in combination is the most effective, identifying an additional 36.1% of abused assets in our dataset, compared to just keywords.

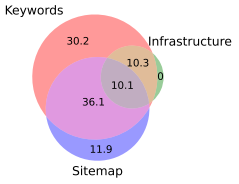


Figure 2: % of detected hijacks with extracted signatures by type.

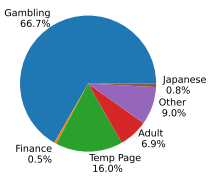


Figure 3: Content classification on hijacked domains.

Keywords. We extracted 56,946 keywords with an average keyword count of 2.72 to classify index HTML files as abused. Table 1 shows the top keywords found, with gambling and adult content as the major sources. Figure 3 shows the full distribution of topics found based on keywords. We list the most popular keywords in the Appendix, Figure 29.

#	Keyword	Count	#	Keyword	Count
1	*HTML Snippet*	4615	2	*HTML Snippet*	4288
3	*HTML Snippet*	4199	4	sex	3257
5	daftar (list)	2930	6	porn	2786
7	situs judi (gambling sites)	2611	8	*HTML Snippet*	2193
9	gacor (hot streak)	2048	10	[j]udi slot online (gambling online)	1892
11	situs slot (slot/gambling sites)	1880	12	slot gacor (hot slot machine)	1564

Table 1: Top 12 keywords for index.html classification.

Abuse dataset. After 3 years of monitoring, we detect 17,698 unique, abused FQDNs (where 1,565 are Second-Level Domains (SLDs), Figure 5) across 11,924 unique SLDs

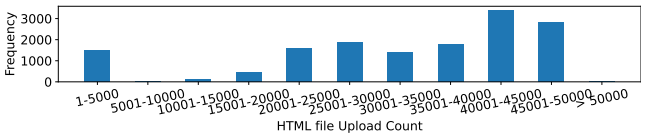


Figure 6: HTML Files uploaded to each abused site

and 218 affected TLDs (see Appendix Table 6 for top 12). These abused domain names point to 15,248 unique CNAMEs (11,654 unique IP addresses). This dataset forms the basis for our subsequent analyses. We notified > 300 affected organizations, which confirmed the abuse.

The Tranco list is a research-focused ranking of the top websites based on their popularity and stability over time [11]. We find 7,049 of 17,698 (39.8%) unique hijacked FQDNs on SLDs included in the Tranco domain list. On average, every tranco-ranked SLD has 2 (1.89) hijacked subdomains. The rank and corresponding unique hijacked subdomain count are illustrated in Figure 4.

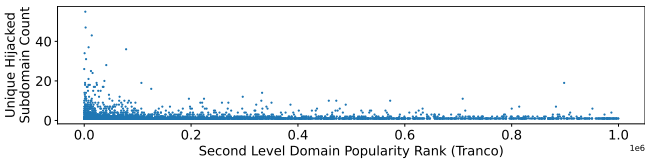


Figure 4: Rank of SLDs and associated hijacked subdomain counts.

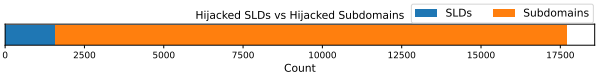


Figure 5: Abused second level domains and sub-domains.

Abuse data volume. We collected 54,325 index HTML files classified as abused content with an average file size of 52.4kB, some of them for the same assets (with changes made by the hackers). We also collected 15,482 sitemaps and analyzed them for the quantity of total malicious HTML content uploaded. Figure 6 shows a histogram of the number of HTML files uploaded, grouped into bins of 5,000. The number of files ranges from 2 to 144,349 HTML files per site, with the clear majority of sites containing many thousands of pages. Abusers uploaded a total of nearly 500M (492,489,492) files with an estimated size of 24TB (25,806,449,380.8kB) and an average of 31,810 HTML files per site.

Ruling out benign changes. There are cases in which changes in content can be legitimate, e.g., changes in parked domains, that display commercial HTML content that changes collectively over time. To rule out legitimate changes applied by registrars that manage multiple domains, we analyze information of the registrars and the owners of the domains. Specifically, we check that clusters of domains with identical changes in content, have different owners and registrars.

To do this we aggregate domains into clusters based on the keywords extracted from their web pages. Identical keyword lists indicate the same page content. By matching the second-

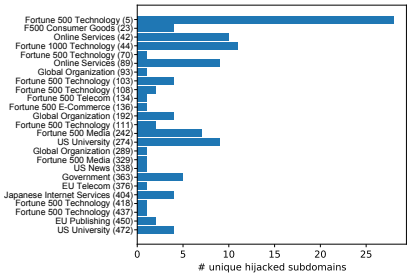


Figure 7: Abuse in top 1M Tranco Domains.

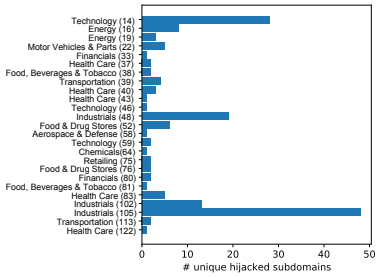


Figure 8: Abuse in Fortune 500 companies.

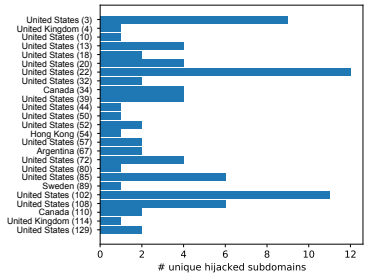


Figure 9: Abuse in QS-ranked universities.

level domains to their respective registrars, we determine the set of unique registrars for each cluster of domains with the same content-changes. We then plot the percentage of these clusters, with at least two domains, by the number of unique registrars observed. This is shown in Figure 10.

Our analysis yielded that in 89% of the cases, where the same change is detected on at least 2 domains, these changes span 2 or more different registrars and owners. In 33% of cases, the changes occur across domains owned by 4 or more registrars; see distribution in Figure 10. This result demonstrates that identical changes in clusters of domains are not made by the registrars, since the domains typically have different registrars.

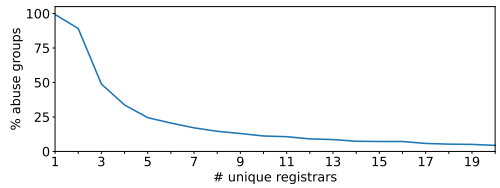


Figure 10: % abuse clusters (grouped by keywords) spanning $\geq$ X registrars.

4 Analysis of Abused Resources

We describe resources of organizations that were abused and identify which cloud platforms host abused resources. To confirm a hijack we have downloaded and manually examined malicious content on hijacked domains. The fact that the same content is seen across unrelated, independent domains and that the content is unrelated to the topics of these domains provides an evidence of abuse by a third party. We also analyzed abuse showing that the content hosted on those hijacked resources are malicious in Section 5.

4.1 Abused Organisations

Popular domains. Among the abused domains we find 8,432 popular websites from the Tranco list [15] (top 25 shown in Figure 7).

Enterprises. We find abuse in 31% of the Fortune 500 companies[2] and 25.4% of the Global 500 companies. Comparing these two lists suggests that the attackers focused on Western countries. The 25 highest-ranking Fortune 500 enterprises abused are shown in Figure 8. Many of the companies were abused more than once, hosting fraudulent content on more than one subdomain at some point in time. Figure 12 shows that the Industrial, Energy and Motor Vehicle sectors have the highest volume of hijacks, but overall the abuse is widespread rather than localized to any one sector.

Universities. We find hijacks of university domains worldwide[5] (top 25 shown in Figure 9). Between May 2020 and 2023 we found 264 abused subdomains in universities globally. University domains have good reputations and are therefore a desirable target for promoting fraudulent content.

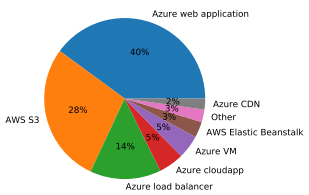


Figure 11: Cloud resources.

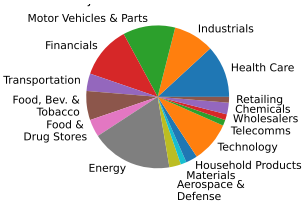


Figure 12: Abused content by sector.

4.2 Abused Providers & Resource Types

Hackers exploit a variety of cloud platforms to claim dangling records. The different services provided by each platform can be determined by the respective cloud domain suffix. Figure 11 and Table 2 show, that Microsoft Azure Cloud services are hosting more than half of the content, followed by AWS S3 static hosting and AWS Elastic Beanstalk, which together make up $\frac{1}{3}$. All other cloud providers only account for small fractions. A 2016 study showed substantially fewer dangling records pointing to Azure than AWS [12].

Cloud Resource	# Monitored	# Abuses	% Abuses
Azure Web Application	690,779	8,347	1.21
Azure VM	565,684	983	0.17
Azure Blob	20,389	-	-
AWS Elasticbeanstalk	138,523	668	0.48
Azure Traffic Manager	140,183	2,980	0.21
Azure Cloud Service	299,494	1,060	0.35
Azure API	17,100	-	-
Azure FrontDoor	14,183	-	-
Heroku App	30,532	146	0.48
Azure CDN	37,360	461	1.23
Azure Service Bus	10,152	-	-
AWS S3	1,137,613	5,876	0.52

Table 2: Abused cloud services among domains monitored.

4.3 The Problem of User-Nameable Resources

We find that the common denominator among the hijacks is *user-nameable* resources. As shown in Table 3, all hijacks we discovered exploited cloud resources that allow free choice of a text-based identifier (blue in Figure 13), enabling easy

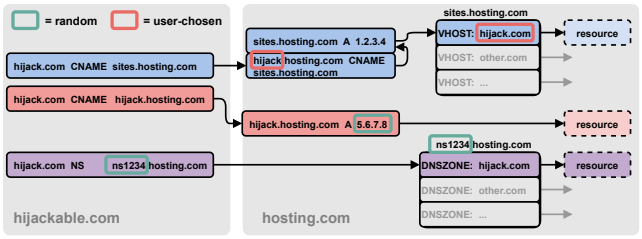


Figure 13: Hijack types: dashed lines are released resources an attacker could re-register to take over the routing and/or content of `hijackable.com`.

re-registration by an attacker.

CNAMEs & internal routing. Such user-chosen names are found at the DNS-level and the virtual hosting-level to route requests to the appropriate resource. At the DNS-level, these names are used in resource's domain, auto-generated by the cloud provider to resolve to the resource's IP address. For example, choosing the name `example` for an Azure website results in the subdomain `example.azurewebsites.net` being generated, resolving to the IP of one of the Azure servers. The server also configures its virtual hosting layer to route incoming requests for `example.azurewebsites.net` to the `example` resource (Figure 14).

Cloud providers typically also allow configuration of custom domains as aliases, so that incoming requests for, e.g., `example.com`, are also directed to the `example` resource. The customer can then conveniently set up a CNAME record directing `example.com` to `example.azurewebsites.net`, allowing the `example` resource to be resolved through `example.com` at the DNS level.

Deterministic re-registration. However, this system poses two problems. First, the resource name chosen by the legitimate user is publicly visible through the DNS record of the auto-generated CNAME. Second, this resource name can be re-registered by an attacker. An attacker who has found a DNS record pointing at `example.azurewebsites.net` can check if the `example` resource still exists and, if not, register this specific resource. As long as the CNAME record linking `example.com` to `example.azurewebsites.net` is never purged, requests for `example.com` (and any other domain in a CNAME chain to `example.azurewebsites.net`) will be hijacked by the attacker.

Randomized identifiers. One effective mitigation of these hijacks is to randomly generate the resource names, because then an attacker is not able to deterministically replicate a target resource. This would maintain the convenience of linking custom domains to the resource through a CNAME record, but provide comparable security to the random IP assignment used for cloud servers with dedicated public IPs.

Cloud resources with dedicated IP addresses are assigned

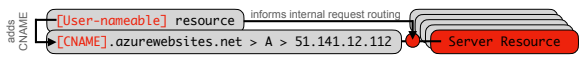


Figure 14: User-nameable resource informs CNAME and internal routing.

Provider	Configurable Sub domain name	Function	Record	Abuses
Azure	[freetext].azurewebsites.net	Web App	CNAME	6,288
Azure	[freetext].trafficmanager.net	Traffic Router	CNAME	1,468
Azure	[freetext].cloudapp.net (legacy naming)	VM	CNAME	1,037
Azure	[freetext].azureedge.net	CDN	CNAME	830
Azure	[freetext].REGION.cloudapp.azure.com	VM	CNAME	928
Azure	[freetext].sip.azurewebsites.windows.net	Web App	CNAME	223
AWS	[freetext].s3-website.REGION.amazonaws.com	Static Hosting	CNAME	2,227
AWS	[freetext].REGION.elasticbeanstalk.com	Orchestration	CNAME	555
Heroku	[freetext].herokuapp.com	Web App	CNAME	139
Pantheon	[test-|dev-|live-][freetext].pantheonsite.io	CMS	CNAME	50
Netlify	[freetext].netlify.app	Web App	CNAME	14

Table 3: Abused resources on cloud platforms with free text entry.

their IP at random from a pool available to the cloud provider (red in Figure 13). Similarly, cloud providers who offer DNS hosting distribute user-created DNS zones randomly across a range of nameservers (purple in Figure 13). When a dangling A or NS record, respectively, points at the IP of such a resource, attackers must register a similar resource repeatedly in the hope of being assigned the desired IP.

Previous work [3, 12] showed strategies to do this effectively, but it is still a probabilistic technique, which, according to our data, attackers do not pursue. Since our collection methodology (see Algorithm 1) also takes A records pointing to cloud IPs into account, such takeovers of specific IPs would also be captured in our dataset. However, we find no instances of such takeovers in our dataset, suggesting that it is not worth the effort compared to the deterministic approach possible with user-nameable resources. This is further underlined by the absence of abused Google Cloud-hosted domains, which are assigned a random subdomain, allowing no user input.

4.4 Abuse Duration

We calculate the approximate lifespan of the abused domains as the difference between the timestamp of the first HTML sample that is recognized as abused and the timestamp of the DNS record that is eventually created by the domain owner to correct the dangling vulnerability. Figure 15 shows the lifespan distribution of the domains in our dataset. A large number of abused domain names are removed within 15 days. At the same time, more than $\frac{1}{3}$ of the domains last longer than 65 days, some more than a year. This gives the attacker time to monetize content by exploiting the reputation of the abused domains. Figure 16 illustrates for each domain the time frame that it was hijacked as a horizontal line from start to end date. The domains are sorted by start date. We see an initial period of hijacks in 2020, followed by a period of relative inactivity in early 2021, and finally a ramping up of activity throughout late 2021, 2022 and 2023. The number of concurrently hijacked domains continuously increases in this period, indicating a growing problem.

5 Characterization of Abuse

In this section we first explain how the attacks with hijacked resources depend on the type of dangling resource that the attacker took over. We then report on different types of attacks we identified that have been launched from the hijacked domains in our dataset. These include SEO, malware distribution, cookie theft and fraudulent certificates.

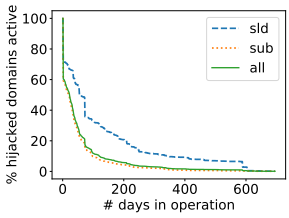


Figure 15: Hijack duration in days.

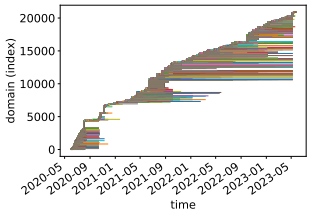


Figure 16: Hijack time frames.

5.1 Abuse Depends on Hijacked Resource

The capabilities of the adversary are dictated by the type of cloud resource used in the hijack (Table 4).

Figure 17: Attacker capabilities based on cloud resource - red indicates attacker-controlled resources, blue indicates hosting-provided resources.

With a dedicated server resource the attacker can host a full webserver (shown in Figure 17 on the right). With other resources the hosting provider may host multiple domains on a single logical webserver through virtual hosting and route to the appropriate resource by domain name. Storage resources such as AWS S3 or Content Management Systems (CMS) such as Pantheon allow control only of static content, which is read and returned by the provider's webserver (Figure 17, left), whereas with traffic management and web applications the provider's server forwards requests to a specified endpoint, where the requests can be processed in full (Figure 17, center).

In the context of cookie stealing, control of the webserver (Figure 17, center & right) affords the attacker both access to cookies in `headers` and those available via `javascript`, thus allowing access to all cookies, whereas control of just the content (Figure 17, left) only enables access to cookies accessible via `javascript`[6], in other words, cookies without the HttpOnly flag. The former also affords the `https` capability, required to access cookies with the Secure flag enabled, whereas the latter scenario does not necessarily afford this capability for the hijacked domain. [16] characterized the limitations that configurations on the victim webserver impose on attacks, e.g., the impact of HttpOnly and Secure flags in cookies or that bypassing CSP requires only `file` and `html` capabilities, while abuse of CORS, postMessage and domain relaxation also requires the `javascript` capability. In our model we show that all of these are possible from static hosting resources. However, depending on the configuration of the target, `https` may be necessary, which generally requires full webserver access to configure a certificate[7]. In addition to the configuration of related domains (shown by [16]), the type

[6] In the case of CMS, the use of JS may require the installation of additional plugins. However, since the attacker controls the resource, this is straightforward.

[7] Hosting providers may offer a dashboard option to configure a certificate, but this is not the default.

Resource	Function	Access	Capabilities
AWS S3	Storage	Static Content	file, content, html, javascript[2]
Pantheon Site	CMS		
Netlify	Web App	Full Webserver	file, content, html, javascript, headers, https
Heroku			
AWS Elastic Beanstalk			
Azure Web Application			
Azure CDN	CDN & Load Balancing		
Azure Load Balancer			
Cloudflare			
Azure VM	Server		

Table 4: Attacker capabilities based on cloud resource

of attacks that can be launched with a hijacked domain are also a function of the cloud resource that the attacker controls.

5.2 Generating Traffic

We find that the main abuse of hijacked, dangling resources is to generate traffic to adversarial services. The attackers exploit the reputation of the hijacked domains to generate page impressions to the content they control to earn money. Once they control the content, sources of income are either advertisements displayed directly on the websites hosted on the hijacked domains or referral (click-through) to another site, where they earn a small amount for each page impression, a higher amount for account registration (Figure 24 in Appendix) and even more for money spent.

Attackers use different techniques to generate traffic and increase the click-through rate to the target site that pays for the traffic. Next we describe the two techniques (SEO and clickjacking) for which we find evidence in our dataset.

5.2.1 Blackhat Search Engine Optimization (SEO)

Search Engine Optimization (SEO) is the process of improving a website's visibility in search engine results. Blackhat SEO or spamdexing[8] involves ethically questionable techniques or violates search engine guidelines. We found that 75% of HTML samples we collected contain some form of (blackhat) SEO. We determine this by manually examining a sample of 100 HTMLs in a sandboxed environment and then checking which of the other sites contain similar content based on keyword features. Specifically, we found the following techniques:

Cloaking. The *Japanese Keyword hack*[9] is one example of cloaking, a technique where content presented to search engine spiders is different from what is presented to the user. About 1% of the sites (Figure 3) featured a large number of randomly generated Japanese pages, which we categorize as the *Japanese Keyword Hack*. If hackers have the ability to add content to a site, they upload a large number of randomly named HTML pages (see Figure 6) with auto-generated Japanese content. These *cloaked* pages are served in parallel to the original site content, but shown only to crawlers, not regular users. Search engines then associate the site and its reputation with the parasitic content. Additional modifications of the `.htaccess` and `robots.txt` files in the

[8] https://en.wikipedia.org/wiki/Spamdexing

[9] https://web.dev/fixing-the-japanese-keyword-hack/

website's root directory point the crawlers to the generated spam pages and away from the legitimate content. We find that 7.17% of SEO creates private link networks and utilizes the Japanese Keyword Hack.

Private link networks. Some websites host a large number of files with the sole purpose of linking to other pages and domains , without any valuable content of their own. The reputation of incoming links contributes to the reputation of the target, so hackers create 2-way link networks across pages on hijacked subdomains, exploiting their reputation; we explain this in Section 5.2.3. In our dataset the abusers uploaded a total of about 500M files with an estimated size of 25.8TB, with an average of 31,810 HTML files per site; statistics are visualized in Figure 6.

Doorway pages. These are low-quality web pages created to rank highly in search results, but link or redirect visitors to a target page that enables monetization. Most of these doorway pages we observed were gambling-related and featured gambling content from Wikipedia. We find that 62.13% of the SEO uses doorway pages.

Keyword stuffing. This technique optimizes keywords, placing them in the content and keyword meta tag: 41% of 58353 HTML pages we analyzed contain the keyword meta tag to help make them more discoverable. Table 5 shows the top 12 keywords used.

#	Keyword	Count	#	Keyword	Count
1	slot	144,108	2	online	77,669
3	judi (gambling)	60,521	4	situs (website)	35,265
5	joker123	23,630	6	terpercaya (trusted)	19,407
7	gacor (hot streak)	18,006	8	agen (agent)	16,939
9	daftar (register)	12,881	10	game	12,113
11	bola (football)	11,688	12	pulsa (credit)	10,467

Table 5: Top 12 meta tag keywords on content hosted on hijacked domains.

5.2.2 Click-Jacking

With this technique an `onClick` event is inserted early in the event bubbling pipeline to intercept a user's mouse click on a legitimate looking hyperlink and redirect it to a malicious JavaScript function. Through our manual inspection we found this method was used on adult-related pages. Instead of navigating to the indicated page, the user is redirected to another server where ads are served.

5.2.3 Reputation of Abused Domains

We find that hijacked subdomains are valuable for hackers due to their reputation (or that of the parent domain), which often takes years to establish. Injecting fraudulent content on hijacked subdomains exploits the historic reputation of the parent domain, ranking high in search engine results. Google is the primary source of traffic to websites and accounts for 85-92% of all search engine traffic for the past 8 years [10]. As such both legitimate and fraudulent content is optimized to rank as high as possible in search results. Details of Google's ranking algorithm and weights of ranking signals are proprietary and change over time. Nevertheless, domain age, as well as other parameters discussed below, play a role in the ranking.

Domain age. Nearly all (98.51%) of the hijacked second-level domains are older than one year and the vast majority is older than a decade. This is clearly seen in Figure 18, which shows the distribution of creation dates obtained through WHOIS for second-level domains in our dataset. Since subdomains inherit their reputation from the parent, we look at the second-level domain for each subdomain.

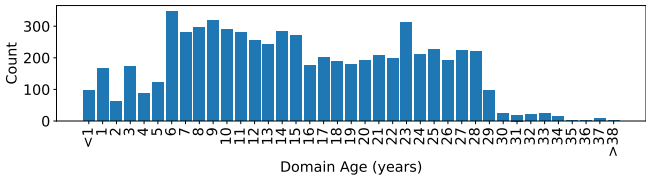


Figure 18: Domain age based on WHOIS creation date of SLDs.

Secure transport. Google also gives preference to sites that are secure. We find that 18.2% of the (sub)domains in our dataset had valid certificates.

Backlinks. Backlinks play an important role in domain selection by attackers. When other reputable websites link to a domain, Google sees this as a sign of trust and authority. This is especially true if the linking sites are themselves high-reputation domains.

5.3 More Page Views Mean More Profit

In summary, scammers leverage the reputation inherited from the parent domain to drive search traffic to fraudulent content on the hijacked subdomain. Since search engines are the primary source of traffic, scammers optimize SEO signals to boost their pages' rankings.

In Appendix, Figure 24 we show a screenshot of the traffic accounting infrastructure of a gambling site that receives traffic from the hijacked domains. The resources behind the hijacked domains are used to relay traffic to the sign-up screen in Figure 24. In this infrastructure a referral code is passed from the content hosted on a hijacked domain to the gambling website. The website pays for traffic based on the referral ID attached to incoming requests, thereby paying the hijackers for each page view, each account sign-up, and money spent on the site. The presence of the referral ID also suggests that website owners and domain hijackers are two different entities. In this abuse we witness an ecosystem, involving multiple entities creating revenue from the hijacked resources.

In summary, we find referral links, a large focus on SEO, ads and gambling, hijacking of cloud resources exclusively with user-chosen names requiring particularly low effort to hijack, and a lack of differentiation of abuse content across a wide range of SLDs (see Section 6). We view this to be strong evidence for a financial motive as the driver for these hijacks.

5.4 Malware Distribution & Flagged Sites

We find almost no evidence of malware distribution, which was considered to be one of the main threats of dangling

[10] https://www.statista.com/statistics/216573/worldwide-market-share-of-search-engines/

records in previous work [16]. Since this is the first work to find exploitation of dangling domains, there is no comparable dataset in previous work. We scanned 58,353 samples of HTML index pages for downloadable executable files for all operating systems, to determine, if hijacked domains are directly used for the distribution of malware or other software and which operating systems are targeted.

VirusTotal executables analysis. We retrieved 2628 binaries using cURL. Among these were 181 unique Android apps (.apk) and a single Windows executable (.exe). Only 2 EXEs were labeled as Trojans by VirusTotal[11]. The vast majority of APKs were gambling apps corresponding to the gambling sites. Our data suggests that the hijacked domains are *not* predominantly used for large-scale malware distribution.

VirusTotal domains analysis. VirusTotal also allows checking if domains are flagged by various antivirus vendors. We queried VirusTotal for our dataset of hijacked domains: only 135 were flagged by at least one vendor, only 18 by two or more. Figure 19 plots these values over time (based on the earliest certificate issuance found for that domain), suggesting that widespread blacklisting takes at least 2 years. Whether due to lack of traffic to these domains, the slow inclusion of domains in blacklists or a general disinterest of AV vendors to include hijacked domains, the small percentage of blocked domains suggests that blacklisting does not effectively protect clients from being served content from hijacked domains.

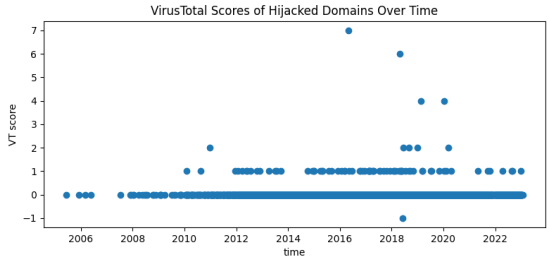


Figure 19: VirusTotal blacklist counts for hijacked domains by date of first certificate issuance.

5.5 Stolen (Authentication) Cookies

As pointed out by [12,16], one of the threats of gaining control of a subdomain's content (e.g., access to a CMS) and/or an entire webserver, is access to cookies. Cookies are used for tracking users across pages via a session identifier, as well as storing authentication tokens which tell the server that a user has previously logged in. As a consequence, stealing authentication cookies enables full access to the website as a logged-in user. This sensitive data can be stolen as they visit the legitimate-looking, but adversary-controlled content or webserver on a hijacked subdomain.

Cookie policies implemented by browsers are such that a cookie is only sent back to the domain that created it, or a subdomain thereof. Cookie access depends on the degree of control an adversary has of a hijacked domain. With full server access, they can read all cookies (including authentication), but if they are only able to inject content onto a running webserver (of a CMS), they are only able to read a subset of browser-accessible cookies: cookies with the `HttpOnly` attribute set to "false" or not set at all (the default).

Stolen cookies of abused domains. Since there is no way to tell if cookies are being actively exfiltrated from HTTP headers on the server, we look for stolen authentication cookies for sale. In a collaboration with a threat intelligence organization we identify 83 unique authentication cookies detected in darknet leaks in the timeframe in which the corresponding dangling domains were detected by us as hijacked. These cookies are linked to 3 different hijacked subdomains originating from 53 unique IP addresses. The low number of leaked authentication cookies we found is not surprising, due to their relatively short expiration time.

5.6 Fraudulent Certificates

In this section we describe our analysis of the fraudulent certificates we found on hijacked domains. Due to availability of HTTP-based Domain Validation, hijackers are in a position to obtain a valid certificate, simply by having access to the webserver root. In fact, many hosting providers, such as Azure, integrate functionality within their dashboards to issue certificates for custom domains that point to the hosting resource. Once a hijacked subdomain is taken over, obtaining a valid certificate is trivial. We can therefore expect hijackers to use certificates, as it allows the use of HTTPS, increasing the efficacy of various attacks. One of the more critical possible abuses for domain hijacks is access to cookies. As detailed in Section 5.5, subdomains will typically receive not just cookies set for that subdomain, but those for the parent domains as well. However, if a cookie has the "Secure" flag set, it will only be sent via HTTPS. So attackers looking to steal, e.g., secure authentication cookies would need to setup HTTPS with a valid certificate for the hijacked subdomain. An interested reader is referred to Section A.2 in Appendix for explanation on different motivations for obtaining a valid but fraudulent certificate for abused resources.

5.6.1 Analysis of Certificates on Abused Domains

Certificate Transparency (CT) logs publicly list all certificates issued by CAs. We use CT to analyze the entire timeline of certificates for all domains in our dataset, looking for anomalies. Across CT history for our dataset we find 24239 single-SAN (single-Subject Alternative Name) certificates and 41877 multi-SAN/wildcard certificates. Since during domain validation hijackers can typically only successfully prove control of a single subdomain that they control, we search the CT history of hijacked subdomains for certificates that contain only a single, non-wildcard subdomain name.

We contrast this set with the certificates issued for these subdomains (Figure 20). We see two distinct time frames (2017-07-31 to 2017-08-14 and 2022-09-09 to 2022-12-16) where a significant number of certificates were individually

[11] https://www.virustotal.com

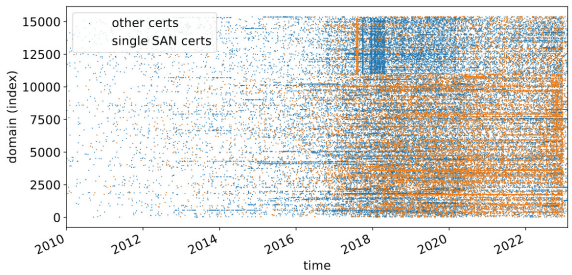


Figure 20: Multi-SAN vs. single-SAN certs issued for hijacked subdomains.

issued across our set of hijacked subdomains. Particularly the former shows a clear anomaly, covering a significant number of "www." subdomains. Taking a closer look at these time frames, we find that the majority of the single-subdomain certificates (95% and 53%, respectively, for each time frame) were issued by Let'sEncrypt, suggesting that these were intentionally issued, as opposed to automatically by a hosting provider. A correlation with our dataset of the abused domains shows that the time frames correspond to campaigns to gather vulnerable subdomains. Based on the dates, we trace this kind of activity back to mid 2017, about a year after the first paper on dangling DNS records was published [12].

5.6.2 CAA Records are not Effective

Hijackers have many options for obtaining a certificate. Cloud providers, such as Azure and AWS, often run their own CAs and provide certificate issuance built into their hosting products, free of charge, which can be leveraged by anyone in control of the hosted resource. Let's Encrypt and ZeroSSL are two additional CAs that provide certificates via domain validation and at no cost.

It has been suggested that domain owners could configure CAA records to allow certificate issuance only by an authorized set of CAs, in order to prevent unauthorized issuance. This is unlikely to be effective, however, since an attacker can simply register an account with one of the authorized CAs and still issue a certificate. This kind of restriction would only be protective if certificate issuance is restricted to a specific *account* with a specific CA ("domain locking"[12]).

In can be argued that, in the case of cyber-criminals gathering large numbers of domains to use for SEO and traffic generation, authorizing only CAs who charge for certificates (i.e., unauthorizing all CAs with free certificates) might disincentivize attackers from issuing certificates at scale, due to an increase in cost. This would, however, necessitate the majority of domain owners switching to a paid CA and setting the appropriate CAA records, in order to function as a deterrent.

This is unlikely to happen due to cost to the legitimate owners. Hence, we find that only 2% of parent domains (and only 0.2% of subdomains) have a CAA record set and only 0.4% (0.01%) specify a CA without free certificate issuance. We also find that half of these domains still had hijacked subdomains with valid certificates. This suggests that CAA records are not a suitable countermeasure for such attacks.

[12] https://docs.digicert.com/en/certcentral/manage-certificates/organization-and-domain-management/domain-locking.html#locking-a-domain

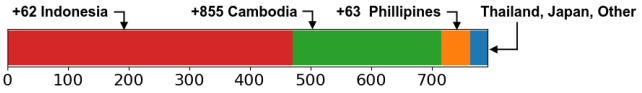


Figure 21: Geo-distribution of phone numbers based on country code.

5.6.3 Certificate Transparency as Countermeasure

CT can be leveraged as a much more effective countermeasure than CAA records. Though reactive, rather than preventative, CT monitoring of one's domain via third-party services is a low-to-zero cost, set-and-forget measure to ensure one is notified whenever a certificate is issued for the domain or one of its subdomains. Should an attacker take over even a long-forgotten subdomain and issue a certificate, an alert is triggered and the domain owner is made aware of the hijack, typically within a few hours. However, the effectiveness of detection rests on the attacker's choice to obtain a certificate.

6 Characterization of Attacker Infrastructure

In this section we characterize the infrastructure used in the hijacks in our abuse dataset. We look for indicators from the infrastructure and the user-facing content side. Due to the large number of hijacks we combine manual analysis of a subset of examples with automated keyword-based approaches.

Hijacked webserver software. Using the Generator Meta tag, we identified that about 22% of the 54,325 collected HTML samples were homepages of WordPress blogs. These are custom WordPress installations on cloud.

Static identifiers. Next we considered other backend links present in the HTML samples. We analyzed the `href` attributes of `<a>` and `<link>` tags contained in the HTML documents served by the hijacked domains. We discovered 792 unique phone numbers through WhatsApp links. Based on the country code, we can see that all of them are based in Asia, primarily Indonesia and Cambodia; see Figure 21. We also discovered 1,884 unique contacts in the form of Telegram, Twitter, Instagram and Facebook accounts/channels/groups and direct chat IDs, as well as 2,671 unique forwarding links provided by URL shortening services. We also discovered 3,553 unique references to IP addresses. Based on WHOIS data, we find that the majority of these IPs belong to various hosting providers (Figure 26 (a) in Appendix). This also matches the concentration of IPs in the US, France and Singapore, based on GeoIP information (Figure 26 (b) in Appendix). The use of cloud hosting does not allow identification of the countries where attackers are operating.

Clustering attacking infrastructure. All of the above data points serve as identifiers. If two identifiers appear on the same HTML page, it is reasonable to assume that they are associated with the same operation. Thus, we cluster these data points based on how many HTML pages of hijacked domains each pair appears on. The resulting relationships between identifiers are visualized as a network graph in Figure 27 in the Appendix. The network graph provides a broad

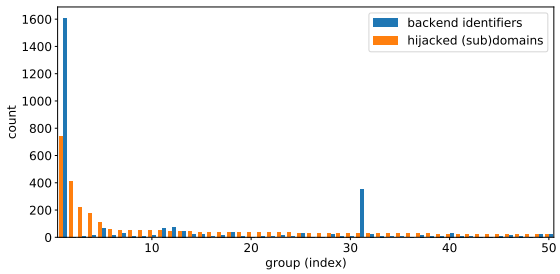


Figure 22: Top 50 clusters sorted by number of hijacked domains.

overview. To delineate concrete groupings we find all nodes that are connected through some path in the network graph. This is accomplished by successive hierarchical clustering, plotted in Figure 28 in Appendix.

Each tick on the x-axis represents an identifier, while the y-axis indicates the distance (on a scale of 0 to 1) at which identifiers are grouped. The minimum distance of 0 indicates that a pair of identifiers are associated with an identical set of hijacked domains and the maximum distance of 1 indicates that two identifiers share no domains at all. The dendrogram displays all groupings up to a cutoff point at a distance of 0.95 (since at the end of the hierarchical clustering process all nodes are merged into one). The cutoff is chosen to achieve the maximal degree of grouping. This is warranted, since the probability of an identifier appearing on two domains by coincidence is very small. Groupings are delineated by color, so all adjacent vertical lines of the same color point to identifiers that have been grouped; the largest grouping is displayed in gray on the right side of the dendrogram. Long, single-color vertical lines indicate identifiers that do not share any domains with other identifiers and thus cannot be grouped.

The hierarchical clustering results in 1,798 clusters, with the vast majority consisting of 1 or 2 identifiers which could not be linked to any others. For the largest grouping, however, it was possible to tie together 1,609 identifiers, associated with 743 hijacked domains. The four next-largest clusters contain 414, 222, 179 and 112 domains, respectively. Figure 22 shows this long-tailed clustering result for the top 50 clusters, sorted by the number of hijacked domains in each cluster. The identifiers cover 8,489 ($\sim\frac{1}{3}$) of the hijacked domains and all are associated with Indonesian gambling.

While the clustering still leaves the possibility for a large number of actors involved, it ties large sets of domains together, showing that at least some actors collect a wide range of diverse domains in a coordinated effort. These are then homogeneously used for the same purpose of referring traffic or manipulating search rankings. The lack of differentiation in how the hijacked domains are abused suggests an attempt at maximizing profit by maximizing the number of domains recruited for a campaign, as opposed to specific targeting of individual domains, e.g., for political reasons.

Our analyses do not yet allow conclusions as to the noticeable bias of observed page content, linked infrastructure, discovered APKs and phone numbers toward Indonesia. However, because our discovery of hijacks begins with a set of domains based on global organizations (e.g., Alexa 1M, Fortune 1000, etc.), there is little reason to assume a biased dataset. Instead, we see a possible explanation in the population size (4th largest in the world) and strict illegality of gambling in Indonesia, leading to a prevalence of online gambling and a need to advertise it through illicit means.

7 Conclusions

Although the threat of take-over of dangling resources in the cloud was explored, there was no evidence in research of real-life abuse. We explore this question empirically with a longitudinal three-year analysis of cloud-hosted resources. Our two key contributions are a methodology for detecting abuse and a longitudinal dataset of abused resources at scale. Our research shows that the abuse of dangling resources on cloud platforms is a real problem that affects a large number of victims in popular and established organizations across different sectors. Our methodology and the findings provide a feasibility proof for identifying abused resources without assuming control over the cloud platform.

Based on our analysis we derive lessons for improving visibility of abuse and developing countermeasures. The hijacks we found show that the attackers target released resources that (1) are cheap and (2) can be directly determined by entering freetext, while avoiding resources that are expensive and require effort to obtain, such as the lottery-based IP assignment from a pool of IP addresses. Therefore, as an easy-to-deploy mitigation we recommend that cloud platforms either do not allow user-created resource names to be publicly visible (e.g., through DNS records) and/or disallow the re-registration of recently released resource names. We also recommend, similarly to previous work, to purge stale DNS records. In addition, cloud platforms should keep track of released resources using our methodology and alert owners of registered domains about changes to the content or sitemap. Since we observe that attackers issue certificates for hijacked domains, we recommend that cloud providers also monitor CT logs for unusual patterns across domains hosted on their platforms to help detect potential large-scale abuse campaigns.

Finally, we point out that although our work focuses on resources on cloud platforms, our results can be used to identify abuse in other third-party services. For instance, while Content Management Systems (CMS) like Wordpress are not included in our dataset, we expect a large number of hijacks of `[freetext].wordpress.com` subdomains, since Wordpress also implements freetext subdomain registration for its blogs.

Acknowledgements

This work has been co-funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) SFB 1119.

References

[1] Eihal Alowaisheq, Siyuan Tang, Zhihao Wang, Fatemah Alharbi, Xiaojing Liao, and XiaoFeng Wang. Zombie awakening: Stealthy hijacking of active domains through dns hosting referral. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1307–1322, 2020.

[2] Omar Alrawi, Charles Lever, Kevin Valakuzhy, Ryan Court, Kevin Z Snow, Fabian Monrose, and Manos Antonakakis. The circle of life: A large-scale study of the iot malware lifecycle. In *USENIX Security Symposium*, pages 3505–3522, 2021.

[3] Kevin Borgolte, Tobias Fiebig, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Cloud strife: Mitigating the security risks of domain-validated certificates. In *Internet Society Symposium on Network and Distributed System Security (NDSS)*, 2018.

[4] European Commission, Content Directorate-General for Communications Networks, Technology, I Paulovics, A Duda, and M Korczynski. *Study on Domain Name System (DNS) abuse*. Publications Office of the European Union, 2022.

[5] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 95–110, 2014.

[6] Zakir Durumeric, Michael Bailey, and J Alex Halderman. An internet-wide view of internet-wide scanning. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 65–78, 2014.

[7] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In *Usenix Security*, volume 2013, 2013.

[8] Liz Izhikevich, Renata Teixeira, and Zakir Durumeric. Lzr: Identifying unexpected internet services. In *USENIX Security Symposium*, pages 3111–3128, 2021.

[9] Doowon Kim, Haehyun Cho, Yonghwi Kwon, Adam Doupé, Sooel Son, Gail-Joon Ahn, and Tudor Dumitras. Security analysis on practices of certificate authorities in the https phishing ecosystem. *ASIA CCS '21: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 407–420, May 2021.

[10] Deepak Kumar, Kelly Shen, Benton Case, Deepali Garg, Galina Alperovich, Dmitry Kuznetsov, Dmitry Kuznetsov, Rajarshi Gupta, and Zakir Durumeric. All things considered: an analysis of iot devices on home networks. In *Proceedings of the 28th USENIX Conference on Security Symposium*, pages 1169–1185, 2019.

[11] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS 2019, February 2019.

[12] Daiping Liu, Shuai Hao, and Haining Wang. All your dns records point to us: Understanding the security threats of dangling dns records. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1414–1425, 2016.

[13] Yang Liu, Armin Sarabi, Jing Zhang, Parinaz Naghizadeh, Manish Karir, Michael Bailey, and Mingyan Liu. Cloudy with a chance of breach: Forecasting cyber security incidents. In *24th USENIX security symposium (USENIX Security 15)*, pages 1009–1024, 2015.

[14] Craig Partridge and Mark Allman. Ethical considerations in network measurement papers. *Communications of the ACM*, 59(10):58–64, 2016.

[15] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. *arXiv preprint arXiv:1806.01156*, 2018.

[16] Marco Squarcina, Mauro Tempesta, Lorenzo Veronese, Stefano Calzavara, and Matteo Maffei. Can i take your subdomain? exploring same-site attacks in the modern web. In *USENIX Security Symposium*, pages 2917–2934, 2021.

[17] Benjamin VanderSloot, Johanna Amann, Matthew Bernhard, Zakir Durumeric, Michael Bailey, and J Alex Halderman. Towards a complete view of the certificate ecosystem. In *Proceedings of the 2016 Internet Measurement Conference*, pages 543–549, 2016.

[18] Mingming Zhang, Xiang Li, Baojun Liu, Jianyu Lu, Yiming Zhang, Jianjun Chen, Haixin Duan, Shuang Hao, and Xiaofeng Zheng. Detecting and measuring security risks of hosting-based dangling domains. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(1):1–28, 2023.

TLD	Count	TLD	Count
1. com	12942	7. de	758
2. org	1069	8. edu	414
3. net	996	9. ca	398
4. uk	758	10. nl	207
5. au	414	11. jp	183
6. br	398	12. co	156

Table 6: Top 12 Top Level Domains (from a total of 218) and their counts.

A Appendix

A.1 Cloud Suffixes

We compiled the list of cloud suffixes using the following sources:

`https://docs.aws.amazon.com/general/latest/gr/rande.html`
`https://learn.microsoft.com/en-us/azure/security/fundamentals/azure-domains`
`https://docs.netlify.com/domains-https/custom-domains/`
`https://docs.pantheon.io/guides/domains/platform-domains`
`https://devcenter.heroku.com/articles/custom-domains`
`https://infogalactic.com/info/List_of_Google_domains`.

Cloud IP ranges were obtained from these provider published sources:

`https://ip-ranges.amazonaws.com/ip-ranges.json`
`https://www.microsoft.com/en-us/download/details.aspx?id=56519`
`https://www.gstatic.com/ipranges/cloud.json`

Based on recent market share data, the AWS, Azure and Google clouds cover 65% of the hosting market, with the rest split across a long tail of providers. Our cloud identification is therefore not 100% complete, but covers a large majority share of the cloud market.

A.2 Motivations for Obtaining a Certificate for a Hijacked Domain

Browser Warnings. Most popular browsers display a warning when attempting to connect to a domain with a self-signed, expired or otherwise invalid certificate. Some end-users might ignore this warning, but many won't. Obtaining and using a valid certificate would remove this barrier and thus increase traffic to the site.

Similarly, browser UI indicators (i.e. green lock icon) increase the users' trust in the legitimacy of the site. It has been shown that phishing sites using HTTPS are more effective [9]. Thus, a certificate for the hijacked site likely serves to increase user interaction.

SEO. The primary type of abuse seen on hijacked domains is SEO spam. A key parameter in search engine rankings is the use of HTTPS. Sites that don't use HTTPS are typically ranked lower. Thus, a certificate would help boost the efficacy of SEO spam.

HSTS. If a hijacked subdomain (or its parent) has previously added the domain to the HSTS list of visiting clients, these clients will only connect to this domain via HTTPS in the future (until expiration of the HSTS setting). Hijackers wishing to capture this traffic, will need to serve HTTPS connections and will thus require a certificate. We queried 1,323 parent domains from our hijacked dataset and found over 16% of non-error responses contained an HSTS header.

Secure Cookies. One of the more critical possible abuses for domain hijacks is access to cookies. As detailed in Section 5.5, subdomains will typically receive not just cookies set for that subdomain, but those for the parent domains as well. However, if a cookie has the "Secure" flag set, it will only be sent via HTTPS. So attackers looking to steal, e.g., secure authentication cookies would need to setup HTTPS with a valid certificate for the hijacked subdomain.

Furthermore, the SameSite flag differentiates HTTPS and HTTP origins. In particular, SameSite=None requires Secure=True, increasing the likelihood that a domain uses secure authentication cookies[13].

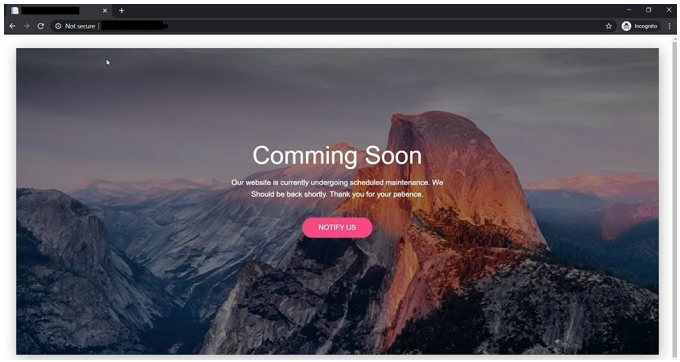


Figure 23: Main page for a Fortune 500's abused site with an error message (May 2020).

[13] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite

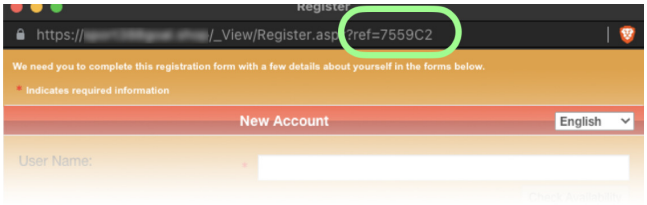


Figure 24: Referral code is passed for traffic accounting.

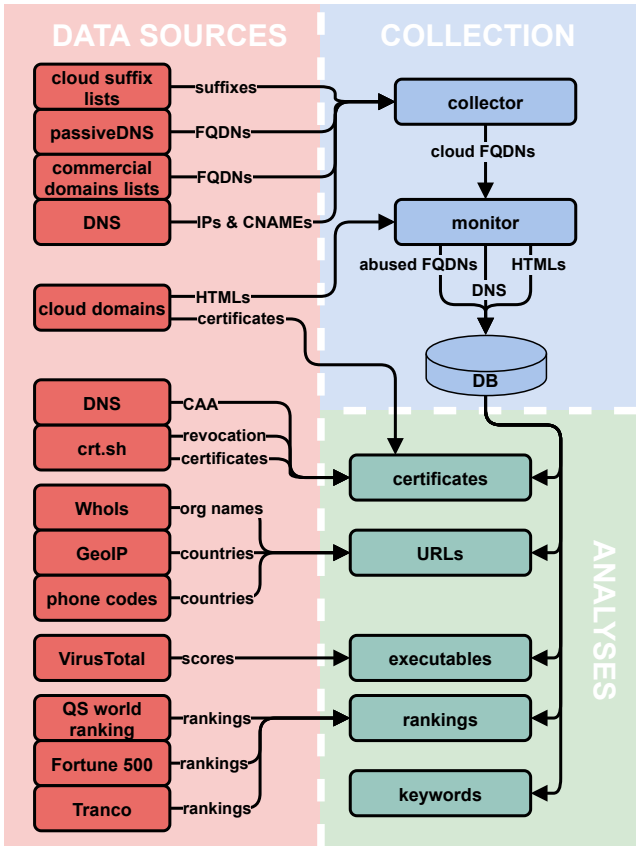


Figure 25: Overview of data collection and analysis process.

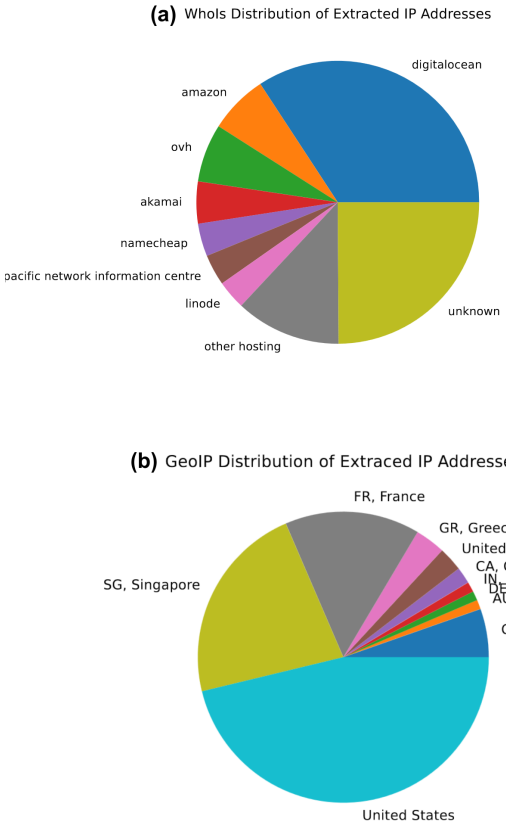


Figure 26: **(a)** Organizations associated with extracted IPs based on WHOIS data. **(b)** Geographical distribution of extracted IPs based on GeoIP data.

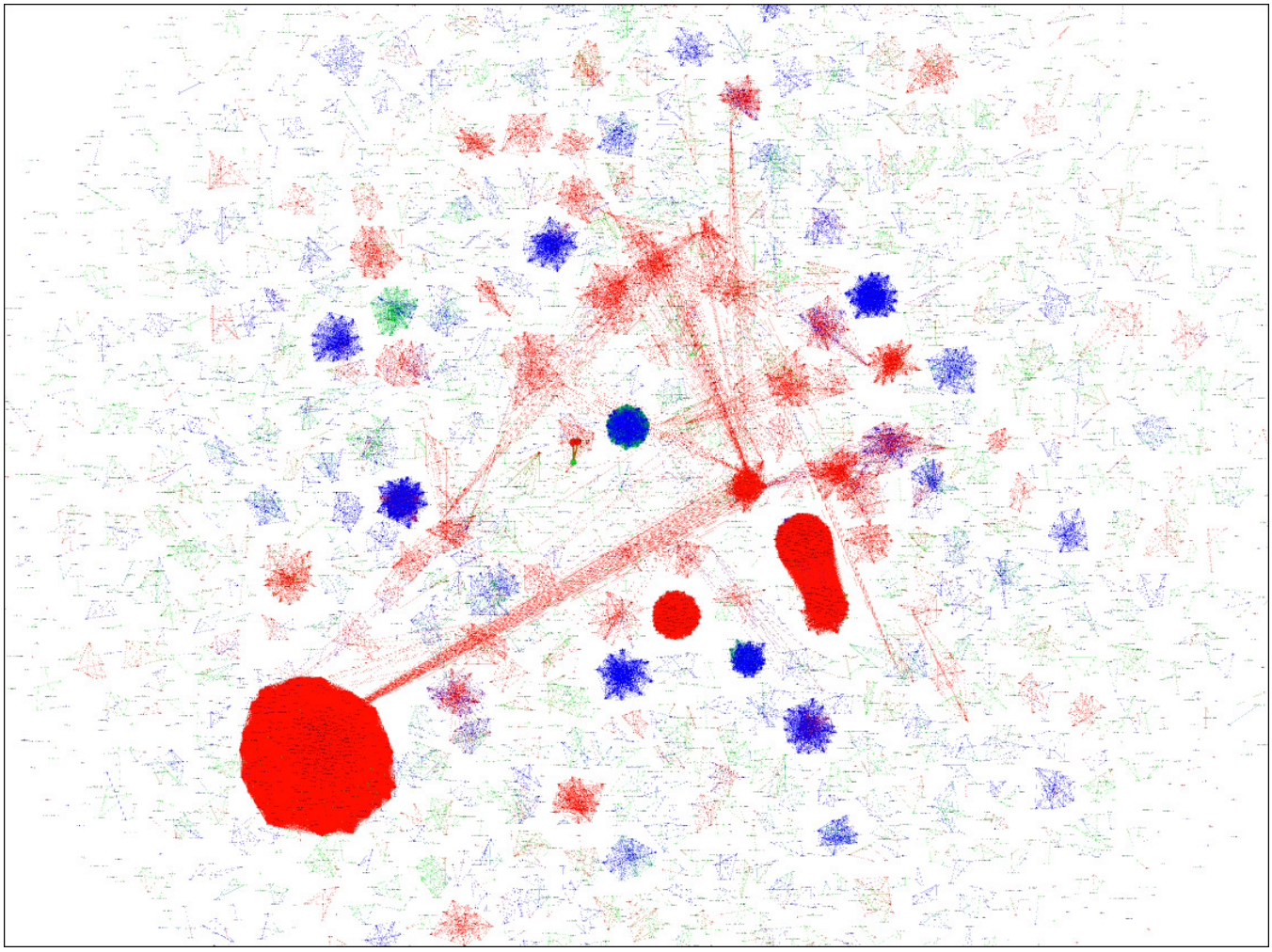


Figure 27: Clustering of extracted identifiers - IPs (red), contact information such as phone numbers, social media channels and chat links (green), and URL shortener links (blue). Node size indicates the number of hijacked domains associated with the identifier, edge thickness indicates the number of shared domains between a pair of identifiers.

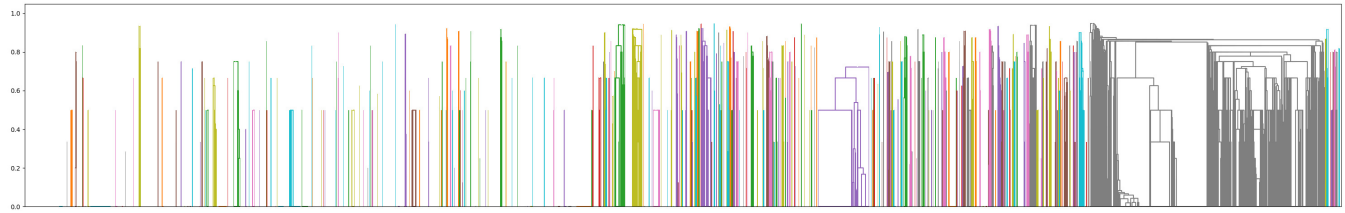


Figure 28: Dendrogram of identifier clustering.

sex	porn
daftar	FAFA800
พนัน	การพนัน
sbobet	DominoQQ
gacor	Rondo Royal
Our website is currently undergoing scheduled maintenance	adult videos and photos</title>
© 2020 <a href="http://	ite checkserverwork.ru just created.
 | <a href="	>SORRY!
<body class="stretched device-xl has-plugin-fitvids has-plugin-easing has-plugin-bootstrap">	The site was created successfully and this page is automatically generated by the system.
The page cannot be displayed because an internal server error has occurred.	We're working to restore all services as soon as possible. Please check back soon
ust another WordPress site	Comming Soon
This page is generated by Softaculous Webuzo, a leading provider of software for the webhosting industry.	Use deployment center to get code published from your client or setup continuous deployment.
Unsere Website wird derzeit planmäßig gewartet	<h3 class="has-text-align-left">fathner situs :</h3>
xinslot	judi dominoqq
situs slot	บาคาร่าออนไลน์
jili slot	fathner :
content='fafa800'	aftar slot
สล็อตออนไลน์	pkv games
judi bola	acor 77
<meta name='author' content='fafa	oto macau
nxdomain 0-_-0	acor 96
jadwal bola	kun demo slot
situs judi	แหล่งเกมที่นักพนัน
#daftarisi a	gacor situs
aftar-judi	live draw sdy
bobetmobile	udi slot online
aftar agen pkv	itus togel
aftar togel	aftar sistus
selamat datang di wordpress! ini adalah pos pertama anda. sunting atau hapus pos tersebut sebagai langkah pertama dalam perjalanan blogging anda.	style="position:fixed; top:0px; left:0px; bottom:0px; right:0px; width:100%; height:100%; border:none; margin:0; padding:0; scrolling= auto overflow:hidden; z-index:999999;"></object>
aftar judiitus gacor	slot gacor

slotxo	aftar sbobet
aftar situs	ominoqq online
ページディレクトリ | <a href="http	<meta property='og:title' content='casino online
.ru/trd"></script>	|<a href="
dzeroki.ru	.win/pops"></script>
>Sitemap | <a href=	/sitemap.xml">ページディレクトリ | <a
document.cookie = "kxxxf=" + keyfin	.ru/push"></script>
{\n "@context": "https://schema.org/",\n "@type": "Product",	著作権 © 2020 日本の無料ブログ．全著作権所有.
onedragon.win/trd	<script type="application/ld+json">
当社のウェブサイトは現在メンテナンス中です	Unsere Website wird derzeit
.win/trd"></script>	.ru/js/popunder.js"></script>
<script type="text/javascript">$(function() {("img.lazy").lazyload({threshold : 200});});</script>	style='color:white;background-color:#555555'> <script type='text/javascript'>setTimeout(function(){{var s=document.createElement('script');
<title>Top adult videos and photos</title>	<meta name="description" content="xxx porn images found for on
يخضع موقعنا حاليًا للصيانة المجدولة	">1 | <a href="
</body>	Как вы здесь оказались? Для перенаправления запросов локального IP-адреса к нужному вам локальному домену создайте алиас в настройках программы.
<!--тут код счетчика-->	.win/js/popunder.js"></script>
onerror="this.src='https://i.imgur.com/	.ru/cry"></script>
<label for="first-name">Your best email address:</label>	Your request did not find this site in the web server!
The IP address for this domain may have changed recently. Check your DNS settings to verify that the domain is set up correctly. It may take 8-24 hours for DNS changes to propagate.	<div class="main-area center-text" style="background-image:url(images/
<title>Как вы здесь оказались?</title>	.ru/dtt"></script>
/sitemap.xml">sitemap | <a	.ru/js/pops.js"></script>
現在作成中です	">脱出 ゲーム sign 攻略
.onlyfuns.win/	"># | <a href="
<a style="text-decoration: none; color: white;" href="	/sitemap.xml">Sitemap | <a
สล็อตแตกง่าย	

Figure 29: List of extracted keywords and code fragments.

CAPA: An Architecture For Operating Cluster Networks With High Availability

Bingzhe Liu* Colin Scott† Mukarram Tariq† Andrew Ferguson† Phillipa Gill† Richard Alimi†
Omid Alipourfard† Deepak Arulkannan† Virginia Jean Beauregard† Patrick Conner†
P. Brighten Godfrey* Xander Lin† Joon Ong† Mayur Patel† Amr Sabaa† Arjun Singh†
Alex Smirnov† Manish Verma† Prerepa V Viswanadham† Amin Vahdat†
{capa-nsdi-paper@google.com}

†Google *UIUC

Abstract

Management operations are a major source of outages for networks. A number of best practices designed to reduce and mitigate such outages are well known, but their enforcement has been challenging, leaving the network vulnerable to inadvertent mistakes and gaps which repeatedly result in outages. We present our experiences with CAPA, Google's "containment and prevention architecture" for regulating management operations on our cluster networking fleet. Our goal with CAPA is to limit the systems where strict adherence to best practices is required, so that availability of the network is not dependent on the good intentions of every engineer and operator. We enumerate the features of CAPA which we have found to be necessary to effectively enforce best practices within a thin "regulation" layer. We evaluate CAPA based on case studies of outages prevented, counter-factual analysis of past incidents, and known limitations. Management-plane-related outages have substantially reduced both in frequency and severity, with a 82% reduction in cumulative duration of incidents normalized to fleet size over five years.

1 Introduction

Cloud applications require high availability from cloud infrastructure. Application deployment patterns vary from non-replicated (single-zone) services to multi-regional replicated services designed for 24/7 global availability [7]. To accommodate this range of deployment patterns, cloud infrastructure must ensure *high baseline availability* within a zone, and *failure domain independence* across zones. Networking, as a baseline dependency for other infrastructure and services, is particularly critical for availability.

Despite the criticality of networking, single-zone and correlated multi-zone network failures take place frequently (*e.g.,* [6, 46, 71] in March 2022 alone). At Google, we find that $\geq 58\%$ of cluster (datacenter) network outages since 2018 result from management operations. Previous research has recommended a number of best practices for executing operations in order to improve availability [1, 8, 25]. These recommendations include defining failure domain boundaries, ensuring progressive and supervised change rollout, defense-in-depth, consistency across planes, invariant monitoring, etc. While these best practices are well understood, major cloud providers continue to suffer outages that in hindsight could have been prevented by application of these practices.

In our experience managing Google's fleet of cluster networks, several challenges make it difficult for engineers and operators to consistently follow best practices: **(C1) Complexity and diversity of system interactions.** Cluster networking has evolved into a complex distributed system of many interdependent subsystems. The cross product of the system's behavioral axes, the variety of operations it supports, and the range of services that perform operations result in a huge surface area where best practices might need to be adhered to. **(C2) Difficulty isolating failure domains.** Failure domain isolation reduces correlated failures. But cluster networks are fundamentally coupled as they must connect to each other and exchange routing messages through a wide area network (WAN) [29, 30]. Additionally, centralized operational teams and systems with global responsibilities create coupling through their day-to-day operations. **(C3) Need to balance safety vs. operational velocity.** Aggressive rate limiting can create a logjam of business-critical network updates. In practice, operators must resort to some degree of concurrency to keep up. Moreover, certain issues might only be triggered in later stages such that progressive rollout alone is not sufficient. **(C4) Software development velocity overhead.** Strict adherence to best practices comes with high development costs, particularly for a foundational service like cluster networking that cannot always make use of common implementations of convenient abstractions (e.g., distributed storage). Mandating a high-level of rigor for best practice adherence across the dozens of teams and hundreds of engineers that maintain individual subcomponents is at odds with the desire to maintain high software development velocity.

In this paper, we ask: is it feasible to limit the systems where strict adherence to best practices is required, so that availability of the network is not dependent on the best in-

tentions of every engineer and operator? We describe our experiences with CAPA, the "containment and prevention architecture" we have designed and implemented with the goal of systematically enforcing best practices while avoiding the need for all subsystems to be encumbered by enforcement. We enumerate all of the features we have found through experience to be necessary in order to achieve our goal.

Following [33], CAPA introduces a new *regulation layer* which separates the (preexisting) *production critical layer* below from the (preexisting) *operations workflow layer* above. CAPA defines and enforces rules for each layer.

- The *production critical layer* encompasses the network control and data planes responsible for handling application payload. The components in this layer adhere to strict failure domain containment.
- The *operations workflow layer* contains capacity planning, modeling, and rollout workflows that produce behavioral intents for the production layer based on business needs. The services in this layer make no changes to the production critical layer directly, and instead proxy intended changes via the regulation layer.
- The *regulation layer* initiates behavioral changes to components in the production critical layer, and modulates changes by providing rate limiting, health supervision, and fast rollback of certain operations.

Going well beyond [33], we describe in detail throughout the paper how CAPA engages with challenges C1-C4. To address the complexity of system interactions (C1) CAPA's regulation layer (i) ensures that management operations first 'drain' (move traffic away from) the data plane entities they are about to mutate as a mitigation against unexpected side effects, and (ii) enforces *protected cross-section bandwidth*, a novel invariant that ensures 'drain' requests (and therefore operations per (i)) are blocked until there is enough available capacity to safely accommodate the operation (§3.1, §4.1).

To prevent correlated failures across zones (C2), the regulation layer rate-limits management operations across the fleet to prevent them from concurrently affecting too many zones, and the production critical layer ensures via ACL enforcement that outbound control messages originating from within a zone are disallowed thereby limiting blast radius and ensuring "low-dependency" bootstrap capability (§4.1, §4.2). One of the few unavoidable exceptions for outgoing ACL enforcement are BGP routing updates, which are needed because cluster networks must maintain connectivity via the WAN. To mitigate correlated failures that propagate through BGP, we have developed *fail-static* fallback routing that allows cluster networks to maintain WAN connectivity even when dynamic BGP routing fails (§4.3). When dynamic BGP routing fails, our border routers continue forwarding packets through pre-programmed static routes to existing traffic engineering tunnels in the data plane. Although these backup routes may be less optimal, many packets will continue arriving at their destination thanks to end-host repathing mechanisms [52, 65] as long as a working path exists in the data plane.

Although CAPA does not directly change the parameters of the safety vs. operational velocity tradeoff (C3), its regulation layer provides a central place where policies such as protected cross-section bandwidth and maximum concurrency can be adjusted (§4.1). Centralized policy definition reduces friction in the ongoing tussle between safety stakeholders and operational velocity stakeholders within the company [12].

In CAPA, only systems in the production critical layer have strict failure domain isolation requirements. Developers at the operations workflow layer are unhindered by these requirements (C4), e.g., they can rely on high-level abstractions and build globally scoped systems which are easier to build and reason about [43]. Since all outputs from the workflow layer are filtered through the regulation layer before propagating to the production critical layer, we largely avoid coupling between globally scoped workflow layer systems and the carefully contained failure domains in the production layer.

CAPA has helped us make considerable progress towards systematic enforcement of production best practices, which improves both baseline availability and failure domain independence. We perform a qualitative retrospective analysis of relevant outages (including before CAPA's deployment) to show that 84% could have been prevented or mitigated by CAPA. Moreover, our fleet's annual incident rate has decreased by more than 73% from pre- to post-deployment. As CAPA was our primary investment in reliability improvements, given the substantial improvements and a number of "near-miss" success stories, it is fair to attribute most of this improvement to the design and deployment of CAPA. At the same time, our work is not complete; we characterize incidents where CAPA did not help, highlighting key limitations.

2 Background

2.1 Cloud deployment patterns

The grouping of servers connected by a datacenter fabric is called a *cluster*, and we therefore refer to the datacenter network fabric as a *cluster network*. Compute resources are presented as *zones* ($\geq$ 1 co-located clusters) and *regions* ($\geq$ 3 zones) to customers. Cloud customers use these building blocks for a range of deployment patterns, from non-replicated services to multi-region replication [7, 15, 23].

In order to provide 99.9% zonal availability, each zone can only tolerate ~4 minutes per month of downtime. This implies that any failure event that requires human intervention will almost certainly render a zone as out of service-level-objective (SLO). Correlated failures (those that simultaneously affect multiple zones) are particularly problematic as they negate the customer's replication efforts.

2.2 Cluster networking

As described in [18, 49, 56], Google's software-defined network (SDN), a.k.a. "cluster network service", includes:

- The data plane switches of a Jupiter [49, 56] fabric, including their operating system and firmware stack.
- SDN applications, responsible for programming switches, tracking link state, traffic matrices, etc.
- The dedicated machines called network control servers (NCS) that run the SDN applications. These machines are completely independent from normal machines.
- The dedicated instance of the Borg [63] cluster scheduler that manages the NCS.
- A separate control plane network (CPN) that provides out-of-band connectivity between the NCS machines and data plane switches.

Figure 5 depicts the relationship between these components.

Steady state behavior refers to how—in the absence of ongoing management operations—the cluster network service treats the user payload (network packets) while responding to changes in the runtime environment (e.g., shifts in traffic, links failures). The primary expectation from users is reachability from cluster machines to destinations both inside the cluster and outside (through the WAN) with expected latency, packet loss rates, and bandwidth.

We define an *outage* as an event where the network service behavior fails to meet user expectations, defined precisely as active prober packet loss or latency exceeding predefined thresholds. Examples include blackholes, capacity loss that leads to congestion, or suboptimal pathing.

2.3 Best practices for operations and their implementation challenges

Since management operations are a common cause of outages, several studies have analyzed outage incidents [1, 8, 25, 44] and proposed best practices for operating networks with high availability. We recap the most relevant practices, as well as challenges we have faced in enforcing them.

Failure domain containment. System designers should identify failure domains (e.g., groupings of physical entities which share power supply, cooling, software control, etc.) and enforce as much as isolation as possible between domains in order to minimize the risk that a failure's "blast radius" will extend beyond domain boundaries.[1] Failure domain isolation also reduces the risk of lockout during disaster recovery, since proper isolation requires systems to avoid cyclic dependencies involved in bootstrapping the failure domain (e.g., the cluster network service cannot depend on a distributed storage service which in turn depends on the cluster network service).

In practice system designers need to strike a balance between minimizing blast radius and minimizing system complexity and cost (C1, C2). For example, some exceptions need to be made in practice for communication (e.g., BGP routing messages) that is allowed to cross failure domain boundaries.

Progressive, supervised change rollout. Faulty operations can lead to service disruptions or violations of failure domain containment. To mitigate these risks, changes should be rolled out gradually. Progressive rollout alone does not improve safety; the health of the system needs to be closely supervised throughout the rollout, otherwise the network remains vulnerable to "slow wrecks" where service behavior gradually but continuously degrades. To keep up with the volume of operations needed to sustain the business, the system must strike a good balance between safety and velocity (e.g., level of concurrency) of operations (C3).

Monitoring and upholding invariants. Safety invariants should be monitored during change rollout to maintain system health [25]. Black-box metrics (those that do not assume any knowledge of the internal implementation details of the network, and therefore capture the user's perspective), such as remote procedure call (RPC) latency or success rates, are effective at detecting failures regardless of root cause as soon as the service behavior begins degrading [1] since they capture the exact conditions that the user experiences, but they are not effective at detecting the degradation before it occurs. Subject matter experts can also identify white-box invariants, i.e. those that leverage implementation details of their subsystems (e.g., "at least 2 replicas should be responsive"), but white-box invariants that generalize to a diversity of system interactions have remained elusive (C1).

Provisioning capacity headroom. Provisioning more capacity than strictly needed to accommodate traffic improves safety. For example, headroom ensures that when a management operation (especially those that require involvement from human technicians) becomes stuck, the operation can remain in a paused state without risking congestion on the unaffected portions of the network. The costs of deploying extra capacity can be amortized by serving low priority (QoS) traffic during periods without ongoing operations.

Fast recovery. Fast recovery limits outage duration. In practice, recovery often requires humans to understand the situation and determine a recovery strategy. In a system with many ongoing changes, it is not always obvious what needs to be rolled back. Systems can also fail in ways that cannot be recovered by rollback, e.g., if the outage wipes out data, or prevents access to the system itself – a real concern for networking. Automated recovery that rolls back too aggressively can also be at odds with operational velocity (C3).

[1] Failure domains are hierarchical. For example, top-of-rack (ToR) switches reside in the same physical failure domain as the machines on the rack; each ToR connects to multiple middleblocks (another failure domain); the middleblocks reside within an aggregation block which are designed to be both physically independent from other blocks (separate power and cooling supplies) and logically independent (separate software control) [18].

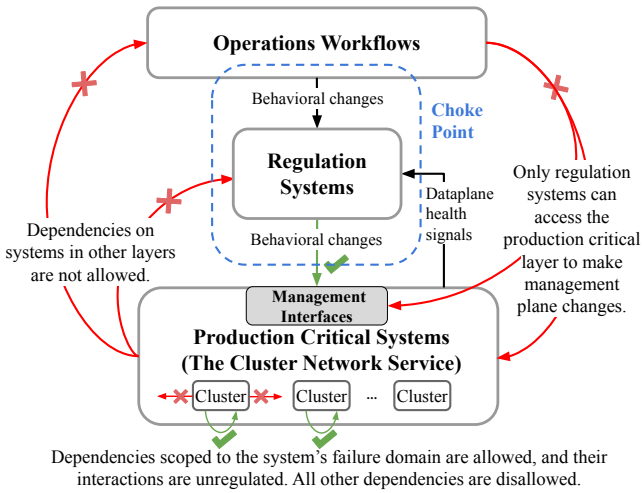


Figure 1: CAPA's layering and enforcement architecture.

3 CAPA architecture Overview

Following [33], CAPA imposes layering onto our existing software and hardware systems so that we can systematically enforce best practices. As shown in Figure 1, these layers consist of: (a) the *production critical layer*, comprising the components (control plane and data plane) that are responsible for delivery of application payload. We must ensure failure independence (§4.2, §4.3) and high baseline availability in this layer to meet service expectations (SLOs). (b) the *regulation layer*, a small number of centrally maintained services with exclusive access to the management interfaces in the production critical layer. It upholds best practices by applying changes progressively (§4.1) while monitoring system health and invoking automatic rollback of operations ("behavioral changes") if needed (§3.1). The regulation layer maintains the key invariant of ensuring sufficient bandwidth at all cross-sections of the network (§3.1), and enforces global rate-limiting to reduce the risk of correlated failures triggered by faulty operations concurrently executing in different zones (§4.1). (c) the *operations workflow layer* contains all other systems responsible for carrying out network management operations. This layer generates the intended behavior (configuration) of the production critical layer, and realizes that intent by invoking operations via the regulation layer.

3.1 Life cycle of an operation

Before detailing the mechanisms introduced by CAPA (§4), we will walk through the life cycle of a management operation as it progresses through CAPA's layers.[2] We choose a capacity expansion as our example. Table 1 lists other operations.

Capacity expansion starting in the workflow layer. Based on projected traffic demands, a forecasting service in the workflow layer predicts that capacity will be needed on the

[2] §6 gives an overview of network management at Google prior to CAPA.

Category of Operation	Frequency
Capacity expansion/contraction	1 per 3.6 months
Switch software upgrade	1 per 4 weeks
Controller software upgrade	1 per 2 weeks
Hardware repairs	1 per 35 minutes

Table 1: Categories of behavioral changes ("operations") that our networks undergo. Frequencies are computed as averages per cluster network across all clusters over one year.

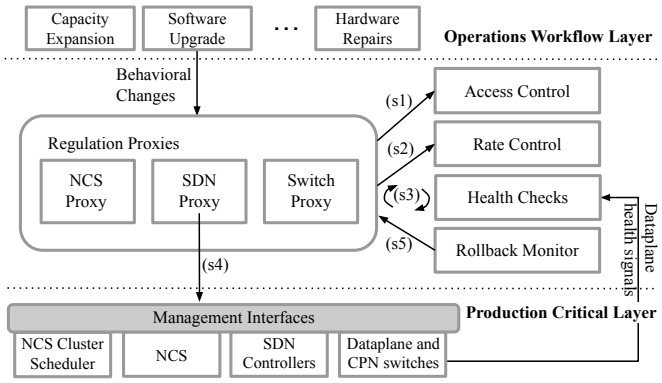


Figure 2: Detailed view of the regulation layer.

WAN-facing links of the fabric border router (FBR) of a cluster network. A capacity expansion workflow is initiated. It begins by retrieving a model of which entities (e.g., fibers, transceivers) will need to be added to the network [43]. The workflow dispatches purchase orders (after human approval) and waits for delivery of the new materials to the datacenter floor. Once the materials arrive at the datacenter floor, the capacity expansion workflow dispatches instructions to technicians on how to wire and qualify (e.g., verify that bit error rate is sufficiently low for) new physical links. The physical changes needed for the capacity expansion are "append-only," meaning that technicians do not need to mutate any pre-existing links carrying live traffic. We reserve spare ports on both the FBR side and the WAN side to make this possible.

Capacity expansion processed by the regulation layer. For scalability of routing updates, we maintain link aggregation ("trunks") at the WAN boundary: we split all WAN-facing physical links into four trunks (each with distinct power and cooling failure domains), and assign a single SDN control domain responsible for routing over each trunk. Trunks are the level of abstraction that the SDN routing stack operates on; BGP sessions, next-hops, metric computations, etc. are maintained on a per-trunk basis.

To bring the physical capacity into service, the capacity expansion workflow needs to reconfigure each of the SDN control domain's configuration for the trunk it routes over. The workflow first drains traffic from the links under the control domain's purview before initiating any reconfiguration, primarily to protect against any *unexpected* side-effects of the operation; as discussed in §5, unexpected side-effects have historically been a prevalent cause of outages. The regulation

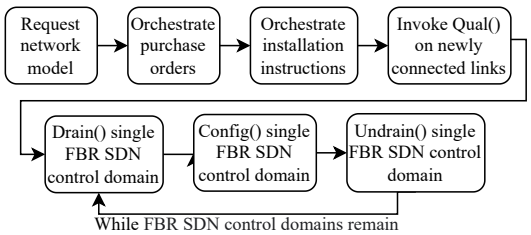


Figure 3: The capacity augment workflow's state machine. The workflow repeatedly invokes APIs in the regulation layer, which are enumerated in Table 2.

layer refuses to process operations that do not already have drains covering the entities they affect.

The capacity expansion workflow invokes the `Drain()` API on the SDN proxy in the regulation layer (Figure 3). The SDN Proxy (Figure 2) validates the workflow's permission to perform `Drain()` (s1), and checks fleetwide rate limits for FBR `Drain()` (s2). Concurrency control and rate limits reduce the scope of temporally correlated failures by limiting the number of domains that may be affected (§4.1).

Enforcing cross-section capacity. The regulation layer enforces that all target elements should be drained before proceeding with the operation: besides protecting against inadvertent impact on live traffic, this invariant allows the `Drain()` API to double as a semaphore that guards the network's bandwidth while allowing concurrent operations.

Figure 4 depicts how `Drain()` doubles as a semaphore. Multiple operations, each requesting drains, can proceed concurrently so long as the effective ("residual") capacity in all relevant cross-sections of the network are above a threshold (see §4.1 for details). Example network cross-sections on a per cluster network basis include: all WAN-facing links, inter-aggregation-block links, intra-aggregation-block links, top-of-rack (ToR) to middleblock links, and control plane network links between SDN controllers and data plane switches [56]. Cross-sections can be overlapping and hierarchical, e.g., we seek to prevent partitions of any particular aggregation block, but we also protect aggregate interblock bandwidth.

Operations are blocked when the threshold is reached, but they can proceed as soon as the previously in-progress operations conclude and restore capacity by issuing an `Undrain()`. The semaphore also gracefully accounts for unplanned maintenance (hardware failures/repairs): failures reduce effective capacity, thus blocking subsequent planned operations.[3]

From a workflow's perspective, `Drain()` is atomic, i.e., if it succeeds, the residual network is in a safe state and the drained entities are safe to operate on. The SDN proxy processes `Drain()` requests in a serial order within each cluster network, which prevents deadlock so long as each workflow requests a single `Drain()` at a time. In the case that there is insufficient effective bandwidth (or some other health

[3]Drain thresholds are chosen to minimize the probability that unplanned failures cause the system to be unable to service customer traffic without congestion. Humans are alerted in the rare event that this occurs.

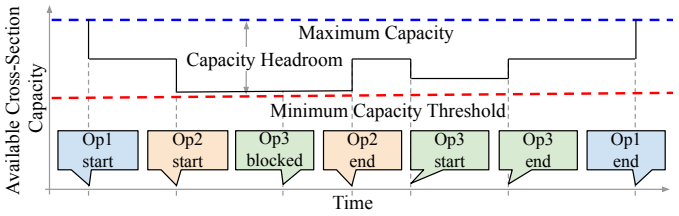


Figure 4: Example of drain thresholds ("ensure minimum cross-section bandwidth") acting as a semaphore for regulating concurrent operations. Op3 is blocked until capacity is released by Op1/Op2.

Interface	Description
SDN Controllers	
Config()	Configuration used by SDN apps. Includes fabric topology, routing policies, etc.
Drain/ Undrain()	Deprefer the targeted elements from the routing solution. May require coordination across domains.
Dataplane and CPN Switches	
Install()	Updates software running on the switch.
Config()	Modifies non-routing behavior, e.g., port speeds.
Qual()	Initiates disruptive bit-error-rate test on a port.
NCS Cluster Scheduler	
Add/Remove Machine()	Modify set of servers running NCS jobs.
Create/Update/ DeleteJob()	Create, update or delete the jobs to run in the cluster.
Config()	Updates resource management policies.
NCS	
Install/Update()	Install or update the firmware and OS of the server.
Restart/ Shutdown()	Reboot or shutdown the server.

Table 2: List of RPC interfaces for management plane operations on various components in the cluster networking service (non-exhaustive).

check fails, as we describe below), the `Drain()` invocation is blocked and leaves no side effects until it is unblocked.

Applying the drain and configuration changes. The SDN Proxy actuates the `Drain()` by invoking management interfaces on the production critical layer (s4). Once the `Drain()` completes, the workflow may proceed with reconfiguration of trunks by invoking a `Config()` API[4] on the SDN Proxy within the regulation layer. Table 2 describes other management APIs provided by the regulation layer.

The proxies execute the same steps (s1-s4) to actuate the reconfiguration. Once reconfiguration is complete, the workflow will issue an `Undrain()` and repeat starting from the `Drain()` for the next FBR SDN control domain.

Continuous health checks. Network health can change rapidly and unexpectedly, e.g., from hardware failures, sudden traffic shifts, or unexpected side-effects of operations.

The regulation layer continuously checks two types of health signals—black-box and white-box signals (introduced

[4]The format of the `Config()` change is arbitrarily defined by application developers; in this case it describes which physical links the routing controllers should aggregate together as a trunk.

in §2.3)—both before (s3) and during (s4) the execution of operations. While black-box signals are a lagging indicator of problems (e.g., discards occur only after congestion reaches unsafe levels), some white-box signals can act as preventive leading signals. Protected cross-section bandwidth is an example of a white-box invariant, and is the only invariant we have found so far that generalizes to all operations. Other examples include ensuring a minimum number of SDN application replicas are live, checks on whether any parts of the cluster networks have entered fail-static [18], and bans on certain orderings of operations that are known to cause issues.

Automated drain rollbacks. When servicing a `Drain()` invocation, the SDN Proxy actuates the request incrementally by repeatedly invoking management interfaces in the production critical layer for a subset of the entities referenced in the `Drain()`. If health checks ever fail (e.g., we detect packet discards, or effective capacity drops below safe thresholds) and only a subset of the entities in the `Drain()` have completed, the SDN Proxy can automatically roll back (s5) the partially executed `Drain()`. If a `Drain()` has already been acknowledged as completed to the workflow, we cannot currently rollback those drains. As part of our future work (§6), we are exploring automated rollback of the operations that proceed after the `Drain()` has been acknowledged.

4 Detailed Design

4.1 Rate-Limiting & Concurrency Control

As described in §3.1, the regulation layer performs rate-limiting and concurrency control. Rate-limiting helps maintain failure independence by limiting temporally correlated outages that arise from concurrent operations. Rate-limiting also maintains baseline availability by increasing the probability that issues are caught early in low-risk cluster networks.

Rate-limiting enforcement. Google uses MALT [43] to represent the elements in our network, their attributes, and their relationships with each other. Using this representation, a global service in the regulation layer builds a fleet-wide model of cluster level failure domains and sub-cluster failure domains (e.g., aggregation blocks) within each cluster. The global service also provides a rate-limiting policy configuration to set the maximum concurrency allowed across (hierarchical) failure domains to balance velocity and production risk. Policies are jointly decided by stakeholders in engineering and operations teams on a per use-case basis.

Equipped with modeling and policy, the global rate-limiting system ensures that only the desired number of entities are operated on concurrently or within a time epoch. For example, we can set policies to limit concurrent capacity expansions to only N clusters globally, or require 15 minutes between successive operations on each of all 4 FBR domains within each cluster such that at most N clusters can lose at most 25% of FBR capacity within a 15-minute window.

The rate-limiting system accounts for the full hierarchy of failure domains in making its decisions, e.g., to perform an operation such as a `Drain()` upon a middleblock in a particular cluster, all rate-limiting policies that apply to `Drain()` operations will be checked. Any ongoing operations in that middleblock will be checked against policies that apply to middleblocks; then, ongoing operations in the aggregation block that the middlebock resides in will be checked against policies that apply to aggregation blocks, and so on through the cluster, zone, region, and global levels [56]. The operation will be allowed to proceed only if all policies defined at all levels of the hierarchy deem that the operation is in compliance with rate-limiting.

Concurrency control and capacity headroom. Rate-limiting policies are defined according to the scope of entities that operations intend to mutate. In some cases, operations affect a larger set of entities than intended. Protected cross-section bandwidth and `Drain()` as a semaphore further protect against unintended side-effects and "slow wrecks" where rollout supervision fails to detect degradation.

The threshold for protected cross-section bandwidth (capacity headroom) is largely determined by the level of redundancy designed into the network. Across most dataplane cross-sections (exceptions in §4.4) we employ 4 independently operable (sub-cluster) failure domains; only 3 are needed to maintain sufficient capacity to avoid outages, leading to 25% capacity headroom. In our example (§3.1), there are 4 independent FBR domains. Similarly, within an aggregation block, we have 4 middleblocks, and we have 4 interblock routing domains [49]. In the control plane network, we generally have 2-way redundancy of switches and paths, so CAPA does not allow draining either unless both are healthy.

4.2 Failure Domain Containment

Separating the operations workflow layer from the production critical layer, and rate-limiting operations through the regulation layer helps prevent temporally correlated failures. However, this protection is only effective if the failure domains themselves are well contained.

Appropriately defined failure domain boundaries should align physical points of failure (e.g., power and cooling) with software control. Once failure domain boundaries are defined, we must audit dependencies between software components to ensure that there is no unintentional communication across boundaries. Most interactions among our system components take place over RPCs.[5] For containment, cluster network service components within one datacenter network are not allowed outbound or inbound *modify*[6] RPCs to other cluster networks or other layers (depicted in Fig. 1). To enforce this rule we use a combination of (1) token-based authentication [26], e.g., ACLs to prevent all outbound RPCs, (2) locking down

[5] Exceptions include BGP (§4.3), NTP (§4.4), and DHCP.

[6] We allow read-only inbound RPCs to facilitate monitoring.

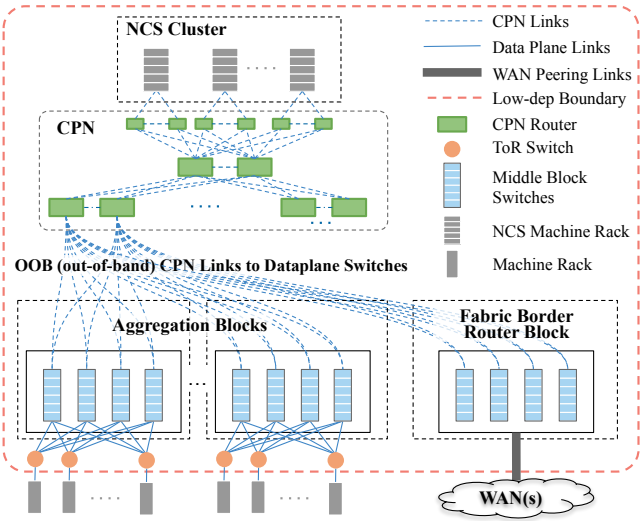


Figure 5: The low-dependency boundary encompasses all elements of the data plane, control plane network (CPN), network control servers (NCS), and their software. RPCs to any services running in normal machine racks or across the WAN are disallowed. (Interblock connectivity and certain other elements not shown for legibility.)

all inbound management interfaces in the production critical layer to the proxies in the regulation layer,[7] and (3) monitoring cross-domain RPCs to/from system components to detect any interfaces that may inadvertently remained unprotected.

Failure domain containment enforcement extends to the transitive closure of all the dependencies of the components in the production critical layer, creating a foundational failure domain containment structure that is robust to erosion as the system evolves. We encourage but do not enforce the same failure domain enforcement rules for sub-cluster failure domains (e.g., individual aggregation blocks) in the cluster.

Low-Dependency Requirements. Our restriction on outbound RPCs implies that all components within the production critical layer must be "low-dependency", meaning that dependencies outside the cluster network service are disallowed. Figure 5 presents all the elements within the low-dependency boundary. All cluster network service software on switches and SDN control applications on NCS machines must be designed without any outside dependencies on software services that run in the normal machine racks either in the same cluster or another cluster. Low-dependency is an expensive requirement to meet as it precludes use of convenient, broadly understood, high-level infrastructure services, such as multi-zonal replicated storage systems, work queues, load balancers, authentication mechanisms, etc.

Ultimately this burden is required considering (i) avoiding circular dependencies on network connectivity simplifies bootstrap and disaster recovery (further discussion in §6), and (ii) the requirement encourages a minimal "trusted computing base" of services that are absolutely needed to maintain steady state behavior. Minimalism reduces the components in the production critical layer and therefore reduces the number of subsystems that can cause an outage.

[7] We are further strengthening these boundaries using separate cryptographic scopes for each failure domain. This reduces the principals that may act across failure domains even in the face of security incidents.

4.3 Static routes for WAN interconnects

Inter-domain BGP routing updates are one of the few unavoidable exceptions to RPC boundary containment. WAN connectivity requirements are in tension with failure domain isolation as BGP messages themselves are a vector for correlated failures. BGP has multiple properties that make it source of historical outages at Google, including:

1. BGP speakers disseminate information rapidly, which can cause multi-zonal outages from a bad update.

2. BGP speakers lack a global view and make uncoordinated best-path decisions; this can result in replicated speakers creating loops (each peer believing the other has the preferred next-hop) and unintended route withdraws (e.g., when loops are detected, or when each speaker deprefers its own next-hops).

3. In default configurations, BGP ties together liveness of the data plane and the control plane through keep-alive messages. Upon three missed keep-alive messages, the peer is assumed to have failed and routes are withdrawn, even if the data plane is healthy. This is particularly problematic for SDN where the BGP speaker is separate from the dataplane switches.

To achieve failure domain isolation we require additional defense-in-depth for BGP. We rely on BGP message filtering [13] for defending against (1) and (2), and implement a static route fallback policy for the WAN and CPN routes for (3). Conceptually, fail-static routing splits BGP-based connectivity into two parallel routing systems: traditional, dynamic (optimal) routing via BGP, and a simpler, rarely changing, safe but suboptimal set of "static" routes. The fail-static policy makes traffic fall back to the static routes whenever dynamic routes become unavailable, thus substantially reducing correlated failures that are triggered by unavailability of the BGP control plane without complete loss of the data plane.[8]

From the perspective of the fabric border routers within the cluster network, static routes are configured as a virtual output port that switches can send traffic out in the event of dropped dynamic BGP messages. We implement static routes in the WAN as either traffic-engineering tunnels that fail-static (for B4 [30]), or redistribution of cached routes even when BGP keep-alive timers have expired.

[8] In the event of complete data plane partition, packet loss will still occur once the packets reach the partition.

Static routes can be updated progressively in accordance with best practices, at the cost of temporary mismatch with primary (BGP) routes. In an application of the end-to-end principle [53], these periods of mismatch are generally acceptable thanks to an end-host mechanism that repaths onto healthy paths [52]. Since cluster networks represent leaf nodes of the WAN, we have so far been able to prevent loops while rolling out updates to static routes (see 4.5 for discussion).

4.4 Exceptions

We allow a few exceptions to the `Drain()`-before-operate rule when we conclude that the benefits of simplification outweigh the risks. For example, except for decommission operations, we do not require `Drain()` for changes to ToR switches as their failure domain is a handful of machines; further, many racks do not have redundant ToRs, so draining the ToRs would require vacating the underlying machines. Similarly, although we require `Drain()` before config changes to (sharded) FBR control domains, we have concluded that sharding of SDN control domains is not worth the complexity cost for our aggregation blocks. We therefore allow config changes to those control domains without requiring the data plane entities under their purview to be drained.

We also allow a few exceptions to the "low-dependency" restriction, which we permit either to ease the transition from a historical dependency, e.g., Network Time Protocol (NTP) from another cluster during bootup, or as an optimization, e.g., host level telemetry to drive optimal traffic engineering. These exceptions create a lockout risk, but we require the teams to have well-tested fallback mechanisms so that the cluster service can recover in absence of these dependencies.

4.5 Known Limitations

No Progress Guarantees. It is possible for workflows to be blocked indefinitely by CAPA's safety policies. In such cases a human operator must intervene. For example, although CAPA prevents workflows from deadlocking on each other (by serializing drains on a per-workflow basis), a single workflow can still deadlock on itself by inadvertently requesting concurrent drains and assuming they will be processed atomically. A deadlocked workflow blocks other workflows.

Drains Can Degrade Application Performance. Our demand forecasting systems seek to provision capacity so that high priority customer traffic can be serviced without congestion. We allow customers to send extra low priority ('BE') traffic in order to amortize the cost of capacity headroom, but we reserve the right to drop BE traffic when maintenance needs to perform drains. That said, CAPA does not formally enforce these contracts with customers; in practice we have found that some customers complain when small percentages of BE traffic is dropped, and applications can also send high priority traffic in excess of planned capacity.

No Provable Guarantee of BGP Loop-Freeness. Although we have never experienced loops introduced by BGP static routes (e.g. a loop introduced when one peer continues to use dynamic routes while the other falls back to static routes), we have no proof that they are impossible.

5 Evaluation

We evaluate CAPA with three methodologies:[9]

1. We enumerate examples of real production incidents where CAPA successfully prevented or mitigated outages, to examine CAPA's mechanisms in practice.

2. We conduct a comprehensive retrospective analysis of outages affecting our cluster networks from 2018 - 2022 that had behavioral change as their root cause to understand where CAPA would have been beneficial and where CAPA still has safety gaps. Some of these outages pre-dated CAPA, due to the fact that we incrementally rolled out CAPA's features over multiple years. That said we verify that CAPA's (current) safety mechanisms do in fact mitigate/prevent outages where CAPA *would have been* applicable by requiring as part of our outage postmortem process that engineers eventually reproduce the outage conditions in an automated test environment to verify that (new) CAPA features work as intended.

3. We examine quantitative data (active network probes) to show incidence trend and severity of management plane related outages. Although this does not establish causality, CAPA was our primary investment in reliability improvements; given quantitative correlation and the first two qualitative evaluations, we can reasonably attribute these gains to CAPA.

For both the qualitative analysis and the quantitative analysis, we only consider outages which exceeded $\geq 5\%$ packet loss across all active probers (§5.4) in at least one cluster network, and lasted for ≥ 5 minutes. We refer to such incidents as "bad fabric incidents." A single outage may cause multiple bad fabric incidents if it affects multiple cluster networks.

5.1 Methodology Discussion

We must cope with three challenges in evaluating CAPA:

1. Our customers will not tolerate us running randomized control trials ("A/B tests") on our deployed reliability mechanisms. Moreover, even if we were able to do so, there is strong evidence that the underlying distribution of outage severity is memoryless ("long-tailed"), implying that we would need to run A/B tests over a multi-year period for statistical soundness.

[9] Since CAPA does not directly change the parameters of the safety vs. operational velocity tradeoff, we consider it out of scope.

2. Cluster networking, operational workflows, fleet size and CAPA's capabilities are constantly evolving; we have incrementally rolled out its features over several years, and continuously make modifications to our safety policies. Hence, there is no single system to be evaluated.

3. We cannot gather a comprehensive list of all "near-miss" success stories where CAPA prevented outages because it is infeasible to precisely define those circumstances.

While we cannot establish causality, we believe our evaluation methodology effectively illuminates CAPA's impact, particularly given that CAPA was our only investment into hardening against faulty management operations. We employ the same qualitative analyses internally to evaluate new investments into availability improvements.

5.2 Success stories

We first describe a selection of incidents where CAPA successfully prevented or mitigated outages.

Layer separation, rate limiting, health monitoring and fail stop. As noted, Google uses a network model [43] to express intended state of our networks. Operational systems continuously monitor differences in the current and intended state of the network, and initiate operations to apply the intended state. A bug in the modeling system mistakenly updated the models for all the clusters globally. Automation, unaware of the buggy nature of the change, detected the differences and initiated a huge number of concurrent operations to apply new configuration to all dataplane switches.

CAPA limited the concurrency and rate of changes, monitored the health of cluster networks, and raised alerts about an unusual backlog of operations. The Drain()-before-operate rule prevented the few operations that were allowed within the rate limits from causing congestion. Separation between the operations workflow layer and the production critical layer prevented a buggy global intent from propagating unabated to the fleet, and rate limiting created temporal separation between operations that—if allowed to execute immediately—would have created correlated failures across multiple zones.

Ensuring disruptive operations are covered by drain. A workflow responsible for decommissioning ToR switches had a bug resulting an intent to decommission ToRs across multiple aggregation blocks instead of the single originally planned block. The bad intent propagated through several planning systems, but the "ensure operations are covered by drain" validation at the regulation layer rejected the downstream intent after detecting that some ToRs were still serving traffic. If the intent had been allowed to propagate to routing systems, the machines under those ToRs would have become partitioned.

Protected cross-section bandwidth. Our switch config rollout workflow first checks if a switch is drained for repairs, and if so skips pushing configs to that switch until the repair case is resolved (since the switch is likely unresponsive). The database that tracks switches under repairs was undergoing a migration to a new format; to reduce human toil during the migration, the config rollout workflow was modified to query either the old or new format. The rollout workflow misinterpreted the new format, causing it to skip pushing configs to switches that were not under repairs. At a later point, another workflow started to upgrade the software of the switches. The upgraded software on the skipped switches was unable to interpret the stale config, causing the switches to crash and consequently be excluded by routing. The result was capacity imbalance at the WAN boundary. The regulation layer detected that cross-section bandwidth had dropped below safe thresholds, and stopped the switch software upgrade workflow from breaking additional switches.

BGP fail static. Maglev [14] is a datacenter-scale load balancing service. It relies on virtual IP addresses (VIPs) which map to machines in the datacenter. Jupiter's traffic engineering system [49] relies on telemetry about traffic volume to program optimal paths to IP prefixes. A bug in Maglev software created telemetry reports with invalid VIP prefixes, which were duly discarded by Jupiter's traffic engineering system. However, without telemetry data, routing entered a fall-back mode using equal-cost-multi-path (ECMP) instead of the more efficient weighted-cost-multi-path (WCMP). As a result, several equal-cost prefixes were aggregated to a single, larger IP prefix that was not present in the export list for the WAN, causing the BGP prefix to be withdrawn. When the WAN stopped receiving dynamic advertisements, CAPA' static routes kicked in, successfully keeping WAN traffic flowing when it otherwise would have started blackholing.

5.3 Retrospective analysis of outages

We retrospectively analyze all outages triggered by management operations that resulted in at least one bad fabric incident between 2018 - 2022. Many incidents pre-dated CAPA. We analyze which of CAPA's mechanisms either *were*, *would have been*, or *still are not* applicable. Our retrospective analysis does not establish statistical significance since the size of the dataset is insufficient; our goal is instead to gain qualitative insights into CAPA in the context of real incidents.

As Table 3 shows, CAPA's mechanisms either were or would have been applicable to 84% of outages. Table 3 demonstrates that no single 'magic-bullet' prevention mechanism exists, due to our systems' wide diversity of failure modes. Some level of rate limiting was in place during most outages, but without additional mechanisms (e.g., protected capacity), management workflows caused "slow wrecks" where the network gradually degraded until an outage occurred. Similarly, gaps in failure domain containment allowed a few operations to affect many failure domains.

5.3.1 Retrospective case studies

GCP Incident 19009. A significant outage [24] affected Google services globally due to misalignment of job schedul-

Mechanism	Prevent or Mitigate
Rate limiting, monitoring, and fail stop	74%
Protected cross-section bandwidth	47%
Failure domain containment	11%
Fail-static for WAN interconnects	21%
At least one applicable mechanism	84%
No applicable mechanism*	16%

Table 3: Counterfactual analysis of which CAPA mechanisms either *were* applicable or *would have been* applicable in mitigating or preventing all outages which resulted in at least one bad fabric incident and which had an operation as root cause trigger from 2018 - 2022. Multiple mechanisms could be applicable for any particular outage. *See §5.3.2.

ing and SDN controller (Orion) failure domains, where a single job controller managed multiple Orion domains. A reconfiguration intended for a single job scheduler triggered cascading failures that brought down all Orion jobs for a large number of zones. This outage underscored the need for considering the transitive closure of dependencies for failure containment: although the SDN controllers, NCS machines, dataplane switches were all in separate failure domains, the job scheduler was not considered as part of the containment.

This outage could have been limited to a single cluster with proper failure domain containment. BGP fail-static would have further mitigated the outage since WAN dataplane connectivity was functional despite failed Orion jobs.

IncidentB. Dataplane switches require a reboot after certain operations, and are expected to signal to regulation layer proxies to trigger reboot. For a new management operation, the switch failed to signal the reboot requirement. A switch configuration rollout exercising this new operation gradually caused a large number of switches to enter an unhealthy state. Orion misinterpreted the lack of responsiveness from these switches as being due to CPN connectivity issues, and then entered massive fail-open (MFO) where it refused to program new routes [18].[10] MFO complicated recovery, since engineers had to find an emergency override to reboot undrained switches rather than typical drain APIs.

If CAPA had required drains before pushing configs to fabric switches (as it does now), impact could have been avoided. Even if drained switches still broke upon receiving the new config, not all switches would have been affected since protected cross-section bandwidth would have halted drain requests (and therefore config pushes) once the affected entities in the cross-section exceeded a threshold.

IncidentC. The workflow responsible for choosing locations of new machine racks on the datacenter floor relies on human provided layout. An error caused racks to be targeted to the wrong datacenter. A ToR turnup workflow concluded that rather than adding new ToRs, it needed to update locations of *existing* ToRs. To do this, the workflow deletes entities in the model and adds them back with updated locations. A different workflow designed to monitor new intents and push configs detected the difference between the delete step and the add-back step and inadvertently decommissioned a large number of live ToRs. Black-box monitoring did not detect packet loss, since the monitoring system itself decommissioned probing jobs running under those deleted ToRs.

The impact could have been avoided if CAPA had checked whether drains were in place before instructing the SDN to remove ToRs.[11] Rate limiting configuration pushes to the controllers of the aggregation blocks would not have stopped the operations since the health signal was also affected.

5.3.2 Retrospective and other known safety gaps.

We share experiences with a selection of outages where CAPA's mechanisms did not fully prevent an outage. We also list other known safety gaps. These highlight that as much as CAPA has improved reliability, there is still work to do.

IncidentD. During a capacity augment, operators inadvertently disconnected undrained physical links. CAPA's `Drain()`-before-operate works well to prevent software from making such mistakes, but not humans in the physical world. We have invested in user experience improvements (e.g., pagers, real-time feedback) to reduce these risks, but actions in the physical world cannot be programmatically stopped.

IncidentE. A bug in interaction between `Drain()` and routing applications caused BGP announcements to be revoked before intra-fabric routing finished moving outbound traffic away from WAN-facing links. This resulted in an abrupt loss of forwarding state and packet loss blips for 7 minutes until the intra-fabric routing converged. We have drawn two lessons from this incident: (i) Unsurprisingly, CAPA's own mechanisms can have bugs and hence it is important to have a multi-layer defense. Draining traffic (even if buggy), together with rate limiting as a second layer of defense, contained the duration and the blast radius. (ii) It is important to monitor that each layer is working as intended, so that multiple layers are not simultaneously impaired thereby increasing risk.

IncidentF. A workflow made a modeling change that affected both switch and Orion configs. Later, an unrelated workflow inadvertently pushed those configs to Orion but not switches. This resulted in a version mismatch, causing switches to fall back to ECMP rather than WCMP. ECMP resulted in higher utilization on links with slower speeds. The issue went undetected until the workflow pushed config to enough domains to cause congestion and packet loss. Although CAPA's cross-section capacity monitoring was in place, the imple-

[10] MFO is a feature in Orion wherein routing apps halt upon unresponsiveness from a large number of dataplane switches. The premise is that switches may become disconnected from the routing apps, but may be serving the dataplane just fine, so it is better to keep the current forwarding state instead of routing around unresponsive switches and steering traffic towards a small number of responsive switches thereby creating congestion.

[11] We except ToRs from `Drain()`-before operate for config pushes direct to a single ToR (§ 4.4), but we still require drains for configuration changes whenever a large multiple ToRs are affected simultaneously.

Metric	2018	2019	2020	2021	2022
Bad Fabric Incidents	1.0x	2.36x	0.28x	0.23x	0.27x
Cumulative Duration	1.0x	5.96x	0.44x	0.10x	0.18x

Table 4: Counts and cumulative duration of "bad fabric incidents" across the fleet (cluster networks with ≥5% loss for ≥5 minutes) that had an operation as their trigger, normalized to the number of switches in the fleet at the beginning of the year, relative to the normalized count for 2018. A single cluster network may undergo ≥1 bad fabric incident. See 5.1 for interpretation of this data.

mentation at the time (now fixed) failed to account for ECMP behavior over heterogeneous link speeds. This outage illustrates a broader challenge: although cross-section capacity is a remarkably effective health signal, we must continuously update our interpretation of capacity signals to keep astride with advancements in routing.

Other known safety gaps: Beyond these three cases, we are aware of a number of other safety gaps:
Organic failures: CAPA is designed to mitigate impact from planned changes rather than organic failures such as link cuts (though it does protect capacity headroom, thereby increasing the chance that the network can accommodate traffic despite failures). Similarly, CAPA does not address outages triggered by coincidental or malicious user behavior (e.g., DDoS attacks, or packets of death). We consider protection against such problems as part of the baseline functionality of the cluster network service which should be addressed through multi-path routing, testing, packet filtering, etc.
Infrequently used disaster recovery tools: In CAPA we have allowed both the workflow and regulation layers to be high dependency, i.e., the services in this layer can depend on higher layer services such as distributed storage that potentially depend on multiple cluster networks. This allows creation of cyclic dependencies which create lockout risk during large-scale outages (e.g., GCP Incident 19009 [24]), where sufficient clusters are degraded to render the higher level services unavailable. To address this gap, we provide separate "breakglass" management tools that can be used when low-dependency recovery is required. Fortunately these tools are infrequently used; but infrequency of use increases the chances of encountering bugs when the tools are actually needed. We address this by regular testing at smaller scale and maximizing the shared libraries between the breakglass tools and systems used on a day-to-day basis.

5.4 Quantitative data

Application-perceived availability is determined by the source, destination, and service class (priority) the application uses to send messages. We employ active probing where a subset of machines continuously send packets of various QoS classes to each other in order to monitor network availability.

Table 4 demonstrates trends in total bad fabric incidents caused by management operations in a five-year period (2018 - 2022). We normalize these counts to the size of our fleet (measured in terms of # of fabric switches across all cluster networks) since the number of behavioral changes (and therefore potential # of outages) we execute is proportional to the size of the fleet. Table 4 shows a 73% reduction in normalized bad fabric incidents over the period.

Table 4 also shows cumulative duration of bad fabric incidents caused by management operations. We observe a 82% reduction in normalized duration. A single large outage in 2019 [24] accounts for 73% of cumulative bad fabric duration.

6 Discussion

Does CAPA apply beyond cluster networking? Failure domain containment enforced via RPC ACLs is a generic mechanism that is already used by other systems (e.g., cluster schedulers). Layered enforcement and regulation (Fig. 1) is also applicable to other production services, but it requires clear separation of management plane interfaces from data-plane interfaces, something that not all services do consistently. Lastly, the idea of defense-in-depth through progressive and supervised rollout is applicable to other services, but it needs to be adapted to the specific context. For example, for cluster scheduling it could mean that the resources allocated to a service should be reduced progressively and should never go below a configured threshold.

Is CAPA provably safe? CAPA's architecture is based on principled reasoning about failure domain containment and separation of operational systems from the cluster network service, but we cannot claim that it is provably safe, especially given reliance on domain specific heuristics, exceptions (§4.4), and known safety gaps (§5.3.2). It is unclear whether provably safe systems for continuously evolving and complex systems like cluster networking are feasible, but the data (§5.2, §5.4) clearly demonstrate that CAPA is effective for real-world large-scale systems, while also providing motivation to continue to invest into improvements.

Should we use progressive updates in the production critical layer itself? In §5, we see a number of incidents that are caused by rapidly propagating updates (e.g., route programming) within the production critical layer. We have pondered whether to rate-limit route programming in the production critical layer itself. To date, we have decided against it, as fast responses to organic events—such as switch/link failures, changes in traffic conditions, or route availability from adjacent domains—are critical for predictable performance and low packet loss in the steady state. While rapid route programming undoubtedly increases risk of failures, we believe the tradeoff is worthwhile so long as failures are rare and contained to one cluster (ideally sub-cluster failure domains).

How has CAPA evolved? What's next? CAPA's design is informed by experience accumulated since Google built its first datacenter network. The datacenters were built with

failure domain separation at physical, data and control plane levels, but this separation was not enforced. We relied on manual management operations (scripts invoked by humans) in the early days, and then factored out management plane functionality into dedicated services to reduce duplication across scripts. Security and reliability concerns led to access controls for these services. Datacenters becoming the basis for Google Cloud accelerated both capacity and feature growth, and underscored the need for more reliable operations. We invested in more advanced network modeling [43], automation in operations, and principles of layering for production operations. Large-scale outages like GCP incident 19009 [24] brought additional emphasis on considering the full transitive closure of dependencies in failure domain containment, BGP fail-static, and low dependency recovery. Our ongoing focus is on preventing erosion of enforcement over time and driving new directions that will further improve the safety and velocity of operations.

Automated rollback of operations beyond `Drain()`: An interesting side-effect of 82% reduction in elapsed outage duration is that newer outages are almost always unique and difficult to prevent with simple principles, even with the benefit of hindsight. This elevates the need for faster recovery. We are working towards fully intent-driven network management that will allow us to quickly roll back to known good state upon failures. For this, we are extending rollback support beyond in-flight `Drain()` operations, as well as *uplevelling* the regulation layer APIs to more directly express intent instead of using the current imperative APIs (Table 2).

Upleveling SDN's management interface: Today, workflows reach their intended state by sequencing low-level imperative APIs. Workflows are not aware of each other's existence or intent. CAPA has created safe guards that prevent outages, but workflows are still vulnerable to data races and indefinite blockages. We are working on up-leveling APIs and increased awareness of concurrent workflows within the regulation and production critical layers to prevent these outcomes.

7 Related Work

Layering for network management. In a keynote lecture, Koley describes ZTN, a three-layered architecture that regulates network management operations applied to Google's B2 and B4 wide area networks [33]. At the operational layer, ZTN describes the importance of device-agnostic modeling with MALT [43] and 'what-if' change planning with TopoPlan [2] in advance of realizing those changes through lower layers. ZTN introduces the concept of isolating operation workflows from data/control planes by interposing a regulation layer which can enforce best practices. CAPA adopts the same principle of interposing regulation.

ZTN only briefly touches on other topics such as rate limiting and invariant monitoring. We go well beyond ZTN by describing the mechanisms we have built into CAPA which we have found to all be necessary to ensure safety, including drain-before-operate, protected cross section bandwidth, fleet-wide rate limiting, ACL enforcement and auditing, low-dependency requirements, and BGP static routes.

Production best practices. Several papers [1, 8, 25, 40, 44] describe reliability best practices for distributed systems. We do not propose novel best practices; we describe our experience in what mechanisms are necessary to systematically enforce them within a thin "regulation" layer.

Network management. The literature on network analysis and testing [28, 55, 67], simulation and emulation [35, 38, 69], verification [10, 16, 19, 20, 31, 32, 50, 57] and synthesis [37, 39, 42, 45, 59, 64] can be applied to change management. These mechanisms test or verify if changes are safe to proceed before releasing them, or automatically generate correct configurations from intent. WAN failure domain containment [34], network update and change planning [2, 3, 36, 51, 54, 58, 68], drain operations [62], impact analysis [21, 60, 68] and failure mitigation [17, 66] are also applicable. These mechanisms allow network operators to manage changes safely and effectively. This paper describes our experience piecing together such mechanisms into a cohesive architecture. While existing literature describes singular aspects of systemically enforcing best practices, we are unaware of any work describing experiences holistically enforcing all best practices.

Failure studies. Prior studies [5, 27, 48, 48] have shown the network to be among the major causes of cloud service outages. Along that line, many failure studies particularly focus on data center networks [4, 9, 11, 22, 25, 37, 38, 41, 47, 48, 61, 66, 70]. Among them, some studies [25, 40, 41] have conclusions that motivate our paper or align with our observations: (1) planned changes and maintenance contribute to a majority of the outages; (2) failures are inevitable in large scale complex systems, and we need to proactively prepare for failure (e.g., with failure domain isolation and fast recovery). Though our paper examines outage case studies as a qualitative evaluation of CAPA, our goal is not to comprehensively analyze the characteristics of observed outages.

8 Conclusion

Production best practices are only effective if they can be reliably enforced. CAPA demonstrates an approach for providing enforcement of best practices without relying on the good intent of every engineer and operator. CAPA has allowed us to maintain availability while facing the practical realities of a continually evolving and large scale production environment.

Even with a 73% reduction in incident rate and 82% reduction in outage duration, the pressure to deliver higher availability, higher operational and feature development velocity, and increased complexity in cluster networking means that this area remains ripe for further innovation.

References

[1] Heather Adkins, Betsy Beyer, Paul Blankinship, Piotr Lewandowski, Ana Oprea, and Adam Stubblefield. *Building Secure and Reliable Systems: Best Practices for Designing, Implementing, and Maintaining Systems*. O'Reilly Media, 2020.

[2] Mohammad Al-Fares, Virginia Beauregard, Kevin Grant, Angus Griffith, Jahangir Hasan, Chen Huang, Quan Leng, Jiayao Li, Alexander Lin, Zhuotao Liu, et al. Change management in physical network lifecycle automation. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 635–653, 2023.

[3] Omid Alipourfard, Jiaqi Gao, Jérémie Koenig, Chris Harshaw, Amin Vahdat, and Minlan Yu. Risk based planning of network changes in evolving data centers. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019*. ACM, 2019.

[4] Peter Bailis and Kyle Kingsbury. The network is reliable. *Commun. ACM*, 2014.

[5] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Ed.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[6] Tom Bedford. Internet outages: Spotify, Discord and more are finally back up. In *TechRadar*, 2022. https://www.techradar.com/news/live/internet-down-spotify-discord-facebook-and-more-are-all-suffering-outages.

[7] Anna Berenberg and Brad Calder. Deployment Archetypes for Cloud Applications. *ACM Computing Surveys*, 2022.

[8] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 1st edition, 2016.

[9] Ayush Bhardwaj, Zhenyu Zhou, and Theophilus A. Benson. A Comprehensive Study of Bugs in Software Defined Networks. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021*. IEEE.

[10] Yiyang Chang, Sanjay G. Rao, and Mohit Tawarmalani. Robust Validation of Network Designs under Uncertain Demands and Failures. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017*. USENIX Association.

[11] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. FBOSS: building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018*. ACM.

[12] David D Clark, John Wroclawski, Karen R Sollins, and Robert Braden. Tussle in cyberspace: defining tomorrow's internet. In *Applications, technologies, architectures, and protocols for computer communications, 2002*.

[13] Jérome Durand, Ivan Pepelnjak, and Gert Doering. BGP operations and security. *RFC*, 7454:1–26, 2015.

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

[15] Google Compute Engine. Designing resilient systems. https://cloud.google.com/compute/docs/tutorials/robustsystems.

[16] Seyed Kaveh Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd D. Millstein, Vyas Sekar, and George Varghese. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*. USENIX Association.

[17] Sifat Ferdousi, Massimo Tornatore, Ferhat Dikbiyik, Charles U. Martel, Sugang Xu, Yusuke Hirota, Yoshinari Awaji, and Biswanath Mukherjee. Joint Progressive Network and Datacenter Recovery After Large-Scale Disasters. *IEEE Trans. Netw. Serv. Manag.*, 2020.

[18] Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. Orion: Google's Software-Defined networking control plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 2021.

[19] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D. Millstein. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15*. USENIX Association.

[20] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the ACM SIGCOMM 2016 Conference*. ACM.

[21] Aaron Gember-Jacobson, Wenfei Wu, Xiujun Li, Aditya Akella, and Ratul Mahajan. Management Plane Analytics. In *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015*. ACM.

[22] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*.

[23] Google. Google Cloud Basics: Geography and regions. https://cloud.google.com/docs/geography-and-regions.

[24] Google. Google Cloud Networking Incident 19009. https://status.cloud.google.com/incident/cloud-networking/19009.

[25] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In *Proceedings of the ACM SIGCOMM 2016 Conference*. ACM.

[26] gRPC Authors. gRPC: Authentication. https://grpc.io/docs/guides/auth/.

[27] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing, 2016*. ACM.

[28] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014*. USENIX Association.

[29] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, 2013.

[30] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally Deployed Software Defined WAN. In *Proceedings of the ACM SIGCOMM Conference*, Hong Kong, China, 2013.

[31] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012*. USENIX Association.

[32] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and Philip Brighten Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013*. USENIX Association.

[33] Bikash Koley. Keynote lecture: The Zero Touch Network. In *International Conference on Network and Service Management*, 2016.

[34] Umesh Krishnaswamy, Rachee Singh, Nikolaj S. Bjørner, and Himanshu Raj. Decentralized cloud wide-area network traffic engineering with BLASTSHIELD. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022*. USENIX Association.

[35] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010.

[36] Bingzhe Liu, Kuan-Yen Chou, Pramod Jamkhedkar, Bilal Anwer, Rakesh K. Sinha, Kostas N. Oikonomou, Matthew Caesar, and Brighten Godfrey. Practical Automation for Management Planes of Service Provider Infrastructure. In *FlexNets '21: Proceedings of the 4th FlexNets Workshop on Flexible Networks Artificial Intelligence Supported Network Flexibility and Agility, 2021*. ACM.

[37] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. Automatic Life Cycle Management of Network Configurations. In *Proceedings of the Afternoon Workshop on Self-Driving Networks, SelfDN@SIGCOMM 2018*. ACM.

[38] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. CrystalNet: Faithfully Emulating Large Production Networks. In *Proceedings of the 26th Symposium on Operating Systems Principles, 2017*. ACM.

[39] Ajay Mahimkar, Carlos Eduardo de Andrade, Rakesh K. Sinha, and Giritharan Rana. A composition framework for change management. In *ACM SIGCOMM 2021 Conference*. ACM.

[40] Ben Maurer. Fail at scale: Reliability in the face of rapid change. *Queue*, 13(8):30–46, sep 2015.

[41] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A Large Scale Study of Data Center Network Reliability. In *Proceedings of the Internet Measurement Conference 2018, IMC 2018*. ACM.

[42] Jeffrey C. Mogul, Alvin AuYoung, Sujata Banerjee, Lucian Popa, Jeongkeun Lee, Jayaram Mudigonda, Puneet Sharma, and Yoshio Turner. Corybantic: towards the modular composition of SDN control programs. In *Twelfth ACM Workshop on Hot Topics in Networks, HotNets-XII, 2013*. ACM.

[43] Jeffrey C. Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with Modeling Network Topologies at Multiple Levels of Abstraction. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 403–418, Santa Clara, CA, February 2020. USENIX Association.

[44] Jeffrey C. Mogul, Rebecca Isaacs, and Brent Welch. Thinking about Availability in Large Service Infrastructures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017*. ACM.

[45] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software defined networks. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013*. USENIX Association.

[46] Jay Peters. Apple has resolved the outage affecting iMessage, Apple Music, the App Store, and other services. In *The Verge*, 2022. https://www.theverge.com/2022/3/21/22989393/apple-is-down-outage-music-imessage-maps-icloud-app-store.

[47] Rahul Potharaju and Navendu Jain. Demystifying the dark side of the middle: a field study of middlebox failures in datacenters. In *Proceedings of the 2013 Internet Measurement Conference, IMC*. ACM.

[48] Rahul Potharaju and Navendu Jain. When the network crumbles: an empirical study of cloud network failures and their impact on services. In *ACM Symposium on Cloud Computing, SOCC '13, 2013*. ACM.

[49] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve D. Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter evolving: transforming Google's datacenter network via optical circuit switches and software-defined networking. In *SIGCOMM '22: ACM SIGCOMM 2022 Conference*. ACM.

[50] Santhosh Prabhu, Kuan-Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020*. USENIX Association.

[51] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015*. ACM.

[52] Mubashir Adnan Qureshi, Yuchung Cheng, Qianwen Yin, Qiaobin Fu, Gautam Kumar, Masoud Moshref, Junhua Yan, Van Jacobson, David Wetherall, and Abdul Kabbani. PLB: congestion signals are simple and effective for network load balancing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022.

[53] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in System Design. *ACM Trans. Comput. Syst.*, 1984.

[54] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. Snowcap: synthesizing network-wide configuration updates. In *ACM SIGCOMM 2021 Conference*. ACM.

[55] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, Hrishikesh B. Acharya, Kyriakos Zarifis, and Scott Shenker. Troubleshooting blackbox SDN control software with minimal causal sequences. In *ACM SIGCOMM 2014 Conference*. ACM.

[56] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Hanying Liu, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and

Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM '15*, 2015.

[57] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. Probabilistic Verification of Network Configurations. In *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. ACM.

[58] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. A network-state management service. In *ACM SIGCOMM 2014 Conference*.

[59] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H. Y. Wong, and Hongyi Zeng. Robotron: Top-down Network Management at Facebook Scale. In *Proceedings of the ACM SIGCOMM 2016 Conference*.

[60] Aisha Syed, Bilal Anwer, Vijay Gopalakrishnan, and Jacobus E. van der Merwe. DEPO: A platform for safe deployment of policy in a software defined infrastructure. In *Proceedings of the 2019 ACM Symposium on SDN Research, SOSR*.

[61] Daniel Turner, Kirill Levchenko, Alex C Snoeren, and Stefan Savage. California fault lines: understanding the causes and impact of network failures. In *Proceedings of the ACM SIGCOMM 2010 Conference*, 2010.

[62] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, Yee Jiun Song, and Tianyin Xu. Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018*. USENIX Association.

[63] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[64] Andreas Voellmy, Junchang Wang, Yang Richard Yang, Bryan Ford, and Paul Hudak. Maple: simplifying SDN programming using algorithmic policies. In *ACM SIGCOMM 2013 Conference*.

[65] David Wetherall, Abdul Kabbani, Van Jacobson, Jim Winget, Yuchung Cheng, Brad Morrey, Uma Parthavi Moravapalle, Phillipa Gill, Steven Knight, and Amin Vahdat. Improving network availability with protective reroute. In *SIGCOMM 2023*.

[66] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: automating datacenter network failure mitigation. In *ACM SIGCOMM 2012 Conference*.

[67] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *Conference on emerging Networking Experiments and Technologies, CoNEXT '12, 2012*. ACM.

[68] Ennan Zhai, Ang Chen, Ruzica Piskac, Mahesh Balakrishnan, Bingchuan Tian, Bo Song, and Haoliang Zhang. Check before You Change: Preventing Correlated Failures in Service Updates. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020*. USENIX Association.

[69] Qizhen Zhang, Kelvin K. W. Ng, Charles W. Kazer, Shen Yan, João Sedoc, and Vincent Liu. MimicNet: fast performance estimates for data center networks with machine learning. In *ACM SIGCOMM 2021 Conference*.

[70] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[71] Ariel Zilber. PlayStation network goes down, sparking panic among gamers. In *The New York Post*, 2022. https://nypost.com/2022/03/23/playstation-network-goes-down-sparking-panic-among-gamers/.

NetAssistant: Dialogue Based Network Diagnosis in Data Center Networks

Haopei Wang, Anubhavnidhi Abhashkumar, Changyu Lin,
Tianrong Zhang, Xiaoming Gu, Ning Ma, Chang Wu, Songlin Liu,
Wei Zhou, Yongbin Dong, Weirong Jiang, Yi Wang
ByteDance

Abstract

In large-scale data center networks, answering network diagnosis queries from users still heavily rely on manual oncall services. A widespread scenario is when network users query whether any network issue is causing problems with their services/applications. However, this approach requires extensive experience and considerable efforts from network engineers who must repeatedly go through lots of monitoring dashboards and logs. It is notoriously slow, error-prone, and costly. We ask: is this the right solution, given the state of the art in network intelligence?

To answer, we first extensively study thousands of real network diagnosis cases and provide insights into how to address these issues more efficiently. Then we propose an AI enabled diagnosis framework and instantiate it in a task-oriented dialogue based diagnosis system, or colloquially, a chatbot, called NETASSISTANT. It accepts questions in natural language and performs proper diagnosis workflows in a timely manner. NETASSISTANT has been deployed and running in the data centers of our company for more than three years with hundreds of usages every day. We show it significantly decreases the number and duration of human involved oncalls. We share our experience on how to make it reliable and trustworthy and showcase how it helps solve real production issues efficiently.

1 Introduction

Providing high visibility is increasingly challenging in modern hyper-scale and heterogeneous data center networks. One fundamental and essential service is handling network diagnosis queries from network users. As networks continue to grow in scale, speed, and link utilization, it has become increasingly challenging for human operators to manually monitor and diagnose network issues. Traditional approaches to network management, which rely on human operators looking at a screen of data to understand the network state, are no longer sufficient. Instead, network management solutions have become more automated than before, with a shift towards software-defined networking (SDN) and automated processes that dynamically drill down based on current conditions and even automatically react to network events. These queries normally come from cloud applications/services generating anomalies which are likely caused by network incidents. For instance, colleagues from the advertisement recommendation team observe network timeout exceptions that lead to the crash of their machine learning jobs. They are eager to ascertain whether it is a network issue and the scope and severity of the issue so that they can make business decisions such as waiting for auto-recover, changing to another backup computing cluster, or temporarily downgrading the recommendation algorithms (e.g., using a simple client side algorithm instead of the server side one).

However, handling these network diagnosis queries has always been time consuming and labor intensive work in large data center networks. The reason behind this comes from both the network user side and the network engineer side. From the perspective of network users, limited by the network domain knowledge and necessary permissions, they normally lack a network-wide view and are not able to access a variety of network monitoring primitives. As a result, network users often need to turn to network engineers for manual assistance. We have a user behavior study to investigate what exactly the needs of network users are in their daily work (Section 2.1). From the perspective of network engineers, they have a variety of network monitoring primitives to monitor, a large amount of monitoring data to process, and too many small but not negligible network incidents to investigate (Section 2.2). As a consequence, extracting useful information from the enormous data and replying to network users becomes highly tedious and time consuming work, and highly relies on the expertise of each network engineer. Therefore, we argue that there is a **gap between the network diagnosis needs of network users and network monitoring primitives in big data center networks.** Network engineers are working hard to bridge the gap with their extensive experience and expertise. And big cloud companies have a considerable number of

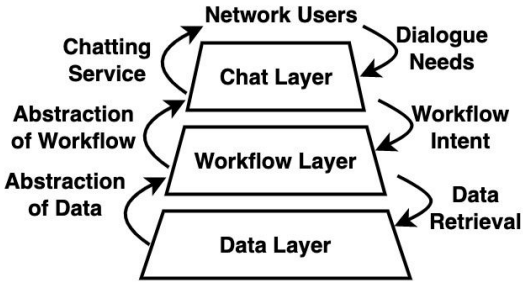


Figure 1: Level of Abstraction in NETASSISTANT

network engineers [11] who take turns to undertake the oncall services.

Researchers have built several querying [13, 15, 19] and diagnosis [12, 16, 17, 20, 23, 25, 28] tools to bridge part of the gap. However, building such automation tools has limited effectiveness since they are ad-hoc, not scalable, and introduce an extra learning curve to network users. Network users may not know these tools, or are not able/willing to learn them. Consequently, we believe a comprehensive solution should be general-purpose, query format independent, easy to leverage the O&M (operation and maintenance) experience from network engineers, and comply with performance and data scale requirements.

Inspired by the technique of task-oriented dialogue systems in the NLP (natural language processing) domain and the top-down idea in network telemetry [24], we come up with an idea to build a virtual assistant (chatbot) to take the networking related queries in natural language and leverage the experience from network engineers to perform proper diagnosis functions. While building a dialogue system/chatbot to help users/customers is already a common practice in many companies ([1, 3, 4, 10]), it is not trivial in the data center networking domain due to the requirement of high accuracy, scalability and efficiency. As shown in Figure 1, we divide the objectives into three levels of needs. Network users have the chat needs from the chat layer. Then the chat layer requires appropriate diagnosis functions from the workflow layer. The workflow layer needs efficient data retrieval from the network monitoring primitives. Conversely, each layer provides a corresponding abstraction to its upper layer. We notice that each layer encounters unique challenges as follows:

- First, networking diagnosis requires precise description and input, while a considerable proportion of queries from network users are not clear enough (Section 2.1). Therefore, it is hard to build a natural language understanding module since any misunderstanding or misconfiguration may cause immeasurable losses.
- Second, building suitable and high-quality methodologies for different diagnosis cases is challenging. We argue that network engineers have the most say on how to select and customize the diagnosis functions based on their experience and expertise. How to automatically leverage the know-how from network engineers and convert it to executable workflows is not trivial.
- Third, the performance bottleneck of a chat system for network questions mainly comes from the retrieval of underlying data. Considering the various types of network monitoring primitives and the huge volume of monitoring data, we need a comprehensive solution for data storage and data retrieval.

We propose our solution, NETASSISTANT, which leverages the technique of task-oriented dialogue systems to provide virtual assistant services to network users. NETASSISTANT takes the queries from network users in natural language, selects appropriate processing workflows that are created by network engineers, and responds to the users in a timely manner. NETASSISTANT contains three novel functional modules to achieve the three layers of abstraction and address the challenges. NETASSISTANT is deployed on top of a large scale distributed network monitoring infrastructure and deals with terabyte level of monitoring data every day.

In summary, we make the following contributions:

- We provide a user study of network users, a quantitative study of network incidents, and monitoring primitives in our production network.
- We propose an end-to-end distributed system, NETASSISTANT, which can make use of the O&M experience of network engineers to answer network diagnosis questions from network users. NETASSISTANT has been deployed and iterated in our production network for over three years and has tens of thousands of usage.
- We share our experience and lessons learned in customer service on the front line for network users.

We construct our paper as follows. Section 2 provides a network user behavior study, a quantitative study of network incidents and monitoring primitives, and motivates this paper. Section 3 describes the system design of NETASSISTANT. Section 4 provides deployment details, case studies, and evaluation results. In Section 5, We share interesting lessons we have learned from our experience of providing network diagnosis and troubleshooting functions. Section 6 and 7 describe related work and conclude the paper.

2 Measurement Study and Motivation

A typical interaction between a network user and a network engineer is that the network user raises oncall questions while the network engineer on duty answers them. The oncall process is time consuming and labor intensive work, and this paper aims to provide a comprehensive solution to help both network users and engineers. Thus, the first step is to understand the needs of network users and the challenges faced by network engineers. In order to accurately grasp the most

real and urgent needs of network users, we leverage years of network oncall history and propose a network user behavior study. To better understand the difficulties and challenges of network diagnosis from the perspective of network engineers, we conduct quantitative analysis of network incidents and monitoring primitives collected from our data centers.

2.1 Network User Behavior Study

We believe grasping the most real and urgent needs of network users is the first step to design a good network diagnosis tool. To achieve this, we leverage the daily network oncall records[1] in our cloud networks, and conduct a user behavior study. These oncall records include network related questions (containing both underlay and overlay network) raised by network users and processing dialogues between network users and engineers.

By Type		**By Objectives**	
Configuration Request	2196	DC/AZ Related	1867
Host Network Issue	837	Host(S) Related	4557
Network Consultation	3073	Link Related	462
Others / Void Oncall	1083	Device Related	113
Total Count	7189	Others	190

Table 1: Network User Behavior Study

Table 1 shows the statistics about the collected oncall records. We collect about 7000 network oncall records from our oncall system and try to categorize them by question types and objectives. We observe that, by question types, most questions can be categorized into **Configuration Request**, **Network Issue**, and **Network Consultation** types. Configuration requests are to seek manual assistance in network related configurations such as BGP, firewalls, gateways, and virtual switches. For example, frequent requests include virtual IP/public IP assignment, firewall allow-list setup, and BGP configuration setup. Host (physical server, virtual machine, etc.) network issue questions are raised when users observe obvious network issues with their host and seek quick fixes. Network consultation questions ask about network health status, which is the category with the highest proportion. The most commonly asked question is "*Is there any problem with the network in xxx*". We believe that answering most of these three types of questions can be automated, while some dangerous operations should be manually confirmed before being taken into action.

We also analyze the oncall questions from their objectives. As summarized in Table 1, most questions are in the dimension of host level, including physical hosts, virtual machines, IP pairs, subnets, and clusters. Besides, questions related to DC (data center) and AZ (available zone) also occupy a certain proportion. The reason behind this is that questions from network users usually originate from the servers they use or the data centers they are aware of.

We also conclude that network users have the following behavioral characteristics when submitting an oncall question.

- First, the questions from network users are normally very broad, e.g., the network health status of a large cluster with thousands of servers or an entire data center.
- Second, the objectives of the questions are sometimes very vague. While data center networks have a strict and accurate naming specification, most network users are unaware of that. For instance, one user may ask about the network health in Santa Clara, which refers to an AZ named US-WEST-1 in the specification[2].
- Third, some questions are even missing a clear intent. Sometimes, users are very anxious to seek manual services and may ask questions like "*Any network incident right now?*" or "*US Redis service is lagging.*" It is hard to understand directly from their questions which part of the network they are asking.

Our tool is designed to take, understand, and respond to network users who will ask similar questions as in oncalls. In Section 2.3, we will discuss the corresponding research challenges in terms of question understanding and processing.

2.2 Network Incidents and Monitoring Primitives Study

Handling network diagnosis requests is challenging for network engineers since the volume of monitoring data and the detected network incidents is huge. Network engineers need to go through a large amount of data to find the possible root causes. Therefore, network diagnosis becomes a time consuming and label intensive work. In order to better understand the difficulties and challenges, we leverage our detected network incidents and collected network monitoring data in our production network to provide a measurement study.

First of all, we aim to know how frequently network incidents happen in daily operations. We define network incidents as **any network abnormal behaviors that could violate the service-level agreement (SLA) of the network.** More specifically, network incidents include abnormalities of the metrics (e.g., packet drop, latency, bandwidth) that can be perceived by the end users and abnormalities monitored from the network components. Network components can be hardware (e.g., switch, circuit, optical module, and network card) or software (e.g., operating system, virtual switch, and configuration). The number of daily network incidents can illustrate the complexity and effort of diagnosing network failures.

In order to avoid bias caused by subjective factors like threshold, we choose the switch running errors/exceptions, which are detected by switch syslog to represent network incidents for the quantitative study. Syslog [8, 9] is collected

[1] This user behavior study only includes data center network oncall questions. But our tool also supports other types of networks, e.g., IT/office networks, edge/CDN networks, etc.

[2] We use our venue city name to represent a data center.

```
11: 49:20 XXXX_XXXX_XXX-02-08-01.S3-3-3
%%01DEVM/1/hwBoardFail_active(l):CID=0x80fa0003-alarmID=0x08130054;The
board partially failed. (EntPhysicalIndex=17301505, EntPhysicalName=LPU slot 8,
EntityType=2, EntityTrapFaultID=132097, Reason=Board CANBUS failed.)

        Jun 21 2023 09:41:15.196: XXXX_XXX_XXXX0-H-01-08.S1-2-6 %PORT-
        3-ERR_FRAME: Received error frames on interface
        HundredGigabitEthernet 0/57. Please check the physical link.

2023-06-21T09: 51:25 XXXX_XXXX_XXXX-K-08-33.S2-1-1
%%10IFNET/5/LINK_UPDOWN: -DevIP=10.X.X.X; Line protocol state on the
interface M-GigabitEthernet0/0/0 changed to down.
```

Figure 2: Sample Syslogs Indicating Switch Exceptions

from switch logging services and contains crucial running information of switches. Sample syslogs are shown in Figure 2, which indicates switch hardware abnormalities like board/port/link failure or software abnormalities like protocol (e.g., BGP, ISIS) down/flapping[3]. We choose one of our data centers containing about 14,000 physical switches and count all syslog errors/exceptions for the **entire month** of June in 2023. The statistics are shown in Table 2.

Error Type		**Error Type**	
Total Count	118,123,984	Linecard Issue	20,690
Queue Issue	57,036,587	Process Exception	83,465
Interface Issue	44,662,097	Frame Issue	2,280,956
Protocol Issue	8,360,567	Others	5,679,622

Table 2: Syslog Statistics for One Month

We can observe that even for just one data center, we encounter around 4 million different kinds of switch abnormalities per day. While most abnormalities will auto-recover very soon and most are in the packet/frame level, it is still annoying for network engineers since it is hard to tell whether the abnormalities are user-insensitive or not. Correspondingly, when network users experience network anomalies, they also need network engineers to help them diagnose and demarcate within such a huge number of switch abnormalities.

Other than the switch syslog, there are also several other types of primitives that monitor the status of different components of the network. For example, we track the packet drop and latency metrics by using the connectivity monitoring primitives and network traffic volume and composition by using the traffic monitoring primitives. We conduct another measurement study for the daily data volume of each type of monitoring primitives for the same data center. The results are shown in Table 3.

From the statistics shown in Table 3, almost every category of monitoring primitives has a huge amount of data every day. Meanwhile, most diagnosis requests from network users often require a combination of different types of data to analyze. For example, we may use connectivity monitoring data (e.g., PingMesh [18]) to check whether there is any connectivity issue and locate the IP pairs of abnormal traffic and then use traffic monitoring data (e.g., sFlow) to further troubleshoot the packer headers to find the root cause. This puts a huge burden on network engineers to retrieve the data reactively. Therefore, an automated tool that can perform various diagnosis procedures will be helpful and unleash network engineers.

[3]While different vendors have different syslog formats, we parse and normalize the syslog according to their specifications.

Monitoring Primitive Category	**Data Volume per Day**
Connectivity (e.g., PingMesh [18], EverFlow [27], etc.)	65GB
Traffic (sFlow, SNMP, etc.)	12TB
Switch Syslog	35GB
Host Monitoring	4.3GB
Routing Configuration	425G
Optical Module (DDM or DOM)	5.5GB
Other Monitoring Primitives	27GB

Table 3: Data Volume of Monitoring Primitives per Day

2.3 Motivation and Research Challenges

We aim to automatically answer questions from network users. An existing idea is the text-to-SQL [22] parsing solution, where the natural language question could be converted into a SQL query. However, this solution is limited in our scenario since data retrieval is just one step in the overall diagnosis process and our network monitoring infrastructure involves variety of data sources in addition to relational databases, for example, non-SQL database, file system, message queue, SSH agent, etc. Therefore, our idea is to leverage NLP technique to map user queries to pre-defined workflows. As illustrated in Figure 1, we model the whole system into three abstraction layers with their unique challenges.

The **chat layer** is responsible for providing a typical task-oriented dialogue service for network users. However, it is challenging since network diagnosis requires precise input while the questions from users are arbitrary (as shown in Section 2.1). This places three requirements on the NLU (Natural Language Understanding) module. First, it should be able to cover various granularity of data center networks, from global regions to small network components. Second, besides the standard terminologies that are strictly defined in the specifications, the NLU module should also understand most common expressions that people might use to describe data center networking in everyday conversations at work. Third, when word slots or the intent is not clear, the chatbot should leverage the dialogue function to guide users to provide more information or ask more precise questions.

Figure 3 illustrates the ideal dialogue service we aim to provide to network users. Our chatbot performs corresponding diagnosis functions to answer user questions if both objectives and the intent are clear. If not, our chatbot will use dialogue to guide network users to provide more information

The **workflow layer** is responsible for providing and performing diagnosis functions for the chat layer. We argue that network engineers have the most say on how to diagnose and troubleshoot network questions as they have extensive expe-

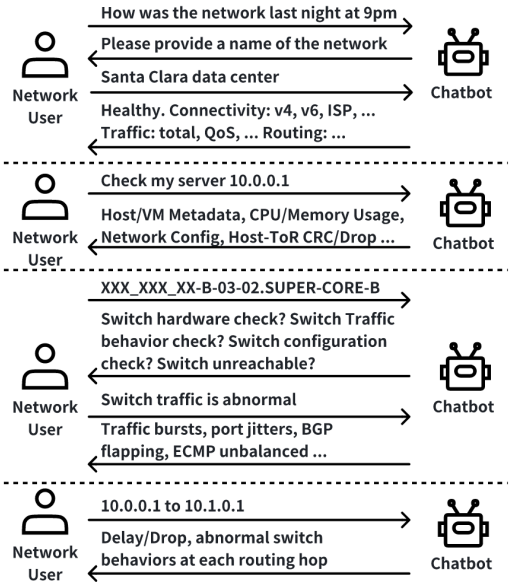


Figure 3: Sample Dialogues between Users and Chatbot

rience in diagnosing and dealing with network problems on the front line and have a deep understanding of the specific network situation, e.g., hardware vendors and models, network architecture, common issues, and user traffic patterns. Inspired by this, our tool introduces a novel functional module to collect and convert the experience from network engineers into a considerable number of executable workflows.

The **data layer** is responsible for providing the abstraction of underlying monitoring data. The performance bottleneck of our chatbot mainly comes from the retrieval of distributed and large volume underlying monitoring data. Reactively analyzing the data takes a long time and is unacceptable for a chatbot. We come up with a solution that combines on-demand querying and proactive alerting to achieve a good trade-off between the response latency and the timeliness of diagnosis results.

3 System Design

In this paper, we propose a virtual assistant tool, named NETASSISTANT, to answer the network diagnosis questions in natural language from network users. The key idea behind is that NETASSISTANT aims to understand what intent and objects network users are asking about and makes responses using corresponding predefined workflows, which are learned from network engineers. To this end, NETASSISTANT is designed as three-fold. First, in offline, NETASSISTANT automatically converted the O&M knowledge and experience from network engineers into multiple executable workflow functions. Second, in runtime, NETASSISTANT analyzes the

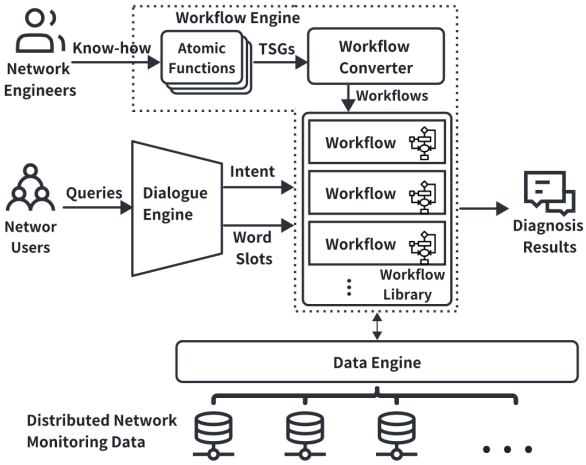


Figure 4: System Design of NETASSISTANT

intent and word slots of user queries, forwards this information as parameters into corresponding workflow functions, and responds to the users with the outputs of the function. Third, NETASSISTANT applies several optimization and trade-offs of the underlying monitoring data storage and retrieval component to meet the performance needs as a real-time chat assistant. In this section, we describe the design and implementation details of NETASSISTANT.

3.1 System Architecture

NETASSISTANT works as an always-on service for the data center networking environment. As shown in Figure 4, NETASSISTANT consists of three main functional modules: 1) **Dialogue Engine**, which provides a dialogue environment for users and understands the intent and word slots from the user queries; 2) **Workflow Engine**, which converts the knowledge and experience from network engineers into workflows and processes proper workflows for each user query; 3) **Data Engine**, which manages all underlying distributed network monitoring data storage and provides high-performance data retrieval. In the following subsections, we describe the design details of each functional module.

3.2 Dialogue Engine

Dialogue Engine module provides network users with a dialogue environment that presents a conversational experience with multiple rounds of question and answer. The input from the user is plain text in natural language. The output form is relatively rich, which can be rich text, pictures, interactive components, and hyperlinks. We leverage the typical framework of a task-oriented dialogue system, and the key challenge is to build the Natural Language Understanding

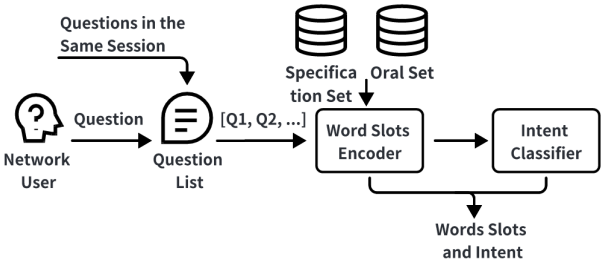


Figure 5: Dialogue Engine Module

(NLU) component. The key target of the NLU component is to understand the **word slots** and **intent**. The intent of each question will decide which diagnosis function (workflow) that we will use, and the word slots will be the input parameters. Figure 5 shows the detailed processing steps for language understanding, which will be explained in the following.

The Dialogue Engine provides session based dialogue management to manage user questions. All user questions belonging to the same session will be converted into a question list for understanding. The first step for understanding is to understand and parse the word slots from the questions. The word slot here could be the name of any component in the data center networks. Therefore, we collect nouns from our production network and categorize them into different types according to their role in the networks, e.g., region, country, state/province, AZ, office, building, pod, subnet/cluster, server, switch, circuit, and optical module.

We put the words into two word sets. The first word set contains terminologies strictly defined in our datacenter network specification, which we call the **Specification Naming Set**. The second word set contains the terminologies and their oral expressions, which we call the **Oral Naming Set**. The Specification Naming Set is a subset of the Oral Naming Set. We build a multi-to-multi mapping between the two sets. It is because one terminology may have multiple oral expressions (both "Santa Clara" and "US West" refer to "US-WEST-1"), and sometimes there may have multiple AZs in the same place (both "US-WEST-1" and "US-WEST-2" locate in "Santa Clara").

We build these two sets for the word slots parsing, standardizing, and encoding. We first use the Oral Naming Set to parse the word slots from the user questions. Then, we use the mapping between the two sets and the Specification Naming Set to standardize each word. After that, we encode each word by using its category name. For example, we encode the word "Santa Clara" with "available_zone_#14" from the original sentence for further intent understanding and parse a parameter {"az_1": ["US-WEST-1"]} for workflow processing.

Then, we process the encoded questions for intent understanding (i.e., text classification). We iterate through two technical solutions for text classification. The first one is based on word-level Convolutional Neural Network (or ConvNet, CNN) [26]. Motivated by the technological breakthrough in the area of Large Language Model (LLM), our second solution is an LLM-based solution. We will detail the two technical solutions in the following.

Word-Level CNN-Based Solution. Our idea is to leverage a training based classifier to classify the category/intent of the given questions. To achieve this, we collect a considerable number of questions for each intent as the training dataset for a word-level CNN classifier offline. In the runtime after the encoding process, the classifier generates a classification score of the text for each intent and selects matched intent(s) based on a predefined threshold.

LLM-Based Solution. Our idea is to leverage the powerful comprehension skills of the LLM to do multiple-choice questions. We use the technique of *few-shot prompting* to add in-context learning where we demonstrate different intents in the prompt and let the language model make selections. One sample of the prompt is:

Intent check_switch_traffic is to check the traffic indicators of a switch. Intent check_host_network is to ... Please select corresponding intents for the following questions: "My devbox is unreachable. VM host_ip_#19."

Since we already have >100 intents, which will make the generated prompt sentence too long, we would like to select a relatively smaller set of intents (around 10) to shorten the prompt. To achieve this, we apply *vector embedding* to both the user questions and the description of intents and utilize a similarity based searching method to select the intents.

Based on the classification results of the user questions, there could be four different cases. 1. If both the intent and words (parameters) are clear, the chatbot will proceed with the selected diagnosis function. 2. If the intent is clear, but some words are missing or have more than one possibility, the chatbot will indicate the user to supplement this part of the information. 3. If the classifier selects more than one intent, the chatbot will indicate the user to select one. 4. If no intent is selected, the chatbot will respond with the manual page containing how to use the chatbot and popular examples.

3.3 Workflow Engine

The Workflow Engine module provides network engineers with the framework to create multiple workflows based on their expertise. The module consists of three primary components: atomic functions, workflow converter, and workflow library, which work seamlessly together to enable efficient workflow creation and execution. The expressive atomic functions allow network engineers to create simple, flexible, and interactive diagnosis logic. The workflow converter converts the logic into executable workflows, and the workflow library supports triggering and executing the workflows at runtime.

Atomic functions: The atomic functions form the build-

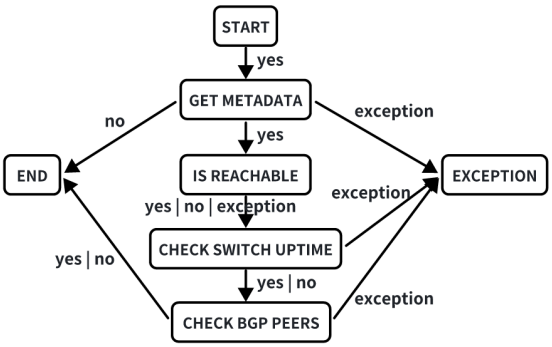


Figure 6: Example of TSG for Switch Reachability Check

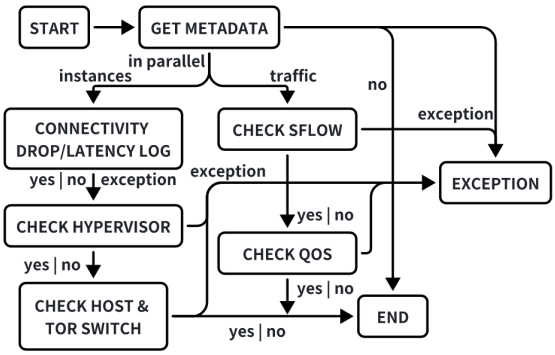


Figure 7: Example of TSG for Storage Service Check

ing blocks of our workflow system, with each encapsulating a particular method unique to a specific network entity and its metrics data. Entities could include switches, links, interfaces, available zones, and more. The decision-making method checks the metrics data of the entity over a time range and determines if there is any abnormality. Additionally, atomic functions may include an optional action process to analyze specific scenarios within the workflow, such as pausing a service on a switch to check if it caused overload or triggering a traceroute to display possible routing paths.

For all common network components, we tailor a set of atomic functions based on the monitoring primitives to meet various diagnosis requirements. For example, based on the sFlow data of a link, we can check if the overall utilization is high, or if there is any kind of traffic violating the QoS rules, or if there is any service traffic experiencing a sudden increase or decrease. There may be multiple algorithm implementations for the same function. The threshold details can be customized based on the experience and expertise of network engineers.

Workflow converter: Network engineers can configure a set of atomic functions to become a **Troubleshooting Guide** (TSG). A TSG comprises multiple atomic functions that allow specifying the data entity to be analyzed within a given time range using decision-making logic. The TSG operates like a state machine, existing at any given point at a specific stage (atomic function). The outcome of each atomic function determines the transitions between stages.

Network engineers can configure these TSGs through a visual interface. The configuration includes the selection and threshold of atomic functions, arranging them in to a flowchart and specifying the input such as switch names or time periods. It is possible to customize different versions of a TSG with the same name depending on the input value. For example, switches from different vendors may have different *check_switch_hardware* TSGs. TSGs can be updated and adjusted dynamically with the change of network architecture.

Workflow Engine then converts the TSGs to executable workflow logic and adds essential exception handling, supplementary background knowledge, associated network events, and common help entries. For instance, all workflows will be added the logic to check if there are any known/ongoing network issues or changes and recommend other tools for non-network issues. Executable workflows are stored in the workflow library.

Workflow library: The workflow library is responsible for selecting and executing proper workflows based on the extracted intents and parameters. Some diagnostic workflows are only available to specific people (e.g., specific network engineers). Thus, we apply role-based access control before executing the workflows. We list the most commonly used workflows in Section 4.2. All monitoring data access will go through the Data Engine which we will describe in Section 3.4.

Figure 6 and 7 presents two simplified examples of TSGs designed to check switch reachability and storage service health (e.g., a SQL cluster). Respectively, upon receiving the switch name or service name as input, the workflows consists of multiple atomic functions, including getting metadata, retrieving monitoring data, detecting abnormalities and making judgments. Finally, if all checks pass and no abnormality is found, the entity is appropriately marked as healthy.

3.4 Data Engine

In order to track various aspects of the network, we utilize multiple monitoring primitives, as outlined in Section 2.2. The Data Engine is responsible for providing the abstraction and query entry of network monitoring data. However, as shown in Table 3, extensive analysis of large data sources is impractical, such as analyzing the last X minutes of sFlow data across multiple links. This is the performance bottleneck of the entire system.

The Data Engine utilizes two strategies to ensure efficient data retrieval. Firstly, data sources that are relatively low in volume, such as switch configurations, can be stored and analyzed in their current form. These sources do not change

frequently, so analyzing them in real time is feasible.

However, high volume data sources are more challenging to manage. While the data generated may contain a significant amount of normal, expected behavior, only a small fraction of it is valuable for diagnosis. As such, we use a proactive approach to monitor the data for anomalies, which are flagged as alerts. We move the execution of some atomic functions that use high volume data sources to become always-on operators generating alerts. The alerts are then tracked, and when we need to query high volume data sources, we retrieve the alerts instead of the full data. For example, only a small portion of syslog data, namely BGP flapping or line card errors, can actually help diagnose underlying network issues. For connectivity monitoring data, normally, only jitters that violate the SLAs will get the attention of network engineers.

Trade-off: Leveraging the proactive alerts achieves a trade-off. We reduce the response time for generating diagnosis results but sacrifice some timeliness since generating alerts is not real-time but periodic and increases some computational burden. We have set the following criteria to determine which data sources are suitable for proactive alerts. First, the data sources should have high volume and only the abnormality is valuable for diagnosis. Second, the quality of data sources should be high enough that the network team may directly get involved in the discovered alerts. Third, the latency increased by the alerts should not violate the service level objectives of incident discovery and incident diagnosis promised by the network team. For instance, the network team promises to detect network issues within 2 minutes, consisting of a delay from monitoring data and a delay from generating alerts.

Implementation of Alerting System: We implement and deploy the alerting system as a standalone distributed system which consumes real time monitoring data and generates **alerts** such as *batch BGP peers down*, *interface flapping*, etc. The alerting system is responsible for generating, aggregating, prioritizing and publishing alerts, and managing the life cycle of each alert. The Data Engine of NETASSISTANT is one of the consumers and it subscribes to a subset of the generated alerts. The alert generating processes are implemented as streaming tasks in the monitoring data collection pipeline before the data storage. This design reduces the overhead of heavy data I/O compared with frequent database queries. The streaming tasks subscribe to the monitoring data from message queues (e.g., Kafka [2]) and utilize a sliding window for periodic detection. The period is usually set to a minute level due to our promises about the service level objectives. Some high priority alerts will directly engage the network team and trigger a fix, which we will discuss this situation in Section 5.1.

4 Evaluation

In this section, we first provide the implementation and deployment details of NETASSISTANT. Then, we share several interesting case studies. At last, we present the evaluation results of our tool in terms of usage, accuracy, and performance.

4.1 Implementation and Deployment

We describe the implementation and deployment details of NETASSISTANT, including Dialogue Engine, Workflow Engine and Data Engine. The Dialogue Engine consists of client-side and server-side. The client-side is built on top of a commercial business chat and collaboration tool as a chatbot application. The chatbot application is able to access content, collect information, conduct operations, and support interaction through messages in both private chat and group chat. The server-side implements NLU and is deployed in a cluster of three server machines. For the NLU component, we collect around 1,000 data entries for each workflow to train the CNN-based classifier and leverage a 13-billion-parameter large language model. The Workflow Engine is deployed in a computing cluster with 14 instances in different data centers for the consideration of disaster tolerance and high throughput. It also contains a front end for the network engineers to create and configure TSGs. The Data Engine is deployed using a server alongside the monitoring data storage in every available zone of our network. The whole system was launched in April 2020 as an always-on company wide service. And it has been continuously iterated in terms of techniques and supported workflows.

4.2 Case Study

We share several diagnosis cases, including user side behavior, diagnosis workflow, and follow-up actions. We also summarise the most commonly used workflows in daily work.

4.2.1 Case 1: Host Network Issue

User Question: A user queried an IP address as her SQL cluster encountered a connection exception to this instance.
Diagnosis Workflow: The bot identified it as an IP of a virtual machine and performed the *check_iaas_ip_network* workflow. After conducting a multi-dimensional diagnosis on this virtual machine, the workflow found that the link between its physical machine and the ToR (Top-of-Rack) switch started to have considerable CRC errors from the SNMP monitoring data, and the ToR switch also reported CRC errors from syslog. Thus, the bot responded with an unhealthy diagnosis conclusion and informed the current oncall network engineer about this issue.
Follow-up Actions: The CRC issue was fixed right after cleaning up the port on the switch.

4.2.2 Case 2: Data Center Network Issue

User Question: A user reported a suspicious network issue in a data center at around 2 AM since her managed service

had a wave of failures, and the log showed many connection errors.

Diagnosis Workflow: The bot performed the *check_idc_network* workflow and found that there was a spike of packet drop and latency increase, which lasted for around 2min according to the connectivity monitoring results. Involved IP pairs can be aggregated into a /48 IPv6 subnet. The workflow further noticed that there were several BGP withdrawn events whose peers pointed to the same switch and several ECMP imbalance exceptions from nearby switches syslog. Thus, the bot made a conclusion that there was a short-lived network issue and the root cause was due to a switch down event.

Follow-up Actions: The bot responded with the diagnosis results, estimated impact and suggestions to the user. Network engineers further confirmed the switch was down and isolated the switch. However, the bot did not see switch unreachable alert from the switch reachability monitoring results the first time, and then the alert was 2 hours late. The network engineers further found that the switch reachability checker only checked the switch management port, which in this case was not down immediately, but the loopback port was down. So, they learned from this case and updated the corresponding TSGs with new switch reachability checking logic.

4.2.3 Case 3: Large Scale Network Failure

User Question: To investigate the cause of performance degradation in a core service, a user queried the network status of her service.

Diagnosis Workflow: The *check_computing_service* workflow discovered multiple ping drops and high latency alerts between regions, along with configuration changes that were made to bring up a plane. The workflow also found the network team had already noticed and engaged in this issue in advance and had an initial conclusion that the configuration change might be the root cause. Therefore, the bot responded with the conclusion from the network team and suggested a downgrade to the service algorithm.

Follow-up Actions: Further investigation by the network engineers revealed that the misconfiguration causing the issue was not on the recently updated devices but rather in previously misconfigured route maps on a Point of Presence (PoP) device. This misconfiguration increased the priority of routes advertised by the PoP, leading to congestion and packet drop. To prevent such issues in the future, they plan to add a verification module (e.g., control plane and data plane verifiers) that will detect control and data plane issues.

4.2.4 Most Commonly Used Workflows

We list the most commonly used workflows in Table 4. Similar workflows are categorized together. Most DC level and IP level workflows support checking a single target or a pair (source and destination), and a time point or period is optional.

Workflows	Explanation
check_pod_network check_az_network check_idc_network check_region_network	Data center level network status workflows, including connectivity (internal, external, overlay, underlay, v4, v6, subnets and etc.), bandwidth & utilization (different types of links, different granularity), switches and existing network incidents & changes.
check_phy_ip_network check_iaas_ip_network check_vip_network check_rdma_network check_p4_network	IP level network status workflows, including software stack check, hardware status check, network environment (nearby switches) check
check_switch_reachable check_switch_hardware check_switch_traffic check_switch_config	Switch health status check, including metrics from switch OS (syslog), protocol (e.g., SNMP, BMP), hardware (e.g., linecard, OTN) and external monitors.
check_direct_connect check_bbone_link check_isp_link	Physical link level status workflows, mainly used by network team, including physical metrics, traffic and protocol status checking.
check_storage_service check_computing_service	Network service level workflow, checking network status of involved servers, upstream and downstream network traffic, QoS management and etc.

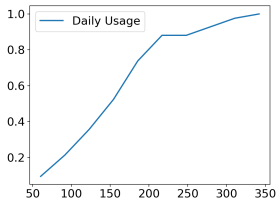
Table 4: Most Commonly Used Workflows

4.3 Usage Evaluation

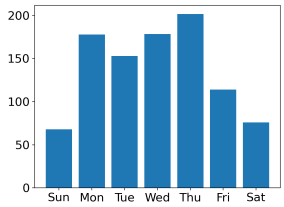
We count the daily usage and the number of distinct users to evaluate the impact of our tool. We collect the usage data for two whole months of July and August 2023. A complete dialogue session counts as **one** use of our tool. The daily usage distribution results are shown in Figure 8. We can observe that there are approximately 100 to 200 uses per day on most days. The distribution by day of the week is interesting. We find that the average usage on weekends is much less than on workdays. This is due to big changes (including software changes and network changes) are not allowed on weekends so that the entire data center network is running relatively stable. We also count and analyze what kinds of users are using our chatbot services. For July and August 2023, there are a total of 476 individual users. Most of them are from the engineering team, e.g., site reliability engineers, computing or storage infrastructure engineers, and network engineers. When there is a big network outage, there will be a wave of heavy usage (normally 20-50) of our chatbot since many people get affected, and they would like to know the situation and impact.

Next, we will measure how our tool reduces the oncall workload for network engineers. Basically, there are three different cases. The first case is that a user does not raise any oncall after using the chatbot. The second case is that a user brings the diagnosis results from the chatbot to an oncall for further operations[4]. The third case is that a user directly raises

[4]Users can raise oncalls either from the oncall platform or through the chatbot, which will directly share the diagnosis results to the oncall.

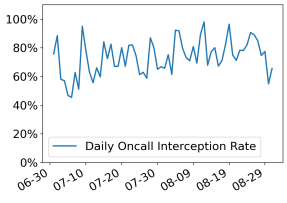


(a) Daily Oncall Usage for July and August in 2023 (CDF)

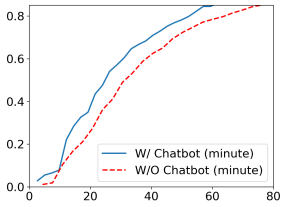


(b) The 50th Percentile Usage by Day of the Week

Figure 8: Daily Usage Results of NETASSISTANT



(a) Daily Oncall Interception Rate



(b) Oncall Duration Time Comparison (CDF)

Figure 9: How NETASSISTANT Benefits Daily Oncall

an oncall without using the chatbot, maybe because she does not know the tool.

We name the **first case** as an **effective use** of our tool. We consider the situation where an individual user makes several **effective uses** within a short time (e.g., an hour) as a successful **oncall interception**. Based on our statistics, the number of intercepted oncalls is about one-third of the total number of effective uses of our tool. The insight behind is that one user may continuously make several queries for just one issue. We further define the **oncall interception rate** as the number of intercepted oncalls divided by the sum of the number of raised oncalls and the number of intercepted oncalls. For the second and third cases, we will compare the oncall time with and without using the chatbot to measure whether the chatbot can benefit the oncall process.

Besides the NETASSISTANT usage data, we also collect our data center network oncall records for July and August 2023 and calculate the daily oncall interception rate and average processing time. The results are shown in Figure 9. We can observe that NETASSISTANT can reduce around 50% - 90% of daily oncalls. In most reduced cases, the user gets a response of health network status from the chatbot and does not continue to raise the oncall service. For the results of oncall time with and without the diagnosis from our tool, we can find that the information provided by NETASSISTANT can shorten the average oncall time. Figure 9(b) shows that NETASSISTANT can save around 20% - 25% of an oncall time in most cases. The saved time should have been spent by network engineers going through various monitoring data. With our tool, network engineers can focus more on the issue fix and communication with the network users.

4.4 Accuracy Evaluation

As a diagnosis tool, NETASSISTANT must achieve high accuracy to gain user trust. In this subsection, we will show our evaluation results of the false positives and false negatives of the diagnosis results. False positive is a non-network issue, but the chatbot diagnoses it as a network issue. To calculate the false positive ratio, we perform manual verification of results that are diagnosed as network issues over a period of

	Mar	Apr	May	Jun	Jul	Aug
FP	9.48%	12.33%	11.6%	10.63%	9.62%	8.45%
FN	0%	0%	0%	0.43%	1.25%	0%

Table 5: Accuracy Evaluation Results of NETASSISTANT

time. We find that the false positives are normally due to noise from the monitoring data. False negative is a network issue that is not detected by the chatbot. For every user-perceived network incident, we have the postmortem session that will check whether it has been (or can be) detected by our chatbot. Based on our experience, false negatives are much more harmful since the results may mislead network users and engineers and delay the progress in fixing issues.

We collect the false positives and false negatives data for half a year (March 2023 - August 2023), as shown in Table 5. We can observe that the false positive ratio is around 10%. We also find that the majority of false positives come from sampling based monitoring data (due to the nonrepresentative data as a result of chance) and switch monitoring data (especially when the switch is under high CPU utilization or software upgrade). Based on our experience, false positives are hard to avoid. Thus, we take a hierarchical approach to the network issues we detect. The chatbot will make an "unhealthy" result only for severe and wide-ranging network issues. For other detected small issues, the chatbot will make a "warning" result to inform the user that there could be a network issue. Compared with false positives, false negatives basically rarely happen. They are mostly due to problems occurring in places not covered by monitoring infrastructure. So, after every false negative happens, we will quickly improve the monitoring items and related diagnostic workflows. For example, in the second case in Section 4.2, we found a false negative generated by the switch reachability monitoring data since the switch loopback port had not been covered, and we fixed this flaw after the postmortem.

4.5 Performance Evaluation

As a dialogue system, the response latency is an important part of the user experience. To address the performance bot-

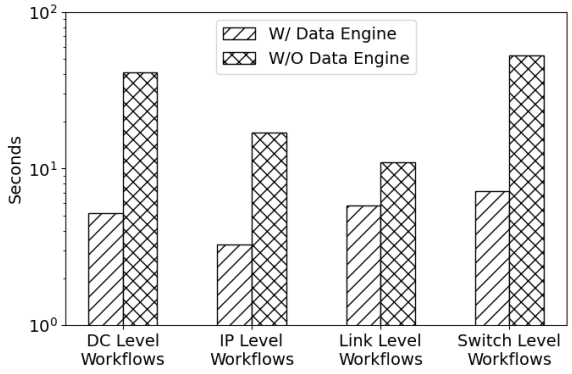


Figure 10: The 99th Percentile Running Time of Workflows

tleneck introduced by monitoring data retrieval, we design and implement the Data Engine module as shown in Section 3.4. To verify the performance improvement, we measure the response latency of the commonly used workflows listed in Table 4 with and without using the proactively generated alerts in Data Engine. We count the 99th percentile response latency for different categories of workflows. As shown in Figure 10, compared with retrieving monitoring data on the fly, leveraging proactively generated alerts can save significant time. The Data Engine reduces the overhead of big data query and transmission. After the optimization, most commonly used workflows take 5-10s for execution. Based on user experience feedback, the 5-10s response delay is acceptable but almost at the limit for a dialogue system. Considering we are still developing and adding new monitoring items to the diagnostic logic, performance optimization will be a continuous work in the future.

5 Experience Learned

NETASSISTANT is designed to help both network users who use the data center networks and network engineers who are engaged in front-line network O&M work. The user experience is the key to the success of our tool. Therefore, in addition to the iteration of the tool itself, we also provide thorough customer services including seminars, office hours, and surveys. We would like to share several lessons we have learned during the development and operation of our tool.

5.1 Can we proactively find alerts, engage the network team, and inform users?

As described in Section 3.4, Data Engine will set up operators to proactively generate alerts for high volume monitoring data. The straightforward question is, can we directly leverage the alerts to engage the network team and inform impacted network users in advance before they ask? The answer is partially yes. We only cover a small portion of high-priority alerts with severe influence. There are a few reasons. Firstly, as explained in Section 2.2, network incidents will flood the network team and users, which is impractical to manage. Secondly, the coverage and granularity of some monitoring techniques are limited by old models, device performance, and vendor support, which affects their quality. Finally, the resource budget is insufficient to proactively run all workflows, so we reserve them only for the high-priority ones.

5.2 Quality user experience

We prioritize ensuring a positive user experience and high engagement with our tool during network diagnosis and troubleshooting. To achieve this, we strongly emphasize the accuracy of our tool, which is achieved through the release of new workflows only after extensive testing. In addition, in our experience, we recognized that users often require further details beyond the initial technical diagnostic report, including information about the network issue, its service impact, and where to turn for help. To address this need, the chatbot provides users with additional details and resources. For instance, the chatbot will provide a brief explanation of the monitoring metrics and why it is (not) a network issue. Moreover, the chatbot can recommend relevant technical documents and similar problem operations and even enter an oncall or ticket request based on the question and diagnosis results.

5.3 Empowering our users

Our natural language understanding component is trained or tuned by user queries. On the contrary, our chatbot can also influence how users ask questions. We observe that after using our chatbot, user questions become more and more accurate and concise, and the communication between network users and network engineers has been dramatically simplified. For example, users get to know more network knowledge and terminologies and better descriptions of their questions. Further, the feedback and suggestions from users can help us polish our tool better. We are happy to see such an effect and think it is a virtuous circle.

5.4 Limitations and Future Work

We understand that our tool, while a crucial advancement in automated network diagnosis and troubleshooting, presents several areas for further research.

First, the state-of-the-art Large Language Model (LLM) has shown significant improvement in natural language understanding and generation, and has great potential to help automate network diagnosis and troubleshooting as a new technology. Our experience shows that LLM is good at learning, extracting and paraphrasing static information from diverse sources. For example, it can accurately and helpfully answer

questions like "How does our QoS system work" and "What is the standard operating procedure for a network change". However, we find it challenging to use LLMs to answer network diagnosis questions due to their limitations in understanding and learning diagnostic logic, as well as processing real-time monitoring data. Therefore, in this project, we leverage LLM for intent understanding but still rely on pre-defined diagnosis workflows.

Secondly, the manual effort involved in building workflows hinders the complete automation of network incident detection and mitigation. Achieving full automation remains a non-trivial task. We plan to fully automate the ability to convert workflows from natural language and past incident cases/tickets. Furthermore, we expect the workflows to be self-adjusting based on feedback from users and network engineers.

Finally, network monitoring primitives form the foundation of our tool, and the quality of the diagnoses is highly dependent on the input monitoring data. The diversity of data center network monitoring data and the different implementations from numerous hardware vendors make data screening and pre-processing very arduous. In the future, we plan to enhance our tool's capabilities with selective data source selection and noise filtering to improve the quality of the results.

6 Related Work

Natural Language Processing: Our work has been heavily inspired by the dialogue systems in the NLP domain [14, 21]. A dialogue system can be categorized into task-oriented or non-task-oriented (also known as open-domain). There have been several task-oriented dialogue system products in industry [1, 3, 4, 10]. Our work attempts to leverage the task-oriented dialogue technique into the data center networking domain and addresses several unique challenges. Besides, a significant breakthrough has recently been made in the field of LLM. Products like ChatGPT [5], LLaMa [6], and PaLM [7] demonstrate strong ability in language understanding and generation. Our work leverages the understanding ability of LLM, and we are still exploring its generation ability.

Querying: Many works have improved the efficiency and effectiveness of querying and summarizing network traffic. Sonata [19] partitions each query across the stream processor and the data plane and dynamically refines each query to ensure it focuses only on traffic satisfying it. BeauCoup [15] is another programmable switch-based system. Using a coupon collector approach, it supports multiple distinct counting queries simultaneously while making only a small constant number of memory accesses per packet. Net2Text [13] uses NLP to format natural language operator queries to database SQL-like queries, executes the query, summarizes the results, and translates the summary back to natural language. We are inspired a lot from Net2Text and further leverage the dialogue system to build a general-purpose querying framework.

Diagnosis: There have been numerous efforts to develop tools that can automate fault localization. One such tool is OmegaGen [20], which combines static and dynamic analysis to track the control and data flow through a program to localize partial software failures at runtime. Another focus of automated fault localization tools is identifying the cause of packet drops. For instance, 007 [12] ranks links based on their relative drop rates using the path of TCP connections suffering from one or more retransmissions. On the other hand, Drift-Bottle [28] takes a more distributed approach, with each switch using the status of flows to infer suspicious links and add lightweight inference headers to packets sent to the operator. VTrace [16] focuses on detecting persistent packet loss over the cloud-scale overlay network. It utilizes the "fast path-slow path" structure of virtual forwarding devices to track and inspect the packets of interest in depth selectively. Zeno [23] generates a causal graph to capture the temporal dependencies between events in a system, such as requests and their corresponding responses, to diagnose performance problems. CloudCanary [25], on the other hand, uses fault graphs to perform real-time audits on service updates to identify the root causes of correlated failure risks and generate improvement plans with increased reliability. Tools like Scouts [17] use machine learning to analyze complex relationships and route incidents to the most likely responsible team. We benefit from these works by allowing network engineers to autonomously select and customize diagnostic algorithms to build workflows.

7 Conclusion

We propose NETASSISTANT, a virtual assistant tool to answer network diagnosis questions and help both data center network users and network engineers. The tool is motivated by our measurement study about network user daily queries and the network monitoring primitives used by network engineers in our production network. NETASSISTANT abstracts the whole diagnosis process into the dialogue layer, workflow layer, and data layer. Accordingly, we design three functional modules to realize these layers and provide dialogue-based network diagnosis to our users. NETASSISTANT has been deployed and used in our company for over three years and the evaluation results show that this tool can significantly reduce the workload of both network users and network engineers and provide excellent user experience and performance.

Acknowledgments

We would like to thank many ByteDance colleagues who have contributed to this work. These include Jiale Feng, He Zhang, Anjian Chen, Dong Zhao, Chengcai Yao, Jing He, Chenxingyu Zhao, Shravan Jagadish Kumar, Chuanxiong Guo, and many others. We also thank our anonymous reviewers and shepherd Arpit Gupta for their valuable comments.

References

[1] Amazon alexa/echo. https://alexa.amazon.com/.

[2] Apache kafka. https://kafka.apache.org/.

[3] Apple siri. https://www.apple.com/siri/.

[4] Google Now. https://assistant.google.com/.

[5] Introducing chatgpt. https://openai.com/blog/chatgpt.

[6] Introducing llama: A foundational, 65-billion-parameter large language model. https://ai.meta.com/blog/large-language-model-llama-meta-ai/.

[7] Introducing palm 2. https://blog.google/technology/ai/google-palm-2-ai-large-language-model/.

[8] Introduction to syslog in cisco routers and switches. https://www.omnisecu.com/ccna-security/introduction-to-syslog-in-cisco-routers-and-switches.php.

[9] Logging - basic syslog and beyond. https://arista.my.site.com/AristaCommunity/s/article/logging-basic-syslog-and-beyond.

[10] Microsoft cortana. https://www.microsoft.com/en-us/cortana.

[11] Network switches, past, present – and an exciting future. https://gblogs.cisco.com/uki/network-switches-past-present-and-an-exciting-future/.

[12] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. Liu, J. Padhye, B. T. Loo, and G. Outhred. 007: Democratically finding the cause of packet drops. In *USENIX NSDI*, pages 419–435, 2018.

[13] R. Birkner, D. Drachlser-Cohen, L. Vanbever, and M. Vechev. Net2text: Query-guided summarization of network forwarding behaviors. In *USENIX NSDI*, pages 609–623, 2018.

[14] H. Chen, X. Liu, D. Yin, and J. Tang. A survey on dialogue systems: Recent advances and new frontiers. *ACM SIGKDD Explorations Newsletter*, 2017.

[15] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *ACM SIGCOMM*, pages 226–239, 2020.

[16] C. Fang, H. Liu, M. Miao, J. Ye, L. Wang, W. Zhang, D. Kang, B. Lyv, P. Cheng, and J. Chen. Vtrace: Automatic diagnostic system for persistent packet loss in cloud-scale overlay network. In *ACM SIGCOMM*, pages 31–43, 2020.

[17] J. Gao, N. Yaseen, R. MacDavid, F. Vieira Frujeri, V. Liu, R. Bianchini, R. Aditya, X. Wang, H. Lee, D. Maltz, M. Yu, and B. Arzani. Scouts: Improving the diagnosis process through domain-customized incident routing. In *ACM SIGCOMM*, pages 253–269, 2020.

[18] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *SIGCOMM*, pages 139–152, 2015.

[19] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*, pages 357–371, 2018.

[20] C. Lou, P. Huang, and S. Smith. Understanding, detecting and localizing partial failures in large system software. In *USENIX NSDI*, pages 559–574, 2020.

[21] J. Ni, T. Young, V. Pandelea, F. Xue, and E. Cambria. Recent advances in deep learning based dialogue systems: a systematic survey. In *Artificial Intelligence Review*, pages 3055–3155, 2022.

[22] B. Qin, B. Hui, L. Wang, M. Yang, J. Li, B. Li, R. Geng, R. Cao, J. Sun, L. Si, F. Huang, and Y. Li. A survey on text-to-sql parsing: Concepts, methods, and future directions. In *arxiv.org/abs/2208.13629*, 2022.

[23] Y. Wu, A. Chen, and L. T. X. Phan. Zeno: Diagnosing performance problems with temporal provenance. In *USENIX NSDI*, pages 395–420, 2019.

[24] M. Yu. Network telemetry: Towards a top-down approach. *ACM SIGCOMM CCR*, pages 11–17, 2019.

[25] E. Zhai, A. Chen, R. Piskac, M. Balakrishnan, B. Tian, B. Song, and H. Zhang. Check before you change: Preventing correlated failures in service updates. In *USENIX NSDI*, pages 575–589, 2020.

[26] X. Zhang, J. Zhao, and Y. LeCun. Character-level convolutional networks for text classification. In *Advances in Neural Information Processing Systems*, 2015.

[27] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-level telemetry in large datacenter networks. In *SIGCOMM*, pages 479–491, 2015.

[28] X. Zuo, Q. Li, J. Xiao, D. Zhao, and J. Yong. Driftbottle: A lightweight and distributed approach to failure localization in general networks. In *CoNEXT*, 2022.

Klonet: an Easy-to-Use and Scalable Platform for Computer Networks Education

Tie Ma[†], Long Luo[†], Hongfang Yu[†], Xi Chen[‡], Jingzhao Xie[†], Chongxi Ma[†],
Yunhan Xie[†], Gang Sun[†], Tianxi Wei[†], Li Chen[◇], Yanwei Xu[¶], Nicholas Zhang[¶]

[†]*University of Electronic Science and Technology of China*
[‡]*Southwest Minzu University* [◇]*Zhongguancun Laboratory*
[¶]*Theory Lab, Central Research Institute, 2012 Labs, Huawei Technologies Co., Ltd.*

Abstract

Currently, one of the simplest and most effective ways for people to gain an in-depth understanding of computer networks is through hands-on practice and experimentation on software platforms. While education is important for the field of computer networks, existing platforms are inadequate in usability and scalability, failing to fully meet all the teaching needs of computer networking education.

This paper describes our experiences in designing and using Klonet, an emulation platform for computer networking education. Klonet is *easy-to-use* for both students and tutors, which has been carefully designed to lower the barrier to use, thus making the practice more efficient. Klonet also demonstrates good *scalability*. It adopts a container-based distributed architecture and a virtual network embedding algorithm customized for this platform. Evaluation experiments show that Klonet exhibits better scalability, such as supporting more students with fewer hardware resources (*i.e.*, servers) and deploying virtual network topologies more quickly. Furthermore, to ensure stability during teaching, Klonet enhances the *robustness* of its upper orchestrator and underlying virtual networks. So far, Klonet has been adopted in 3 universities and 4 courses, serving more than 800 students. We showcase Klonet's usefulness in networking education with real use cases, including a scenario with ∼10,000 emulated routers. We also share our lessons learned from the 4 years of Klonet development and 2 years of operations.

1 Introduction

Practice and experimentation are central to the education of computer networks. Through practice, students are able to obtain hands-on experience with the complicated concepts in their computer network textbooks, thus gaining a deeper understanding of computer networks.

The educational practice of computer networks relies heavily on the field of network experiments which has been the focus of many existing works. Prior work has made great progress in three aspects: testbed [1–5], simulator [6–11], and emulator [12–37]. Testbed offers real hardware for students to manipulate, providing realistic interaction but at a higher cost. Simulator, on the other hand, uses numerical calculations to mimic hardware behavior, offering a cheaper alternative but lacking realistic interaction. Emulator is a compromise between the previous two. It uses software to simulate hardware behavior and is the most promising in education. However, as college educators ourselves, based on years of teaching experience, we found that existing emulators cannot meet our actual needs. We argue that the ideal emulator for computer network education should meet the following goals.

R1. Easy-to-use. A user-friendly educational platform should be easy to use, including getting started, mastering basic operations, *etc.*, especially for students. We know that the first step is always the hardest. If a lot of additional learning or manual configuration is required to perform relevant experiments, students may feel confused and intimidated, especially those with weak background knowledge. For tutors, an ideal platform should also be easy to use in aspects including designing, directing, correcting, and managing experiments, which can increase teaching efficiency instead of being a burden.

R2. Scalable. It also must be scalable, supporting: (1) A large number of concurrent experiments. Since students in a course usually do experiments during the same period, the platform must be able to provide services to all the students in at least one course. The more users use it, the higher the scalability requirements. Scalability requirements increase along with growing student enrollment on the platform. (2) Large-scale emulated network. To cover experiments in as many scenarios as possible, the scale of the emulated/virtual network of each experiment could be small or large. For example, a MAC address learning experiment may require less than 4 nodes, whereas routing learning in a modern datacenter network may require thousands of nodes.

To the best of our knowledge, existing network emulators are insufficient to meet the above two goals. Regarding R1, most emulators [12, 14–35, 37] require students to perform installation and complex software configurations. Considering that most undergraduate students even have not been exposed to the Linux system, this can be challenging for them and dissuade them from the computer network community. Moreover, most emulators [12, 13, 15, 16, 20–34] are initially provided for scientific research and lack consideration for graphical user interface (GUI), auxiliary experimental tools, auxiliary teaching tools, *etc.*, making students and tutors spend a lot of effort

Co-primary authors: Tie Ma and Long Luo. Hongfang Yu is the corresponding author. (Email: yuhf@uestc.edu.cn)

to use them. For R2, some emulators [12,14,16,17,20,37] can only run on a single server, which limits the scale of the virtual network it can emulate and limits the experimental scenarios it can support. Besides, some emulators [15,35] that can be deployed on multiple servers cannot properly map the virtual networks to underlying servers, causing server overloads.

To address this gap, we built Klonet, a network emulation platform to meet these educational goals. Klonet employs several technologies to make it *easy-to-use*. To minimize the difficulty of getting started, Klonet adopts a Browser/Server (B/S) architecture, allowing students and tutors to access and use it without installing any software. Klonet also provides several useful experimental tools such as traffic generator and traffic monitor for students to facilitate their experiments. In order to stimulate students' interest in learning, Klonet carefully provides a GUI that is very important for teaching. Klonet accelerates virtual network creation time, improving user experience and efficiency. For tutors, Klonet implements a container image repository that can easily add various types of nodes, making it possible to build diverse educational scenarios flexibly. Klonet includes a scene repository where students can share experiments and learn from each other, and tutors can quickly automate the creation of experiment environments. Klonet also provides experiment APIs to advanced users such as tutors to improve usability and efficiency.

To achieve *scalability*, Klonet uses light-weight containers as virtual nodes and a novel virtual network embedding algorithm to respectively reduce node overhead and link communication overhead to support more and larger virtual networks. By using container technology, Klonet increases the number of nodes that can be created on a single machine. To support large-scale virtual networks, Klonet employs a distributed orchestration architecture to create overlay virtual networks on a cluster, which enables the size of the virtual networks to scale with the cluster size. The goal of Klonet's virtual network embedding algorithm is to use as few servers as possible without overloading the servers. Klonet also implements a user management model to support multi-users.

As an open educational network emulation platform for students and tutors, *robustness* is critical. Klonet adopts methods such as periodic checks, redundant backups, and monitoring alarms to ensure the stability of its distributed orchestrator, user management model, virtual networks, and cluster.

In our evaluation, we show that Klonet can better utilize fewer physical servers to support more students' experiments faithfully. Klonet has been in development for 4 years, and we have been using it steadily in courses for 2 years. To this day, Klonet has served more than 800 students. To demonstrate Klonet's usage and benefits, we introduce the two selected use cases. The first is a course project about the Software-Defined Network (SDN) [38] and P4 [39], we use this case to showcase Klonet's usability. In the second use case, students collaborate on a routing experiment that contains up to about 10,000 nodes, demonstrating the scalability of Klonet. The lessons we have learned from developing, operating, and using Klonet in education are also presented, as a reference for building a future education platform.

In summary, our main contributions are:

- We design and implement (§4) Klonet, an easy-to-use and scalable network emulation platform for education.
- We evaluate the system performance (§5) and present use cases (§6) to demonstrate its educational applications.
- We summarize lessons (§7) learned from years of experience developing, operating, and using Klonet.

2 Related Work

Here, we discuss existing network emulation platforms used in education, categorizing them as general-purpose and education-purpose. General-purpose emulators may offer less tailored features for education, whereas education-purpose emulators prioritize educational needs. We focus on platform-level related works rather than individual components such as the Kubernetes [40] scheduler.

General-purpose network emulators. Most general emulators are not specifically designed for education but are nonetheless widely adopted in education [41].

Mininet [12] is well-known for its convenience in running small networks on laptops. However, when used in education, several challenges arise: (1) Students often struggle with the installation process. This places performance requirements on student laptops. Since the operating system of most student laptops is Windows, students need to go through the installation and use of virtual machines (VMs). This poses a significant challenge for undergraduate students who lack experience, potentially diminishing their enthusiasm for computer networks. (2) We also consider installing and configuring Mininet on servers provided by the teaching team and let students remote access. However, the original Mininet has a file isolation and scalability issue, making it unable to support multi-student. Though several follow-up variants [14–16,20–22,42] have been proposed to enhance Mininet with different capabilities such as scalability [15,42], ease of use [16], fidelity [20] and additional network scenario [22], none of the variants combining all the advantages, making it not feasible to offer an ideal network emulation service on cluster. (3) Mininet and its variants are primarily developed for research purposes and lack a student-friendly GUI, which is crucial for education.

Emulab [13] is another famous emulator that operates a dedicated cluster and provides users with network emulation service via its website, allowing users to use it without local installation. Emulab has been operated for decades and is robust. Emulab has three issues when used for education: (1) Emulab primarily focus on providing users with bare-mental servers and VMs, while has poor support for light-weight virtualization technologies like container. This makes it costly and inflexible to support courses. (2) Emulab's scalability is

Table 1: Comparison of related network emulators.

Platform Name	Easy-to-use					Scalability
	No Installation Required	GUI and Experiment API	Teaching Tools[1]	Experiment Tools[2]	Rich Node Types	
Mininet [12]	✗	Humble GUI	✗	Limited	✗	✗
Mininet-Hifi [20]	✗	Humble GUI	✗	Limited	✗	✗
Distrinet [15]	✗	No GUI	✗	✗	Limited	✓
Containernet [16]	✗	No GUI	✓	✗	Limited	✗
Vt-Mininet [21]	✗	No GUI	✗	✗	✗	✓
Mininet-Wifi [22]	✗	Humble GUI	✓	✓	✓	✗
Emulab [13]	✓	✓	Limited	✗	Limited	Limited
Netkit [37]	✗	No GUI	✓	✗	✓	✗
Kathará [17]	✗	Humble GUI	✓	✓	✓	✗
Megalos [18]	✗	Humble GUI	✓	✓	✓	✓
GNS3 [19]	✗	✓	✓	✓	✓	✓
SEED [35]	✗	✓	✓	✗	✓	✓
Mini-Internet [36]	✓	No GUI	✓	Limited	✗	✗
IPMininet [14]	✗	No GUI	✓	✓	✗	✗
Klonet (this work)	✓	✓	✓	✓	✓	✓

[1] **Teaching tools** are those designed to facilitate education, *e.g.*, Klonet's scene repository and Mini-Internet's connectivity matrix.
[2] **Experiment tools** are those designed to make experiments easier, *e.g.*, Klonet's traffic generator and IPMininet's IP configuration tools.

limited by its cluster size, and it's hard to deploy on private clusters [37]. (3) Due to the operating mode of Emulab, the physical server needs to start before creating a virtual network, resulting in a long establishment time (as shown in §5.1), which reduces the efficiency of experiments.

There are various emulators [23–27] that focus on evaluating network end-to-end properties by simplifying or abstracting network elements. However, these emulators cannot emulate all types of devices (*e.g.*, switches, routers, firewalls), limiting their ability to provide realistic interactions and serve as ideal emulators for education. Some emulators [21, 28–32] prioritize scalability at the expense of real-world performance, making it unsuitable for education. Some emulators [33, 34] are designed for specific scenarios such as enterprise networks and integrated space and terrestrial networks (ISTNs), lacking support for other network scenarios and are not easy to use for students.

Education-purpose Network emulators. Klonet is not the only network emulator proposed specifically for education. Netkit [37] exploits User-Mode Linux (UML) VM to support networking courses, the container technology chosen by Klonet is light-weight compared to the virtualization technology of Netkit. Kathará [17] is a variation of NetKit which uses Docker container instead of UML VM, but its single host network emulation architecture may cause scalability issues. Megalos [18] improves the scalability of Kathará by leveraging Kubernetes [40]. Kathará and Megalos have been used in many courses [43]. GNS3 [19] is also widely used in education and requires local software installation. Owing to reliance on VMs, GNS3 is a more resource-intensive solution compared to Klonet which employs light-weight container technology. Mini-Internet [36] is an open network emulation platform that emulates a mini version of the internet infrastructure, but it focuses on routing experiments and lacks support for other experiments. Besides, Mini-Internet does not support multi-server deployment, which may make it fail to support a large number of students. SEED emulator [35] was initially developed to support teaching network security education and can scale up the emulation across multi-server deployment. Unfortunately, it needs to embed the network manually, which may overload servers and prevent the courses from proceeding properly. IPMininet [14] enhances the IP configuration and management of the well-known emulator Mininet [12], however, it needs to be installed on the personal computers of students.

In summary, as shown in Table 1, no existing work can achieve the two educational goals of easy-to-use and scalable simultaneously except Klonet.

3 Klonet in a Nutshell

This section gives a macro view of Klonet covering its operating mode, applicable scenarios, and experiment workflow.

Operating mode. Klonet is an open platform that maintains a dedicated private or cloud cluster to offer network emulation services for courses. Its cluster is resilient, allowing operators to adjust its scale as needed. Leveraging light-weight virtualization technologies, Klonet can create (overlay) virtual networks on top of the cluster, enabling the creation of several large-scale or lots of small-scale virtual networks. Students and tutors can conveniently access Klonet via its website. Klonet stays constantly available within the campus network, serving courses during the semester and also serving self-learners. Thanks to Klonet's extensive experiment capabilities, students and tutors can efficiently conduct the majority of their experiments on a single platform.

Applicable scenarios. Klonet supports network experiments with layer 2 and above, allowing network properties

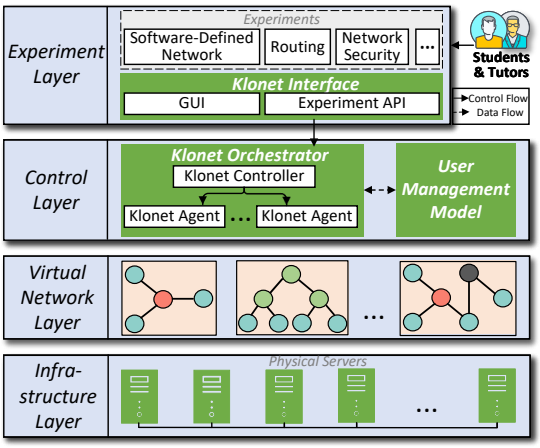


Figure 1: The overall framework of Klonet.

like latency and bandwidth on links to be configured. This versatility makes Klonet suitable for a wide range of network scenarios required in education. It can emulate data center networks, wide-area networks, large-scale enterprise networks, campus networks, *etc.* The topology of these networks can be customized, and end-to-end network experiments can also be conducted. Klonet's virtual network includes virtual Ethernet interfaces that can seamlessly connect with hardware Ethernet interfaces, supporting networks with a mixture of real and virtual devices. Klonet supports rich node types, such as (programmable) switches, routers, controllers, and hosts, enabling experiments in both data plane and control plane. Klonet supports both IPv4 and IPv6.

Experiment workflow. Conducting an experiment on Klonet involves the following steps: (1) Experiment creation. Students create experiments by dragging and dropping node icons onto the canvas, connecting nodes, and configuring basic network properties such as IP addresses. Optionally, students can choose to load pre-configured experiment templates provided by tutors. (2) Development, execution, and observation. Students develop their programs using the terminal on the Klonet web page or via Secure Shell (SSH) connections. They then run their programs and observe the resulting effects. Experiment tools such as traffic generators and monitors are available to assist students during the process. (3) Experiment completion. Once the experiment is finished, students upload their experiments to the Klonet scene repository. Tutors can use these uploaded experiments for grading purposes or allow other students to review and learn from them.

4 Design and Implementation

We first overview the framework and then describe the design adopted by Klonet to meet the educational goals.

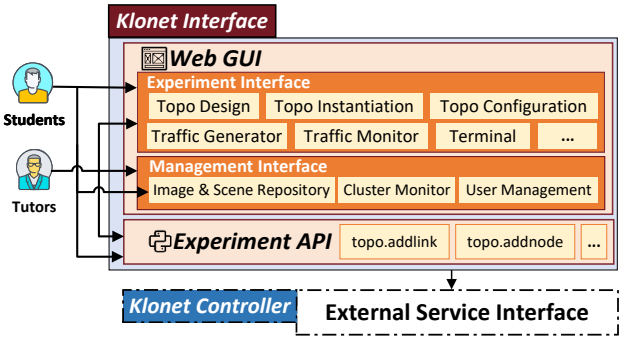


Figure 2: Klonet interface.

4.1 System Overview

As Figure 1 shows, Klonet adopts a four-layer architecture. (1) *Experiment layer:* The direct-to-user front end provides students and tutors with many easy-to-use tools via Klonet's GUI and experiment API, which include virtual network design and emulation, image repository, scene repository, traffic generator, traffic monitor, cluster monitor, *etc.* It receives user requests and communicates with the underlying layers to complete related services. (2) *Control layer:* The back end controls and manages user data, virtual networks, and clusters. This layer consists of an orchestrator distributed on each physical server and a user management model for storing user data, which together support multi-user network experiments on multi-server. (3) *Virtual network layer:* It uses light-weight virtualization technology and cross-server link technology to ensure that the virtual network can be distributed across servers. (4) *Infrastructure layer:* A cluster can provide the storage, computing, and network resources required by Klonet. It can easily remove or add new physical servers for purposes such as cost savings and launching more experiments.

4.2 Achieving Easy-to-use

Unlike research-oriented platforms, easy-to-use is particularly important for education-oriented platforms. We describe how Klonet improves ease of use with designs in terms of software architecture, user interface, reproducibility, experiment tools, and accelerating creation of virtual networks.

Browser/Server (B/S) architecture. To minimize the burden on students and reduce the difficulty in getting started, Klonet adopts a Browser/Server (B/S) architecture that allows students to use it via their browsers. To this end, Klonet is designed with two parts: front-end and back-end. The front-end is implemented in JavaScript, HTML, and CSS containing 16k lines of code. Flask [44], a web development framework is used to implement the back-end containing 45k lines of code. The front-end and back-end interact via HTTP.

Rich Interfaces. As shown in Figure 2, Klonet provides two kinds of interfaces. (1) The GUI is intuitive and is suitable for beginners such as bachelor students to perform simple operations. (2) The experiment API focuses on efficiency and

is more suitable for users with some basic network knowledge, such as master students and tutors, to conveniently and rapidly complete some repetitive and cumbersome operations.

GUI. One web page is designed for experimentation and several for management. The experiment page provides five primary features: (1) Topology design by drag-and-drop of device icons and mouse-based link connection. (2) Virtual network instantiation. (3) Configuration of network properties such as IP addresses and link bandwidths. (4) Interactive access to virtual nodes through SSH or web terminal, offering students an experience similar to real devices. Klonet helps install and configure the SSH service in the nodes. With powerful SSH-based tools (*e.g.*, VScode remote SSH extension [45]), students can efficiently complete more complex tasks. Students can also use the web terminal to conveniently perform simple tasks like running a `ping` command. Based on the Docker daemon [46], Klonet ensures real-time communication of the web terminal through websockets [47]. (5) Traffic generator and monitor tools to assist in experimental implementation. A screenshot of the experiment page is provided in Appendix A.

The management pages add much more support for experimentation. An experiment management page is designed for viewing and accessing experiments. An image/scene repository page is offered for sharing images/scenes, with tutors authorized to manage it. A user management page is designed for tutors to audit and delete student accounts conveniently. To help operators (usually tutors or TAs) quickly locate faulty servers, Klonet provides a cluster monitoring page based on Grafana [48] to view the cluster status (*e.g.*, CPU utilization).

Experiment API. This is designed to enhance the programmability. For simplicity of use, the API is implemented by encapsulating HTTP requests via Python. Listing 1 shows a simple example of using this API. Detailed API documentation is also provided to help users get started quickly.

Image repository and scene repository. Image repository and scene repository are the keys to realizing the reproducibility, including node-level and scene-level, of network experiments.

*Node-level reproducibility relies on an image repository containing node images (*i.e.*, snapshots).* Users can upload images to this repository and later instantiate them as nodes. Since the software dependencies have been packaged into the image, the applications in a node can be reproduced easily. For better usability, Klonet offers three upload methods: *commit* for saving nodes as images, *build* images via scripts, and *compressed file* for external image inclusion. Images are publicly/privately shared and indexed separately via a MySQL database [49], with entities stored in a Docker registry [50] for access across servers.

Scene-level reproducibility relies on a scene repository. We argue that a reproducible scene is a collection of code, software environment, network topology, and code execution flow. This scene repository converts all nodes of an experiment to a set of images via *commit* provided by the image repository, assigning each image a unique ID to help reestablish the corresponding node. The topology description is also saved for rebuilding virtual networks. To facilitate the scene-level reproducibility, Klonet asks users to upload a *replay script* (*i.e.*, written via experiment API) describing code execution flow. This script is optional as manual recreation is possible. MySQL is used to store image indexes, topology descriptions, and replay scripts to provide users with a reproducible scene.

```python
from klonet_api import *
# Get the available images of current student.
images = get_images()
# Select the host(ubuntu) and switch(ovs) image.
ubuntu_image = images["ubuntu"]
ovs_image = images["ovs"]
# Design our topology: h1---s1---h2.
topo = Topo()
h1 = topo.add_node(ubuntu_image, node_name="h1")
h2 = topo.add_node(ubuntu_image, node_name="h2")
s1 = topo.add_node(ovs_image, node_name="s1")
topo.add_link(h1, s1, src_IP="192.168.1.1/24")
topo.add_link(s1, h2, dst_IP="192.168.1.2/24")
# Let Klonet emulate the topology.
deploy(topo)
# Create file in h1 and h2.
exec_cmds_in_nodes(
    {"h1":["touch /log1"], "h2":["touch /log2"]})
```

Listing 1: Simple Experiment API Example.

Auxiliary useful tools. To help students focus on the network experiments themselves, Klonet provides useful tools such as traffic generator, traffic monitor, and typical topology generator. Tutors and TAs can also use these tools to complete tasks such as experimental demonstrations easily.

Traffic generator. Traffic enables the validation of diverse network scenarios. The traffic generator sources three traffic types: (1) Client-server pattern [51] traffic. (2) ON-OFF [52] characteristic traffic. (3) Customizable interval-distributed traffic, *e.g.*, Gaussian. We encapsulate an open source code from GitHub [53] to implement the first traffic type, and implement the latter two based on the Scapy library [54].

Traffic monitor. Feedback helps with experimental debugging, tuning, *etc.* Klonet develops this monitor upon libpcap [55] and reports key performance metrics, throughput, latency, and loss rate, to capture and help analyze traffic between source and destination nodes. It also provides an intuitive GUI for visualizing experimental outcomes.

Typical topology generator. Repeatedly designing some common topologies is tedious. Thus, Klonet features a generator for structures like Star, Fattree, Linear, and Tree, parameterized by variable fields. Custom topologies can also be saved as reusable templates. The typical topology generator is implemented based on the FNSS library [56].

Accelerating virtual network creation. Shortening the virtual network creation time can reduce the waiting time for users to retry experiments, improving experience and efficiency. Klonet applies two parallelization techniques: using

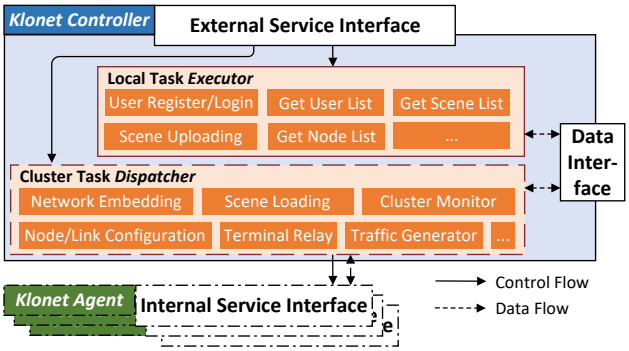


Figure 3: Klonet Controller.

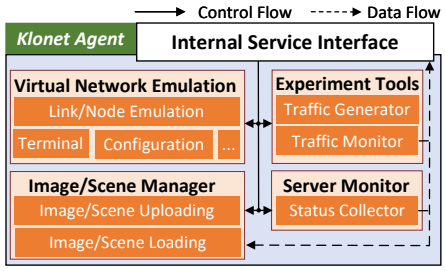


Figure 4: Klonet Agent.

a light-weight concurrency technique called Coroutines [57] to spawn virtual nodes and links concurrently within a server and permitting the orchestrator to distribute virtual network creation across physical machines for additional parallelism.

4.3 Achieving Scalability

Klonet first designs a distributed orchestrator which enables it to scale on a cluster. This poses a challenge in mapping virtual networks, which is addressed by an efficient algorithm design. For effective multi-user support, Klonet implements a user management model for data organization. To support large-scale experiments with fewer physical resources, Klonet leverages light-weight emulation technology to host as many virtual networks as possible on per physical machine.

Klonet orchestrator. Developed based on the Flask [44] framework, the orchestrator consists of a controller and several agents. The controller is installed on one server, while agents are installed on each server. Notably, the orchestrator supports single-server deployment. The controller and agents communicate with each other via HTTP. Most back-end functions of Klonet are implemented in this orchestrator.

Klonet controller. As the orchestrator's core (Figure 3), the controller receives user requests via external service interfaces and interacts with the user management model through data interfaces. It contains a local task executor for authentication and database queries, and a cluster task dispatcher for assigning *cluster tasks*, such as virtual network embedding and link configuration, to appropriate agents for execution.

Klonet agent. As shown in Figure 4, the agent receives instructions from the controller via the internal service interface. It handles on-server tasks including creating virtual networks, collecting server status, monitoring traffic, *etc.*

Virtual Network Embedding Algorithm. The Klonet controller uses a virtual network embedding (VNE) algorithm to map the virtual networks (VNs) to the substrate network (SN). For fidelity performance and efficient resource utilization, the algorithm carefully provides sufficient resources for each VN without overwhelming server capacity in the SN.

Network models and problem formulation. Each VN is modeled as an undirected graph $G = (V, E)$, with each virtual node $v \in V$ having a CPU demand $nw(v)$ and each virtual link $e \in E$ having a bandwidth demand $lw(e)$. To support cross-server deployment of large-scale VNs and efficiently use resources, Klonet partitions a VN into a set of small sub-virtual networks (sub-VNs) $subs = \{s_1, s_2, ..., s_k\}$, where $1 \le k \le |S_{server}|$. *Cut-links* represent inter-sub-VN links, and $cut(s) = \{c^s_1, c^s_2, ..., c^s_{|j|}\}, \forall c^s_i \in E, 0 \le |j| \le |E|$ denotes that connecting sub-VN s.

Let binary variable $x_{s,n}$ denote whether sub-VN s is embedded on server p_n. When embedding G to a cluster with a set S_{server} of servers, with scalability in mind, we expect to minimize the impact of the mapping of every current VN on the capabilities (potential) of the SN to deploy future large-scale VNs under resource constraints, which can be formulated as

$$\text{Minimize} \sum_{p \in S_{server}} I_p(x) \tag{1}$$

$$s.t. \quad \sum_{s \in subs} \sum_{u \in V^s} nw(u) x_{s,n} \le C(p_n), \forall p_n \in S_{server} \tag{2}$$

$$\sum_{s \in subs} \sum_{c \in cut(s)} lw(c) x_{s,n} \le B(p_n), \forall p_n \in S_{server} \tag{3}$$

$$x_{s,n} \in \{0,1\}, \forall s \in subs, p_n \in S_{server} \tag{4}$$

Eq. (1) expresses that the cluster potential is equal to the sum of all server potentials. $I(.)$ is an *Inefficiency Index (IDX)* whose value starts from 0 and grows as the use of CPU resources increases and then slowly decreases to 0 when the resources are exhausted. Besides, it increases as the use of NIC bandwidth grows. To prevent CPU overload, Ineq. (2) constrains the CPU demands of the sub-VNs deployed on server p_n to be less than its remaining CPU quota[1] $C(p_n)$. As the SN of Klonet is a local cluster where all the servers are connected by a powerful switch and each server communicates with the others through the same network interface (NIC), all substrate links connected to a server share its NIC bandwidth capacity for communication. To meet bandwidth constraints, the total bandwidth demands of the cut-links of all subnets deployed on server p_n should be less than its available NIC bandwidth $B(p)$, as expressed in (3). *This differs significantly from most existing VNE efforts, which assume exclusive bandwidth capacity per substrate link.*

[1] CPU quota is a feature of Linux Control Groups (cgroup), which controls how much CPU time a process in a container can use.

Efficient network embedding algorithm. This VNE problem is NP-Hard even with a known value of $I(\cdot)$ and the VN partition solution. A heuristic is thus employed. Through extensive experimentation, we observed that sparsely used servers often outperformed heavily used ones in hosting large VNs, given equal CPU quotas. Therefore, our algorithm is designed to minimize fragmentation by preferentially exhausting resources on previously used servers. See pseudocode in Appendix C for details.

User management model. To appropriately organize experimental information for multi-students, we build a user management model with an easily extensible architecture. We leverage the Redis database [58] to implement the management model. For advanced user and VN information, we use several tables to record them. For the low-level virtual network devices, we model them as several classes with device attributes and relationships, and instantiate devices as objects in the user management model when creating virtual networks. The detailed design can be found in Appendix B.

Virtual network emulation. Another key to scalability is the VN emulation solution. Klonet adopts Docker [59] container technology for node emulation. The reasons are (1) Container light-weight nature versus VM and better file isolation feature than the namespace technology used by emulators like Mininet [12], in line with our goal to design Klonet as a scalable and shared platform; (2) Docker container provides feature-rich node images to facilitate flexible experiment design. Default images in Klonet include: OVS [60] switch, P4 software switch [61], FRR [62] router, quagga [63] router, Ubuntu host [64], Ryu [65] controller, *etc.*

Emulated links are classified as intra-server or inter-server based on their location. Intra-server links adopt virtual Ethernet pair technology, creating virtual NICs in containers and connecting them. The virtual NICs can also be connected to real NICs for mixed virtual/realistic emulation. Inter-server links use VXLAN [66] protocol for Layer 2 connectivity across servers. Users are presented with a transparent view, unaware of distinctions between intra- and inter-server links.

To support more scenarios, Klonet uses `Linux traffic control` [67] to apply queuing disciplines on links, emulating properties, *e.g.*, bandwidth, latency, and packet loss.

4.4 Achieving Robustness

For a shared platform like Klonet, robustness plays a significant role in guaranteeing a smooth education process. We use a variety of technologies to ensure the robustness of each component of Klonet from top to bottom: Klonet orchestrator, user management model, virtual networks, and the cluster.

Robustness of the Klonet orchestrator. Klonet implements two mechanisms to ensure the availability of the orchestrator: regular requests and redundant backups.

Regular requests. Given that the Klonet controller and the Klonet agents are implemented by the web server technology, Klonet confirms that they are alive by periodically sending requests to them and getting corresponding responses. To prevent misjudgments, Klonet sends an odd number of requests at a time and uses the result of the majority of them. Once a controller or agent failure is detected, Klonet restarts it.

Table 2: Major Anomalies of Klonet.

Anomalies	Times
Cluster power outage for regular maintenance.	4
Inter-server link fails due to the firewall.	1
CPU overloaded due to broadcast storm.	1
Node boot failed due to the wrong image.	2
Node boot failed due to no startup command.	2
Lost user management model connection.	1
Image upload failed due to configuration error.	2
Inconsistent MTU among cluster servers leads to communication failures.	1

Redundant backups. During the process of Klonet restarting the controller or agent, the services they provide may temporarily fail. Therefore, Klonet designs redundant backups for them to provide services during the restart process to ensure uninterrupted services.

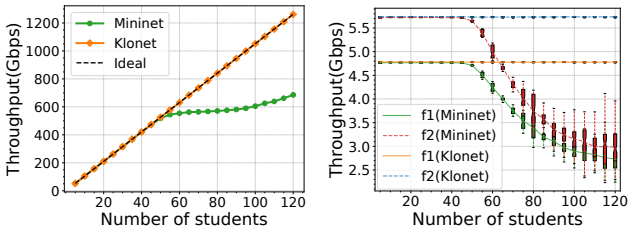
Robustness of the user management model. The methods of ensuring the robustness of the user management model are similar to those of the Klonet orchestrator. First, Klonet leverages an odd number of processes to monitor the status of the model. Second, if a failure is detected, Klonet switches to a backup. The difference is that since the user management model is a storage module, it broadcasts the received instructions to each backup to achieve data synchronization.

Robustness of the virtual networks. The virtual networks can be divided into two parts: nodes and links, and Klonet ensures their robustness respectively.

Nodes. Klonet periodically obtains the running status of each node through the docker-py [68] library. If a node is found to have stopped unexpectedly, Klonet alerts the user and tries to restart the node.

Links. For the intra-server links, they fail when one of their end nodes fails, so Klonet re-establishes the corresponding intra-server links after the nodes are restored. For the inter-server links, Klonet verifies their health through connectivity checks. As inter-server links provide layer 2 communication without requiring IP addresses for end nodes, the traditional `ping` tool does not work. Thus, we design a simple *Layer 2 ping*: Each node periodically sends data frames to the opposite end and gets a response. If the peer does not respond several times, an alert is sent to the corresponding user and operator, indicating an invalid inter-server link.

Robustness of the cluster. The large number of virtual nodes and physical servers that Klonet manages need to be monitored in a unified manner for the convenience of the operator. We use the open-source software Prometheus [69] to monitor and alert the state (CPU, memory, *etc.*) of physical servers and virtual nodes, and visualize the data centrally on the cluster monitoring page (as in §4.2) of GUI.



(a) Total throughput. (b) Throughput distribution.

Figure 5: Evaluation of multi-student concurrent experiments.

Effectiveness. Thanks to efforts in robustness, Klonet ensured the smooth running of the course for two years. During the operation, The major anomalies are shown in Table 2, and most of them were solved quickly due to Klonet's alarm and recovery mechanism.

5 System Performance

5.1 Testbed Evaluation

In this part, we first evaluate Klonet's ability to support simultaneous experiments with varying numbers of students. We also evaluate the speed of Klonet to create a virtual network, which is essential for ease of use and efficiency.

Klonet can support more students to conduct experiments simultaneously. To evaluate how many students Klonet can support to conduct experiments at the same time, we assume that at the same moment, each student is assigned a dumbbell [70] virtual network with 4 hosts and 2 switches, and runs 2 TCP flows (f1 and f2) with bottleneck link bandwidths of 5Gbps and 6Gbps respectively using `iperf3` [71] for 15 seconds. Then we vary the number of students and record the throughput of all flows. Ideally, in a high-fidelity platform, individual flow throughput should remain constant irrespective of student count, while the total throughput scales linearly with student numbers. We believe that a platform cannot support a certain number of students if the experimental results are not faithful. The evaluation is conducted on a Ubuntu 22.04 server with 2 Intel Xeon Gold 5220R CPUs and 32G RAM. To obtain comparable results, we exclude platforms that rely on heavy-weight virtualization technologies such as VM, focusing instead on comparisons with emulators that, like Klonet, use light-weight container-like virtualization techniques. With this aim, we choose the well-known emulator Mininet [12] as the benchmark.

Figure 5(a) shows the total throughput of all f1 and f2 flows, where Mininet has a significant performance drop once the number of students reaches 50, and the total throughput of all students falls short of expectations. In contrast, Klonet demonstrates nearly ideal performance. Figure 5(b) shows the throughput distribution of all flows, where each flow in Klonet maintains the expected throughput while Mininet does not. Klonet can support more students' experiments simultaneously because the node emulation technology used by Klonet has better performance isolation than that of Mininet.

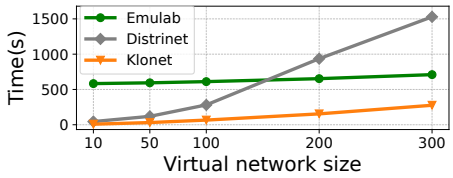
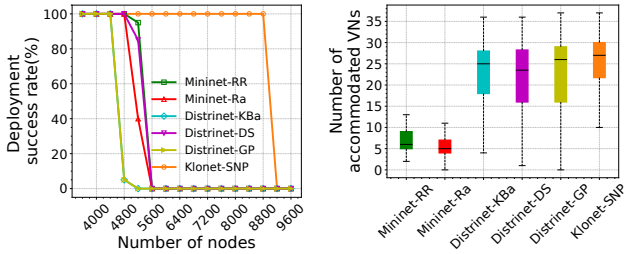


Figure 6: Virtual network creation speed.



(a) Success rate. (b) Success number.

Figure 7: Performance of Klonet's VNE algortihm(SwapNodesPartition). Compared with Mininet(RoundRobin, Random), and Distrinet(KBalance, DivideSwap, GreedyPartition)

Klonet has a faster speed to deploy VNs. We contrast Klonet's virtual networks creation time with Emulab [13] and Distrinet [15], which are representatives of the two platform operating modes: website access and local installation. To ensure consistency of the experimental environment, We request 10 *Emulab d430* bare-metal servers from Emulab and conduct the evaluation on them. We create container virtual networks of different sizes and collect the virtual networks creation time. The virtual network topology is Star, *i.e.*, multiple hosts connected to a central switch.

Figure 6 shows that the creation time for Distrinet experiences the most rapid increase as the network size escalates, mainly attributed to the utilization of LXC containers [72]. Conversely, Emulab necessitates the initiation of bare-metal servers before creating the virtual network, making it the slowest option when the network scale is below 150. In comparison, Klonet consistently exhibits the shortest deployment time up to 300 nodes. Note that the virtual node number in one student's experiment is rarely larger than 300.

5.2 Large-scale Simulation

Our VNE algorithm is one of the keys to Klonet's scalability, we use large-scale simulation to see how large and how many virtual networks it can support. We compare the proposed VNE algorithm with a range of standard algorithms employed by Mininet (cluster edition) [42] and Distrinet [73], which are two leading platforms capable of deploying large-scale virtual networks. We simulate a cluster of 10 servers, each server has

80 CPU cores with 100 CPU quota per core, and 1000Mb bandwidth capacity. To simulate network load, we randomly generate two types of topologies, FatTree and Random. The node and link resource requirements are 10 CPU quota and 10Mb, respectively. If the node or link resource requirements of a virtual network cannot be met, the its deployment is considered a failure. Note that the results of all compared VNE algorithms depend on numerical calculations, so the results are consistent in both simulation and practice.

Figure 7(a) reports the average success rate of VN deployment with varying sizes over 20 runs, decreasing as nodes increase. Mininet and Distrinet fail above 4400 nodes, while Klonet succeeds up to 8800 nodes. Figure 7(b) shows the average number of VNs accommodated over 100 runs. Klonet supports the largest number of VNs with less fluctuation in results. Distrinet ranks second, while Mininet performs worst, only one-third of Klonet. Klonet's better scalability performance benefits from its strategies of reducing cut-link weights and maximizing the potential deployment capacity of the cluster, while Distrinet only considers the former.

6 Use Cases

We have been using Klonet in our teaching practices for 2 years, and Table 3 shows the successful usages facilitated by the technical designs of Klonet.

In the following, we describe how Klonet is used in our university courses. We use two examples, one from an undergraduate-level course (Network Algorithms), and the other graduate-level (Enterprise Networks).

- **Project 1: Playing with algorithms in programmable networks:** In this project, students will learn how to implement classic shortest-path computation algorithms and compare their performance in networks with the programmable control plane or programmable data plane.
- **Project 2: Intra-domain routing:** In this graduate-level project, students will learn how to build and operate their slice of the intra-domain network. They will interconnect networks together and investigate the performance of the OSPF protocol in networks of different sizes.

6.1 Project 1: Playing with algorithms

Project Overview. Students are required to understand the basic principles of path computation algorithms in our lectures. This project helps them to gain hands-on experience with the actual performances of these algorithms in a real programmable network. We ask students to use OpenFlow [38] or P4 [39], which is widely recognized as the control plane programming protocol and data plane programming language, respectively, for implementing path computation algorithms. We request students to evaluate and compare the performance of these algorithms. Moreover, with the increasingly wide application of programmable networks in academia and industry, exposure to programmable networks itself can help students learn advanced topics in the network and engage in related research efforts. Therefore, another goal of this project is to teach basic concepts of programmable networks.

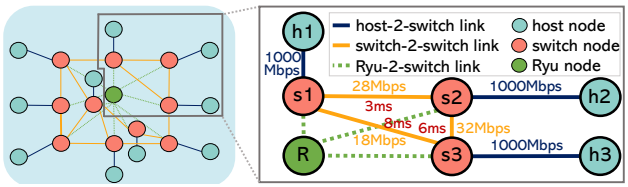


Figure 8: Topology of "Play with Alg." project.

To help students focus on algorithm-related matters rather than network configuration and operation, we use Klonet to pre-create a programmable network, then ask students just to write algorithms and feed the algorithm result into pre-written OpenFlow or P4 program templates. Specifically, since P4 is difficult for undergraduates to get started with, we only let interested students leverage the source routing [74] based on P4 to implement the result of the algorithms.

By the end, we expect the learning outcomes to be:

- Understand how OpenFlow works and master the basic operations of the OpenFlow flow table;
- Write classic path computation algorithms in OpenFlow controllers;
- Identify the effectiveness or limitations of different path computation algorithms;
- (Optional) Appreciate the difference between control plane programming and data plane programming.

To this end, we design three sub-projects in this project. Each sub-project asks students to implement one of the following classic shortest-path algorithms.

Sub-project #1: Use the Depth-First Search (DFS) algorithm to find a path with the minimum latency between two hosts.

Sub-project #2: Use the Dijkstra algorithm to find the shortest path between two hosts.

Sub-project #3: Use the Ford–Fulkerson algorithm to find the maximum network flow and the relative augmenting paths.

Figure 8 depicts the topology used, sub-project #1-2 sets links with different latencies, and sub-project #3 has different link bandwidth. Note that the switch can be an OpenFlow-enabled OpenvSwitch [60] or a P4-enabled BMv2 [61], and the controller is implemented by Ryu [65].

Using Klonet. Klonet helps students, tutors, and teaching assistants (TA) in the entire process of this 5-week project.

• Tutors use Klonet's image repository to pull new types of nodes to design new experiments quickly. Klonet gives tutors the simplicity and flexibility to update their experiments as network technology changes rapidly.

• Students use Klonet's scene repository to obtain the correct and tested version of network scenes. Compared to previous

Table 3: Klonet usages.

Grade	Use Cases	Scale	Servers[1]
2nd year undergraduate	Implement algorithm in programmable networks.	∼150stu/yr.	*S1**3
3rd year undergraduate	Networking virtual devices with Raspberry Pi.	∼40stu/yr.	*S1**1
1st year graduate	Implement network configuration such as NAT.	∼80stu/yr.	*S2**19
1st year graduate	Collaborate to achieve routing.	∼30stu/yr.	①*S1**3, *S4**14, *S5**1, ②*S3**140[2]
Self learners	Verify basic knowledge from textbooks, *etc.*	∼100stu/yr.	*S5**2

[1] Server types: *S1* (40-core 2.10GHz CPU, 256GB RAM), *S2* (2-core 2.40GHz CPU, 4GB RAM), *S3* (8-core 2.49GHz CPU, 32GB RAM), *S4* (8-core 1.70GHz CPU, 32GB RAM), *S5* (32-core 2.30GHz CPU, 128GB RAM)

[2] For this course project, we use different servers and virtual network size in each year. §6.2 describes the second-year course project.

courses without using Klonet, this dramatically reduces configuration problems for students, thus the workload of TAs.

• Klonet helps students focus on learning outcomes. TAs pre-build the VN and share it with students via the *scene repository* of Klonet before the start of each sub-project.

• Klonet allows students to interact with the elements in the networking scenario easily. It is recommended that students establish an SSH connection with the controller node to improve programming efficiency, at the same time use the *web terminal* for simple tasks, such as running the *ping* command in the host or getting the flow table of a switch. Students are also asked to write a replay script (which is just a few lines of code) and upload their sub-project into the *scene repository*.

• Klonet makes it easy for tutors and TAs to help with specific problems from students. In previous years without Klonet, tutors and TAs must deal with students' individual environments on their personal computers. Klonet standardizes the experimentation environment, thus significantly lessening the workload for teaching staff. They can directly recreate the student's networking scene and help the student solve problems.

• Klonet facilitates testing and grading of the project. After students complete a sub-project, the tutor or TA can replay the sub-project uploaded by them in the *scene repository* and grade them. Outstanding sub-projects are selected and shared in *scene repository* for students to replay and self-correct.

6.2 Project 2: Intra-domain Routing

Project overview. This graduate-level project is a hands-on experiment that involves realistic network operations. We intend to let students understand how the real network works and how network professionals operate their networks. Students are asked to build and operate an intra-domain network themselves cooperatively. To make this project realistic, we collaborated with an Internet service company to design the project. We chose the OSPF routing protocol as it is widely deployed in production networks. The students are expected to achieve the following learning outcomes:

- Build and operate their own enterprise network to have a deeper understanding of OSPF.
- Understand the importance of splitting areas.

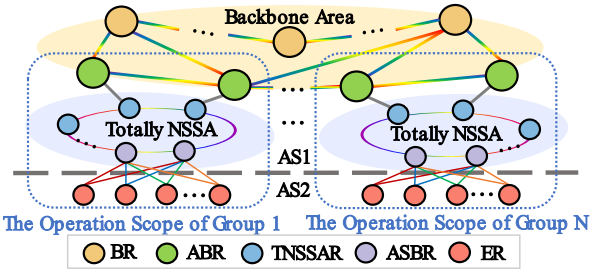


Figure 9: Topology of "Intra-domain Routing" project.

- Observe and learn the benefits of route aggregation.

Our industry collaboration partner provides the setting and configuration skeletons. Figure 9 shows the topology used in this project, with a backbone area and several Totally Not-So-Stubby-Areas (NSSA) [75]. The backbone area is the core, a mesh structure formed by routers randomly connected to each other. Some routers are called Area Border Routers (ABRs), and others are called Backbone Routers (BRs). Every two ABRs are connected to a Totally NSSA, whose topology is similar to the backbone area. Each Totally NSSA consists of Autonomous System Border Routers and Totally NSSA Routers (TNSSARs), in which every two Autonomous System Border Routers (ASBRs) connect to a number of Edge Routers (ERs) of another AS.

Using Klonet. Klonet helps students, tutors, and teaching assistants (TA) throughout this course project. We divide students into groups and let each group operate a set of routers (See Figure 9). The tutor configures BRs. Given the difficulty of this project, it is divided into three stages and each lasts for 6 weeks to configure a single area, multiple areas, and multiple areas with route aggregation, respectively.

• Klonet provides standardized and reproducible experimental settings for students and the teaching staff. The routers are implemented by the FRRouting [62] in the *image repository* of Klonet. At the outset of each stage, TAs pre-build a small VN with a total of about 500 nodes for students. Students are required to configure the routers to satisfy the stage goal and achieve network-wide connectivity. Then, students are required to modify the FRRouting configuration files by using Klonet's experiment APIs. The students are told to have scal-

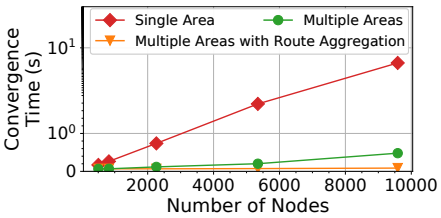


Figure 10: Convergence time of OSPF.

ability in mind, that is, when the number of routers increases or decreases, their codes should still work well.

• Klonet enables easy access to the emulated network devices. Students are mainly required to configure Router ID, Unnumbered interface, Areas and ASes, and Route aggregation to meet the goal of each stage. We have students configure the IP address of the loopback interface as the OSPF router id. The unnumbered interface can directly use the IP address of the loopback interface, thus simplifying the configuration. Students configure the NSSAs and ASes (different OSPF instances) for each router and configure route redistribution on ASBRs. Also, students configure route aggregation on the ABRs and ASBRs to aggregate routes and eliminate the communication overhead.

• Klonet makes it easy for tutors to create failure scenes in a live emulated network. At the end of each stage, the tutor creates a series of topologies of increasing size (up to ∼10,000 routers) and asks students to rerun and debug their code to build network-wide connectivity again. The tutor then takes down one link in the backbone area randomly and measures the convergence time. Finally, the tutor plots the data from the students' joint effort to show them how different network configurations affect network performance, especially convergence time.

• Klonet enables the teaching staff to develop and deploy project-specific tools. We provide a measurement tool for students to obtain the convergence time of OSPF. This tool collects logs from all routers located on each server, and by analyzing the logs, gets the start time (*i.e.*, the time that the breakdown is detected among routers) and the end time (*i.e.*, the latest time that the routing table update is completed among all routers) of the OSPF convergence process. The time synchronization scheme between servers is chrony [76]. Besides, we developed a script called `PingAll` to test the connectivity between routers. The TAs can directly update the image in the repository to distribute these tools to all students.

• The scalability of Klonet makes it possible for students to experience networks running at a realistic and modern scale. Figure 10 shows the convergence performance of different operational strategies at different network scales. Klonet is able to emulate an intra-domain network of 10,000 nodes on our cluster. The result also teaches students the importance of splitting areas and route aggregation for OSPF networks.

6.3 Feedback

To understand the reception of Klonet by students and the teaching staff, we conduct a user survey collected from more than 300 participants (67% bachelor students, 25% master students, 8% tutors and TAs). According to this survey, we have received positive feedback from most students and tutors. Most students give positive comments such as: "It helps me know the algorithms better" and "I definitely can not imagine how these algorithms work in a real network without conducting this project". The comments from teaching staff are also positive, such as: "this platform really helps quickly get started and build the experimental topology"; "it also helps save time on tedious operations and gain real experience"; "using this platform, it is easier for us to realize the experimental design of various scenarios." As mentioned in §7, there is also some negative feedback from students, and the full survey can be found in Appendix D.

The feedback confirms the need for a scalable and easy-to-use network education platform like Klonet.

7 Lessons Learnt

In this section, we introduce several lessons we have learned from developing (L1), operating (L2, L3), and using (L4, L5) Klonet in education. We believe these lessons are valuable to those seeking to establish and operate an educational network emulation platform like Klonet or use Klonet in education.

L1. Connecting to the Internet is important but has side effects. Initially, Klonet nodes had essential tools like `iperf` pre-installed, and none were connected to the Internet. However, during the teaching process, students said that they occasionally need to access the Internet for purposes like downloading Python packages. Therefore, we have all nodes connected to the Internet, which introduces a new issue: While conducting the experiment, network traffic between the two nodes did not follow the expected path specified and was routed to the bridge for the Internet connection.

To address this challenge, Klonet implemented a unique design featuring an on-off button for on-demand Internet connectivity. During the programming phase, students can toggle the button on to enable Internet connection for software installation. In contrast, they should toggle it off during experimentation to preemptively disable the Internet connection, thereby circumventing potential routing interference. It's worth noting that students can still control the node using the web terminal connected to the Docker daemon, not the Internet bridge, even when the Internet connection is offline.

L2. Resources between students need careful isolation. In our first year using Klonet for Project 1, we unexpectedly observed the *broadcast storm problem*, which exhausts CPU and memory resources, crashing the platform. This was caused by some students inadvertently flooding ARP requests to the switch with the Ryu controller. Rather than simply in-

forming students not to use ARP in ring topologies, we saw an opportunity for them to appreciate the importance of path computation algorithms. From the perspective of design, this failure highlights poor resource isolation between students. The root cause is that the container technology is not perfect in resource isolation, with all containers sharing server resources. To observe the effects of broadcast storms without crashing Klonet, we bound each container to a CPU core, limiting the impact of excessive resource usage by a single student and preventing the disruption of other experiments.

L3. Be careful when deploying Klonet or similar platforms in the public cloud. During the COVID-19 pandemic, to facilitate remote education, we deployed Klonet on the public cloud since the campus network is not accessible from the outside. However, we encountered an issue in that initially Klonet was available for a few minutes, but subsequently, broken links between its components were reported.

Upon inspection, the load balancer of the public cloud was found to terminate links between components they do not communicate within a specified period. To remedy this, we adjusted heartbeat intervals between Klonet's components to keep the link alive. Notably, unlike private clusters, these implicit mechanisms in public clouds may bring unexpected challenges.

L4. Knowledge not closely related to computer networking requires effort. Many students expressed their love for designs such as graphic design topology and web terminals, rating them as necessary, interesting and intuitive because they simplify operations and improve efficiency. However, according to student feedback, in addition to networking knowledge itself, any knowledge that is new to them (especially Linux) makes them struggle. This is overlooked in the first year. Therefore, we have supplemented and improved the introduction of relevant background knowledge in lecture notes.

L5. API access should be differentiated for students with different backgrounds. Klonet's rich APIs facilitate customizable experiments, but unrestricted access could potentially overwhelm students. For example, in our algorithm project (§6.1), all APIs are available to students initially in the first year of teaching. Due to the lack of background knowledge, students have no clear boundaries about the task they need to complete (*i.e.*, network algorithm implementation), wasting a lot of time on the preliminary work (*i.e.*, building topology and configuring link properties). We then restricted the APIs available to students in this course and let them use the built and configured topologies we provided in the scene library. With this restriction, we found that students spend significantly more time on algorithm implementation and verification, aligning better with our teaching purposes.

8 Discussion

Applications in research and even industry. The use of Klonet in research or even industry is also very encouraging. Indeed, several researchers have already adopted Klonet to carry out research works, like optimizing clock synchronization in large-scale clusters [77]. However, the research and industry field demands greater fidelity than education. For example, with our evaluation, inter-server links inevitably introduce a ms-level tail latency. While acceptable in education, such latency proves unacceptable where deterministic latency is required. Driven by the requirement from our cooperation projects with large networking companies, we are exploring techniques such as hardware offloading and latency compensation to alleviate this issue. In the future, we will make more optimizations to improve Klonet's fidelity.

Promoting Klonet's use to more courses and universities. Supported by college policy, several other courses are scheduled to use Klonet next semester. We may face more problems with large-scale use, and we will continue to maintain and optimize Klonet. Additionally, several other universities are also planning to introduce Klonet into some courses next year. However, due to many non-technical reasons, they require Klonet to be deployed on their own private cluster. This requires Klonet to be easily installed. Klonet simplifies deployment with a one-click installation script, improving ease of setup compared to Emulab [37]. In the future, we plan to offer a more convenient way for installation, using a VM image that contains Klonet and its dependencies. Technically it is easy, and popularization is more of a non-technical issue.

Limitations of applicable scenarios. While supporting a variety of scenarios, it must be acknowledged that Klonet still has limitations. For example, Klonet can only support terrestrial networks and cannot support emulating the ISTN now. In addition, Klonet uses Docker technology, which does not facilitate the modification of some Linux kernel options. For example, it cannot support experimental scenarios of adjusting the congestion control scheme in the kernel. We plan to introduce support for VMs next year to meet the different levels of configuration requirements in more scenarios.

Continuous optimizing ease of use. To ease the burden on students, we are constantly working hard to improve the ease of use of Klonet. For instance, within the laboratory, we have integrated generative AI to enhance its capabilities (*e.g.* automating topology deployment through natural language), which is currently being refined.

9 Conclusion

This paper presents Klonet, an easy-to-use and scalable network emulation platform for education. We introduce the design and implementation of Klonet. Besides, we give real use cases by using Klonet in our teaching courses which shows it has a great benefit in education. We also introduce our lessons learned from the 4 years of developing and 2 years of using Klonet in education. We plan to make Klonet available as an educational infrastructure, and we hope that Klonet can help the development of computer network education.

Acknowledgements

We would like to thank the anonymous reviewers and our shepherd Eric Eide for their constructive comments and feedback. Special thanks to Emulab for making our performance evaluation possible. We also thank Yi Gao of Linklab 2.0 for his help. We thank all members in Klonet team for their contributions, including but not limited to: Jingshan Duan, Jian Sun, Xiaoyu Yu, Chen Wang, Xunpei Hu, Yifan Su, Chang Xiao, Wei Shan, Bo Tao, Shiman Mei, Yichen He, Dongxu Wu, Jiahao Guo, Kecheng Gong, and Yifan Feng. This work is supported in part by National Key Research and Development Program of China (2021YFB3101001, 2023YFB2904600), National Natural Science Foundation of China (62102066, 62394324), Young Elite Scientists Sponsorship Program by CAST (2022QNRC001).

References

[1] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[2] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of cloudlab. In *USENIX Annual Technical Conference*, pages 1–14, 2019.

[3] Mark Berman, Jeffrey S Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. Geni: A federated testbed for innovative network experiments. *Computer Networks*, 61:5–23, 2014.

[4] Wei Dong, Borui Li, Haoyu Li, Hao Wu, Kaijie Gong, Wenzhao Zhang, and Yi Gao. {LinkLab} 2.0: A multi-tenant programmable {IoT} testbed for experimentation with {Edge-Cloud} integration. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1683–1699, 2023.

[5] Jeongyoon Eo, Zhixiong Niu, Wenxue Cheng, Francis Y Yan, Rui Gao, Jorina Kardhashi, Scott Inglis, Michael Revow, Byung-Gon Chun, Peng Cheng, et al. Opennetlab: Open platform for rl-based congestion control for real-time communications. *Proc. of APNet*, 2022.

[6] The University of Washington NS-3 Consortium. Ns3 official website. https://www.nsnam.org/, 2023.

[7] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *1st International ICST Conference on Simulation Tools and Techniques for Communications, Networks and Systems*, 2010.

[8] Jozef Janitor, František Jakab, and Karol Kniewald. Visual learning tools for teaching/learning computer networks: Cisco networking academy and packet tracer. In *2010 Sixth international conference on networking and services*, pages 351–355. IEEE, 2010.

[9] Qingqing Yang, Xi Peng, Li Chen, Libin Liu, Jingze Zhang, Hong Xu, Baochun Li, and Gong Zhang. Deepqueuenet: towards scalable and generalized network performance estimation with packet-level visibility. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 441–457, 2022.

[10] Kaihui Gao, Li Chen, Dan Li, Vincent Liu, Xizheng Wang, Ran Zhang, and Lu Lu. Dons: Fast and affordable discrete event network simulation with automatic parallelization. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 167–181, 2023.

[11] Qizhen Zhang, Kelvin KW Ng, Charles Kazer, Shen Yan, João Sedoc, and Vincent Liu. Mimicnet: fast performance estimates for data center networks with machine learning. In *Proceedings of the ACM SIGCOMM 2021 Conference*, pages 287–304, 2021.

[12] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6, 2010.

[13] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale virtualization in the emulab network testbed. In *USENIX Annual Technical Conference*, 2008.

[14] Mathieu Jadin, Olivier Tilmans, Maxime Mawait, and Olivier Bonaventure. Educational virtual routing labs with ipmininet. In *ACM SIGCOMM Education Workshop*, 2020.

[15] Giuseppe Di Lena, Andrea Tomassilli, Damien Saucez, Frédéric Giroire, Thierry Turletti, and Chidung Lac. Distrinet: A mininet implementation for the cloud. *ACM SIGCOMM Computer Communication Review*, 51(1):2–9, 2021.

[16] Manuel Peuster, Johannes Kampmeyer, and Holger Karl. Containernet 2.0: A rapid prototyping platform for hybrid service function chains. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 335–337. IEEE, 2018.

[17] Gaetano Bonofiglio, Veronica Iovinella, Gabriele Lospoto, and Giuseppe Di Battista. Kathará: A container-based framework for implementing network function virtualization and software defined networks.

In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2018.

[18] Mariano Scazzariello, Lorenzo Ariemma, Giuseppe Di Battista, and Maurizio Patrignani. Megalos: A scalable architecture for the virtualization of network scenarios. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7. IEEE, 2020.

[19] Jeremy Grossman, et al. Graphical network simulator-3 (gns3). https://www.gns3.com/, 2023.

[20] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 253–264, 2012.

[21] Jiaqi Yan and Dong Jin. Vt-mininet: Virtual-time-enabled mininet for scalable and accurate software-define network emulation. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, pages 1–7, 2015.

[22] Ramon R Fontes, Samira Afzal, Samuel HB Brito, Mateus AS Santos, and Christian Esteve Rothenberg. Mininet-wifi: Emulating software-defined wireless networks. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 384–389. IEEE, 2015.

[23] Mark Carson and Darrin Santay. Nist net: a linux-based network emulation tool. *ACM SIGCOMM Computer Communication Review*, 33(3):111–126, 2003.

[24] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *ACM SIGOPS Operating Systems Review*, 36(SI):271–284, 2002.

[25] Valerio Schiavoni, Etienne Rivière, and Pascal Felber. Splaynet: Distributed user-space topology emulation. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 62–81. Springer, 2013.

[26] Paulo Gouveia, João Neves, Carlos Segarra, Luca Liechti, Shady Issa, Valerio Schiavoni, and Miguel Matos. Kollaps: decentralized and dynamic topology emulation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[27] Jiamin Cao, Ying Liu, Yu Zhou, Lin He, and Mingwei Xu. Turbonet: Faithfully emulating networks with programmable switches. *IEEE/ACM Transactions on Networking*, 2022.

[28] Dimosthenis Pediaditakis, Charalampos Rotsos, and Andrew William Moore. Faithful reproduction of network experiments. In *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 41–52, 2014.

[29] Andreas Grau, Klaus Herrmann, and Kurt Rothermel. Netplace: Efficient runtime minimization of network emulation experiments. In *Proceedings of the 2010 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'10)*, pages 265–272. IEEE, 2010.

[30] Andreas Grau, Klaus Herrmann, and Kurt Rothermel. Netbalance: Reducing the runtime of network emulation using live migration. In *2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–6. IEEE, 2011.

[31] Philip Wette, Martin Dräxler, Arne Schwabe, Felix Wallaschek, Mohammad Hassan Zahraee, and Holger Karl. Maxinet: Distributed emulation of software-defined networks. In *2014 IFIP Networking Conference*, pages 1–9. IEEE, 2014.

[32] Elias Weingärtner, Florian Schmidt, Hendrik Vom Lehn, Tobias Heer, and Klaus Wehrle. Slicetime: A platform for scalable and accurate network emulation. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 253–266, 2011.

[33] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 599–613, 2017.

[34] Zeqi Lai, Hewu Li, Yangtao Deng, Qian Wu, Jun Liu, Yuanjie Li, Jihao Li, Lixin Liu, Weisen Liu, and Jianping Wu. StarryNet: Empowering researchers to evaluate futuristic integrated space and terrestrial networks. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1309–1324, 2023.

[35] Wenliang Du, Honghao Zeng, and Kyungrok Won. Seed emulator: an internet emulator for research and education. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 101–107, 2022.

[36] Thomas Holterbach, Tobias Bü, Tino Rellstab, and Laurent Vanbever. An open platform to teach how the internet practically works. *ACM SIGCOMM Computer Communication Review*, 50(2):45–52, 2020.

[37] Maurizio Pizzonia and Massimo Rimondini. Netkit: network emulation for education. *Software: Practice and Experience*, 46(2):133–165, 2016.

[38] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, mar 2008.

[39] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[40] The Kubernetes Authors. Kubernetes official web site. https://kubernetes.io/, 2023.

[41] Marina Prvan and Julije Ožegović. Methods in teaching computer networks: a literature review. *ACM Transactions on Computing Education (TOCE)*, 20(3):1–35, 2020.

[42] The Mininet Project. Cluster edition prototype. https://github.com/mininet/mininet/wiki/Cluster-Edition-Prototype, 2024.

[43] Kathará. kathara success stories. https://www.kathara.org/stories.html, 2024.

[44] Flask. Flask document. https://flask.palletsprojects.com/en/2.2.x/, 2023.

[45] Microsoft. Vscode remote ssh plugin. https://code.visualstudio.com/docs/remote/ssh, 2023.

[46] Docker Inc. Docker low-level api. https://docker-py.readthedocs.io/en/stable/api.html#docker.api.exec_api.ExecApiMixin.exec_start, 2023.

[47] Ian Fette and Alexey Melnikov. Rfc 6455: The websocket protocol, 2011.

[48] Grafana Labs. Grafana official web site. https://grafana.com/, 2023.

[49] Oracle. Mysql official web site. https://www.mysql.com/, 2023.

[50] Docker Inc. Docker registry document. https://docs.docker.com/registry/, 2023.

[51] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. Enabling ecn in multi-service multi-queue data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 537–549, Santa Clara, CA, March 2016. USENIX Association.

[52] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review*, 40(1):92–99, 2010.

[53] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. Trafficgenerator. https://github.com/HKUST-SING/TrafficGenerator, 2023.

[54] Scapy Community. Scapy. https://scapy.net/, 2023.

[55] The Tcpdump Group. Home | tcpdump & libpcap. https://www.tcpdump.org/, 2023.

[56] FNSS. Fnss official web site. https://fnss.github.io/, 2023.

[57] Python Software Foundation. Coroutines and tasks. https://docs.python.org/3/library/asyncio-task.html#coroutines-and-tasks, 2023.

[58] Redis Ltd. Redis official website. https://redis.io/, 2023.

[59] Docker Inc. Docker official web site. https://www.docker.com/, 2023.

[60] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open {vSwitch}. In *12th USENIX symposium on networked systems design and implementation (NSDI 15)*, pages 117–130, 2015.

[61] Bas Antonin and Fingerhut Andy and Sivaraman Anirudh and Arora Dushyant. Behavioral model (bmv2). https://github.com/p4lang/behavioral-model, 2023.

[62] FRRouting Project. Frrouting. https://frrouting.org/, 2023.

[63] Paul Jakma and David Lamparter. Introduction to the quagga routing suite. *IEEE Network*, 28(2):42–48, 2014.

[64] Docker Inc. Ubuntu image. https://hub.docker.com/_/ubuntu/, 2023.

[65] Ryu SDN Framework Community. Ryu controller. https://github.com/faucetsdn/ryu, 2023.

[66] Mallik Mahalingam, Dinesh Dutt, Kenneth Duda, Puneet Agarwal, Lawrence Kreeger, T Sridhar, Mike Bursell, and Chris Wright. Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. Technical report, 2014.

[67] Linux Foundation. tc(8) - linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html, 2023.

[68] Docker Inc. Docker sdk for python. https://github.com/docker/docker-py, 2023.

[69] Prometheus Authors. Prometheus. https://prometheus.io/, 2023.

[70] P Subramanya, KS Vinayaka, HL Gururaj, and B Ramesh. Performance evaluation of high speed tcp variants in dumbbell network. *IOSR Journal of Computer Engineering*, 16(2):49–53, 2014.

[71] iPerf Group. iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/, 2023.

[72] Canonical Ltd. Linux containers. https://linuxcontainers.org, 2023.

[73] Giuseppe Di Lena, Andrea Tomassilli, Damien Saucez, Frédéric Giroire, Thierry Turletti, and Chidung Lac. Distrinet: A mininet implementation for the cloud. *ACM SIGCOMM Computer Communication Review*, 51(1):2–9, 2021.

[74] Networked Systems Group (NSG). P4-learning. https://github.com/nsg-ethz/p4-learning/tree/master/examples/source_routing, 2023.

[75] Pat Murphy. The ospf not-so-stubby area (nssa) option. Technical report, 2003.

[76] Chrony. Chrony. https://chrony.tuxfamily.org/index.html, 2023.

[77] Zhuochen Fan, Xiaodong Li, Yanwei Xu, Yuqing Li, Tong Yang, and Steve Uhlig. Work-in-progress: A novel clock synchronization system for large-scale clusters. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 519–522. IEEE, 2022.

[78] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

Appendices

A Experiment Page

Figure 11 shows the screenshot of the experiment page of Klonet's GUI. Here is a description of the various elements visible in the screenshot:

• **Klonet Dashboard.** The header at the top left indicates that this is the "Klonet Dashboard", suggesting that the platform is named "Klonet".

• **User and Experiment Information.** In the top right corner, there are details like "demo_user" and "demo_experiment", suggesting the current user's username and the name of the experiment or project they are working on, along with a "Log out" option.

• **Navigation and Action Buttons.** On the top left, there are buttons labeled "Return" and "Save", which are likely used to navigate back to a previous page and to save the current configuration, respectively.

• **Typical Topologies.** On the left sidebar, there are icons representing different network topologies such as "Tree", "Star", "Fattree", and "Linear". These could be templates for quickly setting up network structures.

• **Image Repository.** Below the Typical Topologies, there is an "Images" section with icons representing different node images. Users can upload images to this repository and later instantiate them as nodes.

• **Main Workspace.** The central area of the experiment page shows a visual representation of a network topology. Users can instantiate nodes by dragging and dropping icons from the image repository and are able to create links between nodes by connecting wires between icons. In addition, Klonet provides a control menu for each node, which can be expanded by right-clicking on the icon.

• **Control Menu of a Node.** After right-clicking the node, a contextual menu is open with options such as "Configure", "Terminal", "Delete", "File Upload", and "File Download". Users can interact with each node in the workspace to configure settings, access the web terminal, remove the element, or upload and download files.

• **Experiment Management Panel.** On the right, there is an "Experiment Management" panel where users can control the creation and destruction of the experiment, as well as "Traffic Generation" and "Traffic Measurement" sub-options to generate network traffic and monitor performance metrics such as throughput and latency of network traffic.

• **Entities Section.** Below the management panel, there's an "Entities" section with options to select "Node" or "Link", followed by a "Search" bar, which is used to filter or find specific nodes or links within the network topology. Here, users can interact with nodes (as in the "Control Menu"), delete links, and configure link properties like bandwidth.

B The Design of User Management Model

As shown in Figure 12, We assign each student a *student space* to record the experiment information. In the *student space*, there are several important components:

- The *agent_list* records the IP address of every agent.
- The *VN_list* records the name and corresponding description (*i.e.*, the information of nodes and links) of VNs of students.
- The *VN2subVN*, *subVN2agent* and *subVN_service* describe the information of the sub-VNs derived from virtual network embedding algorithm for all VNs. The *VN2subVN* records the list of sub-VN names, and users can query additional information about a sub-VN using its name. The *subVN2agent* records the IP of the server where each sub-VN is deployed. The *subVN_service* records nodes that need to start some necessary initialization programs when a VN is created.
- <*VN*>_<*node*>, <*VN*>_<*link*> and <*VN*>_<*vxlan*> are the objects of virtual devices. We use the names of VN, link, and node to index an object.

C Details of VNE Algorithm

Algorithm 1: Virtual network embedding algorithm

Input: Virtual network G, Cluster S_{server}

Output: An embedding strategy: $G \xrightarrow{mapping} S_{server}$

Compute $w_G \leftarrow$ the *sum* of the node weights of G
Compute the IDX of each server in S_{server}
if $\exists p_n \in S_{server} : C(p_n) \geq w_G$ **then**
    Embed G on the server p with the largest IDX and has a CPU capacity close to but larger than w_G;
else
    Invoke *strict_partition* (S_{server}, G)
end
Procedure `strict_partition`(S_{server}, G)
    Sort S_{server} by CPU capacity in descending order
    Select the first k' servers whose sum of CPU capacity is just greater than w_G;
    `/* k' is also the number of sub-VNs */`
    Compute the target weights $\boldsymbol{w}$ taken by the partitioning operation for sub-VNs:
    $\boldsymbol{w} = [w_1, ..., w_{k'-1}, w_{k'}] \leftarrow$
    $[\frac{C(p_1)}{w_G}, ..., \frac{C(p_{k'-1})}{w_G}, \frac{w_G - \sum_{n=1}^{k'-1} C(p_n)}{w_G}]$;
    Obtain sub-VNs: $\boldsymbol{s} \leftarrow$ Partition($G, k', \boldsymbol{w}$);
    Embed $\boldsymbol{s}$ to the selected k' servers;
    Adjust the mapping of virtual nodes to satisfy resource constraints and reduce the IDX of servers if necessary.
return *Embedding strategy of sub-VNs*

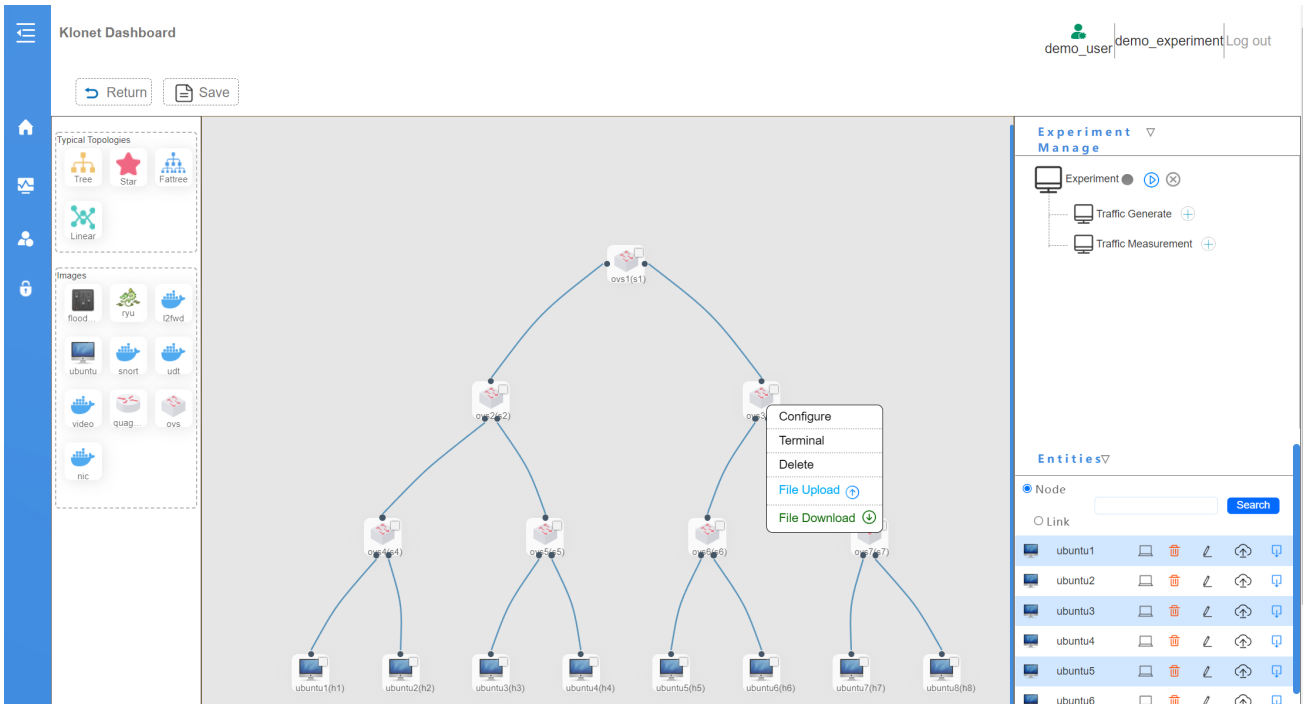


Figure 11: The Experiment Page.

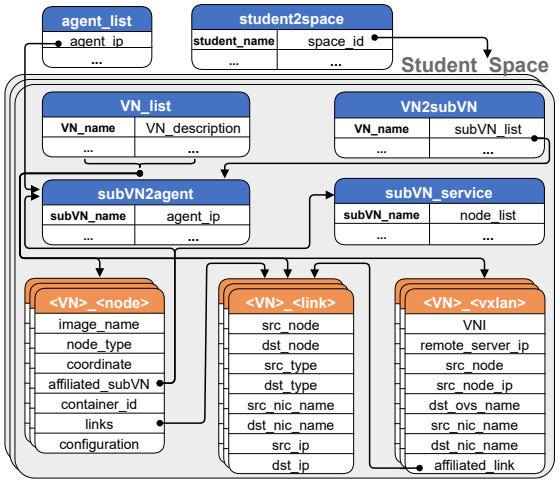


Figure 12: The User Management Model.

As shown in Algorithm 1, If a small VN can be deployed on a single server, we map it to the server with the largest IDX that satisfies the CPU capacity requirement. Otherwise, we will call the *strict_partition* function (described later) to partition this VN into a series of sub-VNs and then map these sub-VNs. To improve resource utilization, *strict_partition* first determines the number k' of sub-VNs and finds a suitable subset of candidate servers for mapping. Then, it calculates the weights $\boldsymbol{w} = [w_1, \cdots, w_{k'-1}, w_{k'}]$ of sub-VNs proportional to the capacity of candidate servers. Taking k' and $\boldsymbol{w}$ as inputs, *strict_partition* adapts a near-optimal graph partition method, METIS [78] to partition the VN into satisfied sub-VNs and then embed these sub-VNs. Finally, we adjust the mapping to meet resource constraints if it does not satisfy resource constraints and to reduce the IDX of servers if possible.

D The User Survey

We conducted an open-ended survey with the question, "What are your thoughts on the course and/or Klonet?" The survey involves 305 participants with three roles (204 bachelor students, 76 master students, 25 tutors and teaching assistants). We analyzed the questionnaire results of the three roles separately. Specifically, we conduct a coarse-grained classification of participants' answers and count the number of people with each type of answer, and then calculate the proportion of this number to the total number of people in that type of role. The results of the survey are shown in Table 4, Table 5, and Table 6.

Ethical Considerations.

- All participants in our survey gave informed and voluntary consent to the use of these data.
- Our institution's ethics review commission also consent to the conduct of this survey and its use. This work complies with all applicable ethical standards of our institution.
- We blur personal information(*e.g.* name) and group it into statistics to ensure maximizing the benefits to an individual.
- All individuals are anonymously hidden in our survey, so individual risk is minimized.
- Our survey has only one open question, so the privacy is respected.
- Also since our question is open, users cannot be induced.

Table 4: Survey result from undergraduate students.

		Comments	Number	Percentage(%)
Undergraduate Student (204)	**Positive**	Klonet is easy to get started.	43	21.08
		Can log in directly and use it very conveniently.	12	5.88
		SSH is helpful.	30	14.71
		Traffic generator or traffic monitor is helpful.	33	16.18
		Visualization is intuitive.	53	25.98
		Learn more knowledge (such as Python) after the course.	89	43.63
		Have deeper understanding of course knowledge.	157	76.96
		Acquired exemplary qualities (such as fearless in tackling difficulties,understand the importance of teamwork).	27	13.24
		Enhanced enthusiasm for learning.	30	14.71
		The course experiments are well designed.	6	2.94
		Feeling good about graphical terminals.	28	13.73
		Being able to realistically interact with network devices is fun.	14	6.86
		The TAs are very nice.	22	10.78
	Negative	Fewer documents, videos, *etc.* are available.	15	7.35
		Access is limited to the campus network; off-campus access is not available.	3	1.47
		Cannot display traffic paths in real-time.	11	5.39
		Front-end display of platform topology cannot be zoomed in and out.	2	0.98
		The first project is overly challenging as it demands a significant amount of time for comprehension of unrelated components (Python, Ryu, Xshell, ARP protocol, LLDP protocol). Despite the provision of a Ryu template, a substantial amount of time was still spent on comprehending the code within the template.	178	87.25
		The curricula and course design are too simple.	28	13.73
		There is little correlation between class content and practical content.	6	2.94

Table 5: Survey result from graduate students.

		Comments	Number	Percentage(%)
Graduate Students (76)	**Positive**	Gained a deeper understanding of OSPF or networking.	58	76.32
		Happy teamwork.	24	31.58
		Klonet is easy to get started.	10	13.16
		Acquired exemplary qualities.	9	11.84
		SSH is helpful.	21	27.63
		Enhanced enthusiasm for learning.	6	7.89
		Can log in directly and use it very conveniently.	5	6.58
	Negative	Configuring routing is difficult.	60	78.95
		Unhappy teamwork.	4	5.26
		Fewer documents, videos, *etc.* are available.	3	3.95
		Cannot use front-end for topology visualization.	11	14.47
		Experiments are too time-consuming and affect the research progress of themselves.	14	18.42
		There is little correlation between class content and practical content.	3	3.95
		A web IDE is required.	1	1.32

Table 6: Survey result from tutors and TAs.

		Comments	Number	Percentage(%)
Tutors and TAs(25)	**Positive**	Consolidated course knowledge.	20	80.00
		Fostered relationships with students.	5	20.00
		Designing experiments is easy.	19	76.00
		Like the image repository or scene repository feature.	22	88.00
		It's nice to be able to automatically scale on a cluster	3	12.00
	Negative	Experimental design is time-consuming.	6	24.00
		Takes some time to debug the software environment.	7	28.00
		Due to campus network limitations, cloud server rental may be required.	5	20.00
		Average level of student knowledge is lower than expected.	20	80.00
		Need to spend some time on offline Q&A.	8	32.00
		Lack of documentation.	2	8.00
		Configuration on the cloud is more cumbersome.	1	4.00
		Sometimes new features need to be developed based on experimental requirements.	2	8.00

EXCHAIN: Exception Dependency Analysis for Root Cause Diagnosis

Ao Li[1], Shan Lu[23], Suman Nath[2], Rohan Padhye[1], Vyas Sekar[1]
[1]*Carnegie Mellon University,* [2]*Microsoft Research,* [3]*University of Chicago*

Abstract

Many failures in large-scale online services stem from incorrect handling of exceptions. We focus on exception-handling failures characterized by three features that make them difficult to diagnose using classical techniques: (1) implicit dependencies across multiple exceptions due to state changes; (2) silent code handling without logging; and (3) separation (in code and in time) between the root cause exception and the failure manifestation. In this paper, we present the design and implementation of ExChain, a framework that helps developers diagnose such exception-dependent failures in test/canary deployment environments. ExChain constructs causal links between exceptions even in the presence of the aforementioned factors. Our key observation is that mishandled exceptions invariably modify critical system states, which impact downstream functions. A key challenge in tracking these states is balancing the tradeoff between performance overhead and accuracy. To this end, ExChain uses state-impact analysis to establish potential causal links between exceptions and uses a novel hybrid taint tracking approach for tracking state propagation. Using ExChain, we were able to successfully identify the root cause for 8 out of 11 reported subtle exception-dependent failures in 10 popular applications. ExChain significantly outperforms state-of-art approaches, while producing several orders of magnitude fewer false positives. ExChain also offers significantly better accuracy-performance tradeoffs relative to baseline static/dynamic analysis alternatives.

1 Introduction

Failures in large-scale production systems continue to be a significant source of frustration for developers and loss of customer satisfaction and revenues for service providers. A common root cause of system failures is *incorrect handling of exceptions* or errors.

Developers today can check whether a failure occurred during the handling of an exception and whether that exception was thrown during another exception handler, etc; i.e., track exception chains. Unfortunately, existing workflows are not useful for diagnosing failures whose root causes are outside the current exception chain, a type of failures that we refer to as **exception-dependent failure (EDFs)**. Exception-dependent failures involve multiple exceptions whose handling periods do *not* overlap (i.e., they do not belong to the same exception chain) and yet the (mis)handling of one *root cause* exception triggers a downstream exception which eventually leads to a failure.

In the context of large systems, diagnosing failures often necessitates an in-depth understanding of EDF. As highlighted by Yuan et al., "Almost all catastrophic failures (92%) stem from the incorrect management of non-fatal errors that are explicitly signaled in software" [59]. Complementing this, our manual analysis of 150 failures across multiple Apache Foundation projects affirms that 85% of these failures originate from exceptions. Unfortunately, existing solutions for root cause diagnosis of EDFs are insufficient; in our evaluation with 11 EDFs, state of the art slicing, log analysis, and statistical debugging techniques could identify the root causes of only three of fewer EDFs.

EDFs differ from simple exception-chains [25] in three key aspects that make them especially challenging to diagnose:

- *Implicit stateful dependencies:* In an explicit exception chain, the final failure-inducing exception is part of a cascaded chain of exceptions triggered by the root cause. In EDFs, however, the root cause exception can lead to a failure not only by just modifying the control flow of a program, but also by subtly changing the state.
- *Silent handling:* Due to the common practice of silent exception handling [59], the root cause exception may not be logged. This makes it difficult to diagnose exception-handling failures that may occur later on.
- *Spatial/temporal separation:* Finally, the root cause and the failure may be spatially and temporally distant from each other. For instance, the root cause exception may be triggered by one user request and the failure may be triggered by a different request that has some data dependency with the root cause.

For these reasons, we argue that it is critical to complement *exception chain* information, which is commonly produced and visualized by standard libraries in languages like Java and Python, with *exception dependency* information for effective failure diagnosis.

In this paper, we present ExChain, a tool that enables developers to diagnose EDFs by automatically inferring exception dependency. ExChain works by instrumenting the production system binary. At run time (e.g., during integration testing or canary deployment), ExChain routines, which were instrumented into the production binary, automatically log all exceptions and their contexts, and perform a dynamic-static hybrid analysis to identify *causal dependencies* among all exceptions; i.e., an exception e_2 causally depends on an exception e_1 if e_1 is responsible for e_2, either *explicitly* (e.g., throw e_2 inside a catch block of e_1) or *implicitly* (e.g., e_1 changes application states that causes e_2). Whenever a failure occurs, developers can query the exception-dependency graph produced by ExChain to see whether the failure was caused by the mishandling of an exception.

ExChain employs a set of novel state-impact analyses to establish potential causal links between exceptions. First, ExChain analyzes how exception-handling code changes program state—we call this *affected state analysis*—by identifying memory locations whose values are impacted by the change in control flow when compared to non-exceptional execution. Second, ExChain analyzes immediate causes for exceptions by tracking control flow backwards from the program locations where exceptions are raised and identifying memory locations whose values are responsible in triggering the exception—we call this *responsible state analysis*. Third, ExChain incorporates taint analysis to monitor the state affected by code addressing one exception as it influences values which activate other exceptions. Notably, ExChain introduces a hybrid algorithm, blending dynamic taint tracking for heap objects with static taint tracking for local primitives. This strategic approach positions ExChain uniquely in the taint-tracking arena, offering accuracy akin to dynamic taint analysis while achieving the overhead benefits of static taint tracking. Collectively, these methods enable ExChain to determine causality between exceptions, proving invaluable for EDF diagnosis.

We evaluated ExChain using 10 diverse applications from the Apache Foundation, spanning various domains and averaging 6K stars. Out of 11 reproducible EDFs instances, ExChain identified root causes for 8, outperforming the state-of-the-art statistical debugging, slicing, and log analysis tools, which pinpointed only 3 or fewer issues.[1] In performance metrics, ExChain introduced an average latency overhead of 8%, half attributable to its techniques and half to underlying JVM tools – tools we aim to optimize in future versions. When juxtaposed with an alternative design employing static taint tracking, ExChain was a mere 2% costlier but detected 5 more root causes. In contrast, while dynamic taint tracking identified the root cause for all failures, it was found to introduce a substantial performance overhead, reaching up to 50 times. This suggests that while accuracy is crucial, the accompanying performance trade-off can be significant. The results show that ExChain's hybrid taint-tracking presents a useful accuracy-overhead trade-off: an accuracy closer to dynamic taint tracking with an overhead closer to static taint tracking. This makes ExChain suitable for test or canary deployments where the modest overhead is acceptable for the ease of failure diagnosis.

The source code of ExChain is available at: https://github.com/aoli-al/exchain.

[1] ExChain could have identified the root causes of two additional failures with a better static taint tracking tool than what our current prototype uses.

```
1  class PageProvider {
2      int counter;
3      Page[] cache;
4  Req. 1void processRequest(Request req) { Req. 2
5  (1)        Page page = cache[req.pageIndex];[1]
6          try {
7  (2)             if (page.isAttached()) {      [2]
8                      logAndThrow(
9                          "attached page.");
10             }
11             page.resolve(req.sessionId);
12             counter++;
13             // render page
14         } catch (StalePageException e1) {
15             // StalePageException is
16 HTTP 200 // swallowed and not logged.
17 (6)          page.refresh(counter);
18         } catch (InitError e2) {      HTTP 500
19             throw new FatalError(e2); [4]
20         }
21     }
22     void logAndThrow(String msg) {
23         logger.info(msg);
24         throw new InitError(msg);     [3]
25     }
26 }
27 class Page {
28     int sessionId;
29     void resolve(int sessionId) {
30 (3)     attach();
31 (4)     if (sessionId != this.sessionId) {
32 (5)         throw new StalePageException();
33         }
34         ...
35         detach();
36     }
37 }
```

The method does not detach the page when a `StalePageException` is thrown.

Figure 1: Wicket [7] fails to detach page when exception occurs [56]. Green circles represents the first request that triggers a `StalePageException` and leaves the page attached. Red squares represents the second request which triggers the `InitError` and causes the system to fail.

2 Motivation

In this section, we present an example to illustrate the notion of exception-dependent failures and discuss why they are an important class of problems that do not yet have good solutions in practice.

2.1 A Motivating Example

We present a simplified real-world example to illustrate

exception-dependent failures (EDFs) and the unique challenges in diagnosing their root causes. To keep the discussion short, we pick a simple application and keep details to a minimum, noting that similar problems also manifest in more complex and popular applications such as Hadoop, HDFS, and Tomcat (as we show in §6).

Figure 1 shows a simplified code snippet from the Apache Wicket web server [7], a popular choice for dynamic web applications. When a user requests a page, Wicket returns the cached version of the page for better performance (Line 5). Next, Wicket resolves the page based on the `req.sessionId` (Line 11). This `sessionId` allows Wicket to differentiate between requests from various users. Specifically, if the `sessionId` associated with a rendered page differs from a new request, Wicket initiates re-rendering for the page. This process is implemented in the `resolve` method. It begins by attaching the page object to the current context (Line 30), ensuring it remains exclusive to the current requester and safeguarded against accidental modifications by concurrent requests. Upon completion, the method detaches the page object. The `resolve` compares the `sessionId` of the page object with that of the requester. In cases of mismatch, it throws a `StalePageException` (Line 32) so that the `PageProvider` will refresh the page (Line 17).

Unfortunately, the `resolve` method in Wicket 9.4.0 has a bug that can result in an HTTP 500 error response. Figure 2 illustrates the sequence of requests that trigger this bug. The issue involves two requests. The first request asks for a stale page, i.e., the `req.sessionId` does not match the current `page.sessionId`, which causes a `StalePageException` at Line 31. Due to this exception, the `resolve` method fails to detach the `page` object, leaving it attached to the current context. The exception is later swallowed and the page is silently refreshed in Line 17. The second request asks for the same page requested by the previous request, and the `cache` array returns correctly. However, since page is still attached, `processRequest` throws an `InitError` exception that eventually causes the HTTP 500 error response. The reporter of the above Wicket issue spent significant time to identify the root cause and to understand how it is causally related to the failure [56].

Note that the final failure (HTTP response 500) causally depends on one or more exceptions. The dependency can be explicit or implicit. An exception e_j *explicitly* depends on another exception e_i if e_i's catch block explicitly throws e_j. In the Wicket example above, `FatalError` explicitly depends on `InitError`. On the other hand, e_j *implicitly* depends on e_i if e_i changes application state in a way that causes e_j; i.e., there is a data-flow between the effect of e_i and the cause of e_j. In the Wicket example, `InitError` implicitly depends on `StalePageException` since the latter leaves the page in the attached state, causing the former.

We observe that diagnosing this exception-handling failure is challenging due to three key factors:

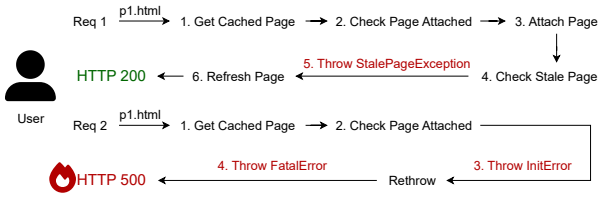


Figure 2: The request flow that triggers the HTTP 500 error when a user requests the same page twice. The labels correspond to the execution flow shown in Figure 1.

F1: Implicit state changes. Exceptions can have unexpected consequences beyond just modifying the control flow of a program. In the case of the `FatalError` exception shown in Figure 2, the presence of a `StalePageException` implicitly modifies the state of the `page` object. Specifically, the `detach` method is not called, leaving the object in an un-detached state that later triggers the `FatalError` exception. Importantly, there is no direct control-flow relationship between the `StalePageException` and the `FatalError`, meaning that simply tracing the execution path of the second request does not reveal the root cause of the failure.

F2: Silent exception handling. The root cause exception `StalePageException` is silently swallowed and not logged (Line 15). This allows the first request to continue normally despite encountering errors, improving overall system reliability and user experience. Such silent exception handling is a common practice among practitioners[59]; however, the fact that the exceptions are not logged or reported to external systems makes it difficult to diagnose failures that may occur later on (e.g., in the second request in this example).

F3: Spatially and temporally distant root cause. The error only surfaces in the second request, which can be temporally distant from when the root cause was triggered by the first request. Many unrelated error messages appear between the temporally-distant root cause and failure, and identifying if an error message is causally related to the failure can be challenging.

These key factors are not specific to the motivating example alone. For example, in a study of 10 Java libraries, Fetzer et al. found that 40% of exceptions caused implicit state changes [23], where both the control- and data-flow of the program are changed. Such state changes can cause implicit EDFs that are difficult to diagnosis. Moreover, an empirical study by Fu and Ryder found that approximately 40% of the exceptions caught by the analyzed applications were completely ignored by the program [25]. Existing empirical evidence shows the ubiquitous of the above factors in diagnosing EDF. Our experiments in §6 sampled 11 reproducible EDFs from ten popular apps (Table 2); six of the EDFs turned out to involve multiple requests/operations, and the root cause exceptions were missing in the logs for five EDFs.

2.2 Prior Work and Limitations

We now briefly discuss why closely related work is not applicable in identifying the root cause of an EDF.

Failure diagnosis. Statistical debugging [39] and backward slicing are two classic approaches to failure diagnosis. The former requires comparing many successful runs and failure runs to identify the failure root cause, a very different usage scenario from ExChain. In our evaluation, even when provided with both types of runs, the leading statistical debugging tool, GZolta, detected 6 fewer root causes than ExChain.

Backward slicing has struggled to balance accuracy and run-time overhead: static slicing [44, 49] cannot scale to analyze large-scale systems precisely. On the other, dynamic slicing techniques, like [5], can drastically slow down program execution—by up to 15 times in our evaluation, rendering them unsuitable for consistent use during routine testing or canary deployments.

Additionally, Sinha et al. suggested an integration of exception semantics with backward tracing, specifically to address null pointer exceptions. However, this technique's scope remains limited as it primarily tracks NULL propagation, making it non-generalizable for other exceptions [49].

Failure monitoring. Existing failure monitoring techniques primarily focus on identifying the failure of distributed systems [4, 13, 14, 17, 22, 26, 27, 28, 37, 38, 54]. However, identifying the failure service does not reveal the root cause of the failure. Panorama [31] and OmegaGen [40] improve the observability of large systems by monitoring grey-failures [30]. Such techniques are not sufficient to identify the root cause for EDFs as not all exceptions are triggered by grey failures; i.e, they do not work for silent exception handling (F2).

Log enhancement and analysis. These techniques focus on improving the log quality [58, 63], like logging more variable values at more selected program locations, and identifying failure-related logs [19] during post-mortem analysis. They are orthogonal to ExChain. ExChain conducts its analysis at run time, without relying on logs. Furthermore, no matter how many variables are logged at how many program locations, exception dependency cannot be figured out without the dependency analysis that we will present later. Notably, in our evaluations, only 3 failures benefitted from analyzing the first exception thrown by the application and the closest exception to the final failure.

2.3 Our Goal

Our goal is to build a tool that can automatically identify, at run time, the causal relationship between root cause exceptions and an EDF, even when the root causes are far from the failure, exceptions are silently swallowed, and dependencies are implicit. More precisely, given an EDF e_f, we aim to produce a DAG such that (1) there is a single sink node e_f, (2) source nodes represent root cause exceptions, and (3) an edge $e_i \rightarrow e_j$ indicates that e_j implicitly or explicitly depends on

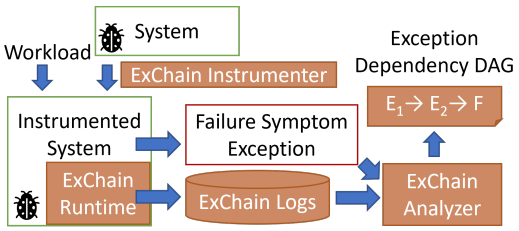


Figure 3: A high-level overview of ExChain.

e_i. In the most common case, the output is a chain of exceptions, starting from the root cause exception and ending at e_f. For example, for the aforementioned Wicket failure in Figure 1, we produce the chain `StalePageException`@32 → `InitError`@9 → `FatalError`@19 (while this notation only shows line numbers, the actual dependencies are between the *objects* corresponding to the exceptions thrown at run-time). Such dependencies can better explain to developers how root cause exceptions lead to the failure.

We aim to make our tool *easy to use*: a developer should be able to use the tool with low manual effort. The tool should be *accurate*: it should be able to find root causes of most EDFs, without generating many false positives. Finally, it should be *efficient*: its run time overhead would be modest. We envision ExChain being deployed in a test, a canary, or a reproduction environment where a modest run time overhead is acceptable for the advantage of an easy-to-use and effective failure diagnosis tool.

3 ExChain Overview

We begin with a high level overview of its workflow and how we envision ExChain being used before diving into the technical challenges.

3.1 A High Level View

ExChain consists of three components as shown in Figure 3: the *instrumenter*, the *runtime*, and the *analyzer*.

To use ExChain, a developer *proactively* uses the automated instrumenter to instrument the target application binaries. The instrumenter does not require application source code or any application-specific configuration. The instrumented binaries are then deployed in the same way as the original binaries (e.g., in an integration testing environment or in a canary deployment).

As the instrumented application executes, ExChain's *runtime* intercepts all exceptions that are raised and saves all exceptions as well as the critical runtime information required to determine their dependencies in the ExChain log file. After a failure, the developer uses ExChain *analyzer* to diagnose the failure. We assume that the developer charged with this incident troubleshooting and remediation knows the final symptom exception of the failure (potentially from the application's own log file). For instance, in our example from earlier, the developer knows that the HTTP 500 error occurred with the `FatalError` exception inside the `Page-`

`Provider` class. The analyzer takes this symptom exception as an input, and uses ExChain logs to output an exception DAG (or, most commonly, an exception chain) that has the given symptom exception as the sink. For instance, applying ExChain to the incident from Section 2, it produces `StalePageException`@32 → `InitError`@9 → `FatalError`@19. The source nodes in the analyzer output represent potential root cause exceptions.

3.2 Technical Challenges

ExChain proactively monitors all exceptions and collects runtime information every time an exception is thrown, treating every exception as a potential threat to the system. This is because predicting whether an exception will eventually cause a failure is impossible in general, and hence a selective interception may miss exceptions that are causally related to the failure. Similarly, a reactive strategy can miss important information if an exception and its information is not available when the failure happens.

The key challenge ExChain addresses is determining possible dependencies between two intercepted exceptions. There are many existing solutions to track explicit dependencies across exceptions, i.e., when one exception is thrown from the `catch` block of another. In this case, simply logging both the caught and the thrown exceptions can trivially capture their explicit dependency. There also exist program analysis tools to automatically construct such explicit dependencies across exceptions [24, 25, 33, 48]. However, these solutions cannot identify *implicit dependencies* of two exceptions where one exception causes state changes, which later causes the second exception.

The core contribution of ExChain is the ability to infer implicit dependencies of exceptions by tracking how an exception changes application states and how the changes cause subsequent exceptions. ExChain needs to address two challenges to achieve this.

First, ExChain needs to identify a set of application states to track. The set should be minimal in order to reduce the tracking overhead. To this end, ExChain uses two novel program analysis techniques to identify a small set of program memory locations to track. In particular, for each exception e_i, it identifies a set A_{e_i} of *affected memory locations* whose values are impacted by e_i and a set R_{e_i} of *responsible memory locations* whose values may cause e_i. For example, if an exception e_1 causes `null` values of the variables v_1 and v_2, and accessing v_1 later leads to a null pointer exception e_2, then $A_{e_1} = \{v_1, v_2\}$ and $R_{e_2} = \{v_1\}$. The fact that A_{e_1} and R_{e_2} overlaps readily implies that e_2 (implicitly) depends on e_1. Note that our abstraction of affected and responsible memory locations captures explicit dependencies as well: if e_2 is thrown in the catch block of e_1, A_{e_1} and R_{e_2} both include e_1 and hence e_2 depends on e_1.

Second, an exception e_2 may depend on another exception e_1 only *indirectly*. For instance, suppose e_1 causes $v_1 = -1$,

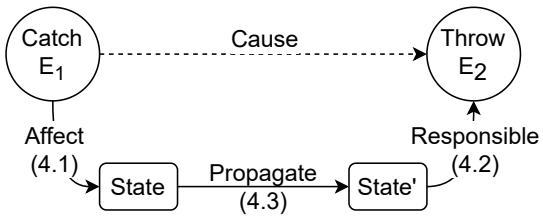


Figure 4: ExChain identifies the affected state of each exception and tracks its propagation. A causal link is established if the state causes another exception directly or indirectly.

which causes $v_2 = -1$ (e.g., via the copy `v2 = v1`), which causes the array index out of bounds exception e_2 (e.g., when executing `arr[v2]`). In this case, e_1's affected memory locations $\{v_1\}$ do not overlap with e_2's responsible memory locations $\{v_2, arr\}$, rather they are related to each other through data- and control-flow. Taint tracking can accurately capture such indirect dependencies or variables; however, it can be prohibitively expensive (up to 50× overhead for some applications in our evaluation in §6). ExChain uses a novel technique that combines static and dynamic taint tracking of a subset of memory locations that is significantly lightweight compared to dynamic taint tracking (although it can miss a small fraction of dependencies). Given the affected memory locations A_{e_1} of exception e_1, the techniques computes $Prop(A_{e_1})$, the set of all memory locations that are tainted by A_{e_1}. Using the information, ExChain decides that an exception e_2 with responsible memory location R_{e_2} depends on e_1 if the intersection of $Prop(A_{e_1})$ and R_{e_2} is not empty. Intuitively, a nonempty intersection means e_1 affects at least one memory location that, through data- and control-flow, affects at least one memory location that is responsible for e_2, and hence e_2 depends on e_1.

3.3 Scope and Limitations

ExChain has several sources of false negatives. First, ExChain cannot identify affected and responsible memory locations that are not initialized when the exception is thrown. Second, ExChain cannot track the state propagation if the exception is thrown or caught by native code or if the state propagates to other systems (e.g. an exception causes a corrupted file or disrupts API functionalities).

The ExChain is specifically tailored for programming languages that utilize exceptions for error handling, such as C# and Java. Its algorithms analyze throw and try/catch statements to track where exceptions are raised and handled. However, ExChain is less effective for languages like C and Rust, which predominantly use return values for error handling, without raising and handling failures explicitly.

4 Detailed Design

Next, we describe the detailed design of ExChain to realize the workflow from the previous section. We start by describ-

Statement	Variable	Memory Location
StalePageException:resolve		
detach();	this	Page
StalePageException:processRequest		
counter++;	counter	PageProvider.counter
page.refresh();	page	Page

Table 1: Affected state analysis for the Wicket example.

ing the main analyses in ExChain (ref. Figure 4). Note that all analyses are *dynamic* unless specified otherwise, and they track *memory locations*, which are either local variables on the stack, objects in the heap, or fields of objects in the heap.

4.1 Affected State Analysis

First, given an exception, our goal is to identify all memory locations whose *values may be affected* by the exception, i.e., their values will differ depending on whether the exception is thrown or not. For example, in Figure 1, let us consider the control-flow of the program if the `resolve` method does not throw the `StalePageException`@32. Firstly, the `resolve` method calls the `detach` method (Line 11), which modifies the internal state of the `Page` object. Next, the `processRequest` method increments the `counter` variable (Line 12). Finally, the `page.refresh()` call (Line 17) is not executed because it is inside the catch block. Therefore, the `StalePageException`@32 affects the memory locations corresponding to the `Page` object referenced by `this` variable at line 35, the `counter` field, and the `Page` object referenced by `page` variable at line 17.

However, obtaining this information at runtime can be challenging. Simply removing the `throw` statement and rerunning the request may not yield the correct result, particularly if the system is stateful.

To enable ExChain to identify memory locations affected by an exception at run-time, we develop a novel a static data-flow analysis that resembles liveness analysis [47]. Given an exception e and its corresponding stack trace ST from the current execution as input, ExChain generates a set of memory locations whose value will be altered by the exception. We represent a stack trace as a sequence of $k \geq 1$ tuples $\langle method, loc, vars \rangle$ corresponding to stack frames when the exception was thrown, where loc is the program location of the call site (for the first $k-1$ frames) or the `throw` statement (for the k-th frame), and $vars$ is a mapping of variables to their values. Our algorithm for computing the affected locations A is as follows:

1. Add thrown exception e to A.
2. For each stack frame $\langle method, loc, vars \rangle \in ST$:
   (a) Identify all instructions I_{aff} that are control dependent on the throw instruction or the corresponding invocation site loc.
   (b) For each instruction $i \in I_{aff}$ which is of the form $x = y$ or $x.f = y$ or $x.foo()$, determine (respectively) the assigned local variable, the assigned object field, or the object on which a method was invoked, and add these locations to A. Intuitively, this is because their value may be impacted by the change in control flow due to the exception. Note that concrete memory locations are obtained by resolving object references and fields via *vars*.
3. Return all affected locations A.

Table 1 shows the analysis result of the `StalePageException`. When the `StalePageException`@32 is thrown, the `detach()` statement is control dependent of the **`throw`** statement. The `detach()` statement is a method invocation of the object referenced by **`this`**. Therefore, ExChain identifies the memory location pointed by **`this`** as affected. The stack trace of this exception also contains the method `processRequest` which is at the invocation site `page.resolve(req.sessionId)` (Line 11); as this call is aborted, the control-dependent statements `counter++` and `page.refresh()` are marked as affected. Correspondingly, the affected state analysis returns two memory locations: (1) the `Page` object referenced by `this` in the `resolve` method and by `page` in the `processRequest` method, and (2) the class field `counter`.

4.2 Responsible State Analysis

Given an exception, we also need to identify all memory locations whose specific values *can cause* the exception. Note that exceptions broadly have two types of causes: (a) exceptions originating at a **`throw`** statement are usually caused by some program condition that is checked by an enclosing **`if`**; and (b) run-time exceptions can be triggered while executing expressions because of the value in some memory location (e.g., if a reference is **`null`** or if a divisor is zero).

As an example, consider the `processRequest` method shown in Figure 1. This method can throw an `ArrayIndexOutOfBoundsException` at Line 5 if `cache.length <= req.pageIndex`. The memory location referenced by `cache` and `req` are responsible for the exception. Next if `page` is attached, the method throws an `InitError` exception (Line 9). In this case, the `Page` object referenced by `page` is the responsible memory location, since it is part of the closest enclosing **`if`** condition. Note that the `InitError` is thrown indirectly by a wrapper method called `logAndThrow`. Therefore, simply analyzing the method that directly throws the exception is not sufficient to identify the responsible location. To address this, we implement several heuristics based on the semantics of the exceptions and the structure of the code.

Exception rethrown. Many exceptions are thrown explicitly in the catch block (e.g. the FatalError in Figure 1). The memory location referenced by the caught exception (e.g. `e2` at Line 18) is responsible for the new exception.

Run-time exceptions without an explicit throw statement. We maintain a list of exceptions that are thrown directly by the runtime while executing individual instructions, and handle

them specially to identify the memory locations responsible for these exceptions. For instance, if a `NullPointerException` is triggered by a method invocation instruction or an object field access, then ExChain identifies the corresponding object reference as the culprit.

Exceptions thrown by throw statements. When an exception is thrown by a `throw` statement, ExChain employs a backward control-flow analysis to identify the memory locations responsible for the exception. The algorithm takes an exception e and its corresponding stack trace ST as input and returns a set of memory locations R that are responsible for the input exception. The stack trace is again represented as a sequence of tuples $\langle method, loc, vars \rangle$ as before. The algorithm is as follows:

1. For each frame $\langle method, loc, vars \rangle \in ST$ starting from the top of the stack:
   (a) Find the closest branch statement of the current frame location loc based on the control-flow graph of the method.
   (b) If no such branch is identified in this method, go to step 1.
   (c) Collect all variables or object fields that are referenced by the condition expression of the branch statement. Use $vars$ to resolve variables to memory locations, and add these to R. Break and go to step 2.
2. Return the set R.

For example, in the `InitError` shown in Figure 1, ExChain first analyzes the `logAndThrow` method at line 24, and then the `processRequest` method at line 9. Since the `throw` statement in question is not dominated by any branch condition in the `logAndThrow` method, the analysis continues up the call stack to the `processRequest` method, where it identifies the closest branch condition as `page.isAttached()`. ExChain then resolves the local variable reference and returns a singelton set containing the memory location referenced by `value` as R. Since this value is the same object included in the affected set for `StalePageException` (ref. Section 4.1), ExChain can establish causality between `StalePageException` and `InitError`.

4.3 Hybrid Taint Flow Analysis

In general, ExChain needs to consider how the values affected by some exception e_i propagate to other values before they become responsible for some other exception e_j (§3.2). This is done using *taint analysis*. The main idea behind taint analysis is to associate some information with program values (e.g., that they are affected by exception e_i) and propagate this to other values that are derived from the former.

Traditional *dynamic* taint analysis works by instrumenting program code to propagate taint information at every instruction, such as copying local variables, performing arithmetic computation, or invoking method calls. This instrumentation introduces excessive overhead to the application [11, 15, 20],

```
1  class Foo {
2      int value = 0;
3      Taint valueT = new Taint("const:0");
4      Taint thisT = new Taint("obj:Foo");
5  }
6  void m() {
7      int i1 = 10;
8      // create a new taint because i1 is created
       ↪ from a constant.
9      Taint i1T = new Taint("const:10");
10     int i2 = i1;
11     // Passing the taint information from i1 to
       ↪ i2.
12     Taint i2T = i1T;
13     Foo e1 = new Foo();
14     Foo e2 = e1;
15     int i3 = e1.value;
16     // Passing the taint information from
       ↪ e1.value to i3.
17     Taint i3T = e1.valueT;
18 }
```

Passing the taint information from e1 to e2 is not necessary!

Figure 5: A simple program to demonstrate how dynamic taint analysis tools tracks the taint tag for heap objects and local variables. The original code is not highlighted.

making it difficult to apply dynamic taint analysis techniques to large, complex enterprise-level applications even in an integration/canary test environment. Alternatively, *static* taint analysis reconstructs the dynamic behavior of a program using only static code analysis [8, 41], which introduces zero overhead to the application while trading off precision.

Hybrid taint analysis. Neither static nor dynamic taint analysis alone can achieve both accurate and efficient taint tracking. Therefore, it is natural to ask if it is possible to combine these two approaches to achieve a high precision and a low overhead. ExChain implements a novel hybrid taint analysis that leverages the following two observations.

First, the main overhead introduced by dynamic taint analysis comes from maintaining the taint information for local primitives, such as integers and booleans. In contrast, tracking heap objects can be done more efficiently and accurately by adding only taint information to the heap object itself.

For example, Figure 5 illustrates an instrumented Java program that tracks taint information using dynamic taint analysis. The fields `valueT` and `thisT` of the class `Foo` are used to track the taint information of `Foo.value` and `Foo` respectively. The dynamic taint analysis tool creates a taint reference for each local primitive and updates the taint information accordingly (Line 9, 12, and 17). For object references, the tool does not create taint references if they point to heap objects whose taint information is already being maintained by corresponding fields (Line 3-4). When the `thisT` field of an object is updated, all references pointing to that object are automatically updated as well. In our evaluation, tracking local primitives introduces 87-5005% overhead but tracking heap objects only introduces 1-10% to the system.

Second, local primitives can be efficiently tracked using

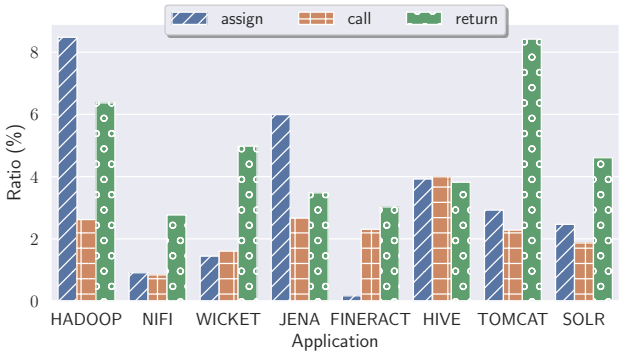


Figure 6: Analysis result of the percentage of data-flows between heap and local objects using CodeQL.

static taint analysis offline and static taint analysis produces more accurate result for local primitives compared to heap objects [29]. Heap objects can be manipulated in various ways by the program, such as being passed between functions or being dynamically allocated and deallocated. This makes it harder to track the flow of data through the program and to accurately determine which inputs have tainted a particular object. On the other hand, local primitives are typically only modified within a single function or block of code, making it easier to trace their flow of data.

ExChain utilizes these observations and dynamically instruments heap objects and adds taint information at run time, introducing only a constant overhead per exception. For local primitives, ExChain uses static taint analysis to track data-flow offline, introducing no overhead at run time.

One limitation of tracking different types of variables differently is that it may miss data flow between heap objects and local primitives. For example, in Figure 5, taint information will be lost between `e1.value` and `i3` because ExChain maintains the taint information of `e1.value` dynamically and the taint information of `i3` statically. This limitation may affect the accuracy of ExChain's analysis.

To better understand the impact, we used CodeQL [16] to statically analyze multiple popular cloud services. Specifically, we focused on three types of statements: assignments, method calls, and method returns. Figure 6 presents the results of our analysis. Across all applications, we found that less than 8.5% of assignment statements, less than 4% of method call statements, and less than 8.4% of method return statements had data flow from heap objects to local primitives. Our findings confirmed that the majority of data flow occurs between heap-to-heap and local-to-local, which can be effectively handled by hybrid taint analysis. In our evaluation of 11 reproduced failures, we found that ExChain can identify the root cause of 6 issues by only tracking heap objects, while four issues require only tracking local primitives. Only one failure requires tracing the data-flow between heap objects and local primitives. Besides, tracking local primitives introduces 17x overhead to the target system compared to only tracking heap objects. Overall, by selectively tracing heap objects and local primitives, ExChain can identify the root cause of most failures with minimal overhead.

4.4 Putting it Together

We now describe how different pieces fit together in the workflow of ExChain as shown in Figure 3. At runtime, ExChain intercepts every exception that is thrown by the application. On each exception, ExChain performs both affected and responsible state analysis and logs the results in the ExChain logs (the results are cached and reused when the same exception is thrown multiple times). It also performs the dynamic part of its hybrid taint analysis. For affected states that are heap objects, ExChain marks them with a unique ID of the exception. For affected states that are local primitives, ExChain logs the stack slot number and the corresponding exception. When analyzing responsible states, ExChain checks if the memory location is a heap object that contains any labels of previous exceptions. If a label is found, ExChain records in its logs the causal link between the current exception and the exception represented by the label. If the memory location is a local primitive, ExChain logs the stack slot number and the corresponding exception.

To diagnose a failure, the user uses the ExChain analyzer offline with the target symptom exception of the failure. The analyzer first performs the static part of its hybrid taint analysis with the stack slots of affected states as sources and the stack slots of responsible states (the states are retrieved from ExChain logs). If the static taint analysis reports a flow from a source to a sink, ExChain reports the causal link between the corresponding exceptions. Finally, the analyzer returns a DAG (or most commonly, a chain) that has the symptom exception as the sink node. The source nodes in the returned DAG represents the root cause exceptions.

5 Implementation

Our ExChain implementation consists of 9,000 lines of code written in Kotlin, Java, and C++. It instruments the compiled bytecode of the target system and attaches dynamic monitors to the runtime. We use JVM Tool Interface (JVMTi) [35] to capture all exceptions thrown by the target application and map local variables and heap fields to memory locations. The affected and responsible state analyzer are developed on top of ASM [9]. Our dynamic taint analysis is based on Phosphor [11], and the static taint analysis is implemented using Soot [53] and FlowDroid [8]. Although our algorithm is not tied to a specific implementation of JDK and JVM, we only execute target applications using OpenJDK 16 because the underlying dynamic taint analysis tool requires APIs that are only available after OpenJDK 16.

Dynamic Entry Point Inference. We have also implemented a dynamic entry point inferring technique that allows the static taint analysis to provide more accurate results.

Client-server architecture for cloud services typically involves multiple public interfaces, each serving as an entry

```
int main() {
    while (true) {
        Request r = waitForNewRequest();
        dispatchRequest(r);
    }
}
class Server {
    @Endpoint("create")
    Response createUser(Request r) {
        method1(); // may throw Exception1
    }
    @Endpoint("remove")
    Response removeUser(Request r) {
        method2(); // may throw Exception2
    }
}
```

Figure 7: A simple web server with two endpoints.

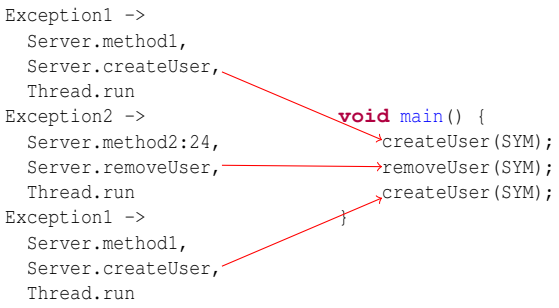


Figure 8: EXCHAIN constructs a main method based on the exception trace collected at runtime.

point to the application. Figure 7 illustrates a simple web server with two public interfaces: `create` and `remove`. The main function of the server is a hot loop that waits for incoming requests and dispatches them to respective endpoints.

The dynamic design of modern web frameworks such as Spring [51] and Wicket [7], which dispatch requests to different worker threads through reflection and dependency injection, makes it difficult for the static taint analyzer to construct an accurate call graph [6]. This design significantly limits the completeness of the static analysis.

To address this problem, EXCHAIN leverages the exception traces collected at runtime. For each exception, EXCHAIN identifies the deepest stack frame that contains application code based on the package name of the caller. It then constructs a main method that invokes the corresponding application code in sequential order. For example, Figure 8 demonstrates an exception trace of Figure 7. EXCHAIN finds the deepest method that contains application code for each exception and creates a new main method that invokes the identified methods sequentially. By doing so, EXCHAIN is able to generate a new main method for the application that starts with the identified application code.

In our evaluation, dynamic entry point inference helps EXCHAIN to identify the root cause for two more issues compared to using the original main method of the application.

Issue	Multi-Run	Cause Logged	# Excp Dist	# Excp Total
WICKET-6908	✓	✓	5	8
JENA-324	✓	✓	792	796
FINERACT-1211	✗	✓	1	58
MAPREDUCE-6654	✗	✓	11	117
HADOOP-17812	✗	✗	1	24
WICKET-6249	✓	✗	7	11
HDFS-4128	✓	✗	7	115
HIVE-13410	✓	✓	15	51
NIFI-8249	✓	✓	1	47
SOLR-16363	✗	✗	1	171
TOMCAT-65131	✗	✗	1	13

Table 2: Basic information of EDFs. 'Multi-Run' indicates if the root cause and the final exception occur in different operations. 'Cause Logged' shows if the root cause exception is logged by the application. 'Dist' shows the number of exceptions thrown between the root cause and the final exception. 'Total' shows the total number of exceptions thrown by the application during the reproduction.

6 Evaluation

We evaluate EXCHAIN to answer the following questions: (1) How does EXCHAIN compare to state-of-the-art failure diagnosis techniques in identifying the root cause for EDFs? (2) How do our analysis techniques help improve the accuracy-performance tradeoffs of EXCHAIN? We conduct all experiments on an Ubuntu server with an Intel Xeon 1290P processor and 128 GB of memory. We set an 8-hour time limit for static taint analysis, with a maximum heap size of 32 GB.

6.1 Methodology

For our evaluation, we looked at popular open source applications maintained by the Apache Foundation. We query the Jira issue tracking system [32] with a list of keywords such as "exception handling". From the result of our query (run on 11-15-2022), we closely examined the latest 30 issues return that are indeed software bugs related to exception handling and have clearly described failure symptoms, and were able to reproduce 11 of them (the remaining ones do not provide an instruction to reproduce the failure).

As such, these 11 *reproducible* incidents spanning 10 open source applications form the core of our evaluation setup. Table 2 shows the basic information of the reproduced 11 issues. We reproduce all failures by running the service in production configuration except for NIFI-8249, which uses a customized class loading mechanism that is not supported by the underlying taint analysis framework. As a result, we reproduced this issue using unit tests. We manually analyze the issue report summary and developer conversations in the ticket to identify the root cause to serve as the ground truth. Note that our system does not need or have access to this issue report.

As shown in Table 2, many of the failures are non-trivial:

Issue	ExChain	Statistical		SL4J	Log	
		Ochiai	Taran.		First	Nearst
WI-6908	✓	✗	✗	N/A	✗	✗
JE-324	✓	✗	✗	✗	✓	✓
FI-1211	✓	N/A	N/A	N/A	✗	✗
MA-6654	✓	✗	✗	N/A	✗	✗
HA-17812	✓	✗	✓	N/A	✗	✗
WI-6249	✓	✗	✗	N/A	✗	✗
HD-4128	✓	✓	✓	N/A	✗	✗
HI-13410	✓	✗	✗	N/A	✓	✗
NI-8249	✗	✗	✗	N/A	✓	✓
SO-16363	✗	N/A	N/A	N/A	✗	✗
TO-65131	✗	N/A	N/A	N/A	✗	✗

Table 3: The analysis result of each issue. Ochiai and Tarantula are two statistical debugging techniques implemented by GZoltar. N/A means GZoltar and Slicer4J are not applicable to the target application. First and Nearest are two debugging techniques that focus on examining the initial exception thrown by the application and the closest exception to the final failure from logs.

their root causes and failures happen in different executions, root causes are not logged, and many (unrelated) exceptions separate the failures from their the root causes.

6.2 End-to-End Evaluation

For the end-to-end evaluation of accuracy, we compare ExChain with three state-of-the-art fault localization techniques: statistical debugging (GZoltar [12] with two different ranking algorithms Ochiai [3] and Tarantula [34]), slicing (Slicer4J [5]) and Log analysis. GZoltar requires workloads that contains both pass and fail cases and we use the unit tests associate with the application. If the existing unit test does not cover the reproduced failure, we manually implement one. GZoltar return a ranked list of statements that may be related to the failure based on their relevance to the failure and Slicer4J returns a ranked list of statements that are data dependent of the failure. If the containing method of a statement reported by the tool that throws the root cause is ranked top 200 of the list, we report a true positive. In our analysis of application logs, we utilized two strategies: "First" representing the initial exception thrown and "Nearest" representing the closest exception to the final failure.

Note that, as discussed in Section 2.2, it is much more costly to use GZoltar and Slicer4J: GZoltar requires many successful runs and failure runs, and Slicer4J requires re-executing a failed run. In contrast, ExChain allows diagnosis right after a failure run.

Table 3 presents the analysis result. ExChain successfully identified the root cause for most issues (8 out of 11). GZoltar only identified the root cause for 2 issues among the 8 on which we could run it. GZoltar was unable to analyze Fineract, Tomcat, and Solr due to incompatibilities with the building system used by these applications. For HDFS-4128, Ochiai identified the root cause in the top 2 predicted statements. However, for HADOOP-17812 and WICKET-6249, Ochiai failed to predict the root cause within the top 200 statements. For Tarantula, it predicted the root cause statements within the top 30 statements for HADOOP-17812 and HDFS-4128. For 5 issues, GZoltar cannot report any root cause statements due to insufficient pass/fail executions. Our experimental results highlight that ExChain is more capable than GZoltar in identifying the root cause for EDFs.

To gain insight into the challenges of diagnosing failures using dynamic variable dependency tracking, we performed a backward slicing for all failures using Slicer4J [5]. However, we were able to use the tool to reproduce only one issue: JENA-324; Slicer4J could not used with the other issues due to incompatible Java versions. For JENA-324, Slicer4J reported 3741 statements related to the final failure, with the root cause identified at the 3628th statement. This suggests that relying on variable dependency for failure diagnosis can lead to information overload, potentially overwhelming developers. Slicer4J can introduce significant performance overhead (up to 15×) Ahmed et al. [5]).

Finally, for the log based approaches, the "First" strategy identified the root cause for three out of the total failures. The "Nearest" strategy pinpointed the root cause for only two distinct failures. It's crucial to note that even in our experiments with smaller workloads, numerous exceptions can occur before and after the root cause, particularly in long-running services.

6.3 Accuracy vs. Performance Tradeoff

To put the accuracy-performance tradeoff of ExChain in context and explain the value of our optimizations, we consider two other hypothetical designs SI+Static and SI+Dynamic (SI stands for affected/responsible state identification). Similar to ExChain, both log exceptions at runtime and identify affected and responsible states using the algorithms mentioned in §4.1 and §4.2. They differ in how they analyze taint flow between affected to responsible states: SI+Static uses fully static taint analysis and SI+Dynamic uses fully dynamic taint analysis. In contrast, ExChain uses a hybrid taint analysis: dynamic analysis of heap objects and static analysis of local primitives. Note, that SI+Static logs all exceptions as well as the corresponding stack traces and runs affected and responsible state analysis offline. SI+Static is expected to offer low run-time overhead but least accurate diagnosis results. Conversely, SI+Dynamic is expected to offer the highest overhead but also the highest diagnosis accuracy.

To evaluate the performance impact of ExChain, we identify benchmarks for 7 out of the 10 applications. For Hadoop, Solr, MapReduce, and HDFS, we used the built-in benchmarks to generate workloads [18, 45, 50, 52]. For Fineract, Wicket, and Tomcat, we use the Apache HTTP benchmarking tool [1] to measure their performance. For each benchmark, we measure both throughput and latency. We encountered issues when attempting to benchmark NIFI in production mode due to dynamic class loading as described earlier. We were

Issue	ExChain		SI+Static		SI+Dynamic	
	TP	FP	TP	FP	TP	FP
WICKET-6908	✓	1	✓	1	✓	0
JENA-324	✓	0	✗	0	✓	0
FINERACT-1211	✓	0	✓	0	✓	0
MAPREDUCE-6654	✓	0	✗	0	✓	0
HADOOP-17812	✓	0	✗	0	✓	0
WICKET-6249	✓	0	✓	0	✓	0
HDFS-4128	✓	0	✗	0	✓	0
HIVE-13410	✓	0	✗	0	✓	0
NIFI-8249	✗	0	✗	0	✓	0
SOLR-16363	✗	0	✗	0	✓	0
TOMCAT-65131	✗	0	✗	0	✓	0
Sum	8/11	1	3/11	1	11/11	0

Table 4: Analysis result of each issue. ExChain cannot identify the root cause for NIFI-8249 and SOLR-16363 because of the imprecise analysis result returned by the underlying static taint analysis tool. ExChain cannot identify the root cause for TOMCAT-65131 because of the data-flow between heap objects and local primitives.

also unable to find a representative benchmark workload for Jena and Hive.[2] Solr only reports throughput and MapReduce only reports latency. All remaining benchmarks report both latency and throughput.

Accuracy Results. Table 4 presents the accuracy results for ExChain and two baselines including the number of true positives (TP) and false positives (FP). A true positive for a technique means that it successfully identifies the root cause exception and correctly reported causal relationship between the root cause exception and the final failure described in the issue. A false positive means that the technique reports an exception that is not mentioned by the reporter and fixing the exception does not prevent the final failure.

The result shows that ExChain successfully identified the root cause for most issues (8 out of 11) with only 1 false positive. SI+Dynamic successfully identified all root causes without any false positive (at the cost of huge run-time overhead that we discuss later). On the other hand, SI+Static could identify root causes for only 3 issues with 1 false positive.

Recall that SI+Dynamic is based on the affected and responsible variables identified by our analysis described in §4. The fact that it can successfully identify all root causes shows the effectiveness of the analysis algorithms.

We also investigated the three failures for which ExChain failed to report the true root cause. For two of the cases (NIFI-8249 and SOLR-16363), ExChain failed because of the imprecise analysis result returned by the underlying static taint analysis while tracking local variables. Only for one case (TOMCAT-65131), ExChain failed because of its design limitation of not being able to track data-flow between heap and local objects.

Performance Result. Figure 9 shows the latency results reported by 6 applications. On average, ExChain incurred 1%-12% overhead on latency, only 2% more than SI+Static. In contrast, SI+Dynamic incurred 87%-5015% overhead. The throughput result in Figure 10 shows a similar trend: ExChain incurred 1%-11% degradation on throughput, while SI+Static also incurred 1%-11% degradation. In contrast, SI+Dynamic incurred 48%-99% degradation.

[2]A third-party benchmark for an old version of Hive was available, but our dynamic taint analysis tool could not instrument the benchmark application.

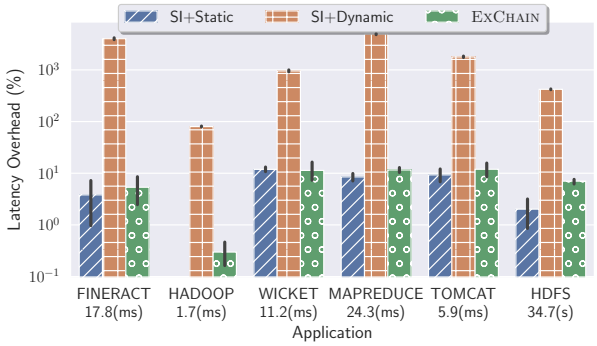


Figure 9: The latency overhead for different applications in *log scale*. Lower is better.

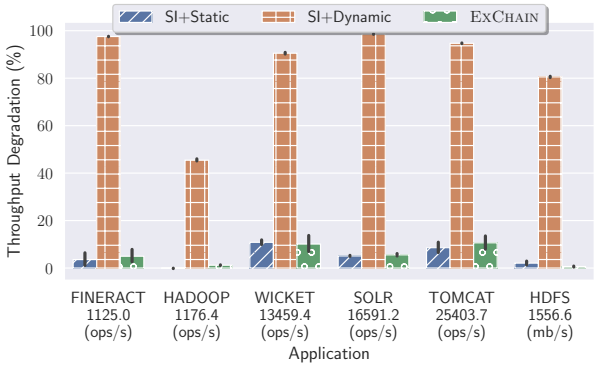


Figure 10: The throughput degradation for different applications. Lower is better.

Our evaluation shows that ExChain achieves a better balance between performance and accuracy than SI+Dynamic and SI+Static. Specifically, it achieves an accuracy closer to SI+Dynamic, with a cost closer to SI+Static. In fact, ExChain successfully identified the root cause of all failures whose affected and responsible states are heap objects, whereas SI+Static only identified the root cause for 3 such failures. Moreover, ExChain reported only one false positive out of 11 issues, demonstrating its high accuracy. In terms of performance, ExChain introduces an average overhead of only 8%, making it feasible to deploy in an integration test or canary environment.

7 Discussion

Our current focus was on using ExChain was in a test/canary environment where a moderate performance overhead ($\approx 8\%$) may be acceptable. One natural question is if our approach may eventually be amenable to be run in production with a lower overhead.

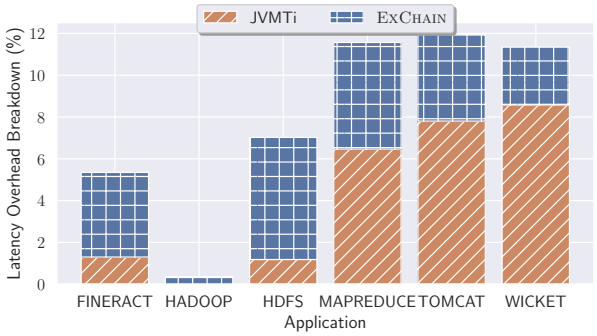


Figure 11: Latency overhead break down for different applications.

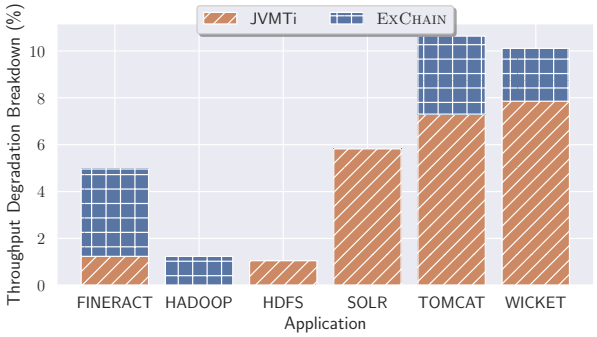


Figure 12: Throughput degradation break down for different applications.

To understand the future feasibility of extending ExChain we investigate the sources of the overhead. We divide the system into two components: JVMTi and ExChain. Figure 11 and Figure 12 show the breakdown of overhead introduced by each component. JVMTi represents the aforementioned overhead due to JVMTi. ExChain represents the overhead introduced by the central design, including logging all exceptions and their corresponding stack traces, computing the affected and responsible states, and storing taint information of heap objects.

From our observations, less than half of the total overhead is attributed to the core components of ExChain, while the rest originates from JVMTi. JVMTi can disable several JIT optimizations when attached to the JVM, which affects the overhead. An alternative way to intercepting exceptions (e.g., through instrumentation or a better JVM mechanism similar to .NET's first-chance-exception[21]) could reduce this overhead substantially to enable closer-to-production acceptable overhead ($\leq 5\%$).

8 Other Related Work

We discussed some key related efforts and their limitations in §2.2. Here, we discuss other related work.

Statistical debugging technique. There is a rich body of work focusing on statistical debugging [2, 12, 34, 43, 55, 57, 61]. Such techniques are effective if the developer provides both failing and passing executions. Unfortunately, such data is not always available. In our evaluation, we show that with existing test suit, GZoltar can only identify the root cause for one EDF. Moreover, statistical debugging aims to identify events (e.g., exceptions) that are *correlated* to failures, rather than finding the causal dependencies among multiple events.

Failure reproducing. Kasikci et al. showed that it is possible to reproduce failures with low overhead instrumentation using hardware features [36]. Pensieve reconstructs failing executions using dependency analysis of runtime events [62]. ExChain is complementary to failure reproducing techniques and help developers to pinpoint the root cause efficiently.

Failure handling testing. ChaosMachine [60] and Filibuster [42] use chaos engineering to test failure handling logic of the application. Such techniques are useful in identifying bugs in failure handling logic.

Speeding up dynamic taint analysis. JetStream uses parallel execution and record and replay techniques to improve the performance of dynamic information flow tracking [46]. Iodine uses static analysis to remote runtime monitors if the data-flow can be determined statically [10]. Both tool show that taint analysis is useful in debugging and failure diagnosis. ExChain uses exception-focused hybrid taint analysis and focuses on identifying the root cause for EDFs.

9 Conclusions

In some sense, ExChain solves a particularly hard problem — the very practices of good software engineering at scale (e.g., throwing exceptions, silent handling) also end up creating subtle exception-dependent failure modes that are incredibly hard to debug! Our key observation is that unlike basic exception chains, EDFs can entail subtle stateful dependencies between the root cause and the eventual failure mode.

In designing ExChain, we addressed fundamental challenges in applying program analysis techniques to balance the performance and overhead in tracking such stateful dependencies in exception handling failures. ExChain helps developers diagnose EDFs using a familiar exception-trace like abstraction akin to traditional debugging workflows. Our evaluation showed that ExChain is able to successfully diagnose subtle issues that stumped expert developers in popular applications with little to no manual effort and that it significantly outperforms state-of-art techniques. While our current implementation offers sufficient performance for test and canary deployments, our core design contributions are amenable to production deployments at scale as well.

10 Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and constructive feedback. This research was supported in part by seed funding from CMU's CyLab and by the NSF grant CCF-2119184. A portion of this work was carried out during an internship at Microsoft Research.

References

[1] ab. ab - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html. Accessed: 2023-02-23.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 89–98, 2007. doi: 10.1109/TAIC.PART.2007.13.

[3] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82:1780–1792, 2009.

[4] Marcos K. Aguilera and Michael Walfish. No time for asynchrony. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, page 3, USA, 2009. USENIX Association.

[5] Khaled Ahmed, Mieszko Lis, and Julia Rubin. Slicer4J: A Dynamic Slicer for Java. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[6] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. Static analysis of java enterprise applications: Frameworks and caches, the elephants in the room. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 794–807, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3386026. URL https://doi.org/10.1145/3385412.3386026.

[7] Apache Wicket. Apache wicket. https://wicket.apache.org/. Accessed: 2023-02-23.

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 259–269, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327848. doi: 10.1145/2594291.2594299. URL https://doi.org/10.1145/2594291.2594299.

[9] asm. Asm: A java bytecode engineering library. https://asm.ow2.io/index.html. Accessed: 2023-02-23.

[10] Subarno Banerjee, David Devecsery, Peter M. Chen, and Satish Narayanasamy. Iodine: Fast dynamic taint tracking using rollback-free optimistic hybrid analysis. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 490–504, 2019. doi: 10.1109/SP.2019.00043.

[11] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, page 83–101, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450325851. doi: 10.1145/2660193.2660212. URL https://doi.org/10.1145/2660193.2660212.

[12] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. Gzoltar: an eclipse plug-in for testing and debugging. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 378–381, 2012. doi: 10.1145/2351676.2351752.

[13] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, mar 1996. ISSN 0004-5411. doi: 10.1145/226643.226647. URL https://doi.org/10.1145/226643.226647.

[14] Wei Chen, S. Toueg, and M.K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(1):13–32, 2002. doi: 10.1109/12.980014.

[15] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, page 196–206, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937346. doi: 10.1145/1273463.1273490. URL https://doi.org/10.1145/1273463.1273490.

[16] Codeql. Codeql. https://codeql.github.com/. Accessed: 2023-02-23.

[17] A. Das, I. Gupta, and A. Motivala. Swim: scalable weakly-consistent infection-style process group membership protocol. In *Proceedings International Conference on Dependable Systems and Networks*, pages 303–312, 2002. doi: 10.1109/DSN.2002.1028914.

[18] dfsio. Hadoop hdfs dfsio. https://github.com/c9n/hadoop/blob/master/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-jobclient/src/test/java/org/apache/hadoop/fs/TestDFSIO.java. Accessed: 2023-02-23.

[19] Pradeep Dogga, Karthik Narasimhan, Anirudh Sivaraman, Shiv Kumar Saini, George Varghese, and Ravi Netravali. Revelio: Ml-generated debugging queries for finding root causes in distributed systems. In *Conference on Machine Learning and Systems*, 2022.

[20] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 32(2), jun 2014. ISSN 0734-2071. doi: 10.1145/2619091. URL https://doi.org/10.1145/2619091.

[21] fce. Appdomain.firstchanceexception event. https://learn.microsoft.com/en-us/dotnet/api/system.appdomain.firstchanceexception?view=net-8.0. Accessed: 2023-02-23.

[22] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computers*, 52(2):99–112, 2003. doi: 10.1109/TC.2003.1176979.

[23] C. Fetzer, K. Hogstedt, and P. Felber. Automatic detection and masking of non-atomic exception handling. In *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, pages 445–454, 2003. doi: 10.1109/DSN.2003.1209955.

[24] C. Fu, A. Milanova, B.G. Ryder, and D.G. Wonnacott. Robustness testing of java server applications. *IEEE Transactions on Software Engineering*, 31(4):292–311, 2005. doi: 10.1109/TSE.2005.51.

[25] Chen Fu and Barbara G. Ryder. Exception-chain analysis: Revealing exception handling architecture in java server applications. In *29th International Conference on Software Engineering (ICSE'07)*, pages 230–239, 2007. doi: 10.1109/ICSE.2007.35.

[26] Trinabh Gupta, Joshua B. Leners, Marcos K. Aguilera, and Michael Walfish. Improving availability in distributed systems with failure informers. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 427–441, Lombard, IL, April 2013. USENIX Association. ISBN 978-1-931971-00-3. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/leners.

[27] Andreas Haeberlen and Petr Kuznetsov. The fault detection problem. In Tarek Abdelzaher, Michel Raynal, and Nicola Santoro, editors, *Principles of Distributed Systems*, pages 99–114, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-10877-8.

[28] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 175–188, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595935915. doi: 10.1145/1294261.1294279. URL https://doi.org/10.1145/1294261.1294279.

[29] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, page 54–61, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581134134. doi: 10.1145/379605.379665. URL https://doi.org/10.1145/379605.379665.

[30] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 150–155, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350686. doi: 10.1145/3102980.3103005. URL https://doi.org/10.1145/3102980.3103005.

[31] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 1–16, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/huang.

[32] jira. Asf jira. https://issues.apache.org/jira/secure/Dashboard.jspa. Accessed: 2023-02-23.

[33] Jang-Wu Jo, Byeong-Mo Chang, Kwangkeun Yi, and Kwang-Moo Choe. An uncaught exception analysis for java. *J. Syst. Softw.*, 72(1):59–69, jun 2004. ISSN 0164-1212. doi: 10.1016/S0164-1212(03)00057-8. URL https://doi.org/10.1016/S0164-1212(03)00057-8.

[34] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, page 467–477, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 158113472X. doi: 10.1145/581339.581397. URL https://doi.org/10.1145/581339.581397.

[35] JVMTI. Jvm(tm) tool interface. https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html. Accessed: 2023-02-23.

[36] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 344–360, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815412. URL https://doi.org/10.1145/2815400.2815412.

[37] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 279–294, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309776. doi: 10.1145/2043556.2043583. URL https://doi.org/10.1145/2043556.2043583.

[38] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Taming uncertainty in distributed systems with help from the network. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450332385. doi: 10.1145/2741948.2741976. URL https://doi.org/10.1145/2741948.2741976.

[39] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on software engineering*, 32(10):831–848, 2006.

[40] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 559–574, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL https://www.usenix.org/conference/nsdi20/presentation/lou.

[41] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 229–240, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316514. doi: 10.1145/2382196.2382223. URL https://doi.org/10.1145/2382196.2382223.

[42] Christopher S. Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. Service-level fault injection testing. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 388–402, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386388. doi: 10.1145/3472883.3487005. URL https://doi.org/10.1145/3472883.3487005.

[43] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3), aug 2011. ISSN 1049-331X. doi: 10.1145/2000791.2000795. URL https://doi.org/10.1145/2000791.2000795.

[44] Mangala Gowri Nanda and Saurabh Sinha. Accurate interprocedural null-dereference analysis for java. In *2009 IEEE 31st International Conference on Software Engineering*, pages 133–143, 2009. doi: 10.1109/ICSE.2009.5070515.

[45] nnbench. Hadoop benchmarking. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/Benchmarking.html. Accessed: 2023-02-23.

[46] Andrew Quinn, David Devecsery, Peter M. Chen, and Jason Flinn. JetStream: Cluster-Scale parallelization of information flow queries. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 451–466, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/quinn.

[47] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

[48] Martin P. Robillard and Gail C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.*, 12 (2):191–221, apr 2003. ISSN 1049-331X. doi: 10.1145/941566.941569. URL https://doi.org/10.1145/941566.941569.

[49] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. Fault localization and repair for java runtime exceptions. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, page 153–164, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583389. doi: 10.1145/1572272.1572291. URL https://doi.org/10.1145/1572272.1572291.

[50] solr. Solr jmh-benchmarks module. https://github.com/apache/solr/tree/main/solr/benchmark. Accessed: 2023-02-23.

[51] spring. Spring | home. https://spring.io/. Accessed: 2023-02-23.

[52] terasort. Package org.apache.hadoop.examples.terasort. https://hadoop.apache.org/docs/r3.0.0/api/org/apache/hadoop/examples/terasort/package-summary.html. Accessed: 2023-02-23.

[53] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13. IBM Press, 1999.

[54] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In Nigel Davies, Seitz Jochen, and Kerry Raymond, editors, *Middleware'98*, pages 55–70, London, 1998. Springer London. ISBN 978-1-4471-1283-9.

[55] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*, pages 364–374, 2009. doi: 10.1109/ICSE.2009.5070536.

[56] WICKET-6908. Wicket-6908. https://issues.apache.org/jira/browse/WICKET-6908. Accessed: 2023-02-23.

[57] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1): 290–308, 2014. doi: 10.1109/TR.2013.2285319.

[58] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 293–306, USA, 2012. USENIX Association. ISBN 9781931971966.

[59] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 249–265, USA, 2014. USENIX Association. ISBN 9781931971164.

[60] Long Zhang, Brice Morin, Philipp Haller, Benoit Baudry, and Martin Monperrus. A chaos engineering system for live analysis and falsification of exception-handling in the jvm. *IEEE Transactions on Software Engineering*, 47(11):2534–2548, 2021. doi: 10.1109/TSE.2019.2954871.

[61] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. Boosting spectrum-based fault localization using pagerank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 261–272, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350761. doi: 10.1145/3092703.3092731. URL https://doi.org/10.1145/3092703.3092731.

[62] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 19–33, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132768. URL https://doi.org/10.1145/3132747.3132768.

[63] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 565–581, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132778. URL https://doi.org/10.1145/3132747.3132778.